**ECE 1310 Exercise 3**

**Date:** 10/22/24, Tuesday, week 9

**Due Date:** 11/12/24, Tuesday, week 12

**Total points:** 76 points

1. (10%) Swapping variable.

In introduction to programming, there is a basic skill that can be very confusing, the swapping or switching of two variables.

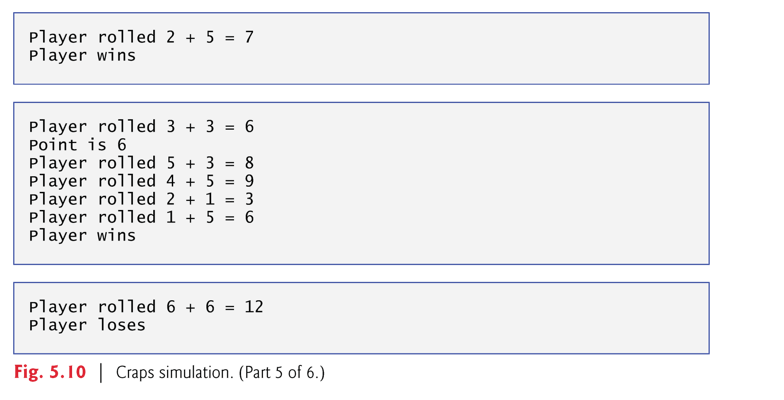
Given two int variables a and b, with a = 3, and b = 4, how can we write code so that a = 4, b = 3? The simple minded way of setting a to b (so that a = 4, the value of b) and then setting b to a (so that b = 3, the value of a) does not work. This says that a = b; b = a; does not set a to 4 and b to 3 after execution.

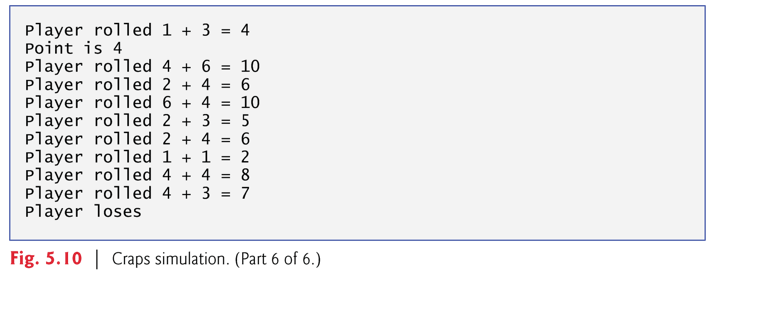
1. Consider the 3 lines of C code: int a = 3, b = 4; a = b; b = a; **Explain** what values of a and b we get after executing these 3 lines of C code. Do we have values of a and b switched after this (i.e. a set to 4, value of b, and b set to 3, value of a)? Why?
2. Write C code / program / function with cout statement(s) to verify your argument in (a) *programmatically* and show the output here (you do NOT have to attach code).
3. The correct way of doing this is: introduce a third temporary variable c to save tentatively the value of a (or the value of b): c = a;, a = b; b = c; Now explain what values of a and b we get after executing such instructions (assuming a = 3 and b = 4 to start with as in (a)). ). Do we have values of a and b switched after this? Why?
4. Write C code / program / function with cout statement(s) to verify your argument in (a) *programmatically* and show the output here (you do NOT have to attach code).

**Game of craps**

Game of craps using two dice (and random number generator) is covered in the class.

Some typical runs of games of craps are in shown in the slides (or in the book)





1. **(20%) Modify / enhance** the source code of figure 5\_10.cpp so that in the case of “Player rolled 2+5 = 7” “Player wins” as in Figure 5.10, part 5 of 6, this is displayed in one line (instead of two lines), and in the case of “Player rolled 6 + 6 = 12”, “Player loses”, as in Figure 5.10, part 6 of 6, this is also displayed in just one line, not two lines.

And also in the case of “Player rolled 3 + 3 = 6”, “point is 6” followed by 4 lines of attempt (I manually counted that), and finally “player wins” (total of 7 lines displayed as in Figure 5-10, part 5 of 6 above), **modify** the code to show just one line “Player rolled 3 + 3 = 6, point is 6, player rolled **4** more times, player wins”.

In the case of “Player rolled 1 + 3 = 4”, “point is 4” followed by 8 lines of attempt (I manually counted that), and finally “player loses” (total of 11 lines displayed as in Figure 5-10, part 6 of 6 above), **modify** the code to show just one line “Player rolled 1 + 3 = 4, point is 4, player rolled **8** more times, player loses”.

Further **modify** / enhance your code of figure 5\_10.cpp as you already modify in the above 3 paragraphs so that you can run in a for loop or while loop until you exit. Run this for 10 “simulated game of craps” or 10 iterations (in Figure 5-10, part 5 of 6, we have 3 simulated games with 2 wins and 1 loss, in part 6 of 6, we have one simulated game that takes 8 throws ending up with 1 loss.). **Add** a variable to count the number of winnings and losses for running this 10 iterations.

1. (12%) Enhance / modify your figure 5\_10.cpp so that you no longer displays the line “Player rolled 6 + 6 = 12, Player loses” or the line “Player rolled 3 + 3 = 6, point is 6, player rolled **4** more times, player wins” (so there will NOT be too many outputs).

Keep the variable from Q2 that computes the total number of winnings and the total number of losses (which is equal to the total number of games minus the total number of winnings).

Add a new variable, total number of throws. The number of throws as we see from slide, figure 5-10, part 5 of 6 is 1 + 4 + 1 = 6 (for 2 + 5 = 7, one throw, for 3 + 3 = 6, 4 throws, and for 6 + 6 = 12, one more throw). The number of throws as we see from slide, figure 5-10, part 6 of 6 is 8 for 1 + 3 = 4 (point is 4). Hence the total number of throws with these 4 games is 1 + 4 + 1 + 8 = 14. We can then derive another variable: average number of throws = 14/ 4 = 3.5 throws (for each game of craps to determine if the player wins or loses).

*The way Q3 differs from Q2 is that in Q2 I ask you to display the result of each game in one line, while in Q3 I only need the summary (very similar to the questions on prime numbers, where in some question, I ask you to display all prime numbers from say 1 to 100, 8 in a rows, (that will display in 4 lines) while in some other question, I ask you to just compute the total number of prime numbers from 1 to 100, which is 25). Displaying all prime numbers from 1 to 100 is fine, but displaying all prime numbers from 1 to 1,000 or 1 to 1,000, 000 is awful or formidable; yet just computing the total number of primes even up to 1,000,000 (or bigger integer) is OK in general.*

1. Run your program for the case of 10 iterations or 10 games similar to Q2. Display the number of winnings (and losses), the total number of throws (and average number of throws)
2. Now run for 1,000 iterations.
3. Run for 1,000,000 iterations.

**Recursive and Iterative way of programming**

It is easier to write code recursively (f(n) is defined in terms of f(n-1)) (usually it is easier than if you write the code iteratively or using for loop). However, the time taken at run time can be awfully big.

In the class, I had shown ways of timing how long some C program runs such as the recursive way of computing Fibonacci numbers up to Fibonacci (50) as shown here (in the code ConsoleApplication4.cpp , which will be provided to you)

Time taken for fibonacci(40) is 4.00 seconds

fibonacci(42) = 267914296

Time taken for fibonacci(42) is 9.00 seconds

fibonacci(45) = 1134903170

Time taken for fibonacci(45) is 64.00 seconds

fibonacci(50) = 3996334433

Time taken for fibonacci(50) is 856.00 seconds

In ConsoleApplication5.cpp (that I will provide to you). I used array of 100 entries to compute Fibonacci numbers up to 100 (there is overflow); but the time taken is very small, say less than 1 second.

The code in ConsoleApplicatoion5.cpp is like below:

cout << "Input an integer \n"; // Note n has to be small like 20, 30

cin >> n;

long long fibo[100]; // declare Fibonacci number of 100 entries

fibo[0] = 0, fibo[1] = 1, fibo[2] = 1;

//compute Fibonacci numbers

for (int i = 2; i < n; i++)

fibo[i] = fibo[i - 1] + fibo[i - 2];

cout << "List of first " << n - 1 << " Fibonacci numbers \n";

1. **(22%) Modify** my ConsoleApplication5.cpp (which is Fibonacci number calculation using for loop and array) so that you do NOT need an array.

Rename my ConsoleApplication5.cpp as FibowithArray.cpp and call your code FiboWithoutArray.cpp.

**Hint:** this means you do NOT declare long long fibo [100];

Change fibo[1] = 1; and fibo[2] = 1 to a = 1; b = 1;

In the for loop

for (int i = 2; i < n; i++)

fibo[i] = fibo[i - 1] + fibo[i - 2];

replace

fibo[i] = fibo[i - 1] + fibo[i - 2];

by c = a + b;

then, you need b = c; followed by a = b; in the same iteration.

Think about this scheme:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 2 | 3 | 5 | 8 | 13 |
| a | b | c |  |  |  |  |
|  | a | b | c |  |  |  |
|  |  | a | b | c |  |  |
|  |  |  | a | b | c |  |

When n = 30, both FibowithArray.cpp and FibowithoutArray.cpp should print out the following

Input an integer

30

List of first 29 Fibonacci numbers

0 1

1 2

3 5

8 13

21 34

55 89

144 233

377 610

987 1597

2584 4181

6765 10946

17711 28657

46368 75025

121393 196418

317811 514229

**Run your program with n = 30 and n = 40** and show the output here.

1. (12%) Factorial computed recursively and iteratively.

Fig5\_28.cpp computes factorial recursively (up to 10), while fig 5\_31.cpp computes factorial iteratively (using for loop). Both of them computes only up to 10 (10! = 3628800).

**Modify** both fig05\_28.cpp and fig05\_31.cpp to use <time.h> (<ctime>) as shown in ConsoleApplication4.cpp.

time\_t start, end;

start = time(0);

cout << "\nfibonacci( 42 ) = " << fibonacci(42) << endl;

end = time(0);

// printf("Time taken to print sum is %.2f milliseconds",

printf("Time taken for fibonacci (42) is %.2f seconds",

difftime(end, start));

Now **display** the time it takes to compute

1. (6%) 15! Recursively (using recursion, like05\_28.cpp)
2. (6%) 15! Iteratively (using for loop as in fig05\_31.cpp)

If it seems too long to compute 15! (say longer than 10 minutes), then compute a smaller number like 14!, 13! Etc.

If it takes a short time to compute 15!, then try bigger number like 20!, 30! Etc. Do NOT worry about overflow now. The main concern is: how long it takes to run.