Advanced C Programming

Fall 2017 :: ECE 264 :: Purdue University

This is for Fall 2017 (2 years ago) only.

Card shuffling #1 (E)

Due 10/6

This exercise, together with HW08, is about card shuffling. You will be implementing an algorithm that generates all possible orderings of a desk of `n` cards after shuffling `k` times.

You may have noticed that shuffling only once does not produce all possible orderings of a deck of cards. For example, if the original deck has 4 cards, the number of all possible orderings is 24. However, shuffling a deck of cards once will give us only 14 orderings, not 24. Moreover, some of the 14 results are the same. Note that 24 is 4!, and 14 is 2^4 - 2. If we are dealing with 5 cards, there are 5! possible orderings, but shuffling once will give us only (2^5 - 2) orderings (with repetitions).

Algorithm

Step 1

For a deck of n cards, divide it into two decks (upper deck and lower deck). Each deck has at least one card. The orders inside each deck is not changed. For example, if the original deck is 2 3 4 5 6 7 8 (each number is a card, 2 is at the top and 8 is at the bottom).

- One possible result is upper: 2 3 lower: 4 5 6 7 8
- Another possible result is upper: 2 3 4 lower: 5 6 7 8
- Yet another possible result is upper: 2 3 4 5 lower: 6 7 8

Step 2

Interleave the two decks. Please be careful that the orders in the upper deck and the lower deck are unchanged. For example, if the upper deck is 2 3 4, in the newly shuffled deck, 2 is still above 3 and 3 is still above 4. Similar, if the lower deck is 5 6 7 8, in the newly shuffled deck, 5 is still above 6 and 6 is still above 7. The order of the cards in the two decks may interleave. The following are some possible results:

A. 2 is between 5 and 6; 3 is between 6 and 7
B. 2 3 are between 5 and 6
C. 2 3 4 are all between 5 and 6
D. 2 3 4 are all above 5 (thus, the new deck is the same as the original deck)

There are other possible results.

Step 3 – combining the first two steps

Suppose the original deck is 234. The are two ways to divide it into an upper deck and a lower deck:

A. upper: 2 lower: 34 or
B. upper: 23 lower: 4

Interleaving them can produce these results:

A. upper: 2 lower: 34
   234 (2 above 3 and 4),
   324 (2 between 3 and 4),
   342 (2 below 3 and 4),
   or
B. upper: 23 lower: 4
   423 (4 above 2 and 3),
   243 (4 between 2 and 3),
   234 (4 below 2 and 3)

Please notice that 432 cannot appear.

Getting started

Run `264get hw07` to get the code. You will be implementing the functions described under *Requirements* below.

Output format

You will use `print_deck(…)` (declared in utility.h) to produce the output. The format is illustrated below. Note that these results are possible orderings of the deck "[6789A23]". Please read the outputs from **top to bottom, left to right**.

```
[6789A23]      [A672893]      [6A27839]
[678A923]      [6A27893]      [A627839]
[67A8923]      [A627893]      [A267839]
[6A78923]      [A267893]      [67A2389]
[A678923]      [678A239]      [6A72389]
[678A293]      [67A8239]      [A672389]
[67A8293]      [6A78239]      [6A27389]
[6A78293]      [A678239]      [A627389]
[A678293]      [67A2839]      [A267389]
[67A2893]      [6A72839]      [6A23789]
[6A72893]      [A672839]      [A623789]
                              [A263789]
                              [A236789]
```

How many possible orderings can the new deck have?

If there are k cards in the original deck, there are k - 1 different ways to divide the original deck to two decks:

1. the upper deck has 1 card, the lower deck has k - 1 cards
2. the upper deck has 2 cards, the lower deck has k - 2 cards
3. the upper deck has 3 cards, the lower deck has k - 3 cards

4. the upper deck has 4 cards, the lower deck has k - 4 cards
   ...
   k-1. the upper deck has k - 1 cards, the lower deck has 1 card

If the upper deck has n cards and the lower deck has m cards, there are $\frac{(n+m)!}{n!m!}$ ways to interleave the two decks, while keeping the orders in the original upper deck and the original lower deck.

If the original deck has k cards (i.e., n + m is k), there are totally $\frac{k!}{1!(k-1)!} + \frac{k!}{2!(k-2)!} + \ldots + \frac{k!}{(k-1)!1!}$ results, as the upper deck may have 1, 2, 3, ..., k - 1 cards. (Remember that both n and m must be one or larger.) This happens to be parts of the binomial expansion:
$(x+y)^k = \sum_{n=0}^{k} \frac{k!}{n!(k-n)!} x^n y^{k-n}$
If the first and the last terms are removed, it becomes $\frac{k!}{1!(k-1)!} + \frac{k!}{2!(k-2)!} + \ldots + \frac{k!}{(k-1)!1!}$

Please be aware that if a new deck appears multiple times, it is counted multiple time. In the previous example, 234 is counted twice.

For a deck of k distinct cards, there are k! possible orders (i.e., permutations).

k	2	3	4	5	6
$k!$	2	6	24	120	720
$2^k - 2$	2	6	14	30	62

As you can see, $2^k - 2$ is equal to $k!$ when k is 2 or 3 and smaller than $k!$ when k is 4 or larger. Thus, shuffling once cannot generate all possible orders, except for only k is 2. Actually, $2^k - 2$ counts the same output multiple times; in the previous example, 2 3 4 is counted twice and 4 3 2 is not generated. For simplicity, ECE 264 does not ask you to determine which orders are missing.

`divide(…)`

This program find the possible outcomes of dividing a deck of cards into a pair of upper and lower decks. The division is done such that if the upper deck is placed above the lower deck, you get back the original deck. After dividing the cards into the upper and the lower decks, they are interleaved into a single deck (using the `interleave(…)`). This is called one round of shuffling.

You have to implement a function to find and store all possible pairings of non-empty upper and lower decks in two arrays.

The `divide(…)` function has three arguments: `orig_deck` is the original deck of cards `upper_deck` and `lower_deck` are two arrays with sufficient amounts of memory upper_deck[i] and lower_deck[i] form a pair. The caller of `divide` (`shuffle(…)`) should allocate enough memory for `upper_deck` and `lower_deck` before calling `divide(…)`.

`shuffle(…)`

You also have to implement a function to perform the shuffling. In this shuffling function, you have to call the division function to get all possible pairings. Before you call the division function, you have to allocate sufficient space such that all possible pairings can be stored. For each pair of upper and lower decks, you have to perform interleaving. You have to deallocate or free the space storing the pairings before you exit from the function.

You are allowed to have additional functions in the file. The output for up to five cards are provided for reference. Please understand that when your submission is graded, the input deck may not include cards 'A', '2', ... Your program must use what is given to the function, without assuming what cards are in the original deck.

Requirements

1. Your submission must contain each of the following files, as specified:

file	contents	
shuffle.c	functions	**interleave**(CardDeck upper_deck, CardDeck lower_deck) → *return type:* void Print all possible interleavings of `upper_deck` with `lower_deck`. • Use `print_deck(…)` to print each resulting order. • Do not make assumptions about the order or composition of the original deck. For example, A may come before or after 2, or may not be present at all. • `print_deck(…)` and `MAX_SIZE` are declared in utility.h. • Tip: There should be no uncertainty in this function. If you think a random number generator is needed, you are on the wrong track. • Tip: To *copy* the elements of one array from one array to another (e.g., the array of cards in a `CardDeck`), you could use `memcpy(…)`. The = operator will simply copy the address, not the elements themselves. • ⚠ You must use `MAX_SIZE` for the maximum length of a deck. Do not use 13. • ⚠ Any memory allocated with `malloc(…)` must be freed. There are penalties for memory leaks.
		divide(CardDeck orig_deck, CardDeck ✶ upper_decks, CardDeck ✶ lower_decks) → *return type:* void Divide a deck into into pairs of upper and lower decks. • `upper_decks` and `lower_decks` are arrays of `CardDeck`. (i.e., Each element is a `CardDeck`.) • Assume `upper_decks` and `lower_decks` are allocated before `divide(…)` is called. • If the original deck has n cards, there should be n - 1 pairs of upper and low decks: • upper deck has 1 card, low deck has n - 1 cards • upper deck has 2 cards, low deck has n - 2 cards • upper deck has 3 cards, low deck has n - 3 cards ... n-1. upper deck has n - 1 cards, low deck has 1 cards • `upper_deck[i]` should contain the first `i+1` cards from `orig_deck`. For example: • `upper_deck[0]` should store the first card in the original deck. • `upper_deck[1]` should store the top two cards in the original deck. • `upper_deck[2]` should store the top three cards in the original deck. • Tip: Remember that within `CardDeck`, `cards` should be an array () • ⚠ Do not call `malloc(…)` in this function.

file	contents	
		`shuffle``(CardDeck orig_deck)` → *return type:* void Generate all possible decks that could result from *one* shuffle. • `orig_deck` contains the number of cards. • The number of upper-low deck pairs should be the number of cards - 1. • Follow these steps: ◦ Calculate the number of upper-low deck pairs. ◦ Allocate enough memory to accommodate the pairs. ◦ Call `divide(…)` to find these pairs. ◦ For each pair of upper-lower deck, call `interleave(…)` to interleave the cards. ◦ Release the allocated memory. • The amount of memory allocated in this function must be determined based on the size of the original deck. The program may fail if the allocated memory has a fixed size (some students like to use 1000 but nobody could explain why 1000). • ⚠ A penalty of 50% of the possible points will be deducted if the amount of allocated memory is not determined by the size of the original deck, e.g., if you use 100, 1000, 10000, or any other fixed number.

2. Your program must output the same lines as any other correct solution. However, the *order* may be different.
3. Do not print anything other than the required output (i.e., no debugging output, etc.).
4. Submissions must meet the code quality standards and the course policies on homework and academic integrity.

Submit

To submit HW07, type `264submit HW07 shuffle.c` from inside your hw07 directory.

In general, to submit any assignment for this course, you will use the following command:

```
264submit ASSIGNMENT FILES…
```

Submit often and early, even well before you are finished. Doing so creates a backup that you can retrieve in case of a problem (e.g., accidentally deleted your files).

To retrieve your most recent submission, type `264get --restore ASSIGNMENT` (e.g., `264get --restore hw07`).

To retrieve an earlier submission, first type `264get --list ASSIGNMENT` to view your past submissions and find the timestamp of the one you want to retrieve. Then, type `264get --restore -t TIMESTAMP ASSIGNMENT`.

Scores will be posted to the Scores page after the deadline for each assignment.

Pre-tester ●

The pre-tester for HW07 has been released and is ready to use.

Using the pretester

The pretester is a tool for checking your work after you believe you are done, and before we have scored it. It is not a substitute for your own checking, but it may help you avoid big surprises by letting you know if your checking was not adequate. To use the pre-tester, first submit your code. Then, type the following command. (*Do this only after you have submitted, and only after you believe your submission is perfect.*)

```
264test hw07
```

Do not ask TAs or instructors which tests you failed.

Keep in mind:

- Pre-testing is intended only for those who believe they are done and believe their submission is perfect.
- The pre-tester is not part of the requirements of this or any other assignment.
- You are responsible for reading the assignment carefully, and ensuring that your code meets all requirements.
- Feedback is limited, to ensure that everyone learns to test their own code.
- If your code is failing some tests, review *your tests* and make sure they are comprehensive enough to catch any bugs (deviations from requirements). Follow the tips given by the pre-tester.
- Code quality issues are not reported by the pre-tester; writing clean code is something you must learn to do from the start, not a clean-up step to do at the end.

Logistics:

- If we discover that we have not checked some significant part of the assignment requirements, we may add additional tests at any time up to the point when scores are released.
- The pre-tester will only be enabled after much of the class has submitted the assignment, and at least a few people have submitted perfect submissions. This is to allow us to test the pre-tester.
- The pre-tester checks your most recent submission. You must submit first.
- You may be limited to running the pre-tester ≤24 times in a 24-hour period. (This is not implemented yet but will be added.)

Q&A

Answers to common questions may be posted here later.

Updates

9/25/2017 Draft.