ECE 15, Fall 2023

Final

Sequence number

Last name

First + middle name(s)

PID

Grade

/ 80

Instructions:

- Do not look at the questions or start writing until it is announced you can do so.
- Make sure you write your PID on EACH page.

- We will only look at the work that is within the dotted box on each page. You can use pages that were part of earlier questions if there is still space. There are also extra pages at the end that you can use as overflow for any of the questions, and you can ask for additional pages as well. Always clearly mark where we can find the work for each question.

- Keep these question pages stapled together.
- Use your own paper for scratch work (and you do not need to turn that in).
- Read each problem completely and thoroughly before beginning.
-
- You must follow the Final Exam Procedures that were posted on Canvas. If you are unsure of anything, ask. As a reminder:
    - Your phone should be turned off and put inside your bag
    - You cannot use any electronic devices.
    - This exam is open book, open notes (provided they are on paper).
    - Follow the Academic Integrity standards.

▼

Question 1 (28 pts)

Complete the questions listed in the box to the right of the program. Important information:

- The programs should not contain any errors (run-time or compilation) unless it is specifically mentioned to check for possible errors.
- Assume this is a **64-bit system**, with **integers 4 bytes** and **doubles 8 bytes**.
- Use **?** if you want to denote a **garbage/unknown value**.

Part a

```
#include <stdio.h>

char *process(char x[]) {
   printf("%d \n", sizeof(x));
   return x+2;
}

int main() {
   char a[5] = "cat";
   char *p = a;

   p = process(a);
   printf("%c \n", p[-2]);

   p = a+1;
   printf("%c \n", p[0]);
   printf("%c \n", ++(*a));
}
```

What gets printed?

Part b

```
#include <stdio.h>

int main() {
   int a = 2.5;
   printf("%d -> %c \n", a, a + 'a');

   a = 9;
   double b = a / (char)2.5;
   printf("%1.2f \n", b);

   if ('=')
      printf("dog \n");
   else
      printf("cat \n");

   a = (8 > 0 && 1 < 2)?(1==0):(0==0);
   printf("%d \n", a);
}
```

What gets printed?

Part c

```
#include <stdio.h>
#define VALUE 4

int dec(int x, int y) {
   int value = 1;
   x = VALUE;
   printf(" %d \n", x);
   return 2+y;
   return 3+y;
}

double inc(double *x, double *y) {
   *x += 1;
   *y -= 2;
   return *x+1;
}

int main() {
   int x = 2, y = 6;
   y = dec(x,y);
   printf(" %d   %d\n", x, y);

   double a = 1, b = 8;
   b = inc(&a, &b);
   printf(" %1.2f   %1.2f\n", a, b);
}
```

What gets printed?

Part d

```
#include <stdio.h>
#include <stdlib.h>

void process(int *x) {
   printf("%d \n", x[0] + 2*x[1] );
   x += 1;
   x[0] += 5;
}

int main() {
   int a[3] = {0,0};
   int *p = (int*)calloc(3,sizeof(int));

   printf("%d \n", sizeof(a) );
   printf("%d \n", sizeof(p) );

   process(p);
   printf("%d \n", p[0] + 2*p[1] );
   free(p);
}
```

What gets printed?

Part e

```c
#include <stdio.h>

typedef enum {mo=4,tu,we,th=9,fi} days;

int main() {
   int a[2][3] = {3, 5, 7, 9, 11, 13};
   days b = we;

   printf("%d \n", a[1][1]);
   printf("%d %d \n", b, b+1);
}
```

What gets printed?

Part f

```c
#include <stdio.h>

int process(int a) {
   a = 1;
   static int c = 3;
   c++;
   return c;
}

int main() {
   int x = 1;
   while (x < 10) {
      x += 3;
      if (x%2 == 0)
            continue;
      printf("%d ", x);
   }
   printf("\n");

   int a = 4, b = 3;
   if (b > 0) {
      int a;
      b = process(3);
      printf("%d %d \n", a, b);
      a = 7;
      b = process(a);
      printf("%d %d \n", a, b);
   }
   printf("%d \n", a);
}
```

What gets printed?

Important Notes for Subsequent Questions

These notes apply to all questions that follow.

1. You are **not** allowed to use **global variables** or functions from **any library that is not already included** (such as a string library).
2. If you need **arrays** of variable length, you need to use dynamic memory allocation.
3. You are allowed and encouraged to use **helper functions**.
4. You will lose points if your code is **overly complicated** or shows a gap in understanding of the core material.
5. Make sure you read the instructions thoroughly. However, the goal of the questions is to test you on the core principles. We are not looking at tricky edge cases.

Question 2 (6 pts)

You must implement the function **`process`**() and call it in the code below. Referring to the code below, the function should process `a` and `b` as follows:

- It should modify `a` so that it is incremented by 1.
- It should return the negative of `b`, but `b` itself should not be modified. The return value should be placed in the variable `result`.

The values in the sample code are given as illustrations. The black box shows the correct outputs for this example. Your code should be generic such that it works if we use different values for `a` and `b`. The goal of this question is for you to show us that you know the basics of passing data to/from functions. It is meant to be short.

```
#include <stdio.h>

int main() {
    int a = 9, b = 5, result;

    // Call your function process() (single line of code)


    printf("%d   %d    %d ", a, b, result);
}
```

```
10    5     -5
```

Your task:

1. Implement the function **`process`**() on the next page.
2. Complete the missing single line in the code above, where the function is called. You cannot make any other modifications to `main()`.

Question 3 (15 pts)

One of the main goals of this question is to further test your ability to pass data to/from functions, which means that you are expected to figure out the parameters and return types.

You must implement four functions. Referring to the code on the next page:

two_digits(): This function should take in a positive integer and return whether this number consists of two digits (i.e., is between 10 and 99). The return value must be such that it can be used in logical expressions (e.g., if statements) as shown in the code on the next page. Specifically, in the example code, the if-statement should evaluate as true if `num` has two digits.

swap(): This function should take in a two-digit positive integer and swap the two digits (e.g., the number 37 should become 73). This function must have a void return type. You may assume it is only called on numbers that have two digits. In the code on the next page, it should swap the two digits of `num`.

swapif(): This function should take in a positive integer and, if the number has two digits, swap the two digits (and otherwise leave the number unchanged). This function must have a void return type.

This function **must** use the functions `two_digits`() and `swap`() that you already implemented. Essentially, this function should implement the same functionality that you already find in the code example on the next page (i.e., what is in the rectangle), but now as a separate function.

process(): This function should take in an array of positive integers (in the code on the next page, the array `vals`) and modify this array. Specifically, for each element in the array that is a two-digit number the two digits must be swapped (and otherwise it should be left unchanged). It must use the `swapif`() function that you already implemented (see above). You need to decide on the parameters (could be one or more) and return type of this function. [Hint: you may assume variable `count` is calculated as in the example code on the next page].

The values in the sample code on the next page are given as illustrations, with the black box showing the correct outputs for this example. Your code should be generic such that it works if we change the content of the array `vals` or the number of elements it contains (although it will never be an empty array), as well as the value of `num`. You may assume that we will only use positive integers (so you do not need to consider what happens with negative numbers).

Your task:

1. Implement the functions **two_digits**(), **swap**(), **swapif**() and **process**() on the next page.
2. Complete the missing lines in the code on the next pages. You cannot make any other modifications to `main()`.

```c
#include <stdio.h>

int main() {
   int vals[] = {6, 92, 123, 34};
   int num = 78;
   int count = sizeof(vals)/sizeof(vals[0]);

   if ( two_digits(num) )
        // Call your function swap() (single line of code)


   // Call your function process() on vals (single line of code)


   printf("num: %d \n", num);
   printf("vals: %d %d %d %d", vals[0], vals[1], vals[2], vals[3]);
}
```

```
num: 87
vals: 6  29  123  43
```

Question 4 (12 pts)

You must implement two functions: **count**() and **alpha**(). Referring to the code below:

count(): This function should take in `symbol` and the string `text`, and return how often the symbol appears in the string. The string should not be modified. When calling this function, the result should be put in the variable `num`.

alpha(): This function should take in the string `text` and produce an output string `letters`. This output string `letters` should consist of subsequent lower case letters of the alphabet (starting with `a` , followed by `b`, etc. until `z`, and then going back to `a`, etc.) and have the same length as the input string `text`. As an example, look at the code below: the input string `text` consists of 36 characters, and the output string `letters` is printed as shown (also 36 characters). The function should not modify the input string `text`.

You must also declare the variable `letters` in `main`() (see the first box) and then pass it appropriately to the function `alpha`(). The values in the sample code are given as illustrations. The black box shows the correct outputs for this example.

```
#include <stdio.h>

int main() {
   char text[] = "It is a very pretty cat! Yes, it is.";
   char symbol = 'e';
   int num;

   // Declare the variable letters; this variable will store the
   // output string of the function alpha() (single line of code)


   // Call your function count() (single line of code)


    // Call your function alpha() (single line of code)


   printf("The letters string: %s \n", letters);
   printf("The symbol occurs %d times", num);
}
```

```
The letters string: abcdefghijklmnopqrstuvwxyzabcdefghij
The symbol occurs 3 times
```

Your code should be generic such that it works if we change the content or length of the string `text` or the character stored in `symbol`. You may assume that `text` is never an empty string. Also, the string initialization will also never contain more than **100** symbols (i.e., there will never be more than 100 symbols between " " in the initialization of `text`). The string can contain letters, spaces, punctuation marks, other symbols, etc.

Try not to make use of the ASCII table (i.e., knowledge of which letter maps to which number); if you do use the table, you will lose 1 point.

Your task:

1. Implement the functions **`count`**() and **`alpha`**() below and on the next page.
2. Complete the missing lines in the code on the previous page. <u>You cannot make any other modifications to `main()`</u>.

Question 5 (6 pts)

In this problem, you are implementing a random roll of the dice. Referring to the code below, the struct Dice implements dice where the field `value` indicates the dice roll, i.e., a number between 1 and 6 (we assume 6-sided dice). The output that is shown is just an example, as the output is in fact random.

You need to implement the function `roll()`. It must set the `value` field of `d` to a number between 1 and 6 chosen in a uniformly random way. Basically, it must simulate a random dice roll. The function must have a void return type. The parameters of this function are for you to decide.

[Hint: the code to seed a random number generator is `srand(time(NULL))`]

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct {
   int value;              // The result of the dice roll
} Dice;

int main() {
    int i;
    Dice d;

    // Any other line(s) of code you may need (optional)


    for (i = 0; i < 10; i++ ) {
        // Any other line(s) of code you may need (optional)


        // Call your function roll() (single line of code)


        printf("%d ", d.value);
    }
}
```

```
2 5 1 2 4 6 1 3 6 4
```

Your task:

(a) Implement the function **`roll`**() below.

(b) Complete the missing lines in the code on the previous page. These should

a. Call the function `roll()`

b. Include any other lines of code you may need to achieve random behavior (i.e., each time you execute the program, you get a different random set of dice roll values). Note that it is possible that you do not need code in every one of these boxes or in any of them.

Question 6 (13 pts)

For this karel problem, **you are not allowed to use variables**. You will receive 0 credit if you use variables. All your functions need to have a void return type and not have any parameters. The repeat() functionality is available. This is similar to what you did in PA1 and PA1b.

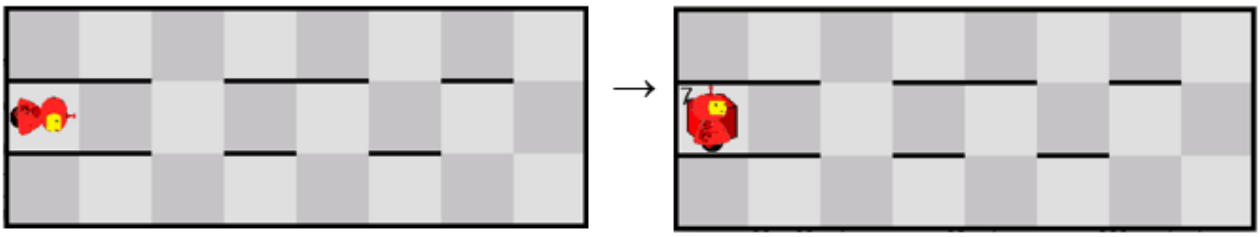
You must use appropriate helper functions and top-down modular design. Your code needs to be well-structured and easy to understand; this will account for a significant portion of the credit that can be earned on this question.

Funtionality:

Karel needs to count how many "doors" there are in his row. For each tile, a door is present if there is an opening (i.e., no wall) to either the north or south or both (in which case there are two doors).

He starts on the west-most tile of the row, facing in a random direction and with an infinite number of items in his bag. At the end, he needs to be back on his starting tile, but his orientation does not matter.

The goal is for him to place as many items on his starting tile as there are doors in his row. The example below illustrates this task. As with the PAs, your code must be flexible to work with different lengths of the row, different positions of the doors, etc. You may assume there is never a door in his starting column (i.e., on his starting tile); however, there can be doors in any other column.



Start state

End state

Your task: Write this karel program on the next page. For your convenience, the code skeleton for a karel program is shown below. You may assume the correct settings and map files are available. You do not need to copy the `karel_setup()` line in your `main`().

You do not need to write your functions in order, so it is fine to define a function after it is first called. This is to make structuring your answer more convenient.

```
#include <karel.h>
// Helper functions would go here
int main() {
   karel_setup("settings/settings.json");
   // Your main code would go here
}
```

PID

---- END ----