CS 382 Fall 2024 — Computer Architecture and Organization — Homework 1

In this homework, we're going to implement control structures using `goto` statements. Even though `goto` is not something you should use normally in C programming, we use it here as a tool to get familiar with un-structured programs, which will benefit our upcoming assembly language learning.

This is an individual assignment.

1 Task 1 (20 pts): Copy a String

In this task, you will write a C code to implement a function called `copy_str()` that copies all the characters in the `src` string to `dst` string. The prototype of the function is declared as follows:

```c
void copy_str(char* src, char* dst);
```

You can assume `dst` has enough space to store all the characters from `src`. Note, you **cannot use** any type of structured loops, such as `for`, `while`, and `do...while`, meaning the only option you can use is `goto` statement. Of course you can use `if-else` structure when needed; however, no statement is allowed in `if` or `else if` or `else` blocks other than a single `goto`.

Requirements

- You must not use structured loops mentioned above;
- You must not include any other headers;
- You must not use any functions provided in `string.h`, especially not `strcpy()` and `strlen()`. Note that your code does not need to calculate string length.

2 Task 2 (30 pts): Calculate Dot Product

In this task, you need to write a C function to calculate the dot product between two vectors. The prototype of the function is declared as follows:

```c
int dot_prod(char* vec_a, char* vec_b, int length, int size_elem);
```

where both `vec_a` and `vec_b` are integer vectors that contain `length` number of integers, and the function will return a single integer as the dot product. `size_elem` is the number of bytes of each element in the vectors.

Note that even though both vectors contain integer values, we do not pass `int*`. In your implementation, you must not cast the entire vector back to `int*`. However, casting one element to `int*` is allowed. As in the previous task, you are not allowed to use loops, so you'd have to use `goto` statements.

Similarly, no statement is allowed in `if` or `else if` or `else` blocks other than a single `goto`.

Requirements

- You must not use structured loops mentioned above;
- You must not cast the entire array into another type; only casting one address to `int*` at a time is allowed.

3 Task 3 (50 pts): Sorting Nibbles

In this task, you'll write a C function to sort all the nibbles (4 bits) in an integer array. The prototype of the function is declared as follows:

```
1 void sort_nib(int* arr, int length);
```

where `arr` is the integer array, while `length` is the number of integers in that array.

For example, say we have an integer array: `int arr[3] = {0x12BFDA09, 0x9089CDBA, 0x56788910}`. One nibble has 4 bits, so each hexadecimal digit represents a nibble. If we treat them as individual numbers and sort them from smallest to largest and print them out as integers, we have:
`0x00011256 0x78889999 0xAABBCDDF`.

You can use any sorting algorithm you like, but do not use functions provided by existing libraries such as `qsort()`.

Hints

- Take a deep breath before you start 😉!
- The first step you want to do in your code is to separate all the nibbles in that array. Even though each nibble takes only four bits, you can still store one nibble in a `char` variable. If the array is like this: `int arr[2] = {0x1234, 0x4321}` you can use bit-wise operations and shifting to create an array like this: `char nibs[16] = {0,0,0,0,1,2,3,4,0,0,0,0,4,3,2,1}`. Note how many `0` are there in `nibs`: each integer takes four bytes which is **eight** nibbles, so you need to make sure leading zeros are also considered in the array;
- After sorting array `nibs`, you just need to re-group nibbles back into integers, and replace them back to `arr`. Note the function performs an **in-place** sorting;
- **10 bonus points** if you use `goto` in your code instead of structured loops. [1]

Requirements

- You must not use existing libraries to sort;
- Write down the sorting algorithm you chose in the comments;
- In the comments state if you'd like to be graded for bonus points. Without the statement no bonus points will be given.

4 Starter Code & Tester

You are also provided with a starter code where you can see how to call and test the functions. However, note that passing the demo test in the starter code does **not** mean your program is entirely correct. You must come up with your own tests, especially edge cases.

[1] Note either you use loops **or** `goto`. There's no partial bonus points for using mixed `goto` and loops.

To help you with testing, we also provided a tester file `tester_m1` (or `tester_x86`, depending on your machine type). Put this tester file in the same directory as your C code, and use the following command to generate an executable:

```
$ gcc main.c tester_m1
```

assuming the C file you wrote is called `main.c`. If there's no errors reported, you can go ahead and run the tester:

```
$ ./a.out -h
```

where `-h` is to show you how to use the tester. To test your code, use the following command:

```
$ ./a.out -t {copystr|dotprod|sortnibs|all}
```

where the name indicates the task you want to test, or `all` to test all three tasks.

5 General Requirements

In addition to each task's requirements listed above, here are additional requirements applied to all tasks:

- Your code must be able to compile successfully and execute without segmentation fault or any other type errors;
- Write your name and honor code pledge at the top of your code as comments;
- You must not change any function prototypes;
- You must not include any other header files other than `stdio.h`;
- You must not use any other library functions; it's ok to create your own helper functions, though;
- Comment your code well – describe what your code does. Meaningless comments and/or comment-less code will be penalized;

6 Grading

The homework will be graded based on a total of 100 points.

- Task 1 (20 pts): 5 test cases in total, **4** points each;
- Task 2 (30 pts): 10 test cases in total, **3** points each;
- Task 3 (50 pts): 10 test cases in total, **5** points each.

After accumulating points from the testing above, we will inspect your code and apply deductibles listed below. The lowest score is 0, so no negative scores:

- Task 1 (20 pts):
  - **-20:** used structured loops or `switch`;
  - **-20:** `if-else` blocks contain non-`goto` statements;
  - **-20:** used any functions from `string.h`;
  - **-20:** function prototype was changed;
- Task 2 (30 pts):

- **-30:** function prototype was changed;
- **-20:** used structured loops or `switch`;
- **-20:** `if-else` blocks contain non-`goto` statements;
- **-10:** didn't use parameter `size_elem` (*e.g.,* hardcoded integer size);

► Task 3 (50 pts):

- **-50:** changed function prototype;
- **-40:** used any sorting functions instead of implementing;
- **-5:** didn't state the sorting algorithm in comments;
- **+10:** didn't use structured loop, and stated in the comments.

► General (only deduct once):

- **-100:** the code does not compile, or executes with run-time error;
- **-10:** no pledge and/or name in C file.

2% of extra credit will be given if the homework is submitted two days ahead of deadline. If it's re-uploaded after the earlybird date, the deal is off.

Deliverable

Submit a single `.c` file on Canvas.