ZJU-UIUC Institute
Final Exam, ECE 220

Tuesday 29 December 2020

Name (pinyin and Hanzi): **SOLUTION**
Student ID: Lab TA Name:

- **Be sure that your exam booklet has 13 pages.**
- **Write your name, Student ID, and lab section TA name on the first page.**
- **Some of C's I/O routines and an LC-3 ISA guide are provided. Unlike the first midterm, Patt and Patel's Appendix A will not be available during the exam.**
- **Do not tear the exam apart other than to remove the last two reference pages.**
- **This is a closed book exam. You may <u>not</u> use a calculator.**
- **You are allowed THREE A4 sheets of notes (both sides).**
- **YOU MAY NOT USE EXTRA PAPER! WRITE ON THE EXAM!**
- **Absolutely no interaction between students is allowed.**
- **Show all work, and clearly indicate any assumptions that you make.**
- **Challenge problems are marked with ***.**
- **Don't panic, and good luck!**

Problem 1	21 points	_____
Problem 2	16 points	_____
Problem 3	25 points	_____
Problem 4	20 points	_____
Problem 5	18 points	_____
Total	100 points	_____

Problem 1 (21 points): Short Answer Questions

1. **(5 points)** A bad TA compiles the code below for LC-3, then types in some Special Input™ for the `scanf`. In response, the program prints out "weird" instead of "main", then terminates. Based on your knowledge of the LC-3 calling convention, and **USING 20 WORDS OR FEWER**, explain what happened.

   Special Input™ overwrote return address on stack with address of `weird`

```
#include <stdio.h>
int weird () {
  printf ("weird");
  return 0;
}
int run () {
  char buffer[10];
  scanf ("%s", buffer);
  return 0;
}
int main() {
    run ();
    printf ("main");
    return 0;
}
```

2. **(6 points)** Consider the C++ declarations shown below.

```
class Base {
                int A;
protected:      int B;
private:        int C;
public:         int D;
};

class Derived: public Base {
private:        int E;
                static void aFunction (void);
public:         int F;
};

Derived instance;

void anotherFunction (void);
```

   1. **(3 points)** CIRCLE ALL FIELDS of `instance` that are accessible by name within the function `Derived::aFunction`.

      A (B) C (D) (E) (F)

   2. **(3 points)** CIRCLE ALL FIELDS of `instance` that are accessible by name within the function `anotherFunction`.

      A B C (D) E (F)

Problem 1, continued:

3. **(5 points)** The Linux man page gives the following function signature for the C library's implementation of quicksort.

```
void qsort (void* base, size_t nmemb, size_t size,
            int (*compar) (const void*, const void*));
```

Note the callback argument **`compar`** used to compare two elements of the array **`base`**. This function must compare two elements of the array and return -1 if the first element should appear before the second, 0 if the two elements are the same, and 1 if the second element should appear before the first.

Your friend has implemented a sophisticated ranking algorithm for Blocky (MP6) players based on the use of a deep neural network (DNN), and has provided the function

```
int32_t player_get_rank (player_t* p);
```

that executes the DNN to calculate a player's rank. **The function takes about five seconds to execute.** To sort the players in decreasing order or rank, your friend has implemented the function below for use with quicksort:

```
int player_sort_by_rank (const void* p1, const void* p2)
{
    int32_t r1 = player_get_rank (p1);
    int32_t r2 = player_get_rank (p2);

    if (r1 > r2) { return -1; }
    if (r2 > r1) { return  1; }
    return 0;
}
```

Unfortunately, **`qsort`** seems to take quite a long time when executed with this function on an array of 1,000 players. **USING 20 OR FEWER WORDS**, suggest a way in which your friend can improve the performance by about a factor of 10.

Calculate rank once for each player and store in a new field of `player_t`

Problem 1, continued:

4. **(5 points)** Consider the following C++ code:

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

class ALPHA {
    private:
        int val;
    public:
        ALPHA (int start) : val (start) { }
        void add (int amt) { val += amt; }
        void add (double amt) { add (ceil (amt)); }
        int value (void) { return val; }
};

int
main ()
{
    ALPHA a (40);

    a.add (1.5);

    printf ("%d\n", a.value ());

    return 0;
}
```

Your friend wrote the code above, compiled it, and executed it. Unfortunately, rather than printing 42 as your friend expects, the program crashes. **USING 15 WORDS OR FEWER**, explain why.

infinite recursion to `ALPHA::add` with `double` argument

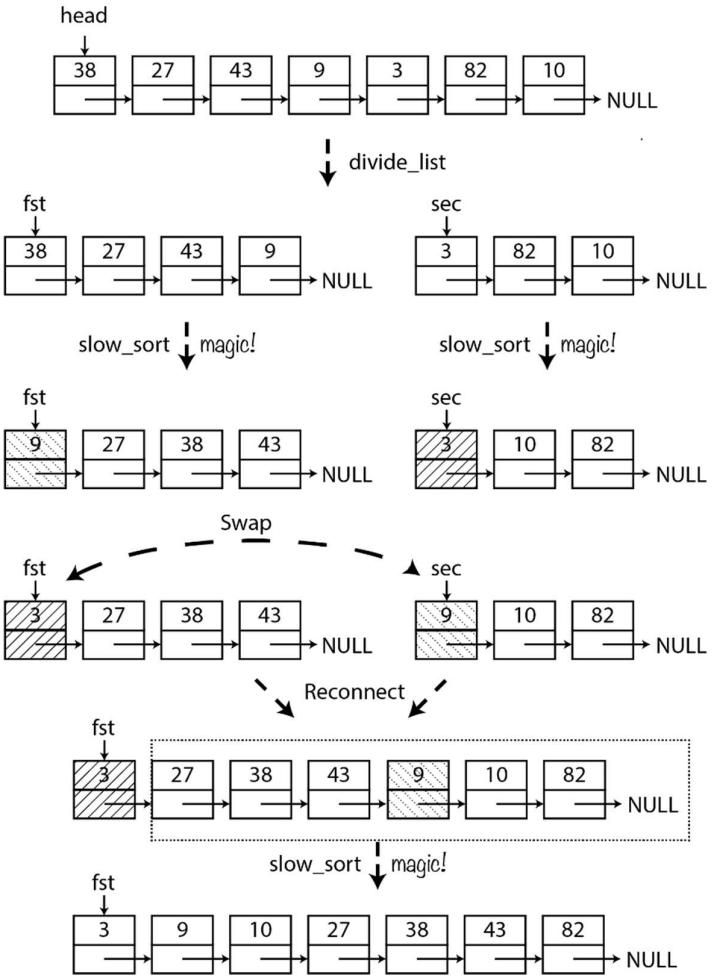
Problem 2 (16 points): Slow Sort

Quick Sort is quick, but difficult to understand. Instead, you must implement the "Slow Sort[1]" algorithm. As you know, writing a recursive function requires a "leap of faith," which means that you believe that your recursive calls work even before you have finished implementing the function. Slow Sort relies on this idea.

`slowsort(A, i, j)`: I am asked to sort $A[i \ldots j]$ from small to large. Here is my strategy:

- If $i \geq j$, nothing needs to be done. I will just go home and sleep.
- Otherwise, split the list by half: $A[i \ldots m]$ and $A[m+1 \ldots j]$, where $m = (i + j) / 2$.
- I call `slowsort` to sort the first and the second half for me. I believe it works.
- Both halves are sorted now. Let me compare the first one of each half, $A[i]$ and $A[m+1]$, and swap them if necessary. Now $A[i]$ must be the smallest one in the whole list!
- I have sorted one element. I feel tired now.
- How about the rest $A[i+1 \ldots j]$? Humm… Let me just call `slowsort` to sort them!
- Look, the list is sorted!

Based on the description above, complete the following code that performs Slow Sort on values stored in a singly linked list (the same input as the Merge Sort problem in the last midterm.)



[1] Andrei Broder and Jorge Stolfi. "Pessimal Algorithms and Simplexity Analysis," 1986.

Problem 2, continued:

The linked list is constructed using the following structure:

```
typedef struct element_t element_t;
struct element_t {
    int32_t    value;
    // other fields not relevant to this problem
    element_t* next;
};
```

Complete the implementation below using the following helper function and **using only the lines provided**:

```
// Divide a linked list starting at head into two equal halves (from MT2).
void divide_list (element_t* head, element_t** firstp, element_t** secondp);

// Swap *a and *b (simply swaps the two element_t*'s; does NOTHING else).
void swap (element_t** a, element_t** b);

element_t* slow_sort (element_t* head) {
    element_t* fst; element_t* sec; element_t* last;

    // If empty list or only one element, done!
    if (NULL == head || NULL == head->next) {
        return head;
    }
    // Otherwise, divide the list into two sublists of equal length.
    divide_list (head, &fst, &sec);

    // Sort each half.

    fst = slow_sort (fst);

    sec = slow_sort (sec);

    // If fst is larger than sec, swap them (you MUST use the swap function).
    if (fst->value > sec->value) {

        swap (&fst->next, &sec->next); // as shown; not needed for correctness

        swap (&fst, &sec);
    }
    // Reconnect fst and sec into a single list.

    for (last = fst; NULL != last->next; last = last->next) { }

    last->next = sec;

    _____

    _____

    _____

    // Sort the rest of the list.

    fst->next = slow_sort (fst->next);
    // Return the sorted list.
    return fst;
}
```

Problem 3 (25 points): Processing a File with I/O

Your task is to write a multi-function calculator in C to process files. The executable file produced has the name **`calculator`**, with the following command-line argument format:

```
./calculator <operation> <input filename> <output filename>
```

The operation is specified by an integer (0, 1, or 2), which is used as an index into the function pointer array **`func_arr`** defined as shown below. All other indices are invalid.

```
int add (int a, int b) {return a + b;}
int magic_1 (int a, int b); // definition not needed for problem
int magic_2 (int a, int b); // definition not needed for problem

typedef int (*operation_t) (int, int);
static operation_t func_arr[3] = {&add, &magic_1, &magic_2};
```

The number of lines in each input file varies, with each line contains two integers and a space between them. You may assume that the input file has the correct format (as specified). One example of the content of a input file **`input.txt`**:
1 1
2 3
4 5

The output file should have the same number of lines as the input file. Every line of the output file should contain one integer, which is the result of applying the operation on the two integers of the corresponding line of the input file. For example, if the following command is run (on the example input above),

```
./calculator 0 input.txt output.txt
```

the program should produce a file called **`output.txt`**, with content:
2
5
9

Complete the code below **by writing portions of code on the following page, using only the lines provided**. Return 0 for success, or -1 for any failure. **Be sure to check for all error conditions.** See the reference sheet for C's I/O functions.

```
//... some headers and other information omitted
int main(int argc, char* argv[]){

    // Check the command line arguments
    if (argc != 4 || strlen(argv[1]) != 1 ||
        argv[1][0] > '2' || argv[1][0] < '0'  ) {return -1};

    // *** Your code for Part 1 is inserted here ***

    FILE* in_file;
    FILE* out_file;
    // *** Your code for Part 2 is inserted here ***

    int a, b;
    // *** Your code for Part 3 is inserted here ***

    // *** Your code for Part 4 is inserted here ***
}
```

Problem 3, continued:

1. **(3 points)** Read the argument checking code (given to you), then write an expression to calculate the operation index given to the program and store it in the variable **`func_index`**.

```
int func_index = argv[1][0] - '0' ;
```

2. **(7 points)** Write the code to prepare streams for I/O files based on the command-line arguments.

```
in_file = fopen (argv[2], "r");
if (NULL == in_file) {
    return -1;
}
out_file = fopen (argv[3], "w");
if (NULL == out_file) {
    fclose (in_file);
    return -1;
}
```

---

3. **(10 points)** Write the code to apply the chosen operation to every line of the input file and write the result to the output file.

```
while (2 == fscanf (in_file, "%d%d", &a, &b)) {
    if (0 > fprintf (out_file, "%d\n", (*(func_arr[func_index])) (a, b))) {
        fclose (in_file);
        fclose (out_file);
        return -1;
    }
}
```

---

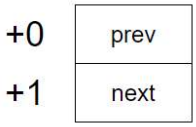
4. **(5 points)** Write the code to release resources and return success.

```
fclose (in_file);
fclose (out_file); // can check return value here instead of fprintf above
return 0;
```

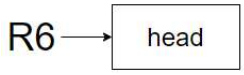
Problem 4 (20 points): Lists and Hierarchies of Structures

Recall that in class we developed container code for cyclic, doubly-linked lists with sentinels. Later, you made use of the code in a lab. The node structure for the list (using a shorter name) appears below, and a diagram of the structure in memory when compiled for LC-3 appears to the right (with offsets).

```
typedef struct dl_t dl_t;
struct dl_t {
      dl_t* prev; // previous element in the list
      dl_t* next; // next element in the list
};
```



1. **(10 points)** Implement the function **`dl_length`** shown below as an LC-3 assembly subroutine. The diagram to the right of the code shows the stack on entry to your subroutine.
   - Your code may **change only R0, R1, R2 and R3**.
   - Do **NOT** set up a stack frame. The local variable **`count`** can be kept in a register of your choice (R0-R3).
   - **USE 15 OR FEWER INSTRUCTIONS** (not counting **`RET`**, provided for you).
   - **Push the return value onto stack** before returning.




```
int16_t dl_length (dl_t* head) {
    int16_t count = 0;
    for (dl_t* elt = head->next; elt != head; elt = elt->next) {
        ++count;
    }
    return count;
}
```

```
DL_LENGTH   AND   R0,R0,#0     ; count

            LDR   R1,R6,#0     ; head

            LDR   R2,R1,#1     ; elt

            NOT   R1,R1        ; R1 <- -head

            ADD   R1,R1,#1

LOOP        ADD   R3,R1,R2     ; R3 <- elt - head

            BRz   DONE

            ADD   R0,R0,#1     ; count++

            LDR   R2,R2,#1     ; elt = elt->next

            BRnzp LOOP

DONE        ADD   R6,R6,#-1    ; return count

            STR   R0,R6,#0

            _____________________________________________

            _____________________________________________

            RET
```

Problem 4, continued:

```
typedef enum {FISH, DOG, CAT, BIRD, AARDVARK, NUM_ANIMAL_TYPES} animal_type_t;

typedef struct animal_t {
      dl_t          dl;       // for inclusion in doubly-linked list
      char*         name;     // animal's name (dynamically allocated)
      animal_type_t type;     // type of animal
} animal_t;

typedef struct bird_t {
      animal_t anm;           // a bird is a type of animal
      int32_t  migratory;     // 1 for migratory, 0 for not migratory
      double   speed;         // speed of the bird (always positive)
} bird_t;

// definitions of other animal types omitted
```

2. **(10 points)** Now we have a bunch of animals contained in a doubly-linked list. Given **head**, a pointer to the sentinel for the list, we want to find the fastest migratory bird in the list. If the list contains no migratory birds, the function should return NULL. You may assume that no two birds have the same speed. Complete the C function below, **using no more lines than are provided for you**.

```
bird_t* find_fastest_migratory_bird (dl_t* head) {

    bird_t*   rval = NULL; // return value
    double    max = -1;    // maximum speed seen
    animal_t* a;
    bird_t*   b;

    for (dl_t* elt = head->next_____________; head != elt_____________________ ;

         elt = elt->next_________________ ) {

        a = (animal_t*)elt;

        b = (bird_t*)elt;

        if (BIRD == a->type && b->migratory && b->speed > max) {

            rval = b;

            max = b->speed;

        }

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

    }
    return rval;
}
```

Problem 5 (18 points): Constructors, Destructors, and Operator Overloading

Read the following C++ code and answer the questions.

```
#include <stdio.h>

class Mystery {
private:
    int x;
public:
    Mystery () { printf("M"); }
    Mystery (int xval) : x(xval + 1) { printf("Y"); }
    const Mystery& operator= (int xval) {
        xval = 1;
        printf("S");
        return *this;
    }
    Mystery (const Mystery& m) : Mystery(m.x + 10) { printf("T"); }
    ~Mystery() { printf("E"); }
};

Mystery c, d;

int main() {
    printf("---START---\n");
    c = d = 0;
    printf("\n");
    Mystery a = 42;
    printf("\n");
    Mystery b = a;
    printf("\n");
    c = a;
    printf("\n---END---");
    return 0;
}
```

1. **(12 points)***** The output of this program has **EXACTLY SIX LINES**. What is the output? Write "blank" for a blank line.

   Line 1: `MM---START---`

   Line 2: `S`

   Line 3: `Y`

   Line 4: `YT`

   Line 5: **blank**

   Line 6: `---END---EEEE`

2. **(6 points)** What are the following values immediately before execution of "`return 0`"? Write "bits" for any value that can't be determined.

   a.x = ___43_____ b.x = ___54_____ c.x = ___43_____ d.x = ___bits_____

some of the routines from C's standard libraries

```
// returns new stream, or NULL on failure
FILE* fopen (const char* path, const char* mode);

// returns 0 on success, or EOF on failure
int fclose (FILE* stream);


// returns char, or EOF on failure
int fgetc (FILE* stream);

// returns s, or NULL on failure
char* fgets (char* s, int size, FILE* stream);

// returns # of elements read, or 0 on failure
size_t fread (void* ptr, size_t size, size_t nmemb, FILE* stream);

// returns # of conversions, or -1 on failure (no conversions)
int fscanf (FILE* stream, const char* format, ...);

// returns # of conversions, or -1 on failure (no conversions)
int sscanf (const char* str, const char* format, ...);


// returns c, or EOF on failure
int fputc (int c, FILE* stream);

// returns value >= 0 on success, < 0 on failure
int fputs (const char* s, FILE* stream);

// returns # of elements written, or 0 on failure
size_t fwrite (const void* ptr, size_t size, size_t nmemb,
               FILE* stream);

// returns # of characters printed, or negative value on failure
int fprintf (FILE* stream, const char* format, ...);

// returns # of characters printed, or negative value on failure
int snprintf (char* str, size_t size, const char* format, ...);


// returns length of string s, not counting terminal NUL
size_t strlen (const char* s);


// rounds x up to the next integral value (smallest integer >= x)
double ceil (double x);
```