ECE220 F20 Answer(UIUC)

Problem 1

1. 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.

```c
#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;
}
```

Answer: Special Input overwrote return address on stack with address of `weird`

1. **栈溢出原理**： `run()`函数中定义了`char buffer[10]`，但`scanf("%s", buffer)`未限制输入长度。若输入超过10字节（如输入`"weird\0"`需7字节，但实际可能更长），会导致缓冲区溢出。
2. **覆盖返回地址**： 溢出数据会写入栈中`run()`的返回地址位置（原指向`main`中`printf("main")`的指令），将其替换为`weird`函数的入口地址。
3. **控制流劫持**： `run()`执行完毕后，CPU从栈弹出（已篡改的）返回地址并跳转至`weird()`，触发其`printf("weird")`，最终程序异常终止。

核心机制： 通过输入操控栈内存，篡改函数返回地址，实现任意代码执行（此处为简单跳转）。

2. 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. LIST ALL FIELDS of `instance` that are accessible by name within the function `Derived::aFunction`.

   Answer: B, D, E, F

2. LIST ALL FIELDS of `instance` that are accessible by name within the function `anotherFUnction`

   Answer: D, F

3. 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 players based on the use of a deep neural network, 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.

   Answer: Calculate rank once for each player and store them in a field of player_t

原始代码中，每次比较两个玩家时，`player_sort_by_rank`函数会**重复调用`player_get_rank`**（每次比较调用两次）。由于`player_get_rank`需5秒/次，排序1000个玩家时，比较次数约为$O(1000 \log 1000) \approx 10^5$次，总耗时高达$10^5 \times 2 \times 5 = 10^6$秒。

因此我们提出优化方案：首先**预计算排名**，在排序前一次性计算所有玩家的排名，并将结果存储在`player_t`结构体的字段中（如`cached_rank`），然后**直接使用缓存值**，比较函数直接读取已缓存的排名值，无需重复调用`player_get_rank`。此方法将耗时操作从$O(n \log n)$次减少至$O(n)$次，性能提升约为20倍

4. 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.

Answer: Infinite recursion to `ALPHA::add` with `double` argument

`add(double amt)`函数中调用了`add(ceil(amt))`。`ceil(amt)`返回的是`double`类型（例如`ceil(1.5)`返回`2.0`），因此会再次调用`add(double)`函数，形成**无限递归**。这会导致程序栈溢出，运行时崩溃。

Problem 2

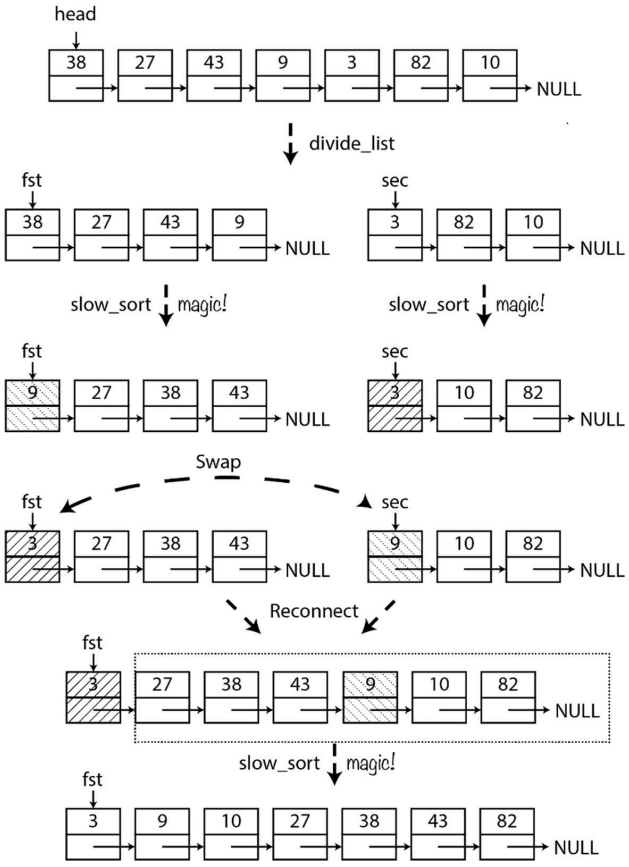
Quick Sort is quick, but difficult to understand. Instead, you must implement the "Slow Sort1" algorithm. As you know, writing a recursive function requires a “leap of faith,” which means that you believe that your recursive calls work even before you have finished implementing the function. Slow Sort 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 $m = \frac{i+j}{2}$
- I call `slowsort` to sort the first and the second half for me. I believe it works.
- Both haves 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 $A[i+1 \ldots j]$? Hmmm...Let me just call `slowsort` to sort them!

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.)



The linked list is constructed using the following structure:

```
typedef struct element_t element_t;
struct element_t{
    int32_t value;
    element_t* next;
};
```

```
// Divide a linked list starting at head into two equal halves
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 (head == NULL || head->next == NULL){return head;}
    // Otherwise, divide the list into 2 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
    if (fst->value > sec->value){swap(&fst, &sec);}
    // Reconnect fst and sec into a single list.
    for (last = fst; last->next != NULL; last = last->next){}
    last->next = sec;
    // Sort the rest of the list.
    fst->next = slow_sort(fst->next);
    // Return the sorted list.
    return fst;
}
```

核心思想：通过递归分割链表、排序子链表、比较头部元素并重组，最终实现排序。其关键在于**“信仰跳跃”**——假设递归调用能正确完成，专注于当前步骤的逻辑。

算法步骤解析

1. **终止条件**：若链表为空或仅一个节点，直接返回（无需排序）。
2. **分割链表 (`divide_list`)**：将链表均分为两半（前半段 ≥ 后半段长度）。例如，原链表 `38 → 27 → 43 → 9 → 3 → 82 → 10`被分割为：
   - `fst`: `38 → 27 → 43 → 9`
   - `sec`: `3 → 82 → 10`
3. **递归排序子链表**：对 `fst` 和 `sec` 分别调用 `slow_sort`，确保子链表已排序。示例中排序后：
   - `fst`: `9 → 27 → 38 → 43`
   - `sec`: `3 → 10 → 82`
4. **比较并交换头部元素**：若 `fst` 的首元素 > `sec` 的首元素，交换两链表指针。示例中 `9 > 3`，交换后：
   - `fst`: `3`
   - `sec`: `9 → 27 → 38 → 43 → 10 → 82`
5. **合并链表**：将 `sec` 连接到 `fst` 的末尾，形成新链表。示例合并后：`3 → 9 → 27 → 38 → 43 → 10 → 82`
6. **递归处理剩余部分**：对 `fst->next`（即 `9 → 27 → 38 → 43 → 10 → 82`）再次调用 `slow_sort`，重复上述步骤。此时，链表被重新分割、排序、比较、合并，直至完全有序。

Problem 3

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

typdef int (*operation_t) (int, int);
static operation_t func_ar[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 an 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 ***
}
```

命令行参数检查：

`if (argc != 4 || strlen(argv[1]) != 1 || argv[1][0] > '2' || argv[1][0] < '0' ) {return -1};`

目的：验证输入参数的合法性

检查内容：

- `argc != 4`：确保程序名、操作类型、输入文件、输出文件共4个参数。
- `strlen(argv[1]) != 1`：操作类型必须是单字符（如 `"0"` 而非 `"10"`）。
- `argv[1][0]` 是否在 `'0'` 到 `'2'` 之间（ASCII字符比较）。

错误处理：不满足条件直接`return -1;`

1. 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'`

   其中`argv[1]`表示第二个参数，而`argv[1][0]`表示该参数的第一个字符，该行代码将操作类型字符转为整数(0,1,2)

2. Write the code th prepare streams for I/O files based on the command-line arguments.

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

3. Write the code to apply the chosen operation to every line of the input file and write the result to the output file.

```
while (fscanf(in_file, "%d %d", &a, &b) == 2){ // 逐行读取输入
    int result = (*func_arr[func_index])(a, b); // 调用对应函数
    if (fprintf(out_file, "%d\n", result) < 0){ // 写入结果
        fclose(in_file);
        fclose(out_file);
        return -1;
    }
}
```

流程:

1. 使用 `fscanf` 从输入文件读取每行的两个整数 `a` 和 `b`。
2. 通过函数指针数组 `*(func_arr[func_index])` 调用对应的运算函数（`add`、`magic_1` 或 `magic_2`）。
3. 将结果写入输出文件，若写入失败（如磁盘空间不足）则关闭文件并返回 `-1`。函数`fprintf`如果成功运行，则返回成功写入流的字符数，如果失败则返回负值。

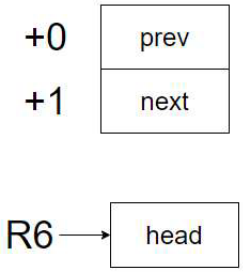
4. Write the code to release resources and return success.

```
fclose(in_file);
fclose(out_file);
return 0;
```

Problem 4

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_1{
    dl_t* prev; // previous element in the list
    dl_t* next; // next element in the list
};
```



1. 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.

```c
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;
}
```

1. Your code may change only R0, R1, R2 and R3.
2. Do NOT set up a stack frame. The local variable count can be kept ina register of your choice (R0-R3).
3. USE 15 OR FEWER INSTRUCTIONS (not counting RET, provided for you).
4. Push the return value onto stack before returning.

Answer:

```
DL_LENGTH
    AND R0, R0, #0 ; count
    LDR R1, R6, #0 ; R1 = head
    LDR R2, R1, #1 ; R2 = elt
    NOT R1, R1
    ADD R1, R1, #1 ; R1 = -head
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
```

2. Now we have a bunch of animals contained in a doubly-linked list. Given `head`, a pointerto 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 samespeed. Complete the C function below, using no more lines than are provided for you.

```c
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
```

Answer:

```c
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; elt != head; elt = elt->next){
        a = (animal_t*)elt;
        b = (bird_t*)elt;
        if (a->type == BIRD && b->migratory && b->speed > max){
            rval = b;
            max = b->speed;
        }
    }
    return val;
}
```

Problem 5

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

```cpp
#include <stdio.h>
class Mystery{
private:
    int x;
public:
    Mystery(){printf("M");} // 无参数构造函数
    Mystery(int xval):x(xval+1){printf("Y");} // 参数化构造函数
```

```
    const Mystery& operator= (int xval){ // 注意，赋值符号的右侧必须是int
        xval = 1;
        printf("S");
        return *this;
    } // 运算符重载
    Mystery(const Mystery& m):Mystery(m.x + 10){printf("T");} // 复制构造函数
    ~Mystery(){printf("E");}
};

Mystery c, d; // 调用无参数构造函数

int main() {
    printf("---START---\n");
    c = d = 0; // 此处调用的是赋值运算符重载
    printf("\n");
    Mystery a = 42; // 此处调用的是参数化构造函数
    printf("\n");
    Mystery b = a; // 调用复制构造函数，首先先由参数化构造函数创建对象，然后printf("T")
    printf("\n");
    c = a; // 调用默认赋值符号，因为=右侧不是int类型，不属于自定义的那个运算符重载
    printf("\n---END---");
    return 0; // 此时所有的对象全都应该用析构函数销毁，一共有4个对象
}
```

1. 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. 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
```

怎样区分“=”运算符重载和复制构造函数?

复制构造函数

- **定义**：用于创建新对象时，用另一个同类对象初始化它。
- **调用场景**：

- 显式初始化（如 `MyClass obj2 = obj1;`）。
- 对象作为参数按值传递（如 `func(obj)`）。
- 函数返回对象（按值返回时）。

- **函数签名**：

```
MyClass(const MyClass& other); // 必须是引用传参，否则会无限递归
```

赋值运算符重载

- **定义**：用于将一个已存在的对象赋值给另一个已存在的对象。
- **调用场景**：
  - 已存在对象之间的赋值（如 `obj2 = obj1;`）。
- **函数签名**：

```
MyClass& operator=(const MyClass& other);
```

关键区别

特性	复制构造函数	赋值运算符重载
是否创建新对象	是（创建新对象并初始化）	否（修改已存在的对象）
调用示例	`MyClass obj2 = obj1;`	`MyClass obj2; obj2 = obj1;`
是否需要处理自赋值	不需要（因为新对象未初始化）	需要（如 `obj = obj`，需避免资源泄漏）
典型用途	初始化新对象	替换已有对象的内容

示例代码

```
class MyClass {
public:
    int* data;

    // 复制构造函数
    MyClass(const MyClass& other) {
        data = new int(*other.data); // 深拷贝
        std::cout << "Copy Constructor called\n";
    }

    // 赋值运算符重载
    MyClass& operator=(const MyClass& other) {
        if (this == &other) return *this; // 处理自赋值
        *data = *other.data; // 深拷贝
        std::cout << "Assignment Operator called\n";
        return *this;
    }
};
```

```
int main() {
    MyClass obj1;
    obj1.data = new int(10);

    MyClass obj2 = obj1; // 调用复制构造函数
    MyClass obj3;
    obj3 = obj1;         // 调用赋值运算符
    return 0;
}
```

4. 常见错误

- **忘记深拷贝**：若类包含指针成员，未实现深拷贝会导致多个对象共享同一块内存，最终引发**双重释放**或**野指针**问题。
- **忽略自赋值检查**：在赋值运算符中，若未检查自赋值（如 `obj = obj`），可能导致资源泄漏或逻辑错误。
- **参数非引用传递**：复制构造函数的参数必须是引用类型（如 `const MyClass&`），否则会无限递归调用复制构造函数本身。

5. 默认行为：若未显式定义，编译器会生成默认的复制构造函数和赋值运算符，但它们仅执行**浅拷贝**（直接复制指针地址而非指向的内存内容）。对于需要动态资源管理的类，必须手动实现深拷贝。