Problem 1 (20 points): Testing and Debugging

(10 points) The recursive function below is meant to return 1 if the length of the string passed is odd, and 0 if the length of the string passed is even. Unfortunately, the function has a bug. In NO MORE THAN 15 WORDS, describe the bug. Then fix the function.

(the bug) _____

_____

```
int strodd (const char* s)
{
    if ('\0' == *(s + 1)) {
        return 1;
    }
    if ('\0' == *s) {
        return 0;
    }
    return (strodd (s + 2));
}
```

(10 points) The function below is meant to check whether two strings, **`s`** and **`t`**, match each other when reversed. For example, "example" matches "elpmaxe" when reversed. If the strings match when reversed, the function should return 1, and should otherwise return 0. Sadly, the function has a bug. In NO MORE THAN 15 WORDS, describe the bug. Then fix the function.

(the bug) _____

_____

```
int backwards (const char* s, const char* t)
{
    const char* u = s;

    while ('\0' != *u) { u++; }

    do {
        if (*--u != *t++) {
          return 0;
      }
    } while (s < u);

    return ('\0' == *t);
}
```

Problem 2 (30 points): Manipulating Linked Lists

Prof. Lumetta needs your help!

He has gathered his books into a linked list of **`book_t`** structures, defined as shown below. The **`next`** field links one book to the next in the list.

```
typedef struct book_t book_t;
struct book_t {
    char*   title;
    char*   author;
    char*   call_number;
    int     due_date;
    book_t* next;
};
```

Prof. Lumetta needs to separate a linked list of "books" (**`book_t`**'s) into two separate lists based on their due dates (the **`due_date`** field).

Help Prof. Lumetta by completing the C function on the following page. The function signature is as follows:

```
book_t* collect_books (book_t** listp, int due_by);
```

The function takes two parameters:

- **`listp`**—a pointer to a pointer to the first book in the original linked list of books
- **`due_by`**—the date used to separate the list

The function must separate the original list into two separate linked lists, each correctly terminated, as follows:

- All books with **`due_date`** field less than or equal to **`due_by`** should be removed from the original list and collected into a new list. The order of these books in the new list does not matter. The function must return a pointer to the first book in the new list.

- Any book with a **`due_date`** larger than **`due_by`** should remain in the original list. The order of these remaining books must not change. The function must update the value to which **`listp`** points to indicate the first book in the original list with a **`due_date`** larger than **`due_by`**.

Problem 2, continued:

```
        /* structure definition replicated for your convenience */

typedef struct book_t book_t;
struct book_t {
    char*   title;
    char*   author;
    char*   call_number;
    int     due_date;
    book_t* next;
};


          /* Complete the function by filling in the blanks. */

book_t* collect_books (book_t** listp, int due_by)
{
    book_t* collected;
    book_t* book;

    ____________________________________________ ;


    while ( ____________________________________ ) {

        book = *listp;

        if (due_by < _____________________________ ) {


            listp = ___________________________________ ;

        } else {

            *listp = ___________________________________ ;


            ____________________________________________ ;


            collected = _______________________________ ;
        }
    }

    ____________________________________________ ;
}
```

Problem 3 (20 points): Data Structures in Memory

This problem refers to the data structure shown below, one of which is similar (NOT identical) to a structure used in Problem 2. Shown below on the left is C code defining these data structures along with the variables `flt, ptr, lib`, and `str_p`. Shown on the right is the LC-3 memory at runtime.

C code:

```
struct library_t {
    char*      name;
    book_t*    books;
    book_t*    borrowed;
    library_t* next;
};
struct book_t {
    char* title;
    char* author;
    book_t* next;
};
float     flt;
float*    ptr;
library*  lib;
char**    str_p;
```

LC-3 Memory

Address	Data	Comments
x4001	x5165	
x4002	x400D	
x4003	x0000	
x4004	xA792	
x4005	x400A	
x4006	x4915	
x4007	x7492	
x4008	x004F	
x4009	x0000	
x400A	x4008	
x400B	x8217	
x400C	x4001	
x400D	x0053	
x400E	x004C	
x400F	x0000	
x4010	xABCD	`flt`
x4011	x400F	`ptr`
x4012	x4004	`lib`
x4013	x4002	`str_p`

Complete the table below by indicating the value and C type for each expression in the left column. If the expression is a structure, provide the start and end address of the structure (for example, Mem[start_addr : end_addr]) instead of a value. The first two rows of the table have been completed for you.

Expression	Value	C type
`flt`	xABCD	float
`&flt`	x4010	float *
`ptr`		
`*ptr`		
`&ptr`		
`lib->books`		
`**str_p`		
`lib->books->next`		
`&(lib->books->next->next)`		
`*(lib->books->name)`		
`*(lib->borrowed)`		
`lib - 1`		

Problem 4 (30 points): Dynamic Allocation and I/O

Prof. Lumetta needs your help with a function designed to read a two-dimensional array of integers (called a "matrix") from a file. The height and width of the matrix are stored on the first line of the file, followed by the (height × width) elements of the matrix.

The function must read the height and width, check that they are valid (from 1 to 100 each), allocate space for the array, then read all elements of the matrix from the file into the dynamically-allocated array.

If anything goes wrong while reading the matrix from the file **`f`**, the function must free any dynamically-allocated memory and return NULL.

The function also takes two other parameters, **`heightPtr`** and **`widthPtr`**. If the matrix is read successfully from the file, the function must write the height and width of the matrix to these pointers, respectively. Otherwise, the values pointed to by the pointers are not relevant (they can be changed, but they will be ignored by the caller).

COMPLETE THE CODE on the next page by filling in the blanks appropriately.

An example of the matrix file format appears below.

```
3 5
1 2 3 4 6
-2 4 1 200 42
0 1 2 72 4
```

Problem 4, continued:

```
int* readIntMatrix (FILE* f, int* heightPtr, int* widthPtr)
{
    char buf[100];
    int* m;
    int  nElts;
    int  i;

    if (NULL == f || NULL == heightPtr || NULL == widthPtr) {
        return NULL;
    }
    if (NULL == fgets (buf, 100, f) ||

        2 != sscanf (buf, "_____________________________", heightPtr, widthPtr) ||

        0 > *heightPtr || 100 < *heightPtr ||

        0 > *widthPtr  || 100 < *widthPtr) {

        return NULL;
    }

    nElts = ________________________________________________ ;


    m = malloc (_____________________________________ * nElts);


    if ( _____________________________________ ) {

        return NULL;
    }

    for (i = 0; nElts > i; i++) {
        if (1 != fscanf (f, "%d", ___________________________________)) {


            _____________________________________________ ;


            _____________________________________________ ;
        }
    }

    ________________________________________________ ;
}
```