Name: **SOLUTION** UID#: DirectoryID:

CMSC216: Practice Exam 1A SOLUTION

Fall 2025
University of Maryland

Exam period: 20 minutes *Points available:* 40 *Weight:* 0% of final grade

Background: Noel Pwanter is experimenting with a linked list application using code shown nearby. The function `list_init()` takes a pointer to a `list_t` struct as its argument. Noel thinks the OLD VERSION, shown below commented, could be improved by directly declaring a pointer as shown in the NEW VERSION. Noel is surprised when Valgrind identifies problems and the program crashes.

```
#include "list.h"

int main(int argc, char *argv[]){
  // ...
  // list_t list;       // OLD VERSION
  // list_init(&list);
  list_t *listptr;      // NEW VERSION
  list_init(listptr);
  // ...
  return 0;
}

// initialize list
void list_init(list_t *list){
  list->head = NULL;
  list->size = 0;
  return;
}
```

```
>> gcc -g list_main.c
>> valgrind ./a.out
==4529== Use of uninitialised value of size 8
==4529==    at 0x109147: list_init (list_main.buggy.c:15)
==4529==    by 0x109133: main (list_main.buggy.c:8)
==4529==
==4529== Invalid write of size 8
==4529==    at 0x10914F: list_init (list_main.buggy.c:15)
==4529==    by 0x10913B: main (list_main.buggy.c:8)
==4529==  Address 0x0 is not stack'd, malloc'd or free'd
==4529==
==4529== Process terminating with default action of
==4529==  signal 11 (SIGSEGV): dumping core
==4529==  Access not within mapped region at address 0x0
==4529==
==4529== HEAP SUMMARY:
==4529==   in use at exit: 0 bytes in 0 blocks
==4529== total heap usage: 0 allocs, 0 frees
```

Problem 1 (15 pts): Answer the following questions about Noel's code.

(A) Describe the problems that Valgrind identifies and what they mean about Noel's code.

SOLUTION: The pointer `listptr` *has not been initialized and based on the Valgrind output is defaulting to address 0x0 (NULL). This triggers an out of bounds access and a segmentation fault.*

(B) Reverting the code to the OLD VERSION will fix this problem. Describe in which Logical Region of memory the `list` is allocated in the OLD VERSION and WHEN the memory associated with `list` will be de-allocated.

SOLUTION: The old version allocates `list` *in the stack where it has enough space for the struct itself. This memory will be de-allocated when* `main()` *returns like all the other local variables in* `main()'s` *stack frame.*

(C) Noel wants her `listptr` as a pointer to the `list` struct to avoid needing to use the `&list` syntax at later points in her code. What code should she write to achieve this? Indicate if your answer would keep the memory for the `list_t struct` in the same place as the OLD VERSION or move it to a different logical region of memory.

SOLUTION: Noel could uncomment the OLD VERSION and connect her pointer to the existing `list` *via* `list_t *listptr = &list;` *This would keep the struct in the stack but provide a pointer to it. The memory would still be de-allocated when* `main()` *returns.*
ALTERNATIVELY Noel could heap-allocate the list: keep the OLD VERSION commented and use
`list_t *listptr = malloc(sizeof(list_t));`
to get space in the heap. She would also then need to add `free(listptr)` *at the end of her* `main()` *function to avoid a memory leak.*

Problem 2 (15 pts): Nearby is a description of the function `equiv_exchange()` along with a `main()` function demonstrating with example calls. Write this function to meet the specification given.

```c
#include "equiv_exch.h"
typedef struct {
  char x[128];
  char y[128];
} strpair_t;

int equiv_exchange(strpair_t *strpair);
// If the x/y fields are strings of
// equal length, swap them and return 1.
// Otherwise do nothing and return 0.
// CONSTRAINT: does NOT use strcpy() or
// memcpy() functions.

int main(){
  int ret;
  strpair_t elrics = {
    .x="Ed", .y="Al"
  };
  ret = equiv_exchange(&elrics);
  printf("ret:%d  x/y: %s %s\n",
         ret, elrics.x, elrics.y);
  // ret:1  x/y: Al Ed

  strpair_t side = {
    .x="Winry", .y="Mustang"
  };
  ret = equiv_exchange(&side);
  printf("ret:%d  x/y: %s %s\n",
         ret, side.x, side.y);
  // ret:0  x/y: Winry Mustang

  strpair_t homonc = {
    .x="Lust", .y="Envy"
  };
  ret = equiv_exchange(&homonc);
  printf("ret:%d  x/y: %s %s\n",
         ret, homonc.x, homonc.y);
  // ret:1  x/y: Envy Lust
  return 0;
}
```

Note CONSTRAINTs: does not use `strcpy()` / `memcpy()`

YOUR CODE HERE

```c
///// SOLUTION /////
int equiv_exchange(strpair_t *strpair){
  int lenx = strlen(strpair->x);
  int leny = strlen(strpair->y);
  if(lenx != leny){
    return 0;
  }
  for(int i=0; i<lenx; i++){
    char c = strpair->x[i];
    strpair->x[i] = strpair->y[i];
    strpair->y[i] = c;
  }
  return 1;
}

// Alternate version uses pointer aliases to shorten the code
// (eliminates repeated ‘strpair->xtstr‘ syntax)
int equiv_exchange_ALT(strpair_t *strpair){
  char *x = strpair->x;
  char *y = strpair->y;
  int lenx = strlen(x);
  int leny = strlen(y);
  if(lenx != leny){
    return 0;
  }
  for(int i = 0; i < strlen(x);i++){
    char tmp = x[i];
    x[i] = y[i];
    y[i] = tmp;
  }
  return 1;
}
```

Problem 3 (10 pts): Write a `main()` function nearby that changes its behavior based on command line arguments in the way demonstrated in the interactive section below. Only printing the indicated messages is required. You do not need to include header files in your solution but should write the whole `main()` function including its prototype.

```
>> gcc command_line_args.c
>> ./a.out
usage: ./a.out {--runA, --runB}
>> ./a.out --runA
Running A version
>> ./a.out --runB
Running B version
>> ./a.out zap
unrecognied option: zap
```

```c
///// SOLUTION ////
#include <stdio.h>   // headers not required
#include <string.h>
int main(int argc, char *argv[]){
  if(argc < 2){
    printf("usage: ./a.out {--runA, --runB}\n");
    return 1;
  }
  char *option = argv[1];
  if( strcmp(option,"--runA")==0 ){
    printf("Running A version\n");
  }
  else if( strcmp(option,"--runB")==0 ){
    printf("Running B version\n");
  }
  else{
    printf("unrecognied option: %s\n",option);
  }
  return 0;
}
```