Name: NumericID#: UID:

CMSC216: Practice Exam 1A SOLUTION

Spring 2024
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 for this (show 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.

```
#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

```
///// 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): Fill in the following table of equivalent ways to write these 8 bit quantities. There are a total of 9 blanks to fill in and the first column indicates which blanks occur in which lines. Assume two's complement encoding for the signed decimal column.

SOLUTION Blank #s	Binary	Hex	Octal	Unsigned Decimal	Signed Decimal
#1 #2 #3	0001 1011	0x1B	0033	27	27
#4 #5 #6	1010 0101	0xA5	0245	165	-91
~x + 1	0101 1011				
#7 #8 #9	1100 0111	0xC7	0307	199	-57
~x + 1	0011 1001				

```
NOTES
- Octal shows leading 0 which is not strictly necessary
- Typical twos’ complement conversion technique show below
  binary representation: invert bits and add 1
```