CS 136L Lecture 10

Valgrind and Address Sanitizer

Code Examples from the notes

Let's look at some of the code examples from the module (included below for reference)

Test 1

```c
// File test1.c
#include <stdio.h>

int main(void){
   int i;
   printf("%d",i);
   return 0;
}
```

Test 2

```
// File test2.c
int main(int argc, char **argv) {
  int arr[2];
  if (arr[argc != 1])
    return 1;
  else
    return 0;
}
```

Test 3

```c
// File test3.c
#include <stdio.h>

int main(void){
   int *p = NULL;
   *p = 5;
   printf("%d\n",*p);
}
```

Test 4

```c
// File test4.c
#include <stdio.h>

int main(void){
   int *p = NULL;
   int x = *p;
   printf("%d\n",x);
}
```

Test 5

```c
//File: test5.c
#include <stdio.h>

void print_element(int *arr, int index){
    printf("%d",arr[index]);
}
int main(void) {
  int a[10];
  print_element(a,10);
}
```

Test 6

```c
// File test6.c
#include <stdlib.h>
#include <stdio.h>

char *get_hello(){
  char str[20] = "Hello World!";
  char *toRet = str;
  return toRet;
}
int main (){
  printf("%s\n",get_hello());
  return 0;
}
```

Test 7

```
// File test7.c
#include <stdlib.h>
#include <stdio.h>

char *get_hello(){
  char str[20] = "Hello World!";
  return &str;
}
int main (){
  printf("%s\n",get_hello());
  return 0;
}
```

Test 8

```c
//file test8.c
#include <stdio.h>

struct vec {
  int x;
  int y;
};

// get_vec(_x,_y) is a helper for creating a
//   vec object with the specified values
struct vec *get_vec(int _x, int _y){
   struct vec myVec;
   struct vec *p = &myVec;
   p->x = _x;
   p->y = _y;
   return p;
}

int main(void){
   struct vec *p1 = get_vec(1,2);
   struct vec *p2 = get_vec(5,10);

   printf("%d,%d\n", p1->x, p1->y);
   printf("%d,%d\n", p2->x, p2->y);

   return 0;
}
```

Test 9

```c
//File: test9.c
#include <stdio.h>
#include <stdlib.h>

struct vec {
  int x;
  int y;
};

// get_vec(_x,_y) is a helper for creating a
//   vec object with the specified values
// memory is heap allocated and must be freed by caller
struct vec *get_vec(int _x, int _y){
   struct vec *p = malloc(sizeof(struct vec));
   p->x = _x;
   p->y = _y;
   return p;
}

int main(void){
   struct vec *p1 = get_vec(1,2);
   struct vec *p2 = get_vec(5,10);

   printf("%d,%d\n", p1->x, p1->y);
   printf("%d,%d\n", p2->x, p2->y);

   return 0;
}
```

Test 10

```
// File: test10.c
#include <stdio.h>
#include <stdlib.h>

const int MAX = 2;
int main(void){
  int *array = malloc(MAX);
  array [0] = 42;
  array [1] = 84;
  printf("Value at index 0 is %d\n", array
   [0]);
  printf("Value at index 1 is %d\n", array
   [1]);
}
```

Test 11

```c
// File: test11.c
#include <stdlib.h>

void memory_cleanup(int *memory){
  free(memory);
}

int main (){
  int elements [3];
  memory_cleanup(elements);
  return 0;
}
```

Test 12

```c
// File test12.c
#include <stdio.h>
#include <stdlib.h>

const int MAX = 10;
void helper(int *array){
  // do something with array
  free(array); // free once done
}
int main(void){
  int *array = malloc(sizeof(int) * MAX);
  for (int i=0; i < MAX; ++i) {
    array[i] = MAX - i;
  }
  helper(array);
  printf("%d\n",array[0]);
  return 0;
}
```

Test 13

```c
// File: test13.c
#include <stdlib.h>
#include <stdio.h>

struct resizing_array{
  int length;   // current number of elements
  int capacity; // maximum number of elements
  int *contents;// heap array of ints with capacity of capacity
};

void resize(struct resizing_array *old){
  int *new_arr = malloc(sizeof(int) * old->capacity * 2);
  for(int i=0; i < old->length; ++i){
    new_arr[i] = old->contents[i];
  }
  old->capacity *= 2;
  old->contents = new_arr;
}

int main (){
  struct resizing_array arr;
  arr.length = 0;
  arr.capacity = 4;
  arr.contents = malloc(sizeof(int) * arr.capacity);

  resize(&arr); // call to test resize functionality
  free(arr.contents);
}
```

Test 14

```c
// File: test14.c
#include <stdlib.h>
#include <stdio.h>

struct resizing_array{
  int length;   // current number of elements
  int capacity; // maximum number of elements
  int *contents;// heap array of ints with capacity of capacity
};

void resize(int *old, int old_capacity){
  int *new_arr = malloc(sizeof(int) * old_capacity * 2);
  for(int i=0; i < old_capacity; ++i){
    new_arr[i] = old[i];
  }
  free(old); //free up old contents
  old = new_arr;
}

int main (){
  struct resizing_array arr;
  arr.length = 0;
  arr.capacity = 4;
  arr.contents = malloc(sizeof(int) * arr.capacity);

  resize(arr.contents, arr.capacity);
  arr.capacity *= 2;
  free(arr.contents);
}
```