CSE 333 22au – Section 2: Structs and Debugging

In this class, it is very helpful to be comfortable with `gdb` and `valgrind` as debugging tools. `gdb` allows you to see the source code and has many useful commands for analyzing your program; `valgrind` catches many types of runtime memory errors.

Plenty of debugging resources can be found on the CSE333 Resources Page:
https://courses.cs.washington.edu/courses/cse333/22au/resources.html

Starting `gdb`

For gdb to work with your C/C++ program, you must compile it using the "-g" flag!
To start up `gdb`, run the following command (the `-tui` flag is optional and enables a text UI).

```
bash$ gdb -tui <program file name>
```

Some essential gdb commands

If you want to know more, ask a TA or investigate the resources at the top of the page.

Setting Breakpoints and Continuing

- `break <filename>:<line #>` Set a new breakpoint
- `info breakpoints` Prints information about the set breakpoints
- `continue` Continue normal execution

Controlling Program Execution

- `run <command_line_args>` Run the program with provided command_line_args
- `next` Go to next instruction, but don't five into functions
- `step` Go to next instruction, and dive into functions
- `finish` Continue until current function returns
- `quit` close gdb

Examining the Current Program

- `list` Shows the current or given source context
- `backtrace` Shows the call stack
- `up` Moves up a stack frame
- `down` Moves down a stack frame
- `frame <#>` Move to a specific stack frame
- `print <expression>` Prints content of variable/memory location/register

Starting `valgrind`

Note that valgrind only analyzes the code reached during a specific execution of your program.
Run the following command:

```
bash$ valgrind --leak-check=full ./<program file name>
```

reverse.c

Download full code on the course website.

```c
/* Ask user for a word and print it forwards and backwards.
 * CSE 333 demo (for debugging),  HP
 */

#define MAX_STR 100   /* length of longest input string */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Return a new string with the contents of s backwards */
char * reverse(char * s) {
  char * result = NULL;            /* the reversed string */
  int L, R;
  char ch;

  /* copy original string then reverse and return the copy */
  strcpy(result, s);

  L = 0;
  R = strlen(result);
  while (L < R) {
    ch = result[L];
    result[L] = result[R];
    result[R] = ch;
    L++; R--;
  }

  return result;
}


/* Ask the user for a string, then print it forwards and backwards.
*/
int main() {
  char line[MAX_STR];    /* original input line */
  char * rev_line;       /* backwards copy from reverse function */

  printf("Please enter a string: ");
  fgets(line, MAX_STR, stdin);
  rev_line = reverse(line);
  printf("The original string was:   >%s<\n", line);
  printf("Backwards, that string is: >%s<\n", rev_line);
  printf("Thank you for trying our program.\n");
  return EXIT_SUCCESS;
}
```

Exercise 1

Draw a memory diagram for the execution of the code above up to the call to `strcpy()` in `reverse()`. Make sure to distinguish between local variables on the Stack and Heap-allocated memory.

Exercise 2

Feel free to make a few code changes based on your findings in Exercise 1. However, the rest of your time for this exercise should be spent in `gdb` and `valgrind` and NOT staring at the code. Find and fix all of the remaining logical and memory errors in the code and try to document/associate each fix with the tool features or output that led you there.

Please use the space below for documenting your errors fixed and tooling assistance.

Exercise 3

Fix any remaining style issues with the code in `reverse.c`.

simplestring.c

```c
/* Buggy code for CSE 333 Section 2
 * 1. Draw a memory diagram for the execution to identify errors.
 * 2. Use gdb and valgrind to identify sources of runtime, logical,
 *  and memory errors.
 * 3. Clean up the code style.
 */
#include <string.h>  // strncpy, strlen
#include <stdio.h>   // printf
#include <stdlib.h>  // malloc, EXIT_SUCCESS, NULL

// A SimpleString stores a C-string and its current length
typedef struct simplestring_st {
    char* word;
    int   length;
} SimpleString;

// Allocate a new SimpleString on the heap initialized with word
// and return pointer to the new SimpleString in dest
void InitWord(char* word, SimpleString* dest);

int main(int argc, char* argv[]) {
    char comp[] = "computer";
    SimpleString ss = {comp, strlen(comp)};
    SimpleString* ss_ptr = &ss;

    // expecting "1. computer, 8"
    printf("1. %s, %d\n", ss_ptr->word, ss_ptr->length);

    char cse[] = "cse333";
    InitWord(cse, ss_ptr);
    // expecting "2. cse333, 6"
    printf("2. %s, %d\n", ss_ptr->word, ss_ptr->length);

    return EXIT_SUCCESS;
}

void InitWord(char* word, SimpleString* dest) {
    dest = (SimpleString*) malloc(sizeof(SimpleString));
    dest->length = strlen(word);
    dest->word = (char*) malloc(sizeof(char) * (dest->length + 1));
    strncpy(dest->word, word, dest->length + 1);
}
```

Exercise 4

Draw a memory diagram of the execution of the above code (`simplestring.c`). Be sure to differentiate between memory on the Stack and Heap-allocated memory.

Exercise 5 (Bonus)

Can you identify and fix the bug causing issues with the initialization of our `SimpleString` structs?