CS270 Recitation 9
"Bad Practices and Pointer Bugs"

Goals

Improve your understanding of C pointers and identify programming practices that lead to serious program errors such as segmentation faults, stack corruption, heap corruption and exploitable code.

The Recitation:

Make a subdirectory called R9 for the recitation and copy the following files into the directory:
http://www.cs.colostate.edu/~cs270/Recitations/R9/morepointers.c
http://www.cs.colostate.edu/~cs270/Recitations/R9/morepointers.h
http://www.cs.colostate.edu/~cs270/Recitations/R9/Makefile

Compile the program and run with different command line arguments. Note that the program accepts a single integer value in the range 1 – 4 as the argument:

```
$ make
$ ./r9 1
```

The program should crash when invoked with the above command (i.e., with ./r9 `1`). Open up `morepointers.c` and `morepointers.h` in `gedit` (or any other editor) and follow the program code.

1. Stack Corruption:

The first function (invoked using ./r9 1) demonstrates stack corruption and memory corruption errors. Your task is to use `gdb` (or `ddd`) to step through the function code and find the cause of the error.
At gdb or ddd prompt:

```
(gdb) break 51
(gdb) run 1
(gdb) continue
(gdb) continue
(gdb) print i
(gdb) print a_array[i]
(gdb) disable 1
(gdb) continue
(gdb) backtrace
```

Backtrace should print the call stack for this program, but instead prints a large number of address locations. The reason is that your program call stack has been corrupted inside the function call. Fix the error and rerun the command. This time you should notice a different error. The values stored inside a_array[] have been overwritten! The TA will describe the problem and provide a solution to fix the error (after the error is fixed, set a breakpoint on line 60 and run backtrace to see the call stack).

2. Accessing Invalid Memory:

The second function (invoked using ./r9 2) demonstrates invalid memory accesses. Explain why the program is not printing the desired output?
To understand the error, debug the program with the following commands:

```
(gdb) break 18
(gdb) break 19
(gdb) break 66
(gdb) break 84
(gdb) run 2
```

```
(gdb) print r
(gdb) print *r
(gdb) continue
(gdb) print result
(gdb) continue
(gdb) print result
(gdb) print *result
(gdb) continue
(gdb) print r
(gdb) print *r
(gdb) continue
```

3. Non-Reentrant Functions & Reentrant Functions:

The next two functions describe the difference between functions that work in a multithreaded program versus functions that do not. Reentrant functions can be safely called in both single and multi--threaded programs.

Fact:
Several GNU libc calls are non-reentrant (but libc also provided re-entrant versions with a _r suffix).

In general returning memory from within a function is considered bad practice. Your program should allocate memory in a very restricted set of functions and call other helper routines by passing addresses to these memory locations.