AE1PGA Lab 11

This weeks lab is to allow you to work on coursework. X2Go contains advanced tools for debugging your programs, which are particular useful to help hunt down memory leaks. If you find these tools too complicated then you can ignore them and continue debugging as we have so far, using `printf` statements to display the flow of control and the values of variables in our programs. If you are interested in them you can explore them in your own time to learn how to use these potentially powerful tools.

In order to use either of these tools your program will need to be compiled with debugging information enabled. This causes gcc to include extra information in the program that relates each instruction and piece of data back to the original source code files. To enable compilation with debugging information, add the "-g" flag to the gcc command to compile your program. Eg,

```
gcc -g -std=c99 -lm -Wall -Wformat -Wwrite-strings int_list.c -o int_list
```

If you forget to do this, both GDB and Valgrind will work but they will not be able to produce very useful output as they cannot tell you where in your source code file your problem is.

GDB

GDB is a debugger made by the same group that makes the gcc compiler. A debugger allows you to see what your program is doing while it is running. You can step through your program line by line seeing what lines are executed and what values the variables have, you can tell it to stop when the program crashes so you can debug the crash, you can tell it to stop every time the value of a variable changes, and many other things as well.

We don't have time for a full introduction to GDB but the book contains a chapter on the use of the GDB debugger as an appendix. This appendix isn't printed in the book but is available as a PDF online. The link to the resources page is on Moodle. Download the appendix and read through it to learn how to use gdb. Then, the next time a program has a problem, you could try using gdb to debug it instead of printf statements.

Since gdb is a command line application, it can be hard to remember it's commands. It has a built-in "help" command but I have also linked to a basic "cheat-sheet" reference card on Moodle.

Valgrind

Valgrind is a tool that can help you discover memory-related problems with your programs such as not free-ing memory allocated with malloc, using pointers which are not pointing to a valid area of memory, and accessing some out-of-bounds array indicies. Although advanced use of Valgrind can be quite complex, minimal use of it may help identify where memory and pointer bugs are in your program.

If you have a program called "list" then you can ask Valgrind to do a complete scan of your programs memory accesses with the following command:

```
valgrind --leak-check=full --track-origins=yes ./list
```

This will run the program and generate some output which describes if there are any memory leaks in your program and where in your source code Valgrind thinks they are. I have put a a deliberately wrong file on Moodle (`valgrind-test.c`). If you compile and run Valgrind on this program, Valgrind will tell you it thinks there are three errors in this program. The first error is:

```
==4277== Conditional jump or move depends on uninitialised value(s)
==4277==    at 0x3582A58D45: _IO_vfscanf (in /lib64/libc-2.12.so)
==4277==    by 0x3582A6444C: __isoc99_scanf (in /lib64/libc-2.12.so)
==4277==    by 0x400613: main (valgrind-test.c:9)
==4277==  Uninitialised value was created by a stack allocation
==4277==    at 0x4005E0: main (valgrind-test.c:5)
```

which tells you you are using an uninitialized variable on line 9 (green highlight) and that that variable was declared on line 5 (blue highlight). If we look at the file we can see we have declared a `char *` but are then using that pointer without giving it a value.

The second error also refers to the same variable but this time we tried to use it on line 11 (green highlight):

```
==4277== Conditional jump or move depends on uninitialised value(s)
==4277==    at 0x4A063E3: free (vg_replace_malloc.c:446)
==4277==    by 0x40061F: main (valgrind-test.c:11)
==4277==  Uninitialised value was created by a stack allocation
==4277==    at 0x4005E0: main (valgrind-test.c:5)
```

The final error says that we are returning from the main function without giving a return value. In this case, there is no real problem because if we don't give a return value from main, the operating system will assume the program terminated normally.

```
==4277== Syscall param exit_group(status) contains uninitialised byte(s)
==4277==    at 0x3582AACE38: _Exit (in /lib64/libc-2.12.so)
==4277==    by 0x3582A35AD1: exit (in /lib64/libc-2.12.so)
==4277==    by 0x3582A1ED63: (below main) (in /lib64/libc-2.12.so)
==4277==  Uninitialised value was created by a stack allocation
==4277==    at 0x4005E0: main (valgrind-test.c:5)
```

This last case highlights that Valgrind can't always tell if something is an error or not. One particular limitation that might be important for you is that is cannot detect out-of-bounds array accesses if the array was declared on the stack with an explicit size (eg, `int xs[20]`). For pointers and malloc/free, however, Valgrind can be quite useful at highlighting potential errors you might have otherwise missed.

Valgrind is not mentioned in the textbook but I have put a link on the Moodle page to the official Valgrind "QuickStart" tutorial where you can find more information about basic and advanced usage.

End