Reading Material
Lab Exercise 3: Debugging C programs

Environment

Again, this guide is for debugging C programs on a school PC. All tools mentioned below are already installed in system.

You could use any tool in your own system, but support from TA is only given for problems using the following methods.

This lab exercise awards marks for proper use of suitable tools.

Debugging

To debug a program, I usually insert *printf* statements (remember to output a newline "... \n" at the end or even use *fprintf*(stderr,...) otherwise you might not see incomplete output generated just before a crash)

Some people also prefer to use a debugger to identify less dramatic problems, rather than inserting printf statements. In the lab you will be asked to demonstrate that you can use a debugger.

Especially when you are using pointers, there is some chance that at some point your program will crash with the message: *Segmentation fault*. If you can't work out what happened, use a debugger (e.g. *man gdb* or *man ddd*)

ddd

man ddd says:

> DDD is a graphical front-end for GDB and other command-line debuggers. Using DDD, you can see what is going on "inside" another program while it executes – or what another program was doing at the moment it crashed.
>
> DDD can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:
>
> - Start your program, specifying anything that might affect its behavior.
> - Make your program stop on specified conditions.
> - Examine what has happened, when your program has stopped.

- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

For example, to find where a program is crashing:

- Run *ddd*, loading the program that is crashing e.g. *ddd arrays*
- Click "Run" on the buttons, or Program->Run
- If you get a pop-up window for arguments etc., just click Run

  The top panel will use a big red arrow to point to just before the line of code where the program crashed. The bottom panel will give more detailed information e.g. about function parameters and the line number.
- Click "Up" on the buttons, or Status->Up

  The top panel will use a big grey arrow to point to just before the line of code where the previous function was called. Again, the bottom panel will give more detailed information.

  Repeat this step if you need to.

For more information about *ddd*, try *ddd –manual* (here) or Norm Matloff's ddd Tutorial.

valgrind

valgrind is well suited to identifying problems with pointers or *malloc* (even if your program seems to be working correctly).

man valgrind says it is "a suite of tools for debugging and profiling programs"

The simplest way to use it is without the *–tool* option, to just check the use of pointers and the heap. This tends to produce a lot of output that can be hard to understand at first, so **use it before you have a problem** to see what the output looks like when everything is ok.

Moreover, *valgrind* can spot potential problems even if they aren't (yet) serious enough to cause your program to crash, so **get into the habit of using it whenever you use pointers and malloc.**

To use it, simply run your program *arrays.c* using:

valgrind ./arrays

To get more detailed information on memory leak, add *–leak-check=full* to your command:

valgrind –leak-check=full ./arrays

A piece of code that causes memory leak is shown in figure 1.
Figure 2 displays the ouput of Valgrind running code given in figure 1.
For more information, please read documentation of Valgrind.

```c
#include <stdlib.h>
void *p;
int main() {
  p = malloc(7);
  p = 0; // The memory is leaked here.
  return 0;
}
```

Figure 1: Example code causing memory leak

```
$ valgrind --leak-check=full ./memory-leak
==31485== Memcheck, a memory error detector
==31485== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==31485== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==31485== Command: ./memory-leak
==31485==
==31485==
==31485== HEAP SUMMARY:
==31485==     in use at exit: 7 bytes in 1 blocks
==31485==   total heap usage: 1 allocs, 0 frees, 7 bytes allocated
==31485==
==31485== 7 bytes in 1 blocks are definitely lost in loss record 1 of 1
==31485==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.
    so)
==31485==    by 0x40053D: main (memory-leak.c:4)
==31485==
==31485== LEAK SUMMARY:
==31485==    definitely lost: 7 bytes in 1 blocks
==31485==    indirectly lost: 0 bytes in 0 blocks
==31485==      possibly lost: 0 bytes in 0 blocks
==31485==    still reachable: 0 bytes in 0 blocks
==31485==         suppressed: 0 bytes in 0 blocks
==31485==
==31485== For counts of detected and suppressed errors, rerun with: -v
==31485== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 1 from 1)
```

Figure 2: Output of error message using Valgrind

Clang AddressSanitizer

Both GCC and Clang/LLVM are compilers for the C programming language. Although we mainly introduced usage of GCC and provided makefile for you, it is up to you to decide which one you prefer to use.

An alternative to Valgrind is Clang AddressSanitizer, a memory error detector that could give warning on the following problems:

- Out-of-bounds accesses to heap, stack and globals;
- Use-after-free;
- Use-after-return (runtime flag ASAN_OPTIONS=detect_stack_use_after_return=1);
- Use-after-scope (clang flag -fsanitize-address-use-after-scope);
- Double-free, invalid free;

- Memory leaks (experimental).

To use it, simply compile and link your program *arrays.c* using:

clang -fsanitize=address arrays.c -o arrays

For more detailed information on the call stack when an error is detected, add *-fno-omit-frame-pointer* and *-O1* to your command:

clang -O1 -g -fsanitize=address -fno-omit-frame-pointer arrays.c -o arrays

Note: Compiling your program with *-fsanitize=address* conflicts with Valgrind as both are essentially trying to do more or less the same thing. If you wish to use valgrind, build your program normally.

Figure 3 displays the output of Clang AddressSanitizer detecting memory leak of the code shown in Figure 1.

```
$ clang -fsanitize=address -g memory-leak.c -o memory-leak; ASAN_OPTIONS=detect_leaks=1
    ./memory-leak

=================================================================
==4702==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 7 byte(s) in 1 object(s) allocated from:
    #0 0x465359 in __interceptor_malloc
    #1 0x47b619 in main memory-leak.c:4
    #2 0x7f0681631b14 in __libc_start_main (/lib64/libc.so.6+0x21b14)

SUMMARY: AddressSanitizer: 7 byte(s) leaked in 1 allocation(s).
```

Figure 3: Output of error message using clang sanitizer

For more information, please read documentation of Clang AddressSanitizer.
For all possible options to use, please check the tables of AddressSanitizerFlags and SanitizerCommonFlags.