CS2002 Debug Exercise Worksheet

Introduction

Debugging is the process of finding causes for run time errors in the code. Most compile time errors can be overcome with careful attention to language syntax and use. As programs grow large with multiple files, code editors do a good job of pointing to locations of potential errors in the code. Unfortunately, it is possible to write code that hides bugs until execution. What options exist, then?

Runtime Errors with Debugging Statements

Consider the following code that is in the file "main.c" on studres:

```c
/*
 * This code is for the debug exercise
 */

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

int sum(int x, int y, int *z) {
    char c = 4;
    return (x + y + *z);
}

int main(int argc, char *argv[]) {

    int i, j, k, *z;
    int result;

    *z = k;

    if (argc != 4) {
        printf("Please specify three numbers as parameters.\n");
        exit(1);
    }

    i = atoi(argv[1]);
    j = atoi(argv[2]);
    k = atoi(argv[3]);
    k *= 2;

    result = sum(i, j, z) + sum(i, j, &k);
    printf("%d\n", result);
    return 0;
}
```

The program has problems that can be found with closer inspection. However, debugging statements are often useful to narrow the search field. Try to compile and run the program above;

```
% gcc main.c
% ./a.out
```

At best there will appear an exit code of 1, at worst a bus error or seg-fault.

This begs the question: How do we find run-time errors?

Task 1. Modify the program to include debug (trace) statements that write to **`stderr`**. In `main()`,

```
fprintf(stderr, "Number of parameters = %i\n", argc);
```

or in appropriate locations in `sum()`,

```
fprintf(stderr, "x=%i\n", x);
fprintf(stderr, "y=%i\n", y);
fprintf(stderr, "z=%i\n", z);
fprintf(stderr, "*z=%i\n", *z);
```

Switching Debug Messages On/Off

A quick shortcut to turn debug messages on and off:

1. Beneath the `#includes`, type **`#define DEBUG`**
   (Alternatively avoid **`#define`** and use `-D` at command line, i.e. **gcc -DDEBUG …**)
2. Before each debugging statement, type **`#ifdef DEBUG`**
3. Make sure to write **`#endif`** after each (set of) debugging statement(s).

So, as an example:

```
…

#define DEBUG // Or omit and set flag as command line option (above).
…
#ifdef DEBUG
  fprintf(stderr, "x = %i\n", x);
  fprintf(stderr, "z = %p\n", z);
#endif
…
```

DEBUG statements can be turned on by setting or omitting the DEBUG flag. The '-D' option tells the pre-processor to `#define` a macro with the string that follows. So, for example,

```
% gcc –Wall -Wextra main.c  -DDEBUG
```

Using the Debugger

This method is independent from printing debugging statements. **Before continuing, comment out the '`#define DEBUG`' line,** or compile without -DDEBUG.

Debuggers are used to inspect program state during execution. This requires additional information. First**, make sure to add the `-g` option** (this is a compile-time option).

```
% gcc -Wall -Wextra -g main.c
```

To launch the executable “a.out” inside of the debugger,

```
% gdb ./a.out  //clang comes with ‘lldb’
```

Once inside,

```
(gdb) set args <args required by main if needed>
(gdb) run
or, equivalently
(gdb) run <args required by main>
```

to launch the program. When (and if) the program crashes, gdb will ‘trap’ the program in its current state and prevent the program from exiting. It will also indicate the location of the crash.

The list of commands that are immediately useful consist of:

```
(gdb) print <what>      // prints variable, memory location, or register
                        // note that “print *var” or “print myS->var” are also valid
(gdb) backtrace         // or ‘bt’ or ‘where’ – prints call stack
(gdb) bt full           // or “where full” – prints all local variables
(gdb) up                // moves up one frame on the execution stack (to caller)
(gdb) down              // moves down one frame to callee function
(gdb) q                 // quit.
```

Additional Notes

It’s rarely the case that the location of the crash is where the problem begins. One can set locations to halt execution and `step’ through, one instruction at a time.

Halting points consist of ‘breakpoints’ at a location in the source code (indicated by function name, filename, or line_no); or ‘watchpoints’ at locations where certain conditions are met, i.e. “x > 5”. Then,

```
(gdb) break <function _name>  // set breakpoint at specified function
(gdb) run                     // run program until any breakpoint is hit or failure/exit
(gdb) step                    // step forward to next instruction, go into functions
(gdb) next                    // next instruction, execute functions but don’t go in.
(gdb) continue                // proceed as normal again until next breakpoint
(gdb) help breakpoints        // show help on using breakpoints
```

There are *lots* of online examples, cheat sheets, and references and `man gdb`.

Try setting a break point at some function or even line within a given file function and try stepping (with ‘s’ or ‘step’) through program execution, examining variable values (using ‘p’ or ‘print’) for example.

If you are using **clang** and don’t want to switch to **gcc**, you should use **lldb.** You can see equivalent commands for **lldb** to those mentioned above at https://lldb.llvm.org/use/tutorial.html