COMP 40 Laboratory: A Self-Guided Tour of the Data Display Debugger

October 21, 2011

1 Introduction

Today's lab is meant to introduce you to DDD, the Data Display, Debugger.

- If you have not used DDD, Section 2 below will help you get familiar with it.
- If you are already familiar with DDD, you can move directly to Section 3 on page 5, which contains some small, focused exercises related to your binary bombs.
- Finally, whether you are familiar with DDD or not, you'll find a selection of little-known commands in Section 4 on page 7. These commands will be useful for defusing bombs.

2 A step-by-step guided experience with DDD

To get an example program, try

```
git clone /comp/40/git/ddd-demo
cd ddd-demo
sh compile
```

This should give you a binary `ppmtrans`. You'll see that the block-major rotation is buggy:

1. Run both of the following:

   ```
   ./ppmtrans -rotate 90 < rand105x104.pbm | pnmscale 5 | display &
   ./ppmtrans -rotate 90 -block-major < rand105x104.pbm | pnmscale 5 | display &
   ```

   (The program is also buggy in that it only accepts images from standard input.)

2. Before running the debugger, *always* check with `valgrind`:

   ```
   valgrind ./ppmtrans -block-major -rotate 90 < rand105x104.pbm > /dev/null
   ```

   In this case `valgrind` won't help us!

3. Start DDD:

   ```
   ddd --debugger "gdb -d /usr/sup/src/cii40" ppmtrans
   ```

4. Close the tip of the day

5. You'll see a window with three panes:

- A blank field where you can display data
- Source code
- A window for interacting with `gdb`

(If you've used DDD for other projects, it remembers the old layout, so you may see something different.)

6. Given the buggy output, I want to know if the big black areas are blocks. So I'm going to stop in `UArray2b_new` to see what happens. To set a breakpoint there, type `UArray2b_new` in the small input window following "`():`" and then hit the stop sign to the right.
7. Next choose "Run..." from the Program menu
8. Fill in the arguments

   ```
   -rotate 90 -block-major < rand105x104.pbm > /tmp/r.ppm
   ```

   and click the Run button.
9. Your source-code window should now show `UArray2b_new` with a red arrow and a stop sign. Double-click the `blockArray` variable.
10. A box should appear in the data window. It shows a pointer value. Because the variable is uninitialized, I can't predict the value.
11. Click the Next button four times. At this point
    - The data display should have changed.
    - The red arrow should be pointing to the declaration of `int bwidth`.
12. Let's display the `blockArray`. Right-click on the box in the display window, and from the menu select "Display *()." That means *follow the pointer*. A box displaying the struct should appear.
13. The display shows that `blocksize` is 73. Whatever is going on, that's significantly larger than the black blotches on the bad output.
14. Use the Next button to step down just before the `for` loop.
15. The display of the `blocks` field changes. Follow that pointer using the "Display *()" from the right-mouse menu.
16. The result says we have 2-by-2 blocks of size 8. If we try to follow the `rows` pointer, we can't see it—it's an abstract type.
17. Let's let `UArray2b_new` finish. Click the Finish button.
18. We were called from `UArray2b_new_64K_block`. Click Finish again.
19. Keep clicking Finish until you get to the assignment

    ```
    image = Pnm_ppmread(stdin, methods);
    ```

20. Further down the same pane you should be able to see some `if` statements that include the case

    ```
    if (rotation == 90)
    ```

    To be sure of stopping there, double-click the `rotation` variable.
21. You should get a data display showing `rotation` is 90. Let's stop at the assignment to `A2 temp`. Double-click in the left margin of that line; a stop sign (breakpoint) should appear.
22. Now let's continue execution until we reach the next breakpoint: click the Cont (continue) button.

23. We hit the breakpoint in `UArray2b_new` again, and it tries to bring back the old data display. It won't work until things are initialized. Single-step using the Next button until the struct appears and the `height`, `width`, and `blocksize` fields are displayed with their correct values.

    DDD remembers what data you were interested in, and the data is displayed on every trip through the same procedure.

24. So far all looks well, so let's continue to the next breakpoint by pressing the Cont button.

25. You're now just before the assignment to `temp`. Take *one* step by clicking the Next button once.

26. We suspect some problem in the `map` or `rotate90` functions. We're going to "step inside" these calls by using the Step button. Click it once.

27. We're in a method suite. "Step inside" the call to `UArray2b_map` by clicking Step again.

28. Double-click the `array2b` parameter to see its value.

29. Right-click the data display to see the fields of the struct.

30. Right-click the `blocks` field in the data display.

31. Using the Next button, single step down to the `for` loop.

32. Hover the mouse over the `bheight` variable. It should show "2." Do the same for the `bwidth` variable.

33. I see no obvious faults in the `map` function. Let's set a breakpoint just before the call to `apply`, by left-clicking in the left margin of that line.

34. Run to the breakpoint by clicking Cont.

35. Hover the mouse over `i`, `j`, `n`, and `temp`. All should have reasonable-looking values. (If a pointer value is roughly nearby other pointer values, it is definitely reasonable. Other kinds of pointers—to global or local variables—might also be reasonable.)

36. Let's "step inside" with the Step button. Keep stepping until you get to a static `height` method. That's not interesting, so let it Finish.

37. Step again; you should be inside the `rotate90` function. If you want you can look at the `src` variable.

    Step again, and you'll be inside a static `at` method. Step once more, and you'll be inside `UArray2b_at`, where you can check to see the parameters are sensible.

38. Finish `UArray2b_at`.

39. Finish the static `at` method.

40. With `rotate90` on the source-code display, use the bottom (`gdb`) pane and type

    ```
    print UArray2b_size(temp)
    ```

    It should confirm the size is 8 (the size of a pointer).

41. Again I see no obvious faults, so let's Finish the `rotate90` procedure.

42. We're back in the `map` function. Right-click the stop sign and choose "Disable breakpoint" from the menu. It should become grayed out.

43. Now let's let the `map` function Finish.

44. Also Finish the static `map_block_major` method.

45. Finish again and we're back in `main`. Let's see if we can find anything wring with `newImage`.

46. Enlarge the data pane by dragging the small square on the right-hand side of the line that separates the data pane from the source-code pane.

47. Double-click `image` in the source-code window.

48. Double-click `newImage` in the source-code window.
49. Follow pointers.
50. You can't double-click either of the `pixels` fields, because they have type `void*`. Instead, in the very top input line, next to `():`, edit in

    ```
    (UArray2b_T)image->pixels
    ```

    and click the "Display" flashlight button.
51. Do the same with

    ```
    (UArray2b_T)newImage->pixels
    ```

52. From the output, we know that pixel $(0, 1)$ is unexpectedly read. In the display window, type

    ```
    (Pnm_rgb)methods->at(newImage->pixels, 0, 1)
    ```

    and click the Display flashlight.
53. Something looks very wrong. What about

    ```
    (Pnm_rgb)methods->at(newImage->pixels, 1, 0)
    ```

    This looks bad.
54. Let's look at the original image. But the data pane is getting crowded. Instead try the `gdb` pane:

    ```
    print *(Pnm_rgb)methods->at(image->pixels, 0, 103)
    print *(Pnm_rgb)methods->at(image->pixels, 0, 104)
    ```

    Oops! my arithmetic is bad, but the `UArray2b` implementation let that reference go without a bounds check!
55. How about

    ```
    print *(Pnm_rgb)methods->at(image->pixels, 0, 102)
    ```

    All these pixels are black. Yet the pixel in the rotated image is not black!
56. Let's try the program again. Type `rotate90` into the input box at the very top, and click the Lookup button with the target.
57. Double-click the assignment to `dest` to get a breakpoint.
58. Right-click to get the Properties menu.
59. In the Condition box type

    ```
    i == 0 && j == 102
    ```

60. Click Apply. You should see a question mark added to the stop sign.
61. Click Close.
62. Let's re-run the program from the start. Click the Run button.
63. We hit a breakpoint in `UArray2b_new`. Disable it with the right mouse button and continue.
64. We hit breakpoint 2 assigning to `temp`. Disable it with the right mouse button and continue.
65. Finally we hit breakpoint 4.
66. Step inside twice to get to `UArray2b_height`.
67. Finish to get to the method suite.

68. Finish again to get back to the `rotate90` function.
69. Use the flashlight to display the `src` variable.
70. Follow the pointer. The original pixel is white. That is, red, blue, and green are all set to 1.
71. Hmm. Maybe we are not pointing at the pixel?
    Double-click `elem` to display a pointer to the pixel.
72. In the display window, type

    ```
    methods->at(a, i, j)
    ```

    and click the Display flashlight.
73. Aha! The `elem` pointer we are given does not actually point to coordinates (i, j).
74. Right-click the breakpoint in `rotate90`, and change the condition to

    ```
    elem != methods->at(a, i, j)
    ```

75. Restart the program, and we can see that it goes wrong almost right away, at (i, j) coordinate $(1, 0)$.

At this point, the best plan is to abandon the debugger and find out why the `map` and `at` functions are inconsistent. We'll find that the math in the *comment* for `map` is consistent with the *code* in `at`, but the comment and the code are inconsistent—the quotient and modulus have been swapped.

One final point: this fault could have been caught easily by placing the conditional assertion

```
if (i + j == 1)
  assert(UArray_at(*temp, n) == UArray2b_at(array2b, i, j));
```

in the loop. I bet this assertion would have cost practically nothing.

3 Reverse engineering your bomb using DDD

The principle behind today's lab is that when you're reverse engineering, if you are not certain of what you're seeing, *work the problem from both ends*. That is, create a short C program, compile it, and see if `objdump` explains the results you are seeing.

`sscanf` and pointers into the stack

If you are not sure of how `sscanf` points, or if you are not understanding address arithmetic related to the stack pointer, then this exercise is for you.

Write and compile a function that reads three unsigned long integers:

```
struct three_unsigned_longs {
  unsigned long a, b, c;
};

void read_three_unsigned_longs(const char *inputline,
                               struct three_unsigned_longs *p);
  /* uses sscanf to read three unsigned long integers from
     'input line', and places the results in the fields of the
     structure pointed to by p */
```

(My implementation of `read_three_unsigned_longs` takes just 3 lines, two of which are used to explode the bomb.) Examine the results with `objdump -d`. Look at your own code and also at the `main` function.

You can get the header file above, a test `main`, and a compile script by

```
git clone /comp/40/git/code
```

Put your function in file `tul.c` and compile with `compile-code-lab`.

`jmp *something` and the `switch` statement

If you see a `jmp` instruction with a star (`*`), you are seeing a *computed goto*, which most likely has to do with a `switch` statement. This stuff is covered pretty well by Bryant and O'Hallaron in Section 3.6.6. The "pidgin C" in Figure 3.15(b) may be helpful, but I would definitely check out the assembly code on line 4 of Figure 3.16, and you can see the jump table on the next page. Everything in your code is similar except that your machine uses 64 bits instead of 32 bits:

- `%eax` becomes `%rax`.
- a 32-bit pointer (4 bytes) becomes a 64-bit pointer (8 bytes), so the multiplier in the effective address becomes 8, not 4.

If you have an instruction like

```
jmpq *0x401b90(,%rax,8)
```

Then you can view this as indexing into an array of 64-bit pointers located at `a` = `0x401b90`, and you are computing `a[rax]`, which in address arithmetic translates to `a + 8 * %rax`.

It's possible to reconstruct the array of pointers by using

```
objdump -s -j .rodata bomb99
```

but *I recommend against this method*. It is far easier just to get the debugger to show you the target address; put

```
((void *)0x401b90)[$rax]
```

in the DDD window and click the flashlight.

If after consulting the book, you are still not sure what is going on, you might want to compile and objdump the file `aspect.c`:

```
#include <stdbool.h>

enum tx { NONE, ROT90, ROT180, ROT270, FLIPH, FLIPV, TRANS };

bool changes_aspect(enum tx transformation) {
  switch (transformation) {
    case NONE:   return false;
    case ROT90:  return true;
    case ROT180: return false;
    case ROT270: return true;
    case FLIPH:  return false;
    case FLIPV:  return false;
    case TRANS:  return true;
  }
  return false;
}
```

Push, pop, and computation on the stack

In Monday's class we'll talk a bit about the call stack and how it works, but for purposes of analyzing stack code, your best bet is to simulate the execution of the code, keeping track of the state of the stack and the registers. A good example program would be the "three unsigned longs" code in the exercise above. Here's what's useful to know:

- The stack grows downward, so younger activations are at smaller addresses.
- Just before a call, the stack pointer `%rsp` must be a multiple of 16 (aligned on a 16-byte boundary).
- The call instruction pushes a pointer to the return address, so just after a call—*and therefore at the beginning of every procedure*—the stack pointer points to the return address, and it is equal to 8 modulo 16. If a function therefore needs to make another call, it has to restore the invariant that the stack pointer before a call is a multiple of 16. This requirement is the source of the mysterious code in some functions that subtracts 8 from the stack pointer without actually using any space on the stack.
- Any `push` instruction subtracts from the stack pointer and writes a word to the stack, all at one go. A `push` instruction is often used to save the value of a register; the value can be restored with a corresponding `pop` instruction.

  Why? Certain registers are *preserved across calls*. A function is allowed to use those registers only if it guarantees to save their values on entry and restore them on exit. The ones you are most likely to see are `%rbx` and `%rbp`; your overview handout has the complete list.
- Keep in mind that although *the stack pointer may move*, the values on the stack do not move. Your best bet is to draw a picture of the stack and to give a name to each location you find there. You can then "play computer" and execute the code by hand, using the names of the locations to see what the code is doing.

4 Commands and menu entries for machine-level debugging

DDD is designed for debugging source code—by continuously displaying the contents of my *data structures*, it really improves my debugging productivity. But at the machine level, there are no stinkin' data structures—just sequences of stuff in memory. Here are some little-used tricks that will help:

- Selecting Machine Code from the View menu gives you a view of the assembly code.
- To single-step by machine instructions, use the Stepi and Nexti commands. The Finish command is also useful.
- The Backtrace command on the Status menu will tell you where you are. It's quite useful for general debugging, but for the binary bomb, you may not need it.
- The Memory command on the Data menu is great for looking at arrays. It can also look at structs if the members of the struct are mostly the same size.
- The Watchpoints command on the Data menu can stop the program when the contents of a memory location change.
- You can view all the registers using the Registers command from the Status menu. But on a modern CPU this is a bit overwhelming. You're better off asking for individual registers in the display window.

  For reasons only Richard Stallman knows, the registers are named using dollar signs, not percent signs. For example, the `%rdi` register is called `$rdi`.
- For best view of a register, you should cast it in a C expression. For example, suppose you think `%rdi` holds a pointer to a string. Then you display `*(char **)$rdi`. Or suppose you think it holds a pointer to a pixel. Then you would display `*(Pnm_rgb)$rdi`.