Worksheet 8: Debugging and Testing

Updated: 28th July, 2015

The objectives of this practical are to gain a theoretical understanding and practical experience in debugging, and to briefly explore unit testing.

Pre-lab Exercises

1. Gdb Commands

Consider the following function, taken from a library manager system:

```
Book *deleteBook(Catalogue *cat, int isbn) {
    Book *book = NULL;
    int i = 0;

    while(i < cat->numBooks && cat->books[i]->isbn != isbn) {
        i++;
    }

    if(i < cat->numBooks) {
        book = cat->books[i];
        cat->numBooks--;

        while(i < cat->numBooks) {
            cat->books[i] = cat->books[i + 1];
            i++;
        }

        cat->books = (Book**)realloc(
            cat->books, cat->numBooks * sizeof(Book*));
    }

    return book;
}
```

Assume that:

- `cat` points to a struct containing a catalogue of 100 books.
- `isbn` matches the ISBN of a book within that catalogue.

What effects will the following gdb commands have, if executed one after another:

(a) `break deleteBook`

(b) `run` (Assume that the breakpoint is hit.)

(c) `print cat`

(d) `print *cat`

(e) `watch book == cat->books[i]`

(f) `continue`

(g) `next`

(h) `next`

(i) `finish`

2. Debugging Tactics

Consider the next function, taken from the same system. It takes pointers to structs representing (a) a catalogue of books, and (b) a list of borrowers.

The function asks the librarian for a borrower ID and an ISBN. If all goes to plan, the book is recorded as no longer being on loan.

```
static void menuReturnBook(Catalogue *cat, BorrowerList *blist) {
    Borrower *borrower;
    Book *book;
    int id, isbn

    id = readInt("Enter borrower ID: ");
    borrower = getBorrower(blist, id);

    if(borrower == NULL) {
        printf("No borrower with that ID was found.\n");
    }
    else {
        isbn = readInt("Enter ISBN of book to return: ")
        book = getBook(cat, isbn);

        if(book == NULL) {
            printf("No book with that ISBN was found.\n");
        }
        else {
            if(returnBook(borrower, book)) {
                printf("Book returned.\n");
            }
            else {
                printf("Book is not on loan to that borrower!\n");
            }
        }
    }
}
```

As you can see, this function relies on four others: `getBorrower()`, `readInt()`, `getBook()` and `returnBook()`.

For each of the following hypothetical situations:

- Suggest two plausible hypotheses — why *could* it be happening? What possible fault/defect in the *other* functions might explain the observations?
- Explain how you would use debugger features to decide between your two hypotheses. Which one is correct? (Or, at least, which one is possible, after ruling out the other?)

  ("Debugger features" includes all gdb functions, but in particular breakpoints and monitoring of variables.)

(a) A segmentation fault occurs immediately after the user enters a borrower ID. (You know the location through valgrind.)

(b) A segmentation fault occurs immediately after the user enters an ISBN.

(c) The message "`Book returned`" is displayed, but the book appears to remain "on loan" (when queried elsewhere).

(d) The message "`No book with that ISBN was found`" is displayed, even though you know it exists in the catalogue.

3. Unit Testing

Outline a unit testing approach for the code from Question 2. In particular:

(a) How would you achieve automation and avoid user input? (Hint: think back to the IO lecture.)

(b) How many unit test functions would you need?

(c) At a minimum, how many different test cases (i.e. sets of input) would you need to test the `menuReturnBook()` function?

Practical Exercises

Obtain a copy of `librarymanager.zip`. This contains the source code and data files for a basic library manager system. Extract all the files into a separate directory. You should see several .c and .h files along with a Makefile and two data files: `cat.txt` and `blist.txt`.

The "Library Manager" has a text-based menu system for keeping track of a catalogue of books and a list of borrowers. It provides facilities to:

- Load and save catalogues and borrower lists.
- Add, delete and list books and borrowers.
- Loan and return books to/from borrowers.

You don't have to write any of this code yourself! Instead, in this practical you'll be *debugging* this system.

1. Debugging Walkthrough

Note: Make sure you follow the *reasoning* here, not just the features of `gdb`. You'll need to stand on your own feet later and make your own debugging decisions!

(a) Compile the library manager with `make`, and run it without parameters:

```
[user@pc]$ ./librarymanager
```

You should see a text-based menu. Type "`c`" to go to the catalogue sub-menu. From there, load a catalogue with "`l`". Type in "`cat.txt`". Go back to the main menu with "`q`".

Now, use the menu to list the books ("k"). Take note of what happens. Have a look at the data file `cat.txt`. It contains three entries, each taking up three lines.

(b) We'll now step through the process you might use to diagnose this error. First, we'll need `gdb`:

```
[user@pc]$ gdb ./libarymanager
```

```
(gdb) run
```

Note: This worksheet will give instructions on the use of `gdb`. You can use `ddd` instead if you choose — it has all the same capabilities.

Perform the same actions as before (loading the catalogue and listing the books). Gdb will give you a somewhat more useful error message.

In particular, gdb tells us exactly what code triggered the segmentation fault occurs. Based on your understanding of worksheet 6, suggest what kind of structure is being accessed here. How many pointers are involved? (Remember: a segmentation fault occurs when we access an invalid pointer.)

(c) We can use gdb to see what's going on. First, let's see the surrounding code:

```
(gdb) list
```

You should see that we're inside a `for` loop. We also know that the first book was correctly printed out. We can confirm this by printing the for loop index `i`:

```
(gdb) print i
```

```
$1 = 1
```

This means that `i == 1` (ignore the `$1`). We're trying to access the *second* array element.

Since the array contains pointers, this is probably the source of the problem. Consider what happens when we dereference the first element:

```
(gdb) print *(cat->books[0])
```

. . . compared to the second (and third):

```
(gdb) print *(cat->books[1])
```

```
(gdb) print *(cat->books[2])
```

(d) So, the question becomes: why does the array only contain one book, when it should contain three? We need to look elsewhere for the answer. So far, we've only looked at the code that *triggers* the problem.

The catalogue loading code is a reasonable place to look, but we don't know where this is. To find it, you could read through all the code, or you could use the debugger. Set a breakpoint at the `catalogueMenu()` function:

```
(gdb) break menusystem.c:catalogueMenu
```

Note: How did we know to set a breakpoint in the `catalogueMenu()` function? An educated guess. You have to start somewhere along the execution path. You could have set a breakpoint in `main()` and stepped through the code from the very beginning, but that might be more time consuming.

Alternatively, you could run the program inside gdb and press Ctrl-C at the right time. Gdb would then pause the program, whatever line happened to be executing.

Now, restart the program:

```
(gdb) run
```

```
Start it from the beginning? (y or n) y
```

If you try to load a catalogue this time, the breakpoint will be hit:

```
Breakpoint 3, catalogueMenu (cat=0x804d008) at
menusystem.c:111
111         int quit = FALSE;
```

We don't care about the preliminaries of this function, only what happens *after* you select an option. Step through the function until then:

```
(gdb) next
```

```
(gdb) next
```

etc.

```
(gdb) next
```

```
Enter an option:
```

Enter "`l`" to load the catalogue as before, then step through again until you reach this line:

```
127                     menuLoadCatalogue(cat);
```

We want to step *into* this function call, because it will take us to the catalogue loading code.

```
(gdb) step
```

Now you should see:

```
menuLoadCatalogue (cat=0x804d008) at menusystem.c:260
256         readLine("Enter name of catalogue file to load:
",
```

List the code again to see where you are (with `list`). You can also open up `menusystem.c` in an editor and scroll to the given line number.

(e) It's clear that this isn't the catalogue loading code — just the UI interface. However, there's a call to `loadCatalogue()` in there. Use `next` to step over the code until you get to the `loadCatalogue()` call, then use `step` to step into it.

You should see that `loadCatalogue()` is located in `catalogue.c`:

```
loadCatalogue (cat=..., filename=... "cat.txt") at
catalogue.c:18 18           int success = FALSE;
```

Open `catalogue.c` in an editor. You should see that `loadCatalogue()` does the actual file IO.

We'll assume that the `do-while` loop is responsible for reading each book record in turn. Obviously the first one is ok, so we want to know what happens on the second iteration, when `i == 1`. Set a watchpoint for this condition:

```
(gdb) watch i == 1
```

Then continue execution (without stepping):

```
(gdb) continue
```

When the watchpoint triggers, it should look like this:

```
Continuing.
Hardware watchpoint 4: i == 1

Old value = 0
New value = 1
loadCatalogue (cat=..., filename=... "cat.txt") at
catalogue.c:56
```

```
56              while(!eof);
```

We're at the end of the `do-while` loop, right after the statement `i++` (as expected). Now we can step through the code to see what happens on the second iteration of the loop.

```
(gdb) next
```

However, we now find ourselves outside the loop! Logically, this means that the `eof` variable must have been `TRUE` (non-zero). Just to check:

```
(gdb) print eof
```

```
$5 = 1
```

This explains the segmentation fault. Only the first array element is being assigned to. The second and third are never initialised.

(f) But why is `eof == 1`? We've only been around the loop once. We can't have executed the statement `eof = TRUE;`, because then `i` would never get to 1.

We can verify this as follows. First, delete all existing breakpoints and watchpoints:

```
(gdb) delete
Delete all breakpoints? (y or n) y
```

Then, put a breakpoint at `eof = TRUE;` (lines 45) and another after the end of the loop (line 54):

```
(gdb) break catalogue.c:45
(gdb) break catalogue.c:54
```

Finally, restart the program:

```
(gdb) run
```

You should notice that the first breakpoint is never hit. Given this, and by looking at the code, you should be able to identify the root cause of the segmentation fault.

2. On Your Own

There is another problem that occurs when trying to add a borrower.

Each borrower has a name and an automatically-generated ID number. When listing borrowers, both should be displayed.

(a) Run the program and identify the symptoms of the problem.

(b) Suggest hypotheses for roughly *where* the problem might occur.

(c) Familiarise yourself with the structures used to record borrower information (in `borrowerlist.h` and `borrower.h`).

> **Note:** `BorrowerList` contains an *array of pointers* to `Borrower` structs.

(d) Starting with the menu system in `menusystem.c`, use debugger features to determine which of these hypotheses is true. Use breakpoints, watchpoints and stepping as needed, and examine the values of key variables, fields, etc. to trace the problem back to its source.

> **Notes:** Remember to keep asking "*why*" until you identify the root cause of the problem.
>
> Don't make assumptions about variables and values! When using a debugger, you can *see* what values a variable has — there's no need to guess.
>
> Don't trawl through the entire source code! You'll need to read and understand *parts* of it as you go. You may also have to consult the documentation (man page) for certain standard C functions.

End of Worksheet