Worksheet 5: Debugging and Testing

Updated: 3rd September, 2018

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:

```
int deleteBook(int *books, int *numBooks, int isbn) {
    int book = 0;
    int i = 0;

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

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

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

        books = (int*)realloc(books, numBooks * sizeof(int));
    }

    return book;
}
```

Assume that:

- `cat` points to an array containing the isbn's 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 books`

(d) `print *books`

(e) `watch book == 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 am array representing (a) the isbn of all the library books, and (b) an array of borrower id's.

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(int *isbnList, int *blist) {
    char *borrower;
    char *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(isbnList, 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 `spellChecker.zip`. This contains the source code and data files for a basic spell checker 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: `small_dictionary.txt` and `test.txt`.

The details for how the spell checker works can be found in the included README.

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

Warning: For this practical you are not required to understand any of the file I/O. None of the bugs that have been introduced to this code are related to file I/O. ie: `input.c` and `output.c` are working and bug free.

(Providing code with no file I/O would boarder on trivial.)

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 spell checker with make, and run it with the following parameters:

```
[user@pc]$ ./check test.txt result.txt
```

You should see following output:

```
Reading settings file ("spellrc")... Done
Reading "small_dictionary.txt"... Done
Segmentation fault (core dumped)
```

(b) This program works with very little user output. So we will use gbd to step through the processing and isolate the error.

We'll now walk through the process you might use to diagnose this error. First, we'll need gdb:

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

```
(gdb) run test.txt result.txt
```

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.

Gdb should give you a somewhat more meaningful error message. You can use the command backtrace to see more than just the current stack frame (essentially the call tree).

In particular, gdb tells us exactly what code triggered the segmentation fault occurs. Based on your understanding of pointers, what could have triggered the segmentation fault?(Remember: a segmentation fault occurs when we access an invalid pointer.)

Remember: input.c is working and bug free.

(c) Lets set a break point for immediately before the function call. This will allow us to check all the parameters being passed to the function.

As we have already passed this point of execution we will need to start again, but first set a break point for line 126 in spellChecker.c:

```
(gdb) break spellChecker.c:126
```

Then use `run` to start execution of the program again.

```
(gdb) run test.txt result.txt
```

Execution of the program will now pause immediately before the function call. As we have identified that `inputFile` is the pointer of interest, lets check its value.

```
(gdb) print inputFile
```

```
$1 = (FILE *) 0x0
```

Memory address 0x0 means the pointer has been set to NULL.

(d) The question is now, how did `inputFile` become NULL.

Lets set a watch point for when `inputFile` was set to NULL. Depending on the versoin of gdb you have, this may cause problems, so we will do it the long winded way. we will also use 0 instead of NULL as gdb does not inherently understand NULL.

```
(gdb) break main
```

```
(gdb) run test.txt result.txt
```

```
(gdb) watch inputFile == 0
```

Once the watch point is set you can continue execution of the program

```
(gdb) continue
```

```
Hardware watchpoint 4: inputFile == 0

Old value = 1
New value = 0
main (argc=3, argv=0x7fffffffdfa8) at spellChecker.c:105
105                     if(inputFile = NULL)
```

The watch point should have been triggered at line 105. If you look closely at this line, you should be able to identify the issue.

Once identified, fix the issue and continue with the next exercise.

2. On Your Own

There are two more problems in the code. You may need to use valgrind in order for the memory issues to become apparent. For each issue make sure you:

- run the program/valgrind and identify the symptoms of the program.
- Suggest hypotheses for roughly *where* the problem might occur
- Use gdb to test your hypotheses.

- Use gdb to identify the cause, and then fix the problem.

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