UNIX and C Programming (COMP1000)

Lecture 5: Debugging and Testing

Updated: 19th August, 2018

Department of Computing
Curtin University

Textbook Reading

For more information, see the weekly reading list on Blackboard.

- Hanly and Koffman do not cover this material in detail. There are two brief sections:
  - **Section 5.10: How to Debug and Test Programs**
  - **Section 6.7: Debugging and Testing a Program System**
- Optional alternative – Kockan (2005), *Programming in C: A Complete Introduction to the C Programming Language*:
  - **Chapter 18: Debugging Programs**
    (This book is used in Engineering Programming).

Outline

Debugging Principles (Overview)

- Test, test, test!
- Make sure you know when something goes wrong.
- Identify the symptoms.
- Guess what might have caused it.
- Identify the code where it happened.
- Determine why it happened.
- Determine why it *really* happened (the "root cause").

Logic Errors

- Your program may compile fine — that doesn't mean it "works"!
- Many faults cannot be found by the compiler.
- Valgrind may not find them either (and if it does, it may still require debugging).
- There are an infinite variety of logic errors:
  - Off-by-one errors;
  - Misplaced semicolons (particularly in loops);
  - Putting parameters in the wrong order;
  - Overflowing an array;
  - etc.

Know Your Bugs

- *Nobody* writes code without making mistakes.
- You must also be able to fix them.
- Before you can fix them, you must find them.
- This is what debugging is all about.

Symptoms / Observations

- Take note of exactly what happens and what doesn't happen.
- Is there a segmentation fault?
- Does the program output an integer instead of a string?
- Does it "forget" to print something out?
- Does it output large, seemingly-random numbers?

Hypotheses

- When you identify a problem, it's useful to guess what the cause might be.
- Start with the symptoms, and consider *possible* causes.
- For instance, if you get a segmentation fault:
  - Maybe you forgot an "&" when using `scanf()` or `fscanf()`.
  - Maybe you're dereferencing an uninitialised pointer.
  - Maybe you're trying to read past the end of an array.
- Use your knowledge and experience (hard at first, but easier as time goes on).
- Rule out as many possible causes as you can — allows you to narrow down the problem.

Isolating the Problem

- Identify *where* the problem first appears.
- Where (in the code) do the first symptoms occur?
- If it's a memory error, valgrind can tell you the line number (if you compile with -g).
- Otherwise:
  - Insert lots of temporary `printf()` statements, to see which code gets executed;
  - OR, **use a debugger**, as discussed later (easier and less messy).

Error States

- Once you know *where*, you must find out *why*.
- Logic errors often happen when a variable gets the wrong value.
- For example:
  - A `char*` variable might be `NULL` instead of a valid pointer.
  - An `int` might be uninitialised, instead of zero.
  - An array index might be larger than the array.
- You have to know:
  1. What values your variables *should* have.
  2. What values they *actually do* have.

  - If you can't work out (1), you don't know what you're doing!
  - Debuggers can help with (2).

Don't Treat the Symptoms!

- You know what line triggers the problem, and why.
- You know that a certain variable is zero when it should be ten.
- What should you do?
- **Certainly not this**:

```
var = 10;          /* No jedi are you, mmmm no. */
```

- You're not done with the "why" yet.
- You need to know *why* the variable has the wrong value.
- Otherwise, you're treating the symptom, not the problem!

The Root Cause

- **You must keep asking “why” until you reach the actual fault.**
- This is the “root cause” of the problem.
- This is what you must actually fix.
- It may be in a different function.
- It may be in a different file.
- Data gets passed all over the program, and faulty data get passed in the same way.

Backtracing

- Keep going backwards in your code to find the root cause.
- What if a function returns the wrong value?
  $\Rightarrow$ Go to that function and find out why.
- What if a function parameter has the wrong value?
  $\Rightarrow$ Go to the function *call* to see why.
- This process can be slow and tedious, but is made easier by debuggers.

Debugging Example #1

Imagine...

- We have a program that:
  - Reads an int from the user (N).
  - Allocates an array of that size (with `malloc()`).
  - Fills the array with the numbers 1 to N.
  - Finds the sum.
  - Prints out the sum.
- If the user enters 1, the program should print 1.
- If the user enters 2, the program should print 3.
- If the user enters 3, the program should print 6.
- etc.

Debugging Example #1 — Symptoms and Hypotheses

- However, no matter what the user enters, the program *actually* prints a large, seemingly random number (e.g. 482674854).
- This is a symptom of an underlying defect.
- We can make a few guesses. Since we're summing an array:
  - Maybe we're not reading user input properly.
  - Maybe we're not initialising the array properly.
  - Maybe we're not initialising the "sum" variable.
- We can discount one possibility:
  - The `malloc()` is probably ok — we would expect a segmentation fault if it wasn't.

Debugging Example #1 — Closer Examination

Starting with our hypotheses:

- Maybe we're not reading user input properly.
  - Print out the value read.
- Maybe we're not initialising the array properly.
  - Assume the user input is correct.
  - Print out the array after it's been filled.
- Maybe we're not initialising the "sum" variable.
  - Assume the array is correct.
  - Print out the sum variable as we're adding up.

Debuggers

- A debugger is a tool that helps you debug your software.
- It lets you poke around inside a running program.
- It won't "figure out" the problem for you — you still have to do that yourself.

What's available?

- gdb — the "GNU Debugger" (text based).
- ddd — the "Data Display Debugger" (graphical, but based on gdb).
- Any good IDE (Integrated Development Environment) will have a debugger built in:
  - NetBeans
  - Eclipse
  - Visual Studio
  - etc.
- (Note: jdb is the Java equivalent of gdb.)

Debugging Information

- To make proper use of a debugger, your executable file should include debugging information.
- This is done during compilation.
- For gcc, use the -g flag.

```
[user@pc]$ gcc -g -c source_file.c
```

- In a makefile, you can add -g to your CFLAGS variable:

```
CFLAGS = -Wall -pedantic -ansi -g
...
gcc $(CFLAGS) -c source_file.c
```

- Debugging information describes how the machine code maps to the source code.
- Without it, you'll have to learn machine code!

Breakpoints

- You can tell the debugger (in advance) to pause your program at a given line.
- This is called a "breakpoint".
- Your program will execute normally, until it hits a breakpoint.
- Then the program will pause.
- At this point, you can use other debugger features to find out what's going on.
- You can then tell the debugger to "continue".
- You can add and remove breakpoints:
  - Before the program is run, AND
  - While the program is paused.

Stepping

- You can tell the debugger to execute the program one line ("step") at a time.
- You can see exactly how the code behaves:
  - Which lines are executed.
  - How variables are modified.
- If you reach a function call, you have two options:
  - Stepping "over" will run an entire function call in one step, pausing after the function.
  - Stepping "into" will run the function line-by-line, pausing after each line.

Monitoring Variables

- Debuggers make it easy to detect an error state (a variable having the wrong value).
- You don't need any extra `printf()` statements.
- The debugger can show you the values of any variable or array (when the program is paused).
- As you step through the code, keep an eye on key variables.
- Make sure you know what they *should* be.

Watchpoints

- A "watchpoint" is like a breakpoint — it causes you program to pause.
- However, you supply an *expression* instead of a line number.
- Whenever the expression is true, the program will pause (no matter what line is executing).
- Use watchpoints to:
  - Find out *where* a variable gets a certain value.
  - Find out what happens after this occurs.
- Note: you can only set watchpoints when the program is paused!

Conditional Breakpoints

- Breakpoint + watchpoint = “conditional breakpoint”.
- A conditional breakpoint will pause when:
  - A given line of code is reached, AND
  - A given expression is true.
- You could use it to pause the program on the 1000’th iteration of a `for` loop.

Post Mortem Debugging

- If your program ends abruptly (e.g. due to seg fault), you can still access debugging information.
- You can see variables as they were at the time of the crash.
- They can tell you why the crash occurred.
- (At this point, you can't step through the code any more).

Using Gdb

- To use gdb (the GNU Debugger), type:

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

  - `program` is the name of your executable.
  - No command-line parameters here (see next slide).
- Gdb will show some copyright, version and disclaimer notices.
- If gdb says "`(no debugging symbols found)`", you forgot to compile with `-g`.
- Gdb will then give you this prompt:

```
(gdb)
```

- This is where you type debugging commands.
- (Note: all of the following commands can be abbreviated!)

Gdb Commands — Breakpoints and Running

Setting Breakpoints

To place a breakpoint at the start of function `doStuff()`:

```
(gdb) break doStuff
```

If the function is in a different file:

```
(gdb) break otherfile.c:doStuff
```

You can also give a line number instead of a function name.

Running Your Program

```
(gdb) run
```

If your program needs command-line parameters:

```
(gdb) run param1 param2 ...
```

Gdb Commands — Stepping

Stepping

To step to the next line (and *into* functions):

```
(gdb) step
```

To step to the next line (and *over* functions):

```
(gdb) next
```

Repeating the Last Command

You can repeat the last command by just hitting enter:

```
(gdb)
```

This is useful when stepping, because you do it a lot.

Gdb Commands — Variables and Expressions

Monitoring Variables

To display the value of the variable `var`:

```
(gdb) print var
```

- This works even if `var` is an entire array.
- You can print any valid C expression.

Setting Watchpoints

This can't be done until the program is running and paused.

```
(gdb) watch var == 10
```

Gdb Commands — Listing Code

- You need to see the source code itself.
- Gdb lets you list code in 10 line chunks.
- You can list the code around a given line:

```
(gdb) list 13
```

- You can list the start of a function:

```
(gdb) list doStuff
```

- To list the *next* 10 lines:

```
(gdb) list
```

- To list the *previous* 10 lines:

```
(gdb) list -
```

Gdb Commands — Other Commands

Continuing

To continue normal execution, after hitting a breakpoint:

```
(gdb) continue
```

Quitting Gdb

```
(gdb) quit
```

Miscellaneous Commands

`bt`	Display a "backtrace".
`info breakpoints`	Lists all breakpoints.
`delete` *n*	Remove breakpoint *n*.
`finish`	Step over the rest of the current function.

DDD

- To use ddd (the “Data Display Debugger”):

```
[user@pc]$ ddd ./program
```

- ddd is graphical, and (possibly) easier to use than gdb.
- However, all the same debugging principles apply.
- Use the menu to “run”, “step”, etc.
- Double-click on a variable to display it.

Debugging Example #2

- The following slide shows a short piece of code, which:
  - Reads an `int` from the user.
  - Fills an array with that value.
  - Prints out the array.
- However, there's a problem:
- If the user enters a number less than 10, the program freezes!
- (Any number ≥ 10 seems to work fine.)

Debugging Example #2 — The Code

```
#define LENGTH 10

int main(void) {
    int array[LENGTH];
    int i, number;
    number = readInt(); /* Assume readInt works */

    for(i = 0; i <= LENGTH; i++) {
        array[i] = number;
    }

    printArray(array, LENGTH); /* Assume this works */
    return 0;
}
```

Debugging Example #2 — Breakpoints

- How can we debug this?
- We must determine what part of the code is freezing.
- We set two breakpoints:
  1. Right before the `for` loop (but after the `readInt()` call).
  2. Right after the `for` loop.
- Then we type "run" in gdb:
  - We hit the 1st breakpoint every time.
  - We only hit the 2nd breakpoint when `number >= 10`.
- Therefore, it's the for loop.

Debugging Example #2 — Watchpoints

- It looks like we have an infinite loop.
- But how can this loop be infinite?

```
for(i = 0; i <= LENGTH; i++) {
    array[i] = number;
}
```

- The `i` variable is incremented each iteration.
- We know that `LENGTH` is 10 — it's a constant.
- When `i > 10`, the loop should end.
- So, we set a watchpoint to check for that expression.
- The watchpoint is never hit!
- For some reason, `i` is never greater than 10.

Debugging Example #2 — Stepping

- We need to find out exactly what the `for` loop is doing.
- Starting at the first breakpoint, we:
  - "`step`" through the loop,
  - use "`print i`" to test the value of `i` at each iteration.
- `i` seems to increment ok.
- However, when `i == 10`, something strange happens.
- The statement "`array[i] = number;`" causes `i` to jump backwards.

Debugging Example #2 — Solution

- Eventually, we realise that we're overflowing the array.
- The array has 10 elements (0–9).
- `array[10]` is outside the array.
- On a hunch, we do the following:

```
(gdb) print &i
```

```
(gdb) print &array[10]
```

- . . . and we discover that the memory addresses are the same!
- In other words, `array[10]` is actually `i`.
- We overwrote the `for` loop index, creating an infinite loop.
- (Our mistake was using "`i <= LENGTH`" instead of "`i < LENGTH`".)

Testing

- Without testing, you have no idea what might be wrong with your code.
- Before (and after) you debug, you must test.
- Testing should not be an afterthought!
- The idea is to *break* your code.
- As a tester, you should be as pedantic, vicious and perverse as possible.
- You must also be systematic.
- You have to *want* your code to fail.
- Then you can fix it.

Automation and Repeatability

- As much as possible, testing should be *automated*.
- Automation leads to *repeatability*.
- If you repeat a test, you should be *guaranteed* the same outcome.
- If you know exactly what tests your program passes, then you (and your users) can have confidence using it.
- That's why you build a test harness.

Automated Test Harnesses

- Your test harness should call your functions to check whether they produce the right answers.
- Your test harness *should not* read anything from the user.
- User input may change from one test to another.
- So, the test outcome (pass or fail) may also change.
- Some input may trigger a fault, while other input does not.
- If your test harness is automated, then the tests it performs are (probably) repeatable.

Unit Testing

- "Unit testing" tests small parts of your code at a time.
- It's much easier to test small things.
- A function is the smallest piece of code you can realistically test.
- Each unit test should be placed inside its own "`test...()`" function.
- For instance, when testing the `calc()` function, call your unit test "`testCalc()`".

Writing Unit Tests

- Select (manually) a few sample inputs.
- Determine (manually) what the function should export.
- Hard code these into the test function.
- Pass each sample input into the function to be tested.
- For each function call:
  - Record the actual answer.
  - Compare the actual answer to the expected answer.
  - Output a message if they differ (i.e. if the test fails).

Unit Testing — Example

A unit test for the `factorial()` function:

```
void testFactorial(void) {
    int inputs[] = {0, 1, 10};
    int expected[] = {1, 1, 3628800};
    int i, actual;

    for(i = 0; i < 3; i++) {
        actual = factorial(inputs[i]);
        if(expected[i] != actual) {
            printf("FAIL: factorial(%d) == %d",
                inputs[i], actual);
        }
    }
}
```

Test Suites

- A test suite is a collection of related tests.
- For example:

```
void testSuiteMath(void) {
    testFactorial();
    testFibonacci();
    testQuantumChromodynamics();
}
```

- Test suites can incorporate other test suites (hierarchically).
- Your test harness (`main()`) should run all test suites.

Regression Testing

- A “regression” happens when a defect is introduced into a previously-working program.
- Obviously, we’d like to stop this from happening.
- This is easy, if you have a comprehensive set of unit tests.
- Simply run your (automated) test harness after a modification.
- Your existing test code can tell you if there’s been a regression.
- This is why you write unit tests!
- (This isn’t fool-proof, but it’s a very good idea.)

Assertions

- Many languages provide an "assert" statement.
- In C, you can write:

```
#include <assert.h>
...
assert(x == y);
```

- assert is a macro that verifies a boolean condition.
- If the condition is TRUE, nothing happens.
- If the condition is FALSE, the program immediately aborts with a message like:

```
prog: prog.c:8: func: Assertion 'x == y' failed.
Aborted
```

- This gives you the source file, line number and condition.

How to Use Assertions

- `assert`s can remain in your code permanently.
- They trap errors, like a "passive" kind of testing.
- They also help document your code.
- Only use `assert` to check for **logic errors** (**not** I/O errors, user errors, etc.)

Disabling Assertions

- Assertions may degrade performance a bit.
- After your code is thoroughly debugged, you can disable them at compile-time:

```
[user@pc]$ gcc -D NDEBUG=1 ...
```

- Leave the `assert`s in your code. When disabled, they simply do nothing.

Assertions — Bad Example

```
int readPositive(void) {
    int input;
    printf("Enter a positive integer: ");
    scanf("%d", &input);
    assert(input > 0); /* Part of the algorithm. */
    return input;
}
```

- Here, the whole program will abort if the input is invalid.
- This is *not* user friendly!
- Never use asserts to check for user errors, I/O errors, etc.
- Never use asserts as part of your algorithm.

Assertions — Good Example

```c
int readPositive(void) {
    int input;
    do {
        printf("Enter a positive integer: ");
        scanf("%d", &input);
    while(input <= 0);
    assert(input > 0); /* Sanity check. */
    return input;
}
```

- Here, the assertion *should* always be `TRUE`.
- If it's `FALSE`, we have a **logic error**.
- That's a *good reason* to abort the whole program.