
Debugging programs

1. You design your program
2. You implement your program
3. You take the typographical errors out
4. Help! It doesn't work!
 - You have to debug your program
 - (– term stems allegedly from 1945 machines: warm, lots of light, lots of bugs causing short-cuts, but used in 19th century already)
 - Requires good detective skills.
 - Look for clues. Look for a motive. Prove that you are right. Remove the bug.

Debugging...

The thing not to do:

- Change something and hope the bug will disappear (this will take you around 4,000,000,000 years)

Instead

- Try and classify the error
 - What kind of errors exist? What do we look for?
- Try non intrusive debugging
 - Hypothesise about the error, and find it.
- Try intrusive debugging
 - Change your program so to hunt it down; for when the previous fails.
- Use tools
 - Debuggers (misnomer: a debugger does *not* debug programs!)

Classification - I

- Design errors, the program is right, but it implements the wrong problem.
- Errors in the code:
 - Your code.
 - * Typos, eg, `for(i=0 ; 1<8 ; i++)` or `a[i][j]` instead of `a[j][i]`.
 - On the interface between yours and someone else's code
 - * `strcmp("Hello", b)` instead of `strcmp(b, "Hello")`
 - Someone else's code
 - * Includes operating system and compiler. Depends on the OS/compiler.
 - Hardware platform (Pentium division)

Classification - II

Reproducibility

- Subsequent runs of a program behave identical: Reproducible.
 - Reproducible bugs can be tracked down over multiple runs.
 - Non-Reproducible bugs cannot! (it may hide without warning)
 - Save any incorrect output/input *immediately*.
- Reproducible bugs are a faulty algorithm
- Non reproducible bugs are caused by either of
 - Reliance on the environment (time)
 - Uninitialised data: `int i ;`
 - Concurrent programs (next year)
 - Faulty hardware.
- Reproducible on one machine: non portable (side effects, byte order)

Non intrusive debugging

1. Check the obvious
2. Preparation
 - Prepare your program.
3. Hypothesise
 - Make a theory as to where the bug may be
4. Locate
 - Locate the bug in the source code
5. Verify
 - Check that the code follows the hypothesis
6. Repair
 - Remove the bug

Check the obvious and Prepare

Make sure that you are not going to chase a red herring.

Check that:

- You are working on current version of the software
- All your files are saved and compiled
- The compiler does not give any errors, even not with `-Wall`.

Prepare input and output:

- Have one or more input files ready.
- Know the expected output.
- You may want to eradicate any User-input (makes debugging a lot less effort).

Hypothesise

Most important phase of debugging

- *come up with an algorithm which could generate the incorrect output*

Reverse engineer the bug!

Reverse engineering is possible if you have inside knowledge of the algorithm

Example

Program Input			Program Output			Expected Output	
1	2	-8	4	-8	2	-4	
1	0	-4	4	-4	2	-2	
2	4	0	0	4	0	2	

Any volunteer to reverse engineer this program?

- We know only that it is a factor 2 out.
- Not enough prior knowledge.

Example

Program Input			Program Output		Expected Output	
1	2	-8	4	-8	2	-4
1	0	-4	4	-4	2	-2
2	4	0	0	4	0	2

Given that the program is supposed to calculate x so that $ax^2 + bx + c = 0$, we can reverse engineer it

- We know that $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
- We know that program is a factor of 2 out

Example

Program Input			Program Output		Expected Output	
1	2	-8	4	-8	2	-4
1	0	-4	4	-4	2	-2
2	4	0	0	4	0	2

Couple of options:

$$x' = \frac{-b \pm \sqrt{b^2 - 4ac}}{a} \quad x'' = \frac{-2b \pm \sqrt{4b^2 - 16ac}}{2a} \quad x''' = \frac{-ab \pm \sqrt{a^2b^2 - 4a^3c}}{a^2}$$

And dozens more: choose the one which is the simplest (Ockham's razor).

Locate, Verify

With the working hypothesis, go and look in the source code (you haven't so far!!)

- Go to the lines which you suspect.
- Compare them to the hypothesis.
- Make sure that the code which is there will indeed generate *all* the incorrect output
 - You may have the wrong hypothesis
 - There may be more than one bug

Repair: correct the program,

- Be extremely careful:
 - There may be more than one bug hiding.
 - The program which you are about to modify may be used somewhere else; in a slightly different context.
- Repairing bugs is not simply patching code. In particular:
 - Try not to add code (you can double the code size every year that way)
 - Make sure there is no dead code floating around
 - ⇒ Tidy up after you are ready

If non intrusive debugging does not work

- You may be chasing a bug which does not exist
- You may be staring at your code too long
 - You may need to explain your code to someone else
- You may not be able to build a hypothesis just on input and output
 - Intrusive debugging.

Intrusive debugging

If the bug is too complex:

- May need more output of the program.
- May need the ability to look inside the program.

Steps for intrusive debugging:

- Strip anything you don't need.
- Observe the output of the program, build a hypothesis as to where the bug is (roughly)
- Change the program so you obtain extra output which will allow you to sharpen your hypothesis.

The ultimate hypothesis will pin the bug down.

What output to produce

Produce as little output as possible.

- If you want to make sure that the sum of two numbers is 12, print the sum of the two numbers, not the two numbers.
- You may want to leave statements in all the time, preceded by an `if(debugging)`, and a global `int debugging=0;`

Debuggers - I

- A debugger allows you to peek in a running program.
- Example:

```
% cc -g x.c
```

```
% gdb a.out
```

```
(gdb) break power
```

```
Breakpoint 1 at 0x10984: file power.c, line 2.
```

```
(gdb) run
```

```
Breakpoint 1, power (x=3, n=10) at x.c:2
```

```
2          if( n == 0 )
```

```
(gdb) continue
```

```
Continuing.
```

```
Breakpoint 1, power (x=3, n=9) at x.c:2
```

```
2          if( n == 0 )
```

```
(gdb)
```

Debuggers - II

What debuggers are good for:

- Peeking inside a program
 - Put a break point in the code, run and print all values that you want to know.
- Post-mortem examination
 - Crashing programs dump a “core-file”, which is the contents of the memory. Debuggers can reconstruct the program state at the time of the crash.

What debuggers don't do:

- Debugging a program
 - they just allow you to look in your running code
- Printing clever output
 - It will print output, but if you want to monitor something complex, you may want to change your code.

Conclusion

- Hypothesise
 - With a hypothesis, you can find a bug
 - Without a hypothesis, you may as well try to find a black cat in coal store at night (without being sure the cat is actually in the coal store).
- Verify
 - Make sure that you understand the bug. Don't simply change something.
- Repair carefully
 - Tidy up.
- Don't chase ghosts.