

Debugging and profiling

Andy Buckley

University of Glasgow

MCnet Computing School,
Mariaspring, 3 September 2018



THE ROYAL
SOCIETY



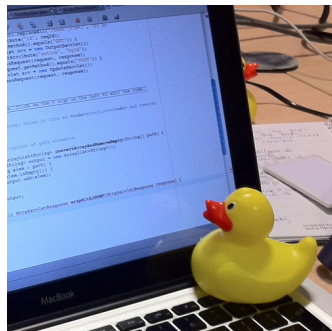
University
of Glasgow



Debugging

Debugging

- ▶ Identifying the cause of an error and fixing it
- ▶ Want to fix the cause, not just the symptom
- ▶ Be patient and methodical, cf. “scientific debugging”
<http://c.learncodethehardway.org/book/ex31.html>
- ▶ Some bug-hunt tricks are language/compiler-specific... although error messages are getting better!



Rubber duck / Cardboard dog
/ long-suffering office-mate
debugging is not to be sniffed
at...

Fixing compilation/linking errors

- ▶ Doesn't apply to Python!
- ▶ There is a distinction between compiler errors and linker errors: even visible *aesthetically*! ⇒ EXERCISE
- ▶ **First, read the error message.** Too many “my compiler said this” ‘bug’ reports
- ▶ Start at the top: many later messages may be spurious, caused by the first issue
- ▶ **clang/LLVM** gives more helpful error messages than GCC (still true circa GCC 5.x, but there's now GCC highlighting)

Preprocessor-time:

Argh...

Compile-time:

```
compile_time.cc:12:11:  
error: expected  
primary-expression before  
âĂŸ:âĂŹ token
```

Link-time:

```
/tmp/cczS5KsC.o: In  
function 'main':  
link_time.cc:(.text+0x1f):  
undefined reference to  
'MyStruct::foo()'
```

Isolating runtime bugs

Common bug types

- ▶ “FPEs”: numeric badness like overflow, nan, div-by-zero;
 - Can turn FPEs into exceptions, cf.
`feenableexcept (FE_OVERFLOW|FE_DIVBYZERO|FE_INVALID)`
- ▶ Crashing bugs: segfaults/SEGV/GPF — memory violations
e.g. out-of-bounds array accesses
- ▶ Hangs
- ▶ *It's just wrong!!* And how do you know?

Isolating runtime bugs

Common bug types

- ▶ “FPEs”: numeric badness like overflow, nan, div-by-zero;
 - Can turn FPEs into exceptions, cf.
`feenableexcept (FE_OVERFLOW|FE_DIVBYZERO|FE_INVALID)`
- ▶ Crashing bugs: segfaults/SEGV/GPF — memory violations
e.g. out-of-bounds array accesses
- ▶ Hangs
- ▶ *It's just wrong!!* And how do you know?
- ▶ Failure doesn't necessarily make itself known at the source... maybe only far downstream
- ▶ Don't dismiss debug printouts!
 - Not clever, but quick and often useful: first port of call.
Debug logging control macros can help
 - “Triangulate” bugs with initial coarse placement and
“binary search” \Rightarrow refinement
- ▶ But funkier stuff available, starting with “real” debuggers...

Debuggers

Python: pdb

- ▶ Nice little debugger available as a Python module, cf.
`python -m pdb myscript.py`
- ▶ Can also be programmatically enabled in code via
`import pdb; pdb.set_trace()`
- ▶ EXERCISE: Investigate the running of a Project Euler code from yesterday, see <http://tjelvarolsson.com/blog/five-exercises-to-master-the-python-debugger/> for help.

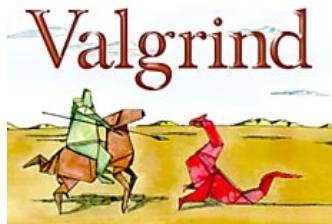
Debuggers

Compiled code: gdb/ldd/lldb

- ▶ Classic debuggers, rather cryptic: `up`, `down`, `n`, `s`, `p`, `bt`, ...
- ▶ Tip: to pass args, `gdb --args myexec foo bar`
- ▶ Tip: to call on scripts:
`gdb --args 'which python' myscript.py`
- ▶ Also useful when extending Python: that's a great way to really screw up memory!
- ▶ EXERCISE: Debug a linked list in C++:
<http://www.enseignement.polytechnique.fr/informatique/profs/Leo.Liberti/teaching/c++/online/exercises/node67.html>

Memory problem debugging — valgrind

- ▶ Valgrind is an amazing suite of tools for *instrumenting* code at runtime
- ▶ Essentially hardware emulation: tends to be sloooooooooooooow
- ▶ Fantastic for memory leak debugging, and also some profiling
- ▶ Main mode: **valgrind**
`--tool=memcheck`
`--leak-check=full myprog arg1`
...
- ▶ EXERCISE: run Valgrind on result of previous debugger fixes



Profiling

Simple profiling with `time`

- ▶ So easy! So useful!
- ▶ `time myprogram`
- ▶ That's it. Can run on cmd sequences, too:
- ▶

```
$ time (seq 100000000 | grep 1987 | sed s/5/r/g | wc -l)
49999

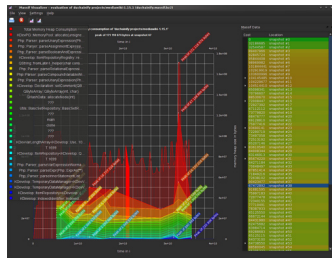
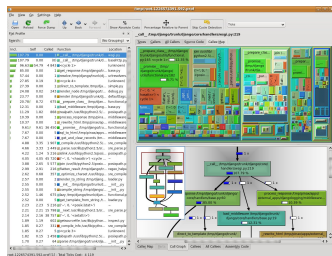
real 0m1.083s
user 0m1.820s
sys 0m0.280s
```
- ▶ EXERCISE: Python `add_numbers`



More valgrind tools

- ▶ More tools: Cachegrind/callgrind monitor function calls, Massif profiles memory allocation (see <http://valgrind.org/info/tools.html>)
- ▶ `valgrind --tool=callgrind`
⇒ `kcachegrind tool`

`valgrind --tool=massif`
⇒ `massif-visualiser tool`
- ▶ EXERCISE: profile the C++ linked list



Python profiling with cProfile

- ▶ `python -m cProfile myscript.py`
- ▶ Instrumentation will slow down
- ▶ EXERCISE:
`time add_numbers_3.py`
`python -m cProfile -s cumtime`
`add_numbers_3.py`
- ▶ `line_profiler` etc. also available, cf.
<https://www.huynh.com/posts/python-performance-analysis>

And more, more, more...

`perf`, `gprof`, the Mac `ddd` and Instruments, and
... may also be useful

A word of caution on profiling

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

— Donald Knuth

