

Computer Labs: Debugging

2º MIEIC

Pedro F. Souto (`pfs@fe.up.pt`)

October 26, 2014

Bugs and Debugging

Problem To err is human

- ▶ This is specially true when the human is a programmer :(

Solution There is none. But we can make it less likely

- ▶ By programming carefully
- ▶ By heeding the compiler warnings
- ▶ By using, if possible, a language different from C/C++
 - ▶ Otherwise, use `assert()` generously
- ▶ By designing good test programs:
 - ▶ If a test program does not detect bugs, most likely it was poorly designed

Using `assert()`

```
//#define NDEBUG // uncomment for public release
#include <assert.h>
void bounds(int i) {
    static int t[100];
    assert( i>=0 && i>100 ); // abort program if false
    ...
}
```

- ▶ `assert()` aborts the program and prints information showing where, when, the condition specified as its argument is false

```
pol: pol.c:50: bounds: Assertion 'i>=0 && i>100' failed
Aborted
```

- ▶ Should not be used in production
 - ▶ Define the `NDEBUG` constant
 - ▶ A program should rarely abort in normal usage
 - ▶ Even if there is nothing else to do, the information provided by `assert()` is not useful to a user
- ▶ Be careful when writing the condition for `assert()`
 - ▶ A bug in the condition may mislead you in a wasted search for a non-existing bug

Debugging and the Scientific Method

Debugging is a **ludic** activity, based on logic

1. Locate/identify the bug (the fun part, sometimes)
2. Fix the bug

Algorithm for identifying a bug:

While the bug has not been found:

1. put forward a hypothesis about where the bug is
2. design a test to prove/reject the hypothesis
3. carry out the test (possibly, changing the code)
4. interpret the test result

Debugging Rule #1: Debug with Purpose

- ▶ Don't just change code and “hope” you'll fix the problem!
 - ▶ I've seen many of you doing it out of despair!
 - ▶ (I confess, that I've done it, but ... it does not help)
- ▶ Use the scientific method
 - ▶ What is the simplest input that produces the bug?
 - ▶ What assumptions have I made about the program operation?
 - ▶ How does the outcome of this test guide me towards finding the problem?
 - ▶ Use pen and paper to keep track of what you've done

Debugging Rule #2: Explain it to Someone Else

- ▶ Often explaining the bug to “someone” makes your neurons “click”. The “someone” may be:
 - ▶ Another group member or colleague
 - ▶ Even someone that is not familiar with the subject
 - ▶ If there is nobody available, you can try explain it to yourself

Debugging Rule #3: Focus on Recent Changes

- ▶ Ask your self:
 - ▶ What code did I change recently?
- ▶ It helps if you:
 - ▶ write and test the code incrementally
 - ▶ use SVN
 - ▶ do regression testing, to make sure that new changes don't break old code
- ▶ However, remember that:
 - ▶ new code may expose bugs in old code

Debugging Rule #4: Get Some Distance ...

- ▶ Sometimes, you can be TOO CLOSE to the code to see the problem
- ▶ Go for a walk, or do something else
- ▶ “Sleep on the problem”
 - ▶ May not be an alternative if your deadline is the following day

Debugging Rule #5: Use Tools

- ▶ Sometimes, bug finding can be very easy by using error detection tools
 - ▶ You just have to use them properly
- ▶ Use **gcc** flags to catch errors at compile time:
 - ▶ `-Wall`, `-Wextra`, `-Wshadow`, `-Wunreachable-code`
- ▶ Use a debugger such as `gdb`
 - ▶ This is not an option in LCOM
- ▶ Use runtime memory debugging tools
 - ▶ E.g. Electric Fence, Valgrind
(not really an option in LCOM)

Debugging Rule #6: Dump State ...

- ▶ For complex programs, reasoning about where the bug is can be hard, and stepping through in a debugger time-consuming
- ▶ Sometimes, it is easier to just “dump state”, i.e. use `printf()`, and scan it for what seems “odd”
 - ▶ This may help you zero in on the problem

Debugging Rule #7: Think Ahead

Once you've fixed such a bug, ask yourself:

- ▶ Can a similar bug exist elsewhere in my code?
 - ▶ Bugs are often a consequence of a misunderstanding of an API
- ▶ How can I avoid a similar bug in the future?
 - ▶ Maybe coding 36 straight hours before the deadline won't help...

Tools of the Trade

Different bugs require different tools:

`printf()` Can be used to:

- ▶ Check simple hypothesis
- ▶ Zero in on hard to reproduce or highly complex bugs

`gdb`

- ▶ Very useful when the program crashes with segfault

Debugging with `printf()`: `debug.h`

```
#include <stdio.h>
#define DEBUG // comment/uncomment as needed

#ifdef DEBUG
    #define print_location() printf( \
        "At file %d, function %s, line %d\n", \
        __FILE__, __FUNCTION__, __LINE__);

    #define print_dbg(...) printf( __VA_ARGS__ )
#else // does nothing, not even generates code!
    #define print_ident()
    #define print_dbg(...)
#endif // DEBUG
```

- ▶ In general, it would be preferable to use `fprintf(stderr, ...)` rather than `printf(...)`, but in Minix it does not work
 - ▶ The problem appears to be with `fprintf()`, as it does not work even with `fprintf(stdout, ...)`
 - ▶ This is probably because the C library used by privileged programs is not the standard C library

Debugging with `printf()`: Usage

```
#include "debug.h"
int main() {
    print_ident();
    print_dbg("dir=%s, count=%d\n", "popo", 5);
    print_dbg("bye\n");
    print_ident();
    return 0;
}
```

At file `po.c`, function `main`, line 18

`dir=popo, count=5`

`bye`

At file `po.c`, function `main`, line 21

- ▶ **Macros do not generate code, if `DEBUG` is not defined**
 - ▶ It is not necessary to comment/uncomment `printf()` in code
- ▶ It may be convenient to define different `DBG_XXX` constants, by using bit-masks you can print debugging messages related to different aspects

Thanks to:

I.e. shamelessly copied material by:

- ▶ **Dave Andersen** (dga@cs.cmu.edu)
 - ▶ Debugging rules
- ▶ **João Cardoso I** (jcard@fe.up.pt)
 - ▶ `assert()`, `printf()`