Computer Labs: Debugging

2º MIEIC

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

November 23, 2010

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

  ```
  po1: po1.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

Degugging 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`
- Use runtime memory debugging tools (not really an option in LCOM)
  - E.g. Electric Fence, Valgrind

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_ident() fprintf(stderr, \
        "At file %d, function %s, line %d\n", \
        __FILE__, __FUNCTION__, __LINE__);

    #define pring_dbg(...) fprintf(stderr, __VA_ARGS__)

#else // does nothing, not enven generates code!

    #define print_ident()
    #define print_dbg(...)

#endif  // DEBUG
```

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

GDB: The GNU Debugger

Run a program and:

- see where it crashes
- suspend its execution to examine program state

Two ways to run it:

1. `gdb binary`: to run binary inside of `gdb`
2. `gdb binary core-file`: to debug a crashed program
   - If you are using `bash`, you can issue:

     `ulimit -a` to find out the size limits for different resources

     `ulimit -c unlimited` to remove any size limit for core dumps

Do not forget, you need to compile your programs with

- the `-g` option and
- NO optimizations

GDB Commands

For Controlling Execution

- `run <cmd-line args>`
- `break <func>`
- `step`
- `next`
- `control-c`

For Getting Info

- `backtrace/where`
- `print <expr>`
- `info locals`
- `list`
- `up/down`

Suspending Program Execution

Breakpoints

Suspend a program at a given execution point, p.ex. to check that the program executes a given instruction

- `break <fun. name>`, e.g. `break draw_pixel`
- `break <file name>:<line num>`, e.g. `break graphics.c:57`

Obs.- There are several other ways to specify `breakpoints` in `gdb`

Watchpoints

Suspend a program when the value of an expression changes

`watch <expr>` , e.g. `watch sum == 15`

Other Operations on Breakpoints/Watchpoints

`info break/watch` info about breakpoints/watchpoints;

`clear <breakpoint>` deletes a breakpoint, that must be specified as in the `break` command;

`delete <breakpoint_no>` deletes the breakpoint whose number is specified;

`disable <breakpoint_no>` disables the breakpoint whose number is specified;

`enable <breakpoint_no>` enables a breakpoint

Examining Variables

Question What is the value of program variables?

Answer Use one of the following commands:

- `print /F EXP` where `EXP` is a C language expression, whose value one wishes to evaluate. `/F` is optional, and allows to specifiy the format to use to show the value. E.g.:

```
print /x ptr
```

- `display /F EXP` shows the value of expression `EXP` everytime the program pauses. (Use `undisplay` to undo the effect of `display`.)
- `printf format-string, arg1, arg2, ...` similar to the C standard library `printf`, but without parenthesis:

```
printf ``%s \n", msg
```

Examining the Stack

Question How did we get to this breakpoint/watchpoint?

Solution Use the command `backtrace` (or `bt`), or `where` which lists the frames in the stack.
Other useful commands are:

- `frame` shows the contents of the current frame
- `up [N]` move `N` frames up in the stack: when `foo()` invokes `bar()` its frame is above `bar()`'s frame
- `down [N]` smove `N` frames down the stack

Obs.- The frame corresponding to the PC has number 0

Obs.- a synonymous of `bt` is `where`

Start/Continue Program Execution

`r {args}` run program with `args`, if any;

`c [N]` continues the program execution, ignoring the `N` passages by this breakpoint

`n [N]` executes the following `N` C instructions;

`s [N]` similar to `n`, except that it steps into a function call;

Obs.- For the last 3 commands `N=1`, if omitted

`Ctrl-C` allows to suspend an out of control program (e.g. executing an endless loop): `gdb` does not terminate

A gdb Session (1/11): Entering and Exiting gdb

```
> gcc -g programa.c -o programa
> gdb protrama
 (gdb) run arg1 arg2 ..
 (gdb) help
List of classes of commands:

breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
tracepoints -- Tracing of program execution without stopping t

[...]
 (gdb) quit
The program is running. Exit anyway? (y or n) y
```

A gdb Session (2/11): where

Shows the stack trace

```
(gdb) run
...
Program received signal SIGABRT, Aborted.
0xffffe410 in __kernel_vsyscall ()
(gdb) where
#0 0xffffe410 in __kernel_vsyscall ()
#1 0x4005b541 in raise () from /lib/tls/libc.so.6
#2 0x4005cdbb in abort () from /lib/tls/libc.so.6
#3 0x40054925 in __assert_fail () from ...
#4 0x08048759 in bounds (i=283) at po1.c:50
#5 0x0804869f in main (argc=1, argv=0xbf943fe4) at po1.c:31
```

A gdb Session (3/11): break

Sets a **breakpoint**, thus suspending a program's execution at specified point/function

```
(gdb) break fact // pause when fact() is invoked
Breakpoint 1 at 0x8048680: file po1.c, line 36.
(gdb) run 5 // run program with arg 5
Starting program: /home/jcard/tmp/po1 5
Breakpoint 1, fact (n=5) at po1.c:36
36 if (n <= 1)
(gdb) info break // show info about breakpoints
Num Type Disp Enb Address What
1 breakpoint keep y 0x0804868d in fact at
po1.c:36
(gdb) delete 1 // remove breakpoint 1
(gdb) disable 1 // inhibit, but does not remove, breakpoint 1
```

A gdb Session: Conditional Pause (4/11)

Pauses program execution at specified point/function if condition is true

```
(gdb) break fact if n==10 // pause in fact()
                          //  only if argument (n) is 10
Breakpoint 1 at 0x8048680: file po1.c, line 36.
(gdb) run 2 5 7 10
Starting program: /home/jcard/tmp/po1 2 5 7 10
Factorial of program args:
n=2.000000 n!=2.000000
n=5.000000 n!=120.000000
n=7.000000 n!=5040.000000
Breakpoint 1, fact (n=10) at po1.c:36
36 if (n <= 1)
```

A gdb Session: watch (5/11)

A **watchpoint** is a breakpoint that pauses execution when the value of an expression changes

```
(gdb) watch sum==15 // pauses whem sum == 15 changes
Hardware watchpoint 2: sum == 15
(gdb) cont // continue program execution
Continuing.
Hardware watchpoint 2: sum == 15
Old value = 0
New value = 1
sum (n=8) at po1.c:55
55 for (i=0; i<n; i++)
(gdb) print i
$2 = 5
(gdb) print sum
$3 = 15
```

A gdb Session: print value of expression (6/11)

Show the value of an expression

```
Breakpoint 2, bounds (i=283) at po1.c:45
45 count ++;
(gdb) list -5 // show previous 5 lines of source code
40
41 void bounds(int i) {
42 static int t[100];
43 static int count=0;
44
(gdb) print i // print the value of local variable i
$8 = 293
(gdb) print count // print the value of local variable count
$9 = 4
(gdb) print t[4]
$10 = 23
(gdb) print t[i]
$11 = 45
```

A gdb Session: set Value of Program Variable (7/11)

Evaluates expression and assigns its value to program variable, without displaying it

```
gdb) print count
$9 = 4
(gdb) set count=34    // assign local variable count the value
(gdb) pr count        // gdb accepts abbreviations,
                      //  as long as they are not ambiguous
$13 = 34
(gdb) cont            // cont(inue) execution
Continuing.
Breakpoint 2, bounds (i=235) at po1.c:45
45 count ++;
(gdb) print count
$14 = 35
(gdb) set var i=5          // alternative to set i=5
(gdb) set var t[i]=34      // set is used for other purposes
```

set is used also for setting the values of GDB's internal parameters

A gdb Session: Execute next line (8/11)

Executes next line, including any function calls

```
(gdb) run 5 3 7 // executes program with args list 5, 3 e 7
Breakpoint 1, main (argc=4, argv=0xbf937b04) at po1.c:12
12 printf("Factorial of program args:\n");
(gdb) next // reached breakpoint: execute next line
Factorial of program args:
13 for (i=1; i<argc; i++)
(gdb) next // and next one
14 foo(atof(argv[i]));
(gdb)  // pressing enter, repeats last command
n=5.000000 n!=120.000000
13 for (i=1; i<argc; i++)
(gdb)
14 foo(atof(argv[i]));
(gdb)
n=3.000000 n!=6.000000
13 for (i=1; i<argc; i++)
```

A gdb Session: step through code (9/11)

Executes until another line is reached, possibly in another function

```
(gdb) run 5 3 7
Breakpoint 1, main (argc=4, argv=0xbffd8984) at po1.c:12
12 printf("Factorial of program args:\n");
(gdb) step // breakpoint reached, execute until next line
Factorial of program args:
13 for (i=1; i<argc; i++)
(gdb)  // <Enter>: repeats previous command
14 foo(atof(argv[i]));
(gdb) step // Pauses at foo's first line
foo (i=5) at po1.c:32
32 printf("n=%f n!=%f\n", i, fact(i));
(gdb)
fact (n=5) at po1.c:36
36 if (n <= 1)
(gdb)
38 return n*fact(n-1);
```

A `gdb` Session: `finish` Function (10/11)

Run until the selected stack fame returns

```
Breakpoint 1, sum (n=100) at po1.c:50
50 int i, sum=0;
(gdb) next
... depois de vários next
(gdb) print i
$3 = 7
... repetir next mais 100 vezes?
(gdb) finish // run until the end of the selected stack frame
Run till exit from #0 sum (n=100) at po1.c:51
0x0804852e in main (argc=1, argv=0xbf94e4a4) at po1.c:11
11 sum(100);
Value returned is $4 = 4950
(gdb)
```

A gdb Session: call Function (11/11)

Executes the specified function with the specified argument

```
(gdb) list foo
31      void foo(float i) {
32          printf("n=%f n!=%f\n", i, fact(i));
33      }
35      float fact(float n) {
36          if (n <= 1)
37              return 1;
38          return n*fact(n-1);
39      }
(gdb) call fact(5) // invoke function fact() with argument 5
$17 = 120
(gdb) call foo(5) // invoke foo()
```

Conclusion

- Debugging is wasteful but unavoidable
  - Program carefully, to reduce the likelihood of bugs
- The use of a debugger like `gdb` may help finding most bugs rather quickly
  - However, learning how to use `gdb` to its full extent is hard (the user manual is over 500 pages)
  - The use of GUI's, such as `ddd` may help
- The most insidious bugs are those
  - In the logic of complex programs
  - Or that are hard to reproduce, such as race conditions

  In this case, the use of printing macros may be essential to zero in the bug

Thanks to:

I.e. shamelessly copied material by:

- Dave Andersen (dga@cs.cmu.edu)
  - Debugging rules
- João Cardoso (jcard@fe.up.pt)
  - `assert()`, `printf()` and `gdb` session

Further Reading

- On `gdb`
  - GDB's Manual
  - GDB's v4 Reference Card