Effective Debugging Methods and Tools for Your Projects

Suraj N. Kurapati

19th October 2005

UCSC SOE Honor Society

Agenda

- Motivation
- Introduction
- Debugging methods
- Debugging tools
- Live demonstration
- Questions

Motivation

We are not taught how to debug programs! Why?

- Debugging is a cognitive process [1] unlike:
  - programming
  - designing
  - testing
- "The end result is knowledge of why there is a problem and what must be done to correct it" [1].
- Principles can be taught, but problem solving is learnt through practice.

Introduction

- What is testing?
  - Testing enables one to determine **whether** certain input causes a program to misbehave. [1].
  - Write test cases that try to break your code!
- What is debugging?
  - Debugging enables one to determine **why** certain input causes a program to misbehave and **what** one must do to correct the misbehavior [1].

Debugging Methods

1. Editing
2. Littering
3. Interacting
4. Repeating
5. Thinking

Debugging by Editing

If the program misbehaves, then:

1. Edit some code
2. Run it again
3. Hope it works!

✘ Edits are random and lack supportive evidence.

✘ Effectiveness limited to short, trivial projects [1].

Debugging by Littering

If the program misbehaves, then litter the output with status messages.

- Littering source code:

```
print "Starting algorithm now..."
  run algorithm
print "Algorithm has finished."
```

- Littered output upon crash:

```
Starting algorithm now...
segfault
```

✔ Vague knowledge of bug's possible whereabouts.

✘ A maintenance headache, even for short projects!

✘ Status messages divert attention from real output [2].

✘ Lots of status messages cause "scroll blindness."

Debugging by Interacting

If the program misbehaves, then use an interactive debugger to [1, 2]:

1. Set breakpoints to interrupt execution
2. View and test the values of variables
3. Observe the functional flow (stack trace)
4. Step through the execution line by line

✔ Output is not littered with status messages.

✔ Bug's possible whereabouts known before any edits.

✔ Programmer's assumptions can be verified quickly.

✔ Disproved assumptions can trigger interruptions.

Debugging by Repeating

If the program misbehaves, then employ the previous methods and repeat as necessary.

✔ Through trial and error, one eventually accumulates a set of useful debugging strategies [1].

✘ New bugs can defeat accumulated strategies.

✘ It takes too long to develop good debugging skills [1].

Debugging by Thinking

"Debugging by thinking means actively seeking methods from intellectual disciplines that solve analogous problems" [1].

- Understand how other professions solve problems [1]:

  1. Detective
  2. Mathematician
  3. Safety expert
  4. Psychologist
  5. Engineer
  6. Computer scientist

✔ Builds a systematic approach based on multidisciplinary understanding of problem solving [1].

✔ Gives insight and provides structure to the way we debug [1].

The Detective's Approach

What guidelines[1] does a detective use to solve crimes?

- Don't guess
- Start by observing
  - Apply cross-disciplinary knowledge
  - Pay attention to unusual details
  - Focus on facts
    * Gather facts before hypothesizing
    * Use a system for organizing facts
    * State the facts to someone else

[1] The following guidelines are direct quotations from reference [1].

- Enumerate possibilities
  - Show how something could be done
  - Use alibis as clues
  - The camouflage effect
    * You're looking right at it
- Eliminate impossible causes
  - Look once, look well
  - Exclude alternative explanations
  - Reason in both directions
  - Watch for red herrings

Debugging Tools

GDB interactive source-level debugger.

DDD graphical front-end for GDB and other debuggers.

Valgrind suite of run-time analysis tools.

- When compiling with GCC, use the "-g" option to pack debugging information into the resulting binary.

The GNU Debugger

- Invocation syntax [3]:

  `gdb` *`program`*
  `gdb` *`program coreDump`*
  `gdb` *`program processId`*

- The basics of interaction:
  - GDB behaves like a standard interactive shell.
  - Press TAB for automatic word completion.
  - Most commands have short abbreviations.
  - When in doubt, ask for help:

    `help`
    `help` *`topic`*

  - When you've had enough, simply:

    `quit`

Environment Configuration

- Set environment variables [3]:

  ```
  set env variable=value ...
  show env
  unset env variable
  ```

- Set program arguments [3]:

  ```
  set args arg1 arg2 ...
  show args
  ```

Breakpoints and Watchpoints

- Set breakpoints [3]:

  `break` *`functionName`*
  `break` `...` `if` *`condition`*
  `rbreak` *`regularExpression`*
  `catch` *`exceptionName`*

- Set watchpoints [3]:

  `watch` *`variable`*

- List breakpoints and watchpoints [3]:

  `info break`
  `info watch`

- Manage breakpoints and watchpoints [3]:

  `enable` *`pointNumber`*
  `disable` *`pointNumber`*
  `clear` *`pointNumber`*

Making Observations

- Start execution of the program [3]:

  `run`
  `run < inputFile`
  `run > outputFile`
  `run 2> errorFile`

- Step through the execution [3]:

  step step into function calls, if any

  next step over function calls, if any

  continue run until program ends or is interrupted

  until *location* pause when given location is reached

  finish pause after current function returns

- View and traverse the program flow [3]:

  `frame`
  `backtrace`
  `up` *`numberOfFrames`*
  `down` *`numberOfFrames`*

- View and test variables [3]:

  `print` *`expression`*
  `display` *`expression`*

The Valgrind Suite

- Powerful run-time analysis tools [4]:

  memcheck a heavyweight memory checker

  addrcheck a lightweight memory checker

  cachegrind a cache profiler

  massif a heap profiler

- Invocation syntax[2] [4]:

  `valgrind --tool=`*`name program arg1 arg2 ...`*

✔ Works directly with pre-compiled binaries.

✔ No need for inclusion of or linking to special libraries.

[2] Memcheck is invoked by default when the "--tool" option is omitted [4].

Live Demonstration

```
#include <stdlib.h>

/* leaks the given number of bytes */
void leakMemory( int howManyBytes ) {
  malloc( howManyBytes * sizeof( char ) );
}

/* causes a segmentation fault */
int segFault() {
  int* p = ( int* ) 0x1F;
  return *p;
}

/* leaks memory and crashes */
int main( int argc, char** argv ) {
  leakMemory( 33 );
  segFault();
  return 0;
}
```

Figure 1: A buggy C program.

References

[1] R.C. Metzger, "Debugging by Thinking: A Multidisciplinary Approach," Burlington, MA: Elsevier Digital Press, 2004.

[2] N. Matloff, "Guide to Faster, Less Frustrating Debugging," [Online document], 2002 Apr 4, [cited 2005 Oct 15], Available HTTP: `http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html`

[3] R.M. Stallman and Cygnus Support, "Debugging with GDB," Boston, MA: Free Software Foundation, 1996.

[4] Valgrind Developers, "Valgrind User Manual Release 3.0.0," [Online document], 2005 Aug 3, [cited 2005 Oct 15], Available HTTP: `http://valgrind.org/docs/manual/manual.html`