CSCI 2132: Software Development

Testing and Debugging

Norbert Zeh

Faculty of Computer Science
Dalhousie University

Winter 2019

Software Testing and Debugging

Bugs = programming errors

- Obvious bugs
- Is it a “bug” or a “feature”?

Testing used to detect bugs

Debugging used to remove bugs

Software Testing

Motivation:

- Ensure robust software
- Maintain reputation
- **Lower cost:** Fixing a bug before release is always cheaper than after release.
- May be critical for security, ensuring user privacy

Software engineer in testing is a whole job!

What Do We Test?

Does the program work?

What does this mean?

- Does the program meet its specifications?

Specification:

- Description of the input
- Description of the output
- Set of conditions
- Specification of output as a function of the input and conditions

What Do We Test?

Does the program work?

What does this mean?

- Does the program meet its specifications?

Specification:

- Description of the input
- Description of the output
- Set of conditions
- Specification of output as a function of the input and conditions

How Do We Test?

Mindset: Try as hard as you can to make your program fail!

Typical test cases:

- Regular cases
- Boundary cases
- Error cases (the code fails when it should)

Types of Testing

White box testing:

- Use knowledge of implementation to guide selection of test cases
- **Goal:** Achieve maximum code coverage (exercise every single line/function of the code)

Black box testing:

- Use specification to guide selection of test cases
- **Goal:** Achieve maximum coverage of cases given in the specification

Debugging

Debugging = methodical process of finding and fixing bugs (defects) in a computer program

Key step: Find where things go wrong in the program

- Track program state:
  - Current location in the program
  - Current values of variables
  - Number of iterations through a loop
- Identify when the observed program state does not match the expected program state.

printf Debugging

Use `printf` statements to print

- Values of variables
- Program location

Example: `printf("Got here\n");`

Cons:

- Requires modifications to the code to be removed later

Pros:

- Debuggers usually only show the current state
- `printf` statements can used to produce a log of the program execution

Strategies for printf Debugging

Linear approach:

- Start adding `printf` statements at the beginning of the program
- Continue until the printout differs from what we expect

Binary search:

- Select halfway point
- Determine if the bug has occurred
- If yes, look in the first half
- Otherwise, look in the second half

Experience:

- We often have an idea roughly where the error occurred
- Apply the above strategies in a limited scope

Using a Debugger

Debugger = tool to run program in a sandbox to observe its behaviour

Features:

- Step through program
- Inspect variables
- Inspect program state (e.g., call stack)
- …

Modern IDEs include a debugger

gdb: GNU Project Debugger

Symbolic (source-level) debugger

Program that allows programmer to

- Access another program's state
- Map the state to the program's source code
  (line numbers, variable names, ...)
- View variable values
- Set breakpoints
- Requires compilation with `-g` option
  (adds source code information to the executable)

Breakpoints

Mark program locations so the debugger stops the program execution each time such a location is reached.

When stopped at a breakpoint, the programmer can:

- Inspect variable contents
- Single-step through code
- Resume execution until the next breakpoint is reached

gdb Commands

`run`: start running the program

`break line_number`: Set a breakpoint at line `line_number`

`break function_name`: Set a breakpoint at the entry point of function `function_name`

`next`: Execute next step in the program (Step over function call: function call = 1 step)

`step`: Execute next step in the program (Step into function call)

`break 316`

`break call_me`

`step`

`next`

```
311 }
312
313 int call_me(int x) {
314   int y = 0;
315   while (x) {
316     y = (y<<1) + (x&1);
317     x >>= 1;
318   }
319   return y;
320 }
```

```
422
423 void calling_you() {
424   int x = 0;
425   int y = call_me(x);
426   printf("%d\n", y);
430 }
431
```

Basic Debugger Operations

- Set breakpoints
- Examine variables at breakpoints or trace execution of the code
- **Strategy:** linear or binary
- Do this until the bug is found
- Advantage over printf-style debugging: No recompilation