

CS 6015: Software Engineering

Spring 2024

Lecture 6: Debugging

This Week

- Designing a program
- Debugging (Lab 2)
- Assignment 3 released

Next Week

- Defensive programming
- Testing / Code Coverage

Plan

- Software bug
- Debugging vs testing
- Error types
- Debugging strategies
- Debugging methods
- Tools



Jan Arkesteijn CC-BY 2.0

Admiral Grace Murray Hopper



9/9

0800 Antan started

1000 " stopped - antan ✓

1300 (032) MP - MC

(033) PRO-2

connect

Relays 6-2 in 033 failed special speed test
in relay

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multy Adder Test.

1545



Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1630 Antan started.

1700 closed down.

{ 1.2700 9.037 847 025

9.037 846 795 connect.

~~1.582 649 090~~
~~2.130 476 415~~ (033) 4.615 925 059 (-2)

2.130 476 415

2.130 676 415

Relay
214.5
Relay 3370

Software bug and debugging?

- **Software bug:** is an error in a computer program that causes it to produce an **incorrect** result, or to behave in unintended ways.
- What is **debugging**?
 - Debug= De+Bug
 - Meaning make the program bug free

Debugging vs Testing

- When testing a program:
 - All test cases **produce the expected** output, then the program is **successfully tested**.
 - One or more test cases fails, then the program is **incorrect**. It contains **errors / bugs**.
- Testing **only** reveals the presence of errors
- The debugging process helps identifying the causes and fixing the errors.

Error Types

- ***Syntax or type errors:***

- Caught by the compiler and reported via error messages that indicates the cause of error.

- ***Typos and other simple errors:***

- Undetected by the compiler such as misplaced parentheses or - instead of +.
 - $x+y/2$ instead of $(x+y)/2$

- ***Implementation errors:***

- The high-level algorithm of a program is not correctly translated to code.

- ***Logical errors:***

- The high-level algorithm was accurately converted to code, but the algorithm itself was incorrect.
- If the algorithm is logically flawed, the programmer must re-think the algorithm.

Difficulties in Debugging

- **The symptoms may not give clear indications about the cause**
- **Symptoms may be difficult to reproduce**
- **Errors may be correlated**
- **Fixing an error may introduce new errors**
 - This is the result of trying to do quick hacks to fix the error, without understanding the overall design and the invariants that the program is supposed to maintain.

A clean design and careful thinking can avoid many of these cases.

Debugging Strategies

- **Incremental and bottom-up program development.**
 - Develop the program *incrementally* and test it often.
 - The search for bugs is limited to small added code fragments.
 - *Bottom-up development*: once a piece of code has been successfully tested, its behavior won't change when more code is incrementally added later.
- **Backtracking**
 - Start from the point where the problem occurred and go back through the code to see how that might have happened
- **Binary search**
 - Explore the code using a divide-and-conquer approach, to quickly pin down the bug
 - Reduces the code that needs inspection to half
 - Repeating the process a few times will quickly lead to the actual problem

Debugging Strategies

- **Git tools**
 - git diff
 - git bisect

Debugging Strategies

- **Problem simplification**
 - Eliminate portions of the code that are not relevant to the bug
 - Simplify data rather than code
 - If the size of the input data is too large, repeatedly cut parts of it and check if the bug is still present.
 - When the data set is small enough, the cause may be easier to understand.

Debugging Methods

- **Instrument program to log information**
 - Print statements: Effective in some cases, but difficult when the volume of logged information becomes huge
 - Visualization tools can also help understanding the printed data
- **A better approach: Use debuggers**
 - Replaces the manual instrumentation
 - Gives all the needed run-time information without generating large, hard-to-read log files.

Debugging Methods

- **Instrument program with assertions**
 - A program stops as soon as an assertion (assert statement) fails

Tools to debug

- LLDB:
 - Part of LLVM framework
 - Default debugger in Xcode
 - To compile and run program with debug support:

```
$ clang++ -g -O0 main.cpp <other .cpp files> -o <executable_name>  
$ lldb <executable_name>
```

-g Builds executable with debugging symbols
-O0 Optimization Level 0 (No optimization, default)

LLDB commands

```
#include <iostream>
using namespace std;

int divint(int, int);
int main()
{
    int x = 5, y = 2;
    cout << divint(x, y);

    x = 3; y = 0;
    cout << divint(x, y);

    return 0;
}

int divint(int a, int b)
{
    return a / b;
}
```

Example

// set a breakpoint at divint

\$ (lldb) **breakpoint set --name divint** // alternative **b divint** or **br s -n divint**

// set a breakpoint in file1.cpp at line 12

\$ (lldb) **b file1.cpp:12** or **b 12**

// breakpoints are listed and enumerated

\$ (lldb) **breakpoint list** // alternative **br l**

// deletes all breakpoints or specific one by adding #

\$ (lldb) **breakpoint delete** or **br del 1** (deletes breakpoint 1)

\$ (lldb) **breakpoint enable/disable 1** // Enables/disables breakpoint 1

LLDB: examining the stack frame and variables

Command	Description
<ul style="list-style-type: none">- frame variable- Alternative: fr v	Shows the arguments and local variables for the current frame fr v -a (shows the local variables for the current frame) fr v var (show content of local variable var) p var (alternative of fr v var)
<ul style="list-style-type: none">- target variable- Alternative: ta v	Shows the global/static variables defined in the current source file ta v gvar (show the content of global variable gvar)
bt or thread backtrace	Prints the stack trace of the current thread bt all (shows the stack backtraces for all threads)
frame info	Lists information about the currently selected frame
frame select 12	Selects a different stack frame with #12
up and down	Goes up and down a level in the stack
exit	Quits lldb

LLDB: more commands

Command	Description
<ul style="list-style-type: none">- <i>memory read --size 4 --format x --count 4 0xbffff3c0</i>- Alternative: <i>me r -s4 -fx -c4 0xbffff3c0</i>	Reads memory from address 0xbffff3c0 and show 4 hex uint32_t values
<ul style="list-style-type: none">- <i>memory read --outfile /tmp/mem.txt -count 512 0xbffff3c0</i>- Alternative: <i>me r -o/tmp/mem.txt -c512 0xbffff3c0</i>	Reads 512 bytes of memory from address 0xbffff3c0 and save the results to a local file as text.
<ul style="list-style-type: none">- <i>thread list</i>	List the threads in your program

Recommended reference: <https://lldb.llvm.org/use/map.html>

Address Sanitizer or ASAN

- Fast memory error to detect:
 - Out-of-bounds access to heap, stack and globals
 - Use-after-free
 - Use-after-scope
 - Double-free, invalid free
 - Memory leaks

Backup slides

Tools to debug

- GDB (**G**NU **D**ebugger):
 - popular debugger for Unix systems
 - Uses a simple command line interface
 - To compile and run program with debug support:

```
$ clang++ -g -O0 main.cpp <other .cpp files> -o <executable_name>
```

```
$ gdb <executable_name>
```

-g Builds executable with debugging symbols
-O0 Optimization Level 0 (No optimization, default)

GDB Breakpoints

```
#include <iostream>
using namespace std;

int divint(int, int);
int main()
{
    int x = 5, y = 2;
    cout << divint(x, y);

    x = 3; y = 0;
    cout << divint(x, y);

    return 0;
}

int divint(int a, int b)
{
    return a / b;
}
```

Command	Description
<i>b main</i>	Puts a breakpoint at the beginning of the program
<i>b</i>	Puts a breakpoint at the current line
<i>b N</i>	Puts a breakpoint at line N
<i>b +N</i>	Puts a breakpoint N lines down from the current line
<i>b fn</i>	Puts a breakpoint at the beginning of function "fn"
<i>d N</i>	Deletes breakpoint number N
<i>info break</i>	List breakpoints

Example

\$ (gdb) **b main.cpp:divint**

\$ (gdb) **b main.cpp:10**

\$ (gdb) **info break** // breakpoints are listed and enumerated

\$ (gdb) **d** // deletes all breakpoints

GDB: other commands

Command	Description
<i>r</i>	Runs the program until a breakpoint or error
<i>c</i>	Continues running the program until the next breakpoint or error
<i>f</i>	Runs until the current function is finished
<i>s</i>	Runs the next line of the program
<i>s N</i>	Runs the next N lines of the program
<i>n</i>	Like s, but it does not step into functions
<i>l</i>	List code around current line

GDB: other commands

Command	Description
<i>p var</i>	Prints the current value of the variable "var"
<i>Watch var</i>	Stops whenever <i>var</i> changes its value, and prints out old and new value of <i>var</i>
<i>bt</i>	backtrace: Prints a stack trace, equivalent command: <i>where</i>
<i>up</i>	Goes up a level in the stack
<i>down</i>	Goes down a level in the stack
<i>quit or q</i>	Quits gdb