

## An Introduction to The Arcane Mysteries of The Black Arte of Program Debugging

Adapted from Ian Parberry  
University of North Texas

## Contents of This Lecture

1. Introduction
2. Debugging Tools
3. Debugging with `Printf`
4. Other Debugging Methods


Debugging 0. Contents 2

Section 1

## INTRODUCTION

Debugging 1. Introduction 3

## A Taxonomy of Bugs




1. Your program crashes.
2. Your program does the wrong thing.

Notice that failing to compile is not a bug.

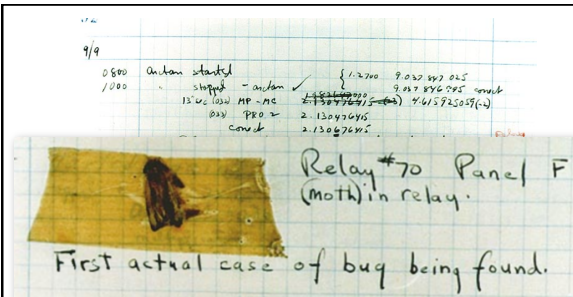
Debugging 1. Introduction 4

## Historical Note



- The term *bug* is usually attributed to Admiral Grace Murray Hopper in 1947.
- When colleagues traced a fault in Harvard University's Mark II Computer to a moth stuck in Relay 70 in Panel F, she remarked that they were "debugging" the system.
- The remains of the moth can still be seen in the log book...

Debugging 1. Introduction 5



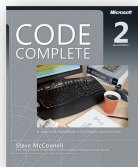
Debugging 1. Introduction 6



**Theodore Rubin:** The problem is not that there are problems. The problem is expecting otherwise and thinking that having problems is a problem.

## Will You Have Bugs?

The problem is not that there are bugs. The problem is expecting otherwise and thinking that having bugs is a problem.



An average programmer generates 15-50 bugs per 1000 lines of code.

Steve McConnell  
*Code Complete*



Section 2

## DEBUGGING TOOLS

## Debuggers

- There are specialized debugging tools for programmers, for example, the Gnu Debugger `gdb` on Unix.
- These let you interrupt the computation and examine the contents of memory.
- They are good for catching low-level bugs, but often the big picture is hidden by too much information (aka “can’t see the wood for the trees”).

## Using the Gnu Debugger



Compile with the `-g` switch so that the compiler generates extra information for the debugger to use. For example,

```
> g++ -g prog1.cpp
```

To run the debugger:

```
> gdb a.out
```

## About the Gnu Debugger



- `gdb` has an interactive shell.
- It can recall history with the arrow keys, auto-complete words (most of the time) with the TAB key, and has other nice features.
- If you're ever confused about a command or just want more information, use the `help` command, with or without an argument:

```
(gdb) help [command]
```

Debugging

2. Debugging Tools

13

## Using `gdb`



- To run the program, just use:  

```
(gdb) run
```
- If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), then the program should run fine here too.
- If the program did have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

```
Program received signal SIGSEGV,  
Segmentation fault. 0x000000000400524 in  
sum array region (arr=0x7fffc902a270,  
r1=2, c1=5, r2=4, c2=6) at sum-array-  
region2.c:12
```

Debugging

2. Debugging Tools

14

## Setting Breakpoints



- Breakpoints can be used to stop the program run at a designated point in the code.
- The simplest way is the command `break`, which sets a breakpoint at a specified file-line pair:  

```
(gdb) break file1.cpp:6
```
- This sets a breakpoint at line 6 of `file1.cpp`.
- Now, if the program ever reaches that location when running, the debugger will pause and prompt you for another command.

Debugging

2. Debugging Tools

15

## Fun With Breakpoints



- Once you've set a breakpoint, you can try using the `run` command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).
- You can proceed onto the next breakpoint by typing `continue`.
- You can single-step (execute just the next line of code) by typing `step`. This gives you really fine-grained control over how the program proceeds.

Debugging

2. Debugging Tools

16

## Querying Variables



- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line.
- However, sooner or later you're going to want to see things like the values of variables, etc.
- The `print` command prints the value of the variable specified, and `print/x` prints the value in hexadecimal:

```
(gdb) print myvar
```

Debugging

2. Debugging Tools

17

## Watchpoints



- Whereas breakpoints interrupt the program at a particular place in the code, *watchpoints* interrupt the program whenever a watched variable's value is modified.
- For example, suppose we do this:  

```
(gdb) watch myvar
```

Now, whenever `myvar`'s value is modified, the program will interrupt and print out the old and new values.
- Which `myvar`? The one that's currently in scope.

Debugging

2. Debugging Tools

18

## Other Useful Commands



`backtrace` - produces a stack trace of the function calls that lead to a segmentation fault

`where` - same as `backtrace`; you can think of this version as working even when you're still in the middle of the program

`finish` - runs until the current function is finished

`delete` - deletes a specified breakpoint

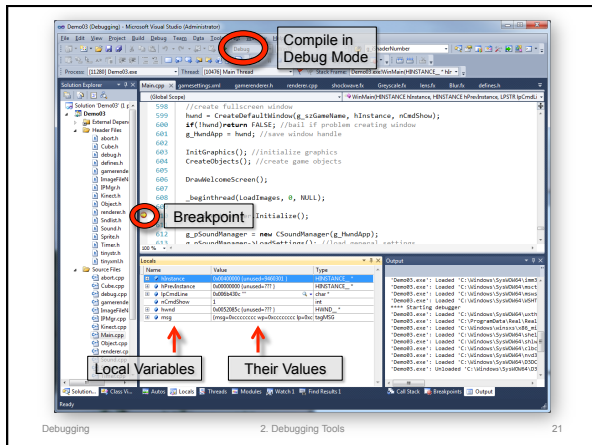
`info breakpoints` - shows information about all breakpoints

For more information, RTFM.

## Debugging in Visual Studio



- Compile in Debug mode (equivalent to `-g`)
- Set breakpoints by right-clicking on a line in the code and selecting `Breakpoint` from the menu.
- When you run your program, the IDE will stop your code at the breakpoint and display local variables, etc.

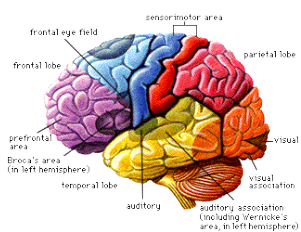


## Debugging Hardware

Closed platforms such as game consoles have development kits with a custom debug version of the hardware.



## The Primary Debugging Tool



## DEBUGGING WITH PRINTF

## How to Debug a Crashed Program



- What do you do when your program crashes?
- Experiment with your program and **think**.
- Try to get some idea about where in the code the crash occurs, if you can.
- The first task is to reproduce the bug - find a series of actions that is guaranteed to make the bug occur every time.
- That should give you some clue as to where the bug might be.


Debugging

3. Debugging with `Printf`

25

## Aside: Reproducing the Bug



- Reproducing bugs can be difficult.
- Some bugs are not easily reproducible.
- In your professional life you may have a *Quality Assurance* (abbreviated QA) team tasked with finding bugs. 
- But they may not tell you how to reproduce it.
- Knowing how to reproduce it may tell you enough about the bug to figure out what's causing it.

Debugging

3. Debugging with `Printf`

26



## How to Debug... Continued

- Start by adding `printf`s that output messages on function entry and exit.
- Do this for the suspicious functions - or all of them if you have to.
- Look at the output file after your program crashes. If you see an `Entering function foo()` message and no `Exiting function foo()` message, then you know where the crash occurred.

Debugging

3. Debugging with `Printf`

27



## How to Debug... Continued

- When you've found the function in which the crash occurred, add code to localize on which line it happened.
- When you've found the line of code that's bad, add code that prints out the values used on that line before you get to it.
- Look at the values output immediately before the crash. **Think Hard**. Are they right? If not, what should they be? And how did they get to be bad?

Debugging

3. Debugging with `Printf`

28

## More Quotes

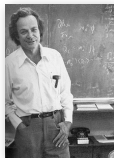
### The Feynman Problem Solving Algorithm:

1. Write down the problem.
2. Think real hard.
3. Write down the solution.

**Albert Einstein:** It's not that I'm so smart, it's just that I stay with problems longer.

**Voltaire:** No problem can stand the assault of sustained thinking.

**Norman Vincent Peale:** When a problem comes along, study it until you are completely knowledgeable. Then find that weak spot, break the problem apart, and the rest will be easy.



Debugging

3. Debugging with `Printf`

29



## Debug Output

- Output to the debugger is good. But what if your program crashes the debugger? Or the OS?
- Output to a file is good. But it can be annoying to have to reopen it every time after you run your program.
- Neither of these is real-time.
- Output to a remote debugger on another computer is good. You get output in real time. But it is ephemeral.
- You'll want to use all three at some point in your professional life.

Debugging

3. Debugging with `Printf`

30

## Warnings



- There are some bugs that can't be caught with debug `printf`s - in particular, bugs in timing and scheduling of multithreaded processes.
- Adding debug output will slow down your program and change its execution profile, which may make the bug go away. Try using as few debug outputs as possible.

Debugging

3. Debugging with `printf`

31

## Segmentation Faults

- Your program is trying to access a location in memory that it can not access.
- Often segfaults are related to NULL pointers:
  - e.g. try to assign a value to a non-existing object  
`int *some_pointer = NULL;`  
`*some_pointer = 5;`
- A recursive program without a basis case will eventually cause a stack overflow, i.e. it tries to use more memory than has been allocated to it.

Debugging

1. Introduction

32

## What To Do About Segfaults

- Start adding “print” (or “cout” for C++) statements backwards from the point where the segfault occurred to determine the contents of your variables.
- Systematically run your program for “basic” input if applicable.
  - Eventually you will find which variable does not contain the expect content.
  - Keep backtracking from there until you find the code where the problem is caused.

Debugging

1. Introduction

33

Section 4

## OTHER DEBUGGING METHODS

Debugging

4. Other Debugging Methods

34

## Defensive Programming

- Test for pre-conditions even when you know they're going to be true ('cos you made it so).
- Write your code in small chunks and debug each chunk before moving on to the next one.
- Keep old versions. Use a Revision Control System (RCS) to keep track of them.



Debugging

4. Other Debugging Methods

35

## Diff

```
183  
184 // Compose menu frame.  
185 // Compose a frame of menu screen animation. This is just the but  
186 // now, but we could add a glitzy animation if we wished.  
187  
188 ID: BDDI CGameRenderer::ComposeMenuFrame(Y, //Align menu screen  
189 n_d3device->BeginScene(); //set vertex buffer to background  
190 n_d3device->SetStreamSource(0, n_background, 0, sizeof(D3D  
191 n_d3device->SetFVF(D3DFVF_VERTEX); //flexible vertex i  
192 n_d3device->SetTexture(0, m_texture); //set menutextu  
193 SetWorldMatrix(0, 0f);  
194 SetViewMatrix((float)g_nScreenWidth, (float)g_nScreenHeight,  
195 n_d3device->DrawPrimitive(D3DPT_TRIANGLESTRIP, 2, 2);  
196 g_pInputManager->DrawButtons(); //draw buttons on screen  
197 n_d3device->EndScene();  
198 return TRUE;  
199 } //ComposeMenuFrame  
200  
201 // Process a frame of animation of menu screen.
```

Debugging

4. Other Debugging Methods

36

## Diff Makes all the Difference

- `diff` is a Unix utility that compares two text files and tells you where they differ.
- Run `diff` on the latest two versions of your code to see what has changed.
- Selectively comment out new code until the bug goes away.
- This can take quite a lot of time.



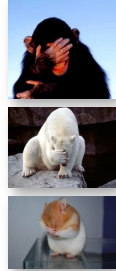
Debugging

4. Other Debugging Methods

37

## The Facepalm Method: Social Debugging

- Walk through your code line by line with another person, explaining it as you go along.
- Nine times out of ten you will spot the bug yourself and be horribly embarrassed by it.
- Never underestimate the power of embarrassment as a debugging tool.
- This works best with somebody you would prefer to impress.



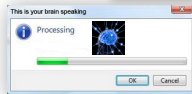
Debugging

4. Other Debugging Methods

38

## Distractions

- During the long and arduous debugging process, you will be tempted to stop for coffee, procrastinate, go on Facebook, do other things.
- Your manager will probably think that you aren't making any actual progress.
- However, the unconscious part of your brain is *always* on the job.
- Keeping the conscious part of your brain distracted can actually help the unconscious part work on the problem.



Debugging

4. Other Debugging Methods

39

## It's Elementary

- Emulate Sherlock Holmes: Gather evidence and use logic.
- "When you have eliminated the impossible, whatever remains, *however improbable*, must be the truth."
- You are a truly a professional programmer when your bugs take:
  - 3 days to find
  - 30 seconds to fix



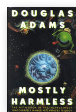
Debugging

4. Other Debugging Methods

40

## Boundary Conditions

We all like to congregate at boundary conditions. Where land meets water. Where earth meets air. Where bodies meet mind. Where space meets time. We like to be on one side, and look at the other.



Douglas Adams  
*Mostly Harmless*

Debugging

4. Other Debugging Methods

41

## Types of Boundary Condition

Bugs like to collect around boundary conditions, so start looking there first. These include:

- The first time around a loop.
- The last time around a loop.
- The code following a loop.
- The code at the start of a function.
- The code after a function returns.



Debugging

4. Other Debugging Methods

42



## Pre- and Post-Conditions

For each unit of code (block, function, loop):

- Are the pre-conditions met? That is, are the conditions required for its correct execution hold at the time of execution?
- Does it meet the post-conditions? That is, does it meet the conditions required for the correct execution of subsequent code?

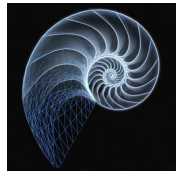
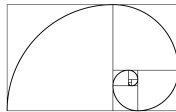
## The Trace

- Grab a pencil and paper and pretend to be a computer executing your program.
- It is a method of desperation because it is so time-consuming and tedious.
- Let's try it on a simple function to compute Fibonacci numbers.



## Fibonacci Numbers

- The first two Fibonacci numbers are 0 and 1.
- From then on, the next Fibonacci number is the sum of the previous two.  
0, 1, 1, 2, 3, 5, 8, 13...
- Count them starting at 0, so for example, the 4<sup>th</sup> Fibonacci number is 3.



## Computing Fibonacci Numbers

```
int fib(int n){
    //return nth Fibonacci number
    if(n <= 0) return 0; else{
        int a = 0, b = 1, c;
        for(int i = 2; i <= n, i++){
            c = a + b; a = b; b = c;
        } //for
        return b;
    } //else
} //fib(n)
```

## Trace of Fibonacci Numbers Code

```
int fib(int n){
    if(n<=0) return 0;
    else{
        int a=0, b=1, c;
        for(int i=2; i<=n, i++){
            c=a+b; a=b; b=c;
        } //for
        return b;
    } //else
} //fib(n)
```

n

a

b

c

i

return

## Trace of Fibonacci Numbers Code

```
int fib(int n){
    if(n<=0) return 0;
    else{
        int a=0, b=1, c;
        for(int i=2; i<=n, i++){
            c=a+b; a=b; b=c;
        } //for
        return b;
    } //else
} //fib(n)
```

n

a

b

c

i

return



### Trace of Fibonacci Numbers Code

```
int fib(int n){
if(n<=0) return 0;
else{
int a=0, b=1, c;
for(int i=2; i<=n, i++){
c=a+b; a=b; b=c;
} //for
return b;
} //else
} //fib(n)
```

n

a

b

c

i

return

### Trace of Fibonacci Numbers Code

```
int fib(int n){
if(n<=0) return 0;
else{
int a=0, b=1, c;
for(int i=2; i<=n, i++){
c=a+b; a=b; b=c;
} //for
return b;
} //else
} //fib(n)
```

n

a

b

c

i

return

### Trace of Fibonacci Numbers Code

```
int fib(int n){
if(n<=0) return 0;
else{
int a=0, b=1, c;
for(int i=2; i<=n, i++){
c=a+b; a=b; b=c;
} //for
return b;
} //else
} //fib(n)
```

n

a

b

c

i

return

### Trace of Fibonacci Numbers Code

```
int fib(int n){
if(n<=0) return 0;
else{
int a=0, b=1, c;
for(int i=2; i<=n, i++){
c=a+b; a=b; b=c;
} //for
return b;
} //else
} //fib(n)
```

n

a

b

c

i

return

### Trace of Fibonacci Numbers Code

```
int fib(int n){
if(n<=0) return 0;
else{
int a=0, b=1, c;
for(int i=2; i<=n, i++){
c=a+b; a=b; b=c;
} //for
return b;
} //else
} //fib(n)
```

n

a

b

c

i

return

### Trace of Fibonacci Numbers Code

```
int fib(int n){
if(n<=0) return 0;
else{
int a=0, b=1, c;
for(int i=2; i<=n, i++){
c=a+b; a=b; b=c;
} //for
return b;
} //else
} //fib(n)
```

n

a

b

c

i

return

### Trace of Fibonacci Numbers Code

```
int fib(int n){
  if(n<=0) return 0;
  else{
    int a=0, b=1, c;
    for(int i=2; i<=n, i++){
      c=a+b; a=b; b=c;
    } //for
    return b;
  } //else
} //fib(n)
```

n   
a   
b   
c   
i

return

### Trace of Fibonacci Numbers Code

```
int fib(int n){
  if(n<=0) return 0;
  else{
    int a=0, b=1, c;
    for(int i=2; i<=n, i++){
      c=a+b; a=b; b=c;
    } //for
    return b;
  } //else
} //fib(n)
```

n   
a   
b   
c   
i

return

### Trace of Fibonacci Numbers Code

```
int fib(int n){
  if(n<=0) return 0;
  else{
    int a=0, b=1, c;
    for(int i=2; i<=n, i++){
      c=a+b; a=b; b=c;
    } //for
    return b;
  } //else
} //fib(n)
```

n   
a   
b   
c   
i

return

### More Debugging Methods

- Exceptions
- Assertions
- Logging
- Post-mortem

