

CSE 374 Programming Concepts and Tools

Lecture 11 - Debugging, GDB

Last week

C pointers

- Data representations
- Memory
- Pointers

Dynamic Memory Allocation

- `malloc` and `free`

Memory Leaks

- Valgrind

Today

Today: Debugging, GDB

- Debugging 
- Debugging C specifically     
- GDB + Demo

Next week:

- Struct, Compiler, Makefile
- C Data Structures
- More testing!

Review: Common Memory Errors

- Dereferencing a non-pointer
- Accessing freed memory
- Double free
- Out-of-bounds access
- Reading memory before initialization
- Wrong allocation size
- Forgetting to free memory ("memory leak")

Review: Finding and Fixing Memory Errors

Valgrind is a tool that simulates your program to find memory errors

It can detect **all** of the memory errors we talked about! 🤖

It catches pointer errors during execution – prints summary of heap usage, including details of memory leaks

```
valgrind [options] ./myprogram args args...
```

- Useful option: `--leak-check=full`
 - Displays more details about each memory leak

Debugging

What is a Bug?

A bug is a difference between the design of a program and its implementation

- (From [Ko & Meyers \(2004\)](#))
- Edison is known to have used this term all the way back in the [1870s](#).
 - “Little faults and difficulties”

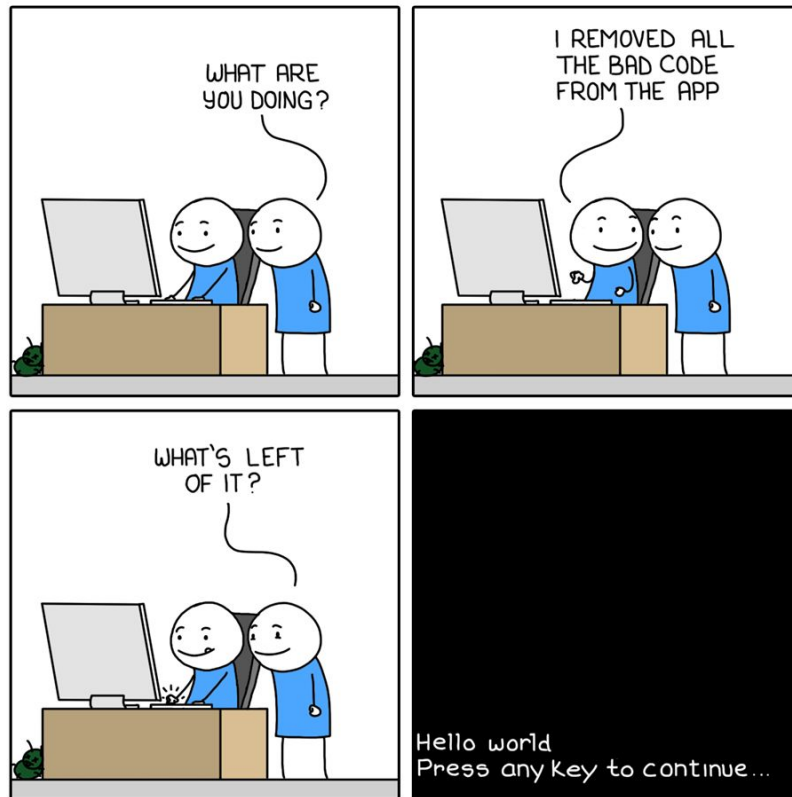
In other words, we expected something different from what is happening.

Examples of bugs:

- Expected factorial(5) to be 120, but it returned 0
- Expected program to finish successfully, but crashed and printed "segmentation fault"
- Expected normal output to be printed, but instead printed strange symbols

How do you avoid debugging? Avoid writing code!

BUG FREE



MONKEYUSER.COM

Discussion

Turn and talk:

What are some bugs or challenges you have faced while implementing HW3?

(If you have not started, what bugs do you anticipate you might encounter?)

How Do We Debug?

Basic outline of what debugging and problem solving looks like:

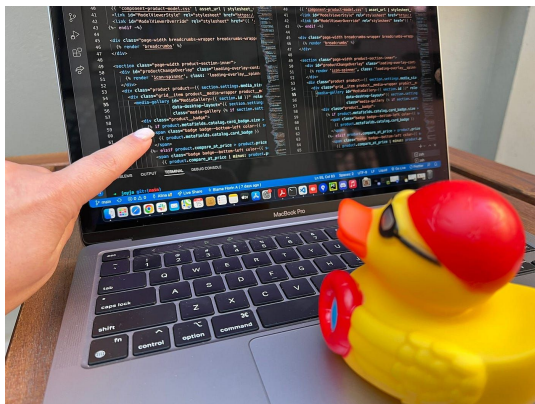
1. Identify and describe the difference between expected and actual behavior
2. Hypothesize possible causes
3. Investigate possible causes, test theories
 - a. If not found, your hypothesis may be incorrect -- go back to step 2
4. Fix the code which was causing the bug

Vast majority of the time spent in steps 2 & 3. **Document** what you did!

Rubber Duck

Some other strategies:

- Rubber duck debugging (originates from The Pragmatic Programmer)
 - Explain to problem to yourself/a rubber duck
- Take a break
 - Sometimes walking away, eating something, taking a nap can help you figure it out!



Polling Time!

Please answer in the Poll: Lec 11 [assignment on Gradescope!](#)

Part of debugging is learning how to describe the problem you're having, which you can do without even seeing the code.

Which of these bug descriptions do you think is best? (No wrong answers!)

- ~~factorial() does not return correct output~~
- ~~factorial() always returns 0~~
- factorial(5) does not return the correct number
- ☒ factorial(5) returns 0

Demo: factorial

Please answer in the Poll [assignment on Gradescope!](#)

Hypothesize

Now, let's look at the code for
`factorial()`

Select *all* the places where the error *could*
be coming from

- The `if` condition
- The code within the `if` branch
- The code within the `else` branch
- Somewhere else

```
int factorial(int x) {  
    if (x == 0) {  
        return x;  
    } else {  
        return x * factorial(x - 1);  
    }  
}
```

Investigate

Let's investigate the base case and recursive case

- Base case is the "if" branch
- Recursive case is the "else" branch

```
int factorial(int x) {  
    if (x == 0) {  
        return x;  
    } else {  
        return x * factorial(x-1);  
    }  
}
```

Case	Input	Math Equivalent	Expected	Actual
Base	factorial(0)	$0! = 1$	1	???
Recursive	factorial(1)	$1! = 1$	1	???
Recursive	factorial(2)	$2! = 1 * 2$	2	???
Recursive	factorial(3)	$3! = 1 * 2 * 3$	6	???

Investigate - Testing

One way to investigate is to write code to test different inputs

If we do this, we find that the base case has a problem

```
int factorial(int x) {  
    if (x == 0) {  
        return x;  
    } else {  
        return x * factorial(x-1);  
    }  
}
```

Case	Input	Math Equivalent	Expected	Actual
Base	factorial(0)	$0! = 1$	1	0
Recursive	factorial(1)	$1! = 1$	1	0
Recursive	factorial(2)	$2! = 1 * 2$	2	0
Recursive	factorial(3)	$3! = 1 * 2 * 3$	6	0

Fix

```
int factorial(int x) {  
    if (x == 0) {  
        return x;  
    } else {  
        return x * factorial(x-1);  
    }  
}
```

```
int factorial(int x) {  
    if (x == 0) {  
        return 1;  
    } else {  
        return x * factorial(x-1);  
    }  
}
```

Case	Input	Math Equivalent	Expected	Actual
Base	factorial(0)	$0! = 1$	1	1
Recursive	factorial(1)	$1! = 1$	1	1
Recursive	factorial(2)	$2! = 1 * 2$	2	2
Recursive	factorial(3)	$3! = 1 * 2 * 3$	6	6

Testing

Pros

- **Prevent bugs** by testing early and often
- Forces you to think about edge cases
- Helps solidify your mental model of what a program should be doing

Cons

- Time consuming to write
- Only as good as the tests you write
- Doesn't necessarily guarantee correctness, can only detect incorrectness

We'll practice writing tests in HW4!

Debugging in C

What are some debugging techniques for C?

Debugging in C

C is unsafe

- No bounds checking
- Manual memory management
 - Null pointers
 - Double free
- Missing null terminator `\0` in strings
- Unsafe casting

Vague error messages

Java

```
Exception in thread "main"  
java.lang.NullPointerException:  
Cannot invoke "String.length()"   
because "name" is null  
    at Main.main(Main.java:4)
```

C

Segmentation fault (core dumped)



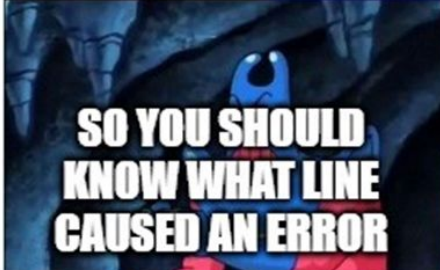
Sometimes it will pretend there's no error



YOU ARE C



YUP



**SO YOU SHOULD
KNOW WHAT LINE
CAUSED AN ERROR**



YUP



THAT'S GOOD



SO WHAT'S MY ERROR?



**SEGMENTATION
FAULT**

C Debugging Techniques (1/2)

Add print statements: `printf("Input string: %s\n", line);`

- Check if certain code is reachable
- Check current state of variables

Comment out (or delete) code: `// isolate bugs`

- Tests to determine whether removed code was source of problem
- Test one function at a time
- [Delta Debugging](#) ★

Write tests ★ (We'll do this in HW5!)

- Unit tests = test of input and output of singular code modules
- Often many tests to one function
 - Edge cases: empty string, `NULL`, extreme values, ...

C Debugging Techniques (2/2)

Read documentation (and in our case HW specs) ★

- Software engineering: Review the spec for what you're trying to implement!
- Research: Design the spec and code in tandem, revising the spec as we go

Type errors/warnings into Google ★

- `gcc -Wall` will show you all compiler warning output

Use a debugger ★

- Lets you control program execution line by line
- Lets you see current state of variables
- *Don't expect the debugger to do the work, use it as a tool to test theories*
- In C: **GDB**

GDB

GDB

GDB = **G**NU Project **d**ebugger

- Standard component of Linux
- Supports multiple languages
- Runs as a CLI

```
Breakpoint 1, factorial (x=0) at factorial.c:19
19         return x;
(gdb) p x
$1 = 0
(gdb) c
Continuing.
5 factorial is 0
[Inferior 1 (process 1570668) exited normally]
(gdb) quit
```

What can you do with GDB?

- Control program execution
 - Pause
 - Resume
 - Step through your code line-by-line
- Observe program behavior
 - Print variables
 - Examine memory locations
- Modify memory
 - Change variable/values store in memory

Compiling code for debugging

Turn on all warnings

Output file (executable)

```
gcc -g -Wall -std=c11 -o executable_name c_file
```

Generate flags for
GDB

C standard
version

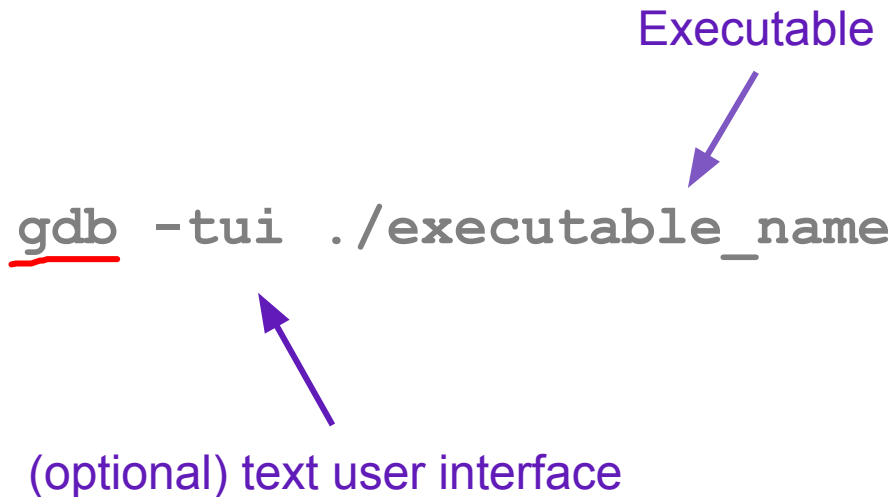
Source file

Running program in GDB

gdb -tui ./executable_name

Executable

(optional) text user interface



Concepts in debugger

Breakpoint

`break <line number/fxn name>`

Pause program execution at a certain line.



Continue

`continue`

Resume program execution until exit/error/next breakpoint.

```
1  /* Include C libraries*/
2  void foo();
3
4  int main(int argc, char* argv[]) {
5      printf("hi there\n");
6      foo();
7      printf("bye\n");
8      exit(0);
9  }
10
11 void foo() {
12     // In foo function
13     printf("foo\n");
14     printf("done\n");
15 }
```

Concepts in debugger

We can also step through the program line by line...

Step into

step

Run the current line (and if it's a function call, go inside that function).

Ex: go into the `foo()` function



```
1  /* Include C libraries*/
2  void foo();
3
4  int main(int argc, char* argv[]) {
5      printf("hi there\n");
6      foo();
7      printf("bye\n");
8      exit(0);
9  }
10
11 void foo() {
12     // In foo function
13     printf("foo\n");
14     printf("done\n");
15 }
```

Concepts in debugger

We can also step through the program line by line...

Step out

`finish`

Finish the current function call.
Pause at the line after function call.



```
1  /* Include C libraries*/
2  void foo();
3
4  int main(int argc, char* argv[]) {
5      printf("hi there\n");
6      foo();
7      printf("bye\n");
8      exit(0);
9  }
10
11 void foo() {
12     // In foo function
13     printf("foo\n");
14     printf("done\n");
15 }
```

Concepts in debugger

We can also step through the program line by line...

Step over

next

Run the current line (and if it's a function call, run through its entirety).

Pause at the line after function call.



```
1  /* Include C libraries*/
2  void foo();
3
4  int main(int argc, char* argv[]) {
5      printf("hi there\n");
6      foo();
7      printf("bye\n");
8      exit(0);
9  }
10
11 void foo() {
12     // In foo function
13     printf("foo\n");
14     printf("done\n");
15 }
```

GDB Most Important Commands (1/2)

<code>gdb [-tui] ./executable_name</code>	Start GDB
<code>run [args] ...</code>	Run the program with the given arguments
<code>start</code>	Run the program & pause at main function
<code>quit</code>	Quit GDB
<code>tui enable/disable</code>	See the code while debugging
<code>refresh</code>	Redraw and refresh the screen for the test UI (TUI).
<code>break <function name></code> <code>break <file name:line number></code>	Set a breakpoint on a certain line or function
<code>list</code>	Display source code (list <line number>, list <function name>)
<code>info locals</code>	Print all locals (but not parameters)

GDB Most Important Commands (2/2)

<code>next</code>	Move to the next line, skipping over function calls
<code>step</code>	Move to the next line, going into function calls
<code>continue</code>	Resume execution until the next breakpoint
<code>clear</code>	Remove breakpoints
<code>print [expr]</code>	Print value for expression
<code>display [expr]</code>	Re-evaluate and print expression every time execution pauses. <code>undisplay</code> to remove an expression from list.
<code>x [expr]</code>	Examine memory at address <code>expr</code> in various formats.
<code>backtrace</code>	Print the functions that were called to get here. Show the call stack
<code>ctrl-c</code>	Terminate the running program but stay in GDB

reverse.c

Reverses an input string that it reads from terminal.

Expected behavior:

- Input string: hello
- Output string: olleh

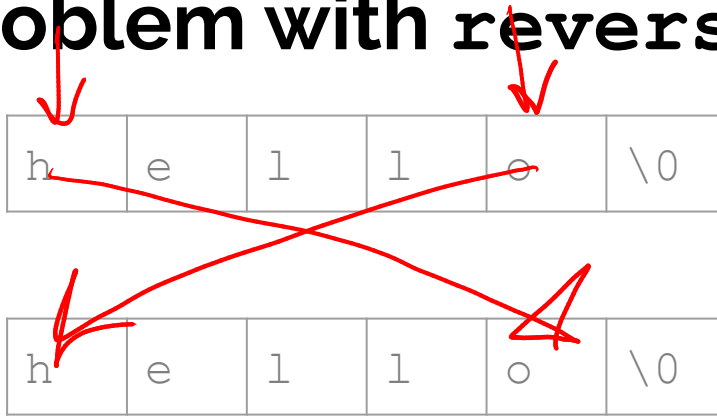
```
// Ask for a string, print it forwards & backwards.
int main() {
    ...
    rev_line = reverse(line);
    ...
    return 0;
}

char* reverse(char* s) {
    char* result = NULL; // the reversed string
    int L, R;
    // copy original string then reverse and return the
    copy
    strcpy(result, s);
    L = 0;
    R = strlen(result);
    while (L < R) {
        char temp = result[L];
        result[L] = result[R];
        result[R] = temp;
        L++; R--;
    }
    return result;
}
```

Demo: debug reverse .c

The Problem with `reverse.c`

Input



h	e	l	l	e	\0
---	---	---	---	---	----

Output

h	e	l	l	o	\0
---	---	---	---	---	----

What gets printed out after reversing?

The Problem with `reverse.c`

Input

h	e	l	l	o	\0
---	---	---	---	---	----

Output

h \0	e o	l l	l l	o e	\0 h
-----------------	----------------	----------------	----------------	----------------	-----------------

Output is an empty C string. Zero characters followed by a null terminator.

Debugging Segmentation Fault

If we get a segmentation fault:

1. Compile with debugging symbols using

```
gcc -g -Wall -std=c11 -o executable_name c_file
```

1. **gdb ./executable_name**

2. Type "**run**" into GDB

3. When you get a segfault, type "**backtrace**" or "**bt**"

4. Look at the line numbers from the backtrace, starting from the top

OR: **valgrind -leak-check=full ./executable_name**

Assignment Reminders

Ex9 due before Wed morning

- GDB tutorial

HW3 due this Friday