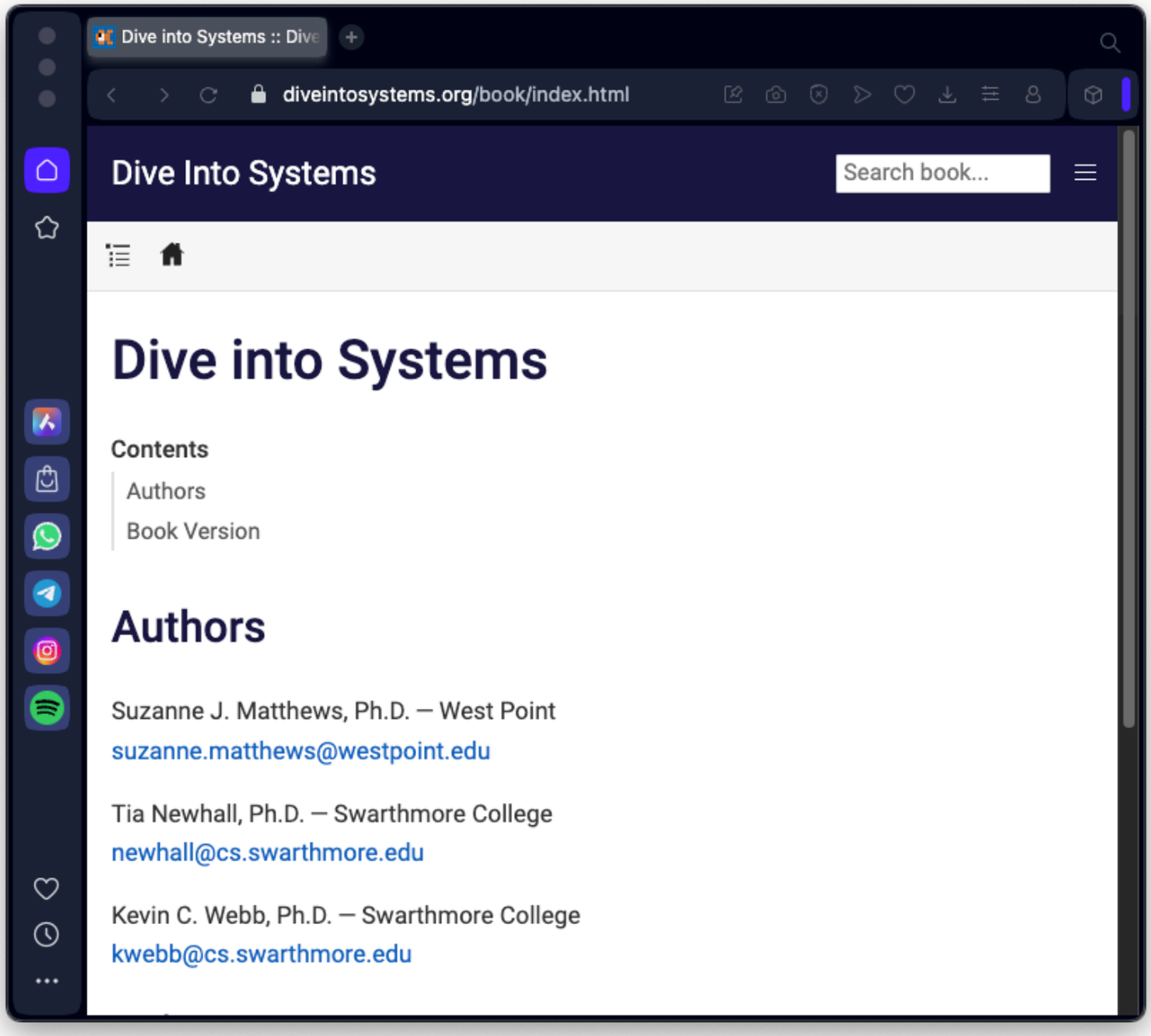
3. C Debugging Tools

For COMSC 142

Sam Bowne

Jan 20, 2025

Free online textbook



- https://diveintosystems.org/book/index.html

Topics

3.1. Debugging with GDB

3.2. GDB Commands in Detail

3.3. Debugging Memory with Valgrind

3.4. Advanced GDB Features

3.5. Debugging Assembly Code

3.6. Debugging Multi-threaded Programs

3.1. Debugging with GDB

GDB Actions

- Start a program and step through it line by line
- Pause the execution of a program when it reaches certain points in its code
- Pause the execution of a program on user-specified conditions
- Show the values of variables at the point in execution that a program is paused
- Continue a program's execution after a pause
- Examine the program's execution state at the point when it crashes
- Examine the contents of any stack frame on the call stack

GDB Features

- **Breakpoints**
  - Stop execution so the user can examine variables and memory
- **Data Display Debugger** (DDD)
  - GUI wrapper around a command-line debugger program (GDB, for example)

3.1.1. Getting Started with GDB

- Compile with the **-g** or **-g3** option
  - **gcc -g**
  - Adds debugging information to the binary executable
    - Allows source code debugging
- Avoid compiler optimizations
  - Omit **-O2**

GDB Commands

Command	Description
`break`	Set a breakpoint
`run`	Start program running from the beginning
`cont`	Continue execution of the program until it hits a breakpoint
`quit`	Quit the GDB session
`next`	Allow program to execute the next line of C code and then pause it
`step`	Allow program to execute the next line of C code; if the next line contains a function call, step into the function and pause
`list`	List C source code around pause point or specified point
`print`	Print out the value of a program variable (or expression)
`where`	Print the call stack
`frame`	Move into the context of a specific stack frame

Demo: badprog

```
/*************************************************/
int main(int argc, char *argv[]) {

    int arr[5] = { 17, 21, 44, 2, 60 };

    int max = arr[0];

    if ( findAndReturnMax(arr, 5, max) != 0 ) {
        printf("strange error\n");
        exit(1);
    }
    printf("max value in the array is %d\n", max);

    return 0;
}
```

Demo: badprog

```
/*****************************************
 * this function should find the largest element in the array and
 * "return" it through max
 *      array: array of integer values
 *      len: size of the array
 *      max: set to the largest value in the array
 *      returns: 0 on success and non-zero on an error
 */
int findAndReturnMax(int *array1, int len, int max) {

    int i;

    if (!array1 || (len <=0) ) {
        return -1;
    }
    max = array1[0];
    for (i=1; i <= len; i++) {
        if (max < array1[i]) {
            max = array1[i];
        }
    }
    return 0;
}
```

Demo: badprog

- wget https://samsclass.info/COMSC-142/proj/badprog.c
- gcc -g -o badprog badprog.c
- ./badprog
- Gives wrong answer

Demo: badprog

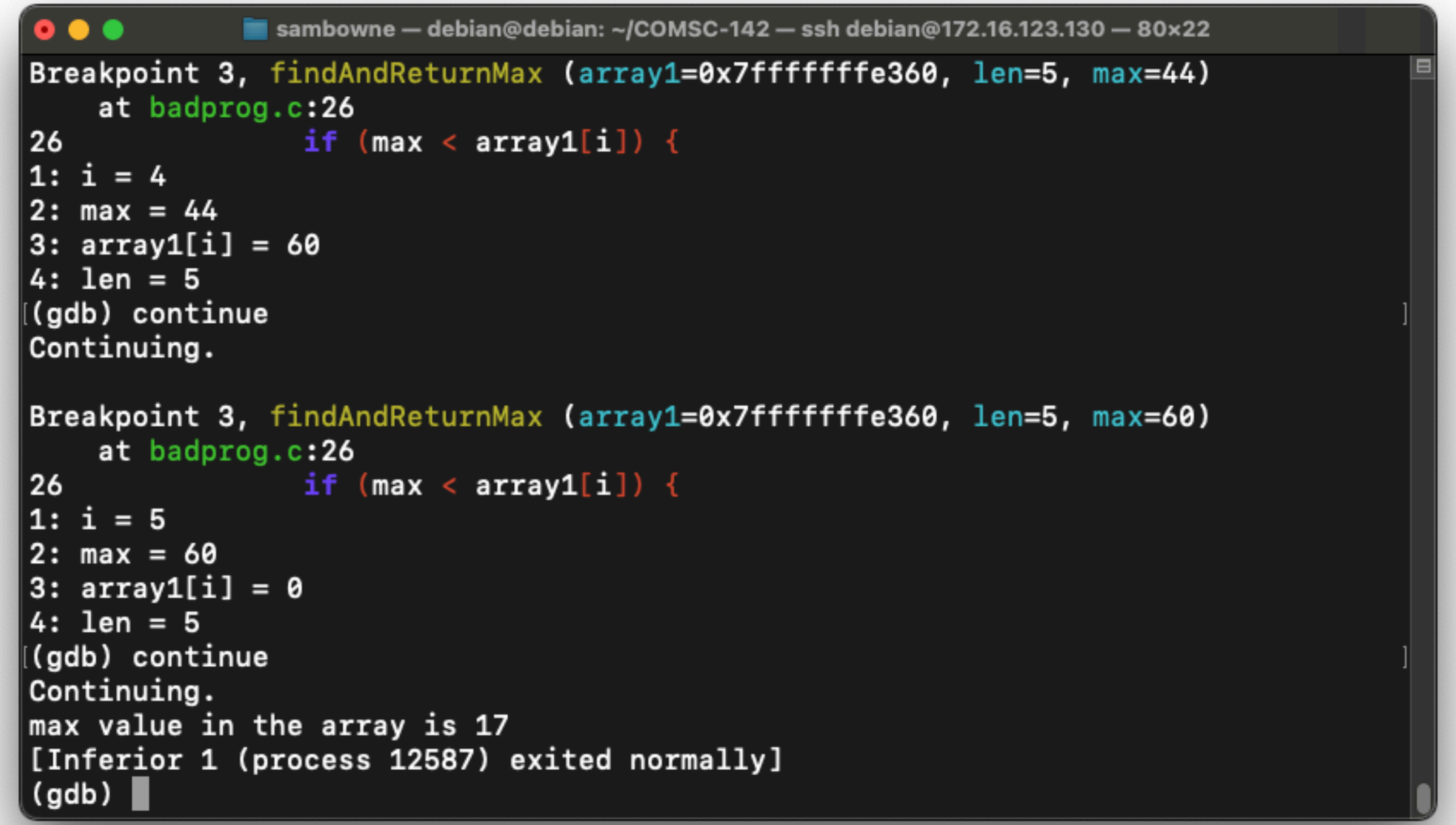
- gdb -q badprog
- break main
- run
- list
- list findAndReturnMax
- next
- print max
- next
- print max
- next
- print max

The max is 17 and never anything else

Demo: badprog

- delete
- y
- break findAndReturnMax
- run
- y
- list findAndReturnMax
- list
- delete
- y
- break 26
- display i
- display max
- display array1[i]
- continue
- continue
- continue
- continue

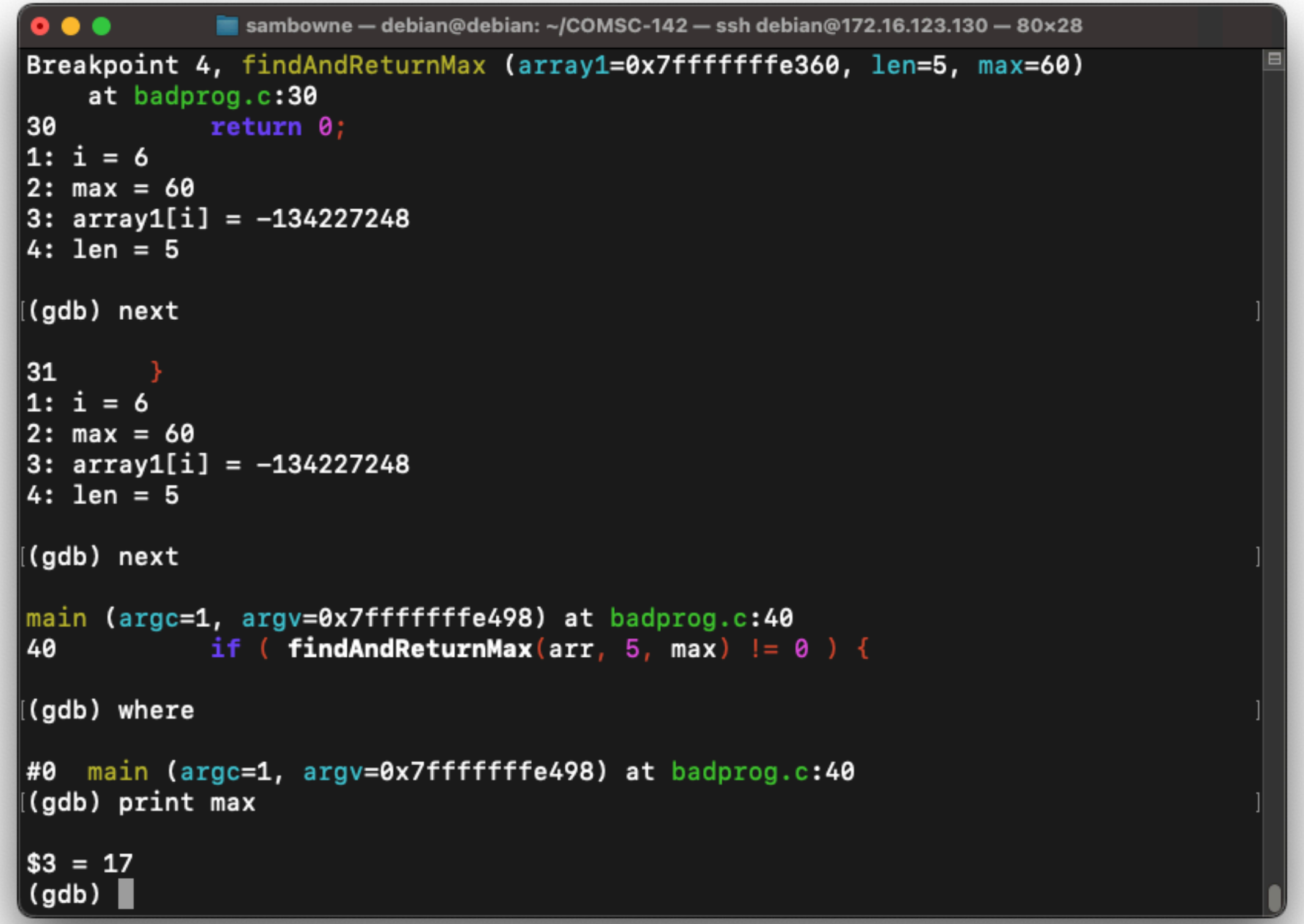
- Loop goes too far, reading past array
- **max** is not passed back to **main**



Demo: badprog

- run
- y
- where
- frame 1
- print arr
- print max
- delete
- y
- break 30
- continue
- print max
- next
- next
- where
- print max

max is not passed to **main**



segfaulter

```
int func(int *array1, int len, int max) {

    int i;

    max = array1[0];
    for (i=1; i <= len; i++) {
        if (max < array1[i]) {
            max = array1[i];
        }
    }
    return 0;
}

int main(int argc, char *argv[]) {

    int *arr = NULL;
    int max = 6;

    if (initfunc(arr, 100) != 0 ) {
        printf("init error\n");
        exit(1);
    }

    if ( func(arr, 100, max) != 0 ) {
        printf("func error\n");
        exit(1);
    }
    printf("max value in the array is %d\n", max);
    exit(0);
}
```

Demo: segfaulter

- wget https://samsclass.info/COMSC-142/proj/segfaulter.c
- gcc -g -o segfaulter segfaulter.c
- ./segfaulter
- gdb -q segfaulter
- run
- where
- list

Demo: segfaulter

- print i
- print array[i]
- print array
- frame 1
- list
- print arr

NULL pointer dereference

Should allocate space for array, not set it to NULL

3.2. GDB Commands in Detail

3.2.1. Keyboard Shortcuts in GDB

- **Tab** completes a command
- **Up** and **Down** keys repeat commands
- **Return** repeats previous command

3.2.2. Common GDB Commands

- **help**
- **break**
- **run**
- **continue**
- **step**
  - Goes to next line of source code
- **next**
  - Goes to next line of source code, treating a function call as one line

3.2.2. Common GDB Commands

- **until**
  - executes until the specified line number
- **quit**
- **list**
- **where**
- **frame**
- **enable, disable, ignore, delete, clear**
  - breakpoints
- **condition**
  - Sets conditional breakpoints

Commands for Examining and Evaluating Program State and Expressions

- **print**
  - prints value one time
- **display**
  - Automatically shows value at each breakpoint
- **x**
  - Examines memory
- **whatis**
  - Shows data type of an expression

Commands for Examining and Evaluating Program State and Expressions

- **set**
  - Sets a variable's value
- **info**
  - Shows program and debugger state
- **help info**

Ch 3a

3.3. Debugging Memory with Valgrind

Memcheck

- Part of valgrind
- Highlights heap memory errors, like:

- Reading (getting) a value from uninitialized memory. For example:

```
int *ptr, x;
ptr = malloc(sizeof(int) * 10);
x = ptr[3];    // reading from uninitialized memory
```

- Reading (getting) or writing (setting) a value at an unallocated memory location, which often indicates an array out-of-bounds error. For example:

```
ptr[11] = 100;  // writing to unallocated memory (no 11th element)
x = ptr[11];    // reading from unallocated memory
```

Memcheck

- Highlights heap memory errors, like:

- Freeing already freed memory. For example:

```
free(ptr);
free(ptr); // freeing the same pointer a second time
```

- Memory leaks. A **memory leak** is a chunk of allocated heap memory space that is not referred to by any pointer variable in the program, and thus it cannot be freed. That is, a memory leak occurs when a program loses the address of an allocated chunk of heap space. For example:

```
ptr = malloc(sizeof(int) * 10);
ptr = malloc(sizeof(int) * 5);  // memory leak of first malloc of 10 ints
```

bigfish

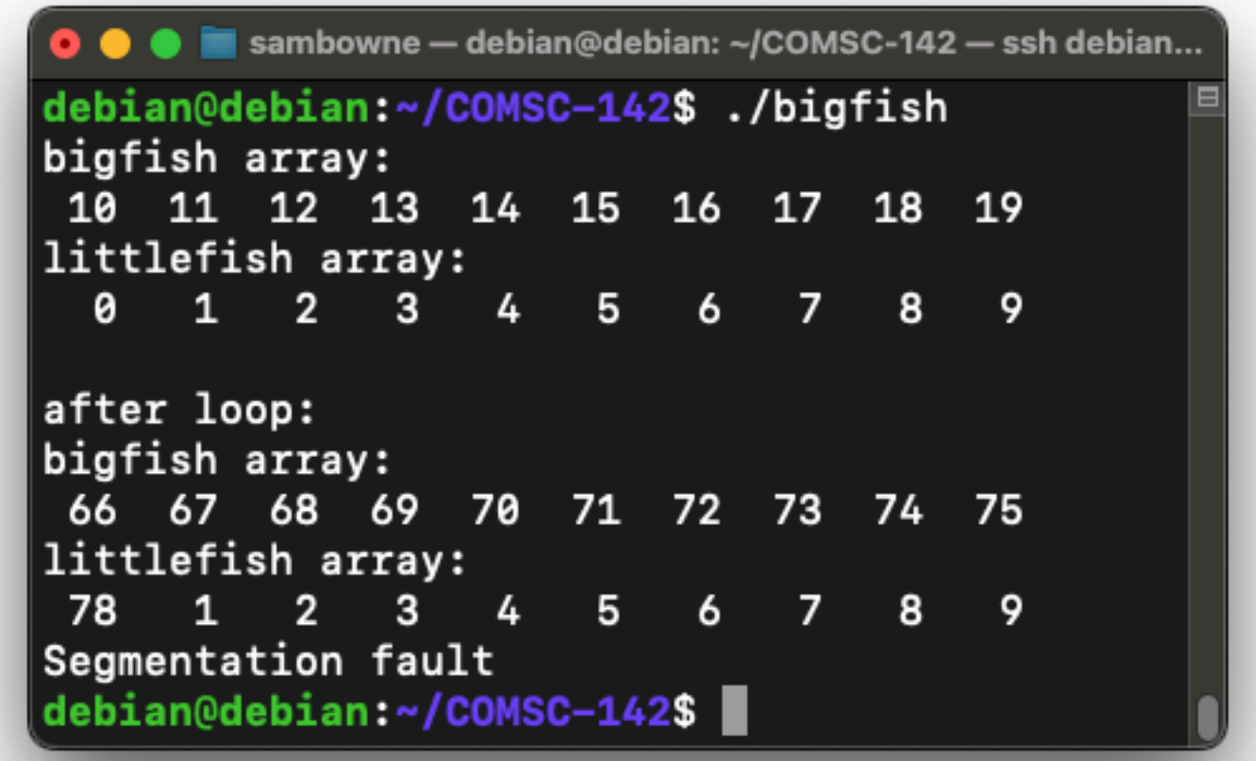
- Doesn't crash at write to **bigfish** array beyond bounds
- Crashes at **free(littlefish)**

```
int main(int argc, char *argv[]) {
    int *bigfish, *littlefish, i;

    // allocate space for two int arrays
    bigfish = (int *)malloc(sizeof(int)*10);
    littlefish = (int *)malloc(sizeof(int)*10);
    if (!bigfish || !littlefish) {
        printf("Error: malloc failed\n");
        exit(1);
    }
    for (i=0; i < 10; i++) {
        bigfish[i] = 10+i;
        littlefish[i] = i;
    }
    print_array(bigfish,10, "bigfish");
    print_array(littlefish,10, "littlefish");

    // here is a bad Heap memory access
    // (write beyond bounds of allocated memory):
    for (i=0; i < 13; i++) {
        bigfish[i] = 66+i;
    }
    printf("\nafter loop:\n");
    print_array(bigfish,10, "bigfish");
    print_array(littlefish,10, "littlefish");

    free(bigfish);
    free(littlefish);  // program will crash here
    return 0;
}
```

Demo: bigfish

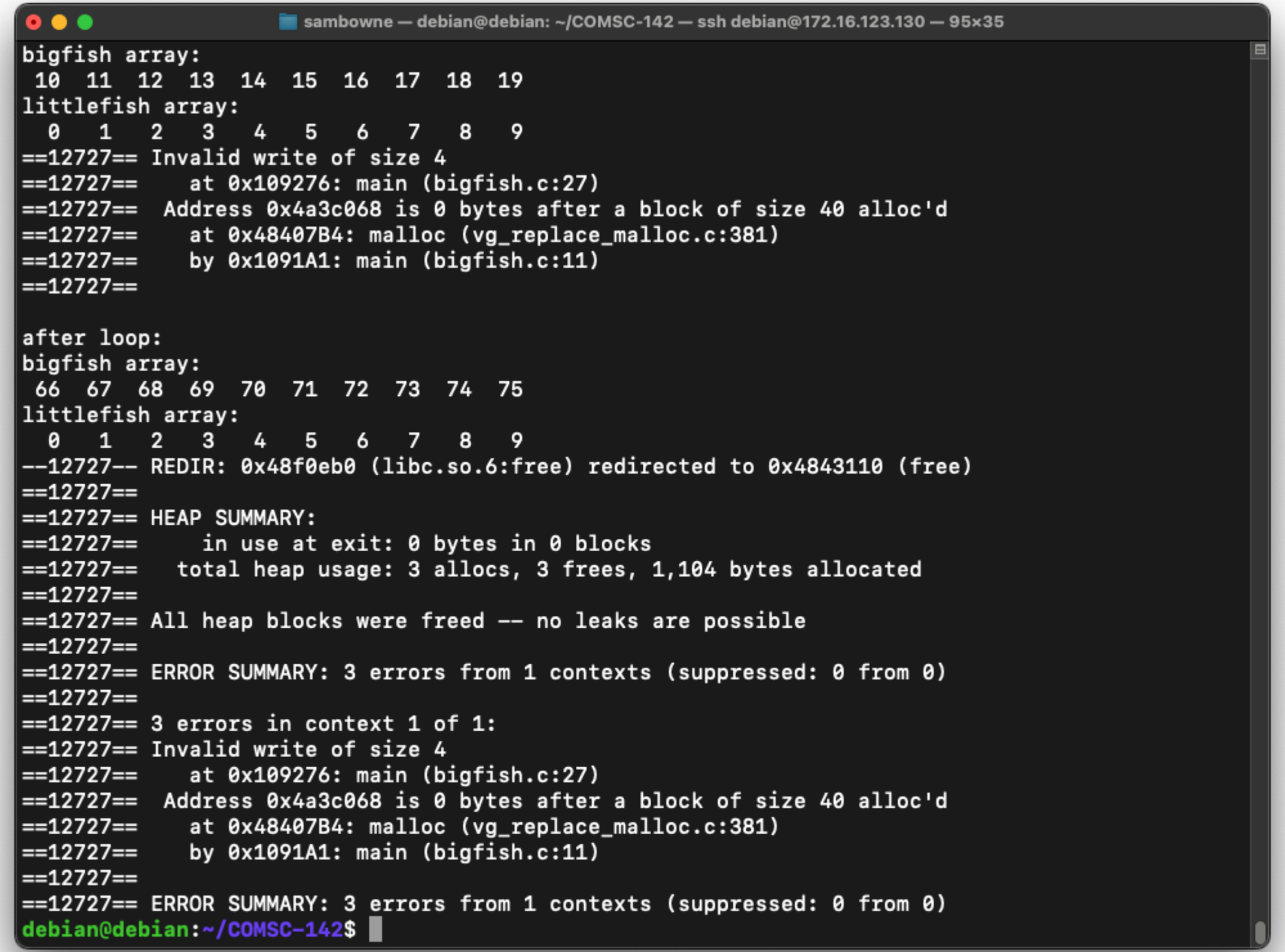
- wget https://samsclass.info/COMSC-142/proj/bigfish.c
- gcc -g -o bigfish bigfish.c
- ./bigfish

- Doesn't crash at write beyond bounds
- Crashes at **free(littlefish)**



Demo: bigfish

- sudo apt install valgrind -y
- valgrind -v ./bigfish


sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 95×35
bigfish array:
 10 11 12 13 14 15 16 17 18 19
littlefish array:
  0   1   2   3   4   5   6   7   8   9
==12727== Invalid write of size 4
==12727==   at 0x109276: main (bigfish.c:27)
==12727== Address 0x4a3c068 is 0 bytes after a block of size 40 alloc'd
==12727==   at 0x48407B4: malloc (vg_replace_malloc.c:381)
==12727==   by 0x1091A1: main (bigfish.c:11)
==12727==
after loop:
bigfish array:
 66 67 68 69 70 71 72 73 74 75
littlefish array:
  0   1   2   3   4   5   6   7   8   9
--12727-- REDIR: 0x48f0eb0 (libc.so.6:free) redirected to 0x4843110 (free)
==12727==
==12727== HEAP SUMMARY:
==12727==   in use at exit: 0 bytes in 0 blocks
==12727==   total heap usage: 3 allocs, 3 frees, 1,104 bytes allocated
==12727==
==12727== All heap blocks were freed -- no leaks are possible
==12727==
==12727== ERROR SUMMARY: 3 errors from 1 contexts (suppressed: 0 from 0)
==12727==
==12727== 3 errors in context 1 of 1:
==12727== Invalid write of size 4
==12727==   at 0x109276: main (bigfish.c:27)
==12727== Address 0x4a3c068 is 0 bytes after a block of size 40 alloc'd
==12727==   at 0x48407B4: malloc (vg_replace_malloc.c:381)
==12727==   by 0x1091A1: main (bigfish.c:11)
==12727==
==12727== ERROR SUMMARY: 3 errors from 1 contexts (suppressed: 0 from 0)
debian@debian:~/COMSC-142$

Skip this section

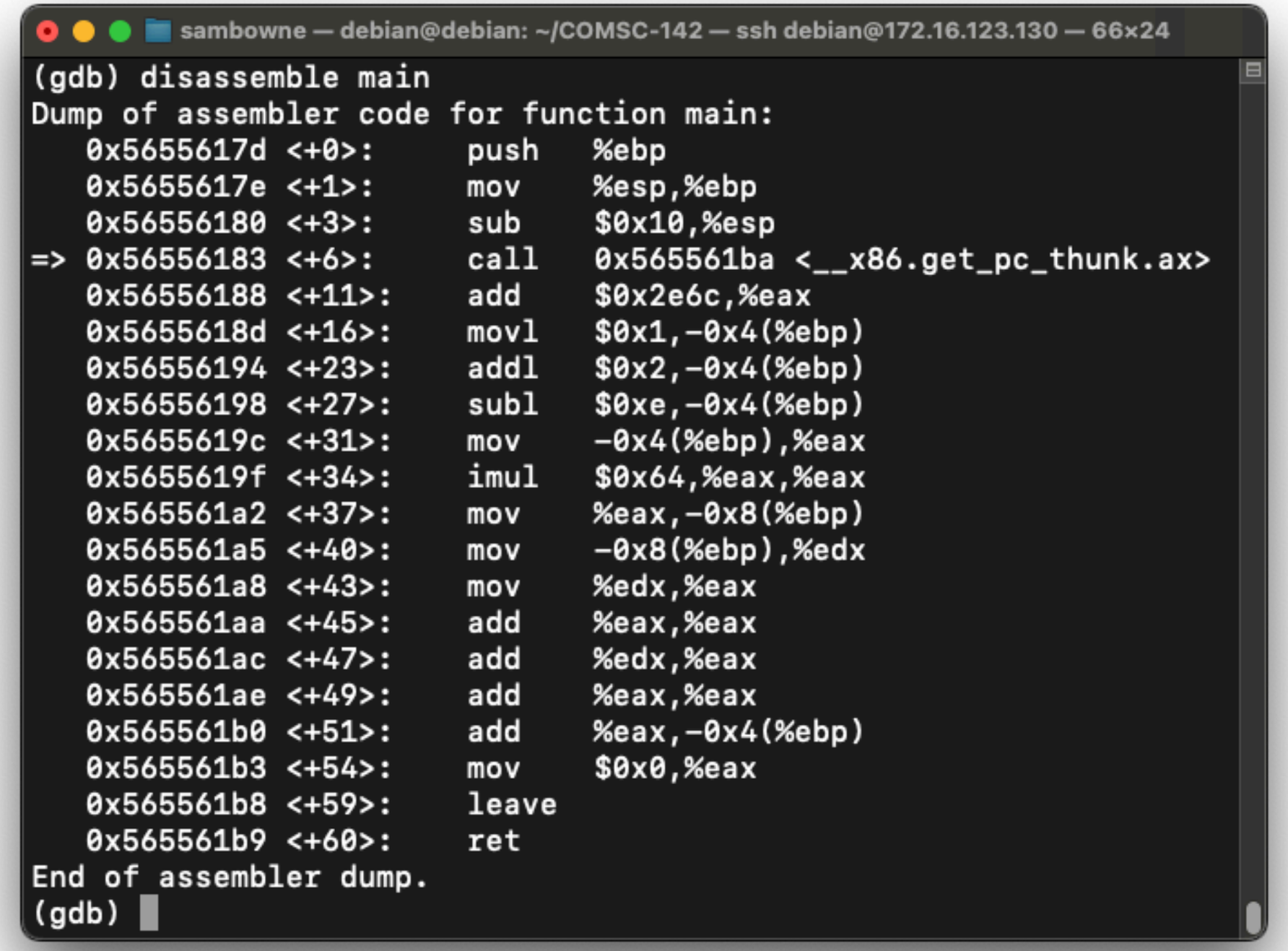
3.4. Advanced GDB Features

3.5. Debugging Assembly Code

Demo: simpleops

- wget https://samsclass.info/COMSC-142/proj/simpleops.c
- sudo apt update
- sudo apt install build-essential gcc-multilib gdb -y
- gcc -m32 -o simpleops simpleops.c
- gdb -q simpleops
- break main
- run
- set style enabled off
- disassemble main

Demo: simpleops



Useful GDB Commands

```
(gdb) break *0x080483c1    # Set breakpoint at instruction at 0x080483c1
```

The program's execution can be executed one assembly instruction at a time using `si` or `ni` to step into or execute the next instruction:

```
(gdb) ni     # Execute the next instruction

(gdb) si     # Execute next instruction; if it is a call instruction,
             # then step into the function
```

Useful GDB Commands

```
(gdb) print $eax    # print the value stored in register eax
```

The `display` command automatically displays values upon reaching a breakpoint:

```
(gdb) display $eax
(gdb) display $edx
```

The `info registers` command shows all of the values stored in the machine registers:

```
(gdb) info registers
```

Graphical Debugger

Use Linux with graphics

- sudo apt install ddd -y
- ddd simpleops

- Failed on Kali
- Not worth the bother

Skip this section

3.6. Debugging Multi-threaded Programs

Ch 3b