15-122 S23 PILOT

DEBUGGING IN C BOOTCAMP

TODAY'S AGENDA:

01.

Print Debugging

How do we make an effective print statement & where do we put them?

02.

Using Valgrind

What do all these errors mean???

03.

Contracts & Test Cases

How do contracts work in C & how do we write good edge cases?

01.

PRINT DEBUGGING

How do we make an effective print statement & where do we put them?

PRINTING IN C

In C, we use the **`printf`** function

Example usage: `printf("%d\n", 15122);`

Note that `printf` can take in more than one argument!

Format specifiers: indicates what “kind” of thing we want printed

Include variables in function call that we want printed

FORMAT SPECIFIERS

TYPE	SPECIFIER	EX. VARIABLE	EX. USAGE
decimal integers	`%d`	`int x = 300;`	`printf("%d\n", x);`
characters	`%c`	`char y = 'a';`	`printf("%c\n", y);`
strings	`%s`	`char* z = "boo";`	`printf("%s\n", z);`

These strings should be NUL-terminated, as seen in lab

SO... *WHAT* DO I PRINT?

Loop index variables:

- **Pros:** tells us which iteration we're at
- **Cons:** can get messy with big loops

Changing variables:

- **Pros:** helps show what's changing
- **Cons:** doesn't tell us where things are changing

Conditional branch indicators:

- **Pros:** catch incorrect if conditions
- **Cons:** doesn't show what's changing

OKAY… *WHERE* DO I PRINT?

Do you have lots of conditions?

A print statement in each “case” can tell you which you’re stepping into

Do you have (small) loops?

A print statement in the beginning of the loop can tell you which iteration you’re in

Are you modifying a value?

If you’re unsure a value is being modified correctly, printing it **before** and **after** you modify it tells you if your changes are right

EXAMPLE 1: FIBONACCI [PRINT]

Take a look at the **FILE**: `ex1.c`

There’s ONE **BUG** in the `fib` function

TA STEP-THROUGH

EXAMPLE 2: FIZZEDBUZZED [PRINT]

Take a look at the **FILE**: `ex2.c`

There's A FEW **BUGS** in the `fizzed_and_buzzed` function

Try putting in print statements to see what's going on!

[10 MINS]

REMINDER: WHAT & WHERE TO PRINT

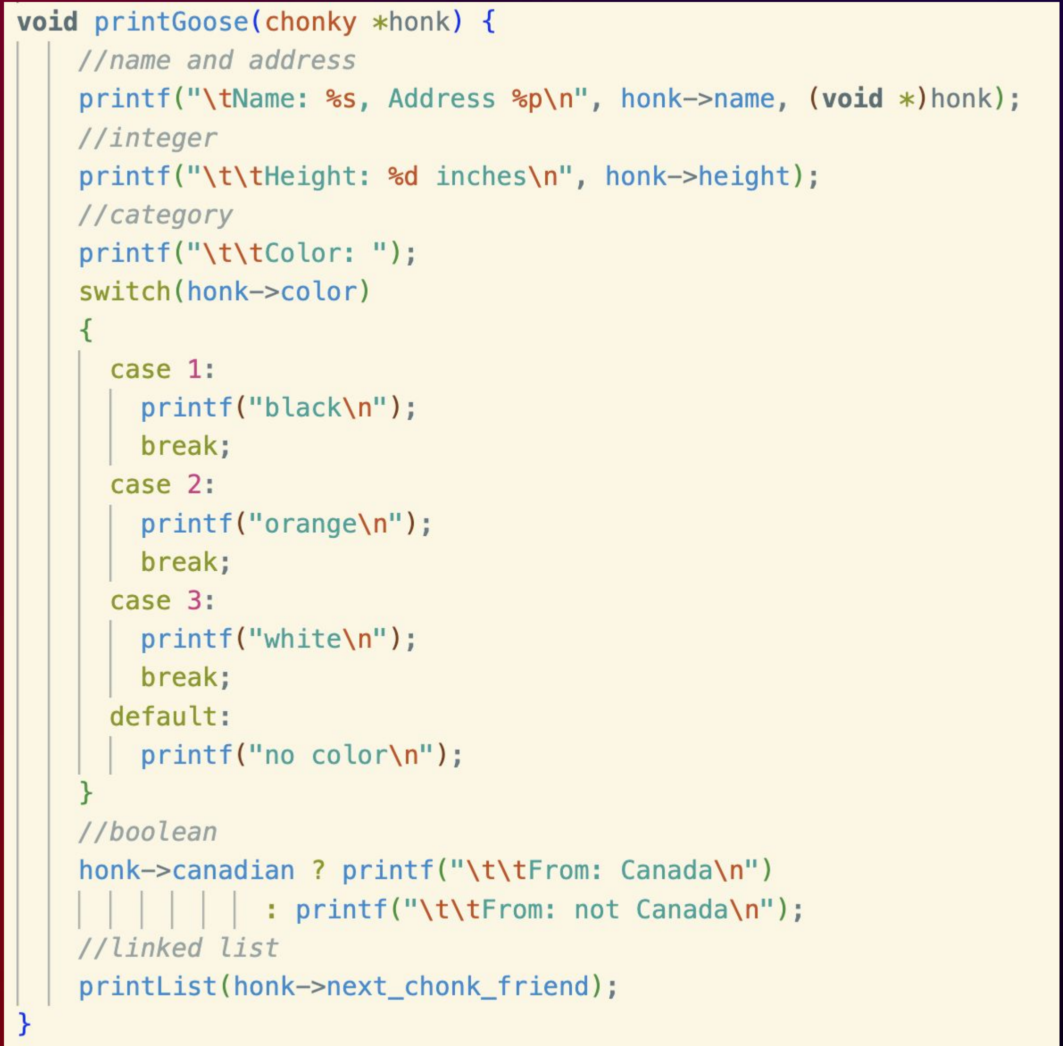
WHAT TO PRINT	PROS & CONS	WHERE TO PRINT
Loop index variables	**Pros:** tells us which iteration we're at **Cons:** can get messy with big loops	Beginning of loop so that counter is printed at start of each iteration
Changing variables	**Pros:** helps show what's changing **Cons:** doesn't tell us where things are changing	Right **before** and right **after** the variable is modified – perhaps before and after function calls that change the variable
Conditional branch indicators	**Pros:** catch incorrect if conditions **Cons:** doesn't show what's changing	A different print statement in each "case" of conditionals

INFOMISSION: PRINT STRUCTS

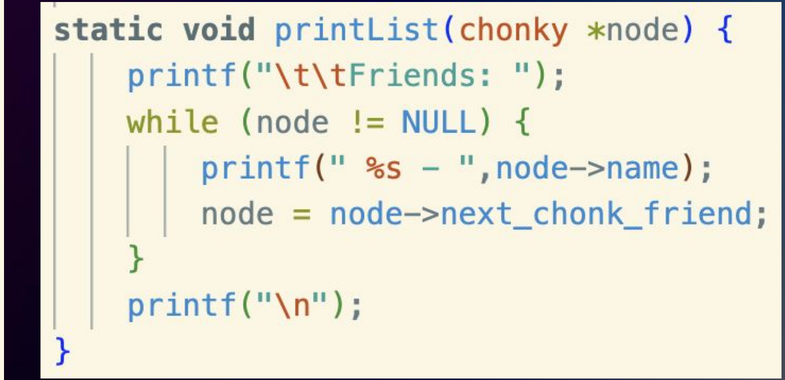
Scenario: we have a Goose structure with

- **Name** (string)
- **Height** (int)
- **Color** (int - categorical)
- **Canadian-ness** (bool)
- **Friends** (linked-list)

Goose Printer



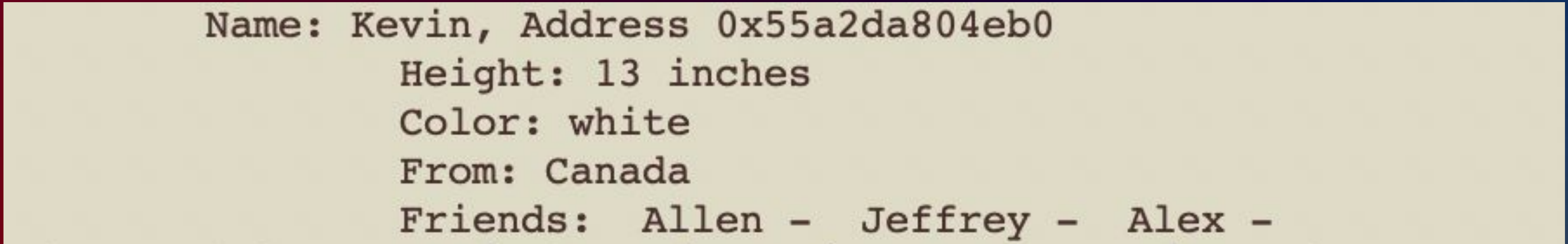
```c
void printGoose(chonky *honk) {
    //name and address
    printf("\tName: %s, Address %p\n", honk->name, (void *)honk);
    //integer
    printf("\t\tHeight: %d inches\n", honk->height);
    //category
    printf("\t\tColor: ");
    switch(honk->color)
    {
      case 1:
        printf("black\n");
        break;
      case 2:
        printf("orange\n");
        break;
      case 3:
        printf("white\n");
        break;
      default:
        printf("no color\n");
    }
    //boolean
    honk->canadian ? printf("\t\tFrom: Canada\n")
                   : printf("\t\tFrom: not Canada\n");
    //linked list
    printList(honk->next_chonk_friend);
}
```



```c
static void printList(chonky *node) {
    printf("\t\tFriends: ");
    while (node != NULL) {
        printf(" %s - ",node->name);
        node = node->next_chonk_friend;
    }
    printf("\n");
}
```

INFOMISSION: PRINT STRUCTS

Output:



```
        Name: Kevin, Address 0x55a2da804eb0
                Height: 13 inches
                Color: white
                From: Canada
                Friends:  Allen -  Jeffrey -  Alex -
```

02.

USING VALGRIND

What do all these errors mean???

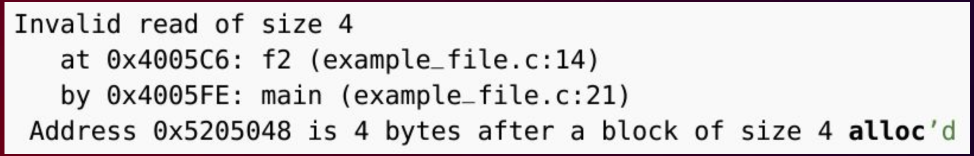
A SUPER USEFUL TOOL: GUIDE TO SUCCESS!

Found under **“Guides to Success”** on our **Autolab course page**

Gives explanations for all kinds of Valgrind output

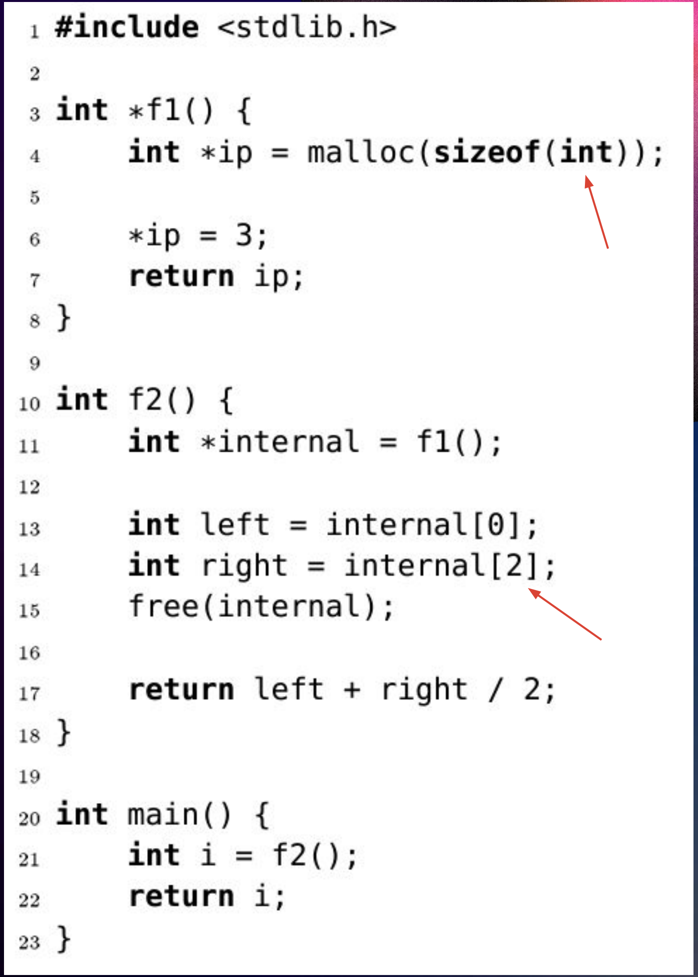
Invalid Read

Invalid read: accessing memory that was not allocated



```
Invalid read of size 4
   at 0x4005C6: f2 (example_file.c:14)
   by 0x4005FE: main (example_file.c:21)
 Address 0x5205048 is 4 bytes after a block of size 4 alloc'd
```

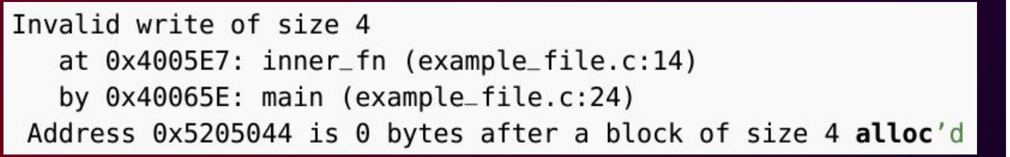
Usually off-by-one array indices or pointer arithmetic with wrong types



```
 1 #include <stdlib.h>
 2
 3 int *f1() {
 4       int *ip = malloc(sizeof(int));
 5
 6       *ip = 3;
 7       return ip;
 8 }
 9
10 int f2() {
11       int *internal = f1();
12
13       int left = internal[0];
14       int right = internal[2];
15       free(internal);
16
17       return left + right / 2;
18 }
19
20 int main() {
21       int i = f2();
22       return i;
23 }
```

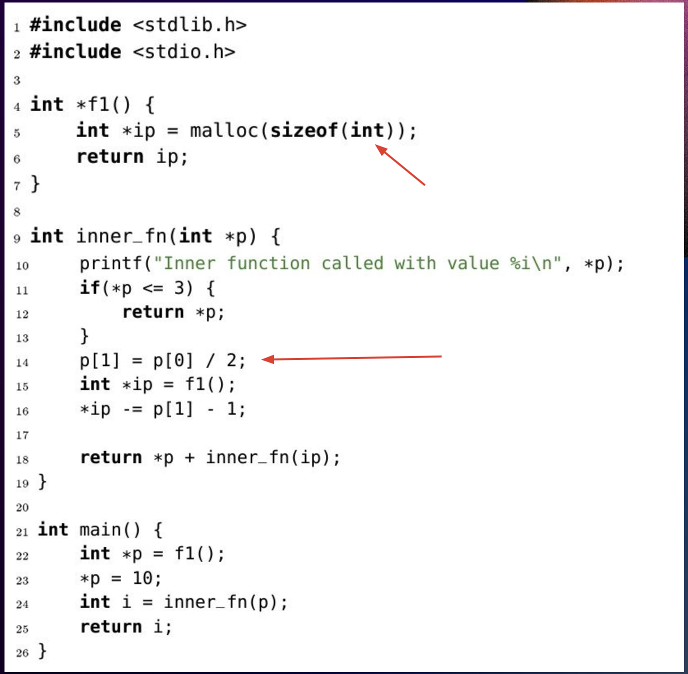
Invalid Write

Invalid <u>write</u>: writing/initializing memory that was not allocated



```
Invalid write of size 4
   at 0x4005E7: inner_fn (example_file.c:14)
   by 0x40065E: main (example_file.c:24)
 Address 0x5205044 is 0 bytes after a block of size 4 alloc'd
```

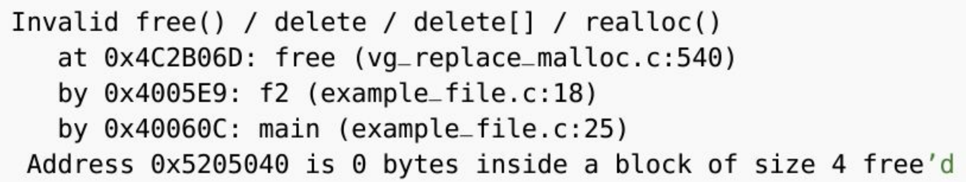
Usually off-by-one array indices or pointer arithmetic with wrong types



```c
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int *f1() {
5      int *ip = malloc(sizeof(int));
6      return ip;
7  }
8
9  int inner_fn(int *p) {
10     printf("Inner function called with value %i\n", *p);
11     if(*p <= 3) {
12         return *p;
13     }
14     p[1] = p[0] / 2;
15     int *ip = f1();
16     *ip -= p[1] - 1;
17
18     return *p + inner_fn(ip);
19 }
20
21 int main() {
22     int *p = f1();
23     *p = 10;
24     int i = inner_fn(p);
25     return i;
26 }
```

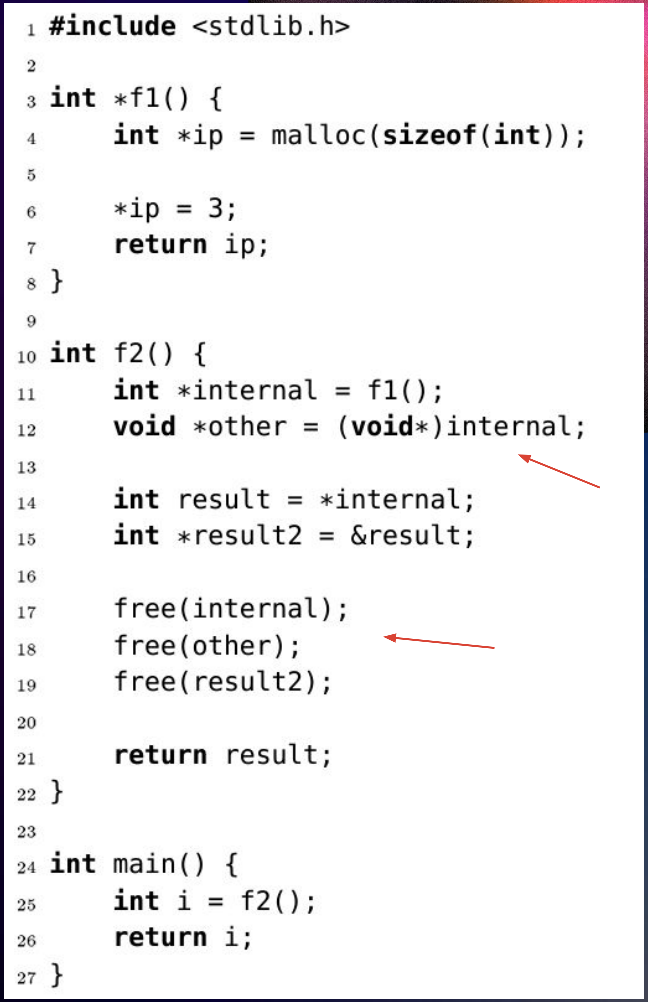
Invalid Free

Invalid <u>free</u>: trying to free memory that is not allocated



```
Invalid free() / delete / delete[] / realloc()
   at 0x4C2B06D: free (vg_replace_malloc.c:540)
   by 0x4005E9: f2 (example_file.c:18)
   by 0x40060C: main (example_file.c:25)
 Address 0x5205040 is 0 bytes inside a block of size 4 free’d
```

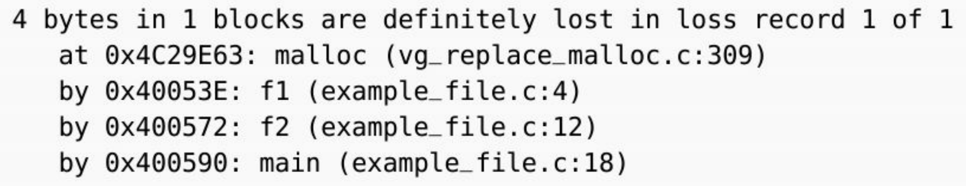
Freeing pointers that were already freed or not allocated



```c
1  #include <stdlib.h>
2
3  int *f1() {
4      int *ip = malloc(sizeof(int));
5
6      *ip = 3;
7      return ip;
8  }
9
10 int f2() {
11     int *internal = f1();
12     void *other = (void*)internal;
13
14     int result = *internal;
15     int *result2 = &result;
16
17     free(internal);
18     free(other);
19     free(result2);
20
21     return result;
22 }
23
24 int main() {
25     int i = f2();
26     return i;
27 }
```

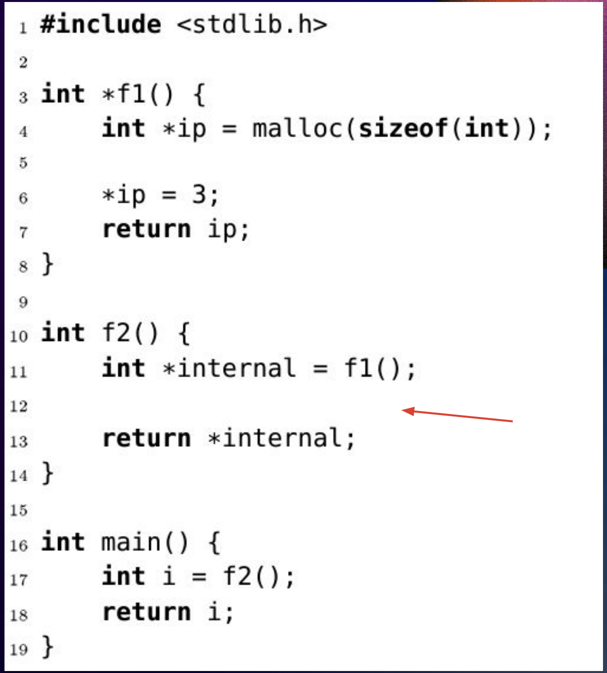
Leaked Memory

Leaked memory: forgetting to free allocated memory



```
4 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4C29E63: malloc (vg_replace_malloc.c:309)
   by 0x40053E: f1 (example_file.c:4)
   by 0x400572: f2 (example_file.c:12)
   by 0x400590: main (example_file.c:18)
```

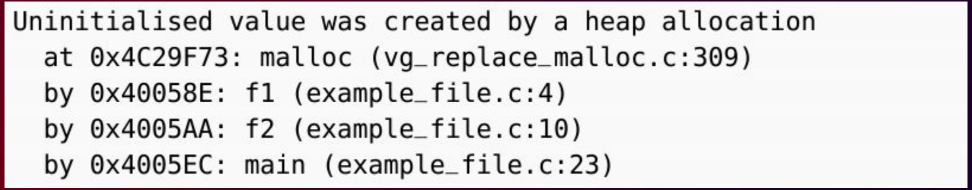
Use flag `--leak-check=full` in valgrind call to get more details



```
1  #include <stdlib.h>
2
3  int *f1() {
4      int *ip = malloc(sizeof(int));
5
6      *ip = 3;
7      return ip;
8  }
9
10 int f2() {
11     int *internal = f1();
12
13     return *internal;
14 }
15
16 int main() {
17     int i = f2();
18     return i;
19 }
```

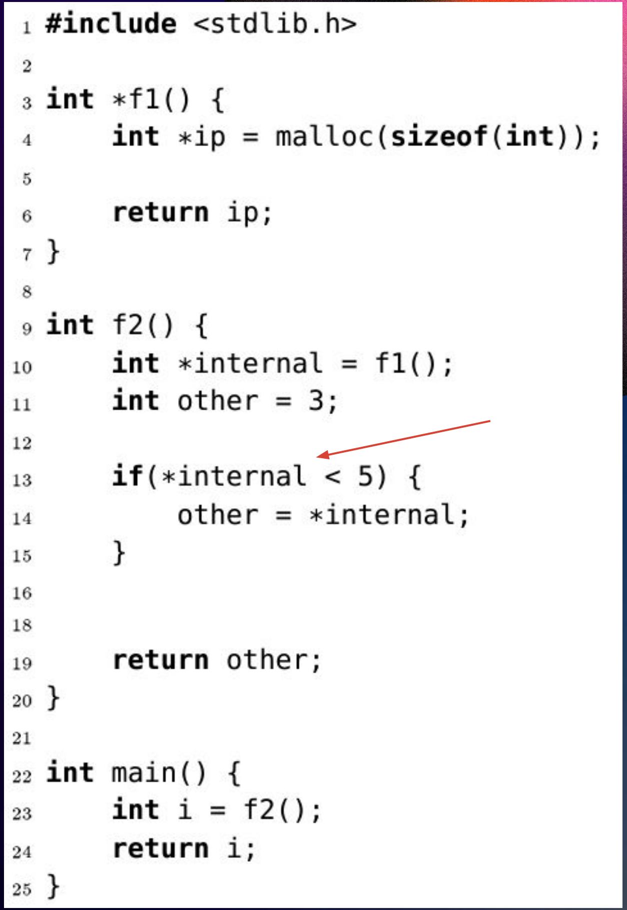
Uninitialized Values

Uninitialized values: using allocated memory without initializing



```
Uninitialised value was created by a heap allocation
  at 0x4C29F73: malloc (vg_replace_malloc.c:309)
  by 0x40058E: f1 (example_file.c:4)
  by 0x4005AA: f2 (example_file.c:10)
  by 0x4005EC: main (example_file.c:23)
```

Use flag `--track-origins=yes` to see where value was allocated



```
 1 #include <stdlib.h>
 2
 3 int *f1() {
 4       int *ip = malloc(sizeof(int));
 5
 6       return ip;
 7 }
 8
 9 int f2() {
10       int *internal = f1();
11       int other = 3;
12
13       if(*internal < 5) {
14             other = *internal;
15       }
16
18
19       return other;
20 }
21
22 int main() {
23       int i = f2();
24       return i;
25 }
```

EXAMPLE 3: REMOVE DUPLICATES

Take a look at the **FILE**: `ex3.c`

There's A FEW **BUGS** in the file

TA STEP-THROUGH

EXAMPLE 4: PASCAL'S TRIANGLE

Take a look at the **FILE**: `ex4.c`

There's A FEW **BUGS** in the `main` and `generate` functions

Try running the file with `valgrind` to see what's going on!

[20 MINS]

03.

CONTRACTS & TEST CASES

How do contracts work in C & how do we write good edge cases?

CONTRACTS IN C0 VS. C

IN C0:	IN C:
	`#include "lib/contracts.h"`
`//@requires ___;`	`REQUIRES(___);`
`//@ensures ___;`	`ENSURES(___);`
`//@loop_invariant ___;`	No `LOOP_INVARIANT`... but we can use
`//@assert ___;`	`ASSERT(___);`

WHAT CONTRACTS TO WRITE?

PRECONDITIONS

Does this function depend on features of the input?

POSTCONDITIONS

Where is the output of this function used; are there assumptions we should meet?

LOOP INVARIANTS

Is there something in the loop you know **must** stay the same throughout?

HOW DO I WRITE GOOD TEST CASES???

EDGE CASES

- Edge cases are often forgotten in implementation
- Small values, large values, empty data structures, long data structures

ITERATING THROUGH ALL CASES

- Not recommended for larger problems
- Usually gives us an idea of a range for which implementation is incorrect

COMMON ARBITRARY CASES

- Helps make sure your function actually works as expected

EXAMPLE 5: PIXEL COLOR TRANSFORMATIONS

Take a look at the **FILE**: `ex5.c`

There's A FEW **BUGS** in the file

TA STEP-THROUGH

EXAMPLE 6: DNA ENTANGLEMENT

Take a look at the **FILE**: `ex6.c`

There's A FEW **BUGS** in the function `twist_my_dna`

1. Write contracts to get it to stop infinite looping!
2. Try writing test cases in the `test()` function to see what's going on!
3. Use the test cases to find the bugs! Write contracts to help isolate the reasons for the bugs

[10 MINS]

THANKS FOR COMING!

PLEASE GIVE US FEEDBACK!