



# **15-122 Bootcamp: Debugging Fundamentals**

Summer 2023



Sign In Here!



# Today's Agenda

01

Print Debugging

02

Writing Test Cases

03

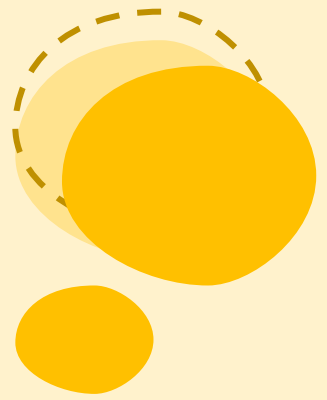
Writing Contracts

**NOTE:** Don't worry about getting to the challenge exercises! They're supposed to be a challenge and feel free to take them home to work on.



# Print Debugging

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



# Printing in C0 –Review

**printf** function:

**Example usage:**

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

```
→ Class: 15122
```

**printf** can take in more than one argument!

**Example usage:**

```
printf("Age: %d, Height: %d (cm)\n", 20, 170);
```

```
→ Age: 20, Height: 170 (cm)
```

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

**NOTE:** remember to “\n” to flush output!

# Format Specifiers

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

# So... *What* Do I Print?

## Loop index variables:

- tells which iteration we're at

## Changing variables:

- helps show what's changing

## Conditional branch indicators:

- catch incorrect if conditions



# 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



# TA Example: Close Pairs

print-example.c0



# Basic Exercise: Sum Elements

Open the file `print-basic.c0`

Write the function `print_arrays` using the provided format in file and use it to help you debug `sum_elements`.

# Classic Exercise: Fib

Open the file `print-classic.c0`

Add print statements to the function `fib` to find the bugs.

# Challenge Exercise: Fizzed and Buzzed

Open the file `print-chal.c0`

Add print statements to the function `fizzed_and_buzzed` to find the bugs. The full description of the rules of `fizzed_and_buzzed` are in the handout.

The background is a solid yellow color. It features several abstract decorative elements: a large, light-yellow curved shape on the left side; a dashed yellow line curving from the top left towards the bottom right; and in the bottom right corner, a small solid yellow circle and a larger solid yellow circle, both with dashed yellow outlines.

# Writing Test Cases

How do we write test cases that  
catch all of our bugs?

# What Test Cases Should I Write?

## Edge cases:

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

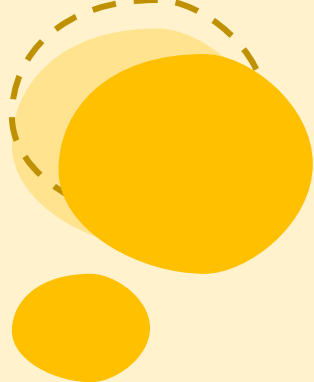
## Stress testing:

- If needed, writing test cases that push the function to its limits can show where it breaks
- Large inputs, empty inputs, many tests to check accuracy & efficiency

## Basic cases & cases from the writeup:

- Helps make sure your function actually works as expected
- Do these first!

# But... How Do I Format It?



Most C0 test cases look like this:

```
assert(my_function(input) == expected);
```

**NOTE:** make sure to use `assert` instead of `//@assert` — we still want our test cases to run when contracts are disabled!

If your function outputs strings:

```
assert(string_equal(my_function(input), "expected"));
```



# Quick Preview on How Strings Work

- Part of this section uses the relationship between character arrays and strings! This is something you'll see moving forward in this course – but not just yet – so here's a quick preview on how they work.
- We need to use a NUL-terminator, which is a character represented by `'\0'`, at the end of the array so that the functions we use know where the string ends. (You'll learn more about this later – just know that this fact is true for now.) That means any array we use to represent a string has a length of `string_length(s) + 1`.

# Quick Preview on How Strings Work

A string can be represented by a char array like so:

```
#use <string>
```

```
string s = "hello";
```

```
char[] arr = string_to_chararray(s);
```

```
→ arr = ['h', 'e', 'l', 'l', 'o', '\0']
```

```
string res = string_from_chararray(arr);
```

```
→ res = "hello"
```



# TA Example: String Repeat

test-example.c0

# Basic Exercise: Remix Pixel

Open the file `test-basic.c0`

Write test cases for the function `remix_pixel` to catch and fix the bug.  
Note that there is only one bug in this file!

# Classic Exercise: Longest Sequence

Open the file `test-classic.c0`

Write test cases for the function `longest_sequence` to catch and fix the bug. Note that there is only one bug in this file!

# Writing Contracts

How do we write contracts that ensure  
safety and correctness of code?

# Why Do We Use Contracts?

**SAFETY & CORRECTNESS:** we want to make sure our code is safe & does what it's supposed to!

Examples of unsafe code: accessing elements of array that don't exist, dividing by zero, dereferencing NULL pointers

Usually correctness is **ensured** by our postconditions

# What Contracts Do I Write?

The slide features decorative yellow elements. In the top right corner, there are three overlapping circles: a large solid yellow circle, a medium solid yellow circle, and a small dashed yellow circle. In the bottom right corner, there is a stylized yellow hand with fingers spread.

## 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 Debug with Contracts?

Use `//@assert` statements!

- If you need something to be true (or if you're not sure if it is) before a certain line of code, you can `//@assert` it to check.
- You can also catch infinite loops really easily with contracts (in both `for` and `while` loops:
  - `for` loops: limit iterating variable (`//@loop_invariant 0 <= i && i <= n;`)
  - `while` loops: limit a loop counter (`//@assert counter < 30;`)

# TA Example: Zero Sum Triplets

contr-example.c0



# Basic Exercise: NBA Farm

Open the file `contr-basic.c0`

Write contracts in the function `NBA_farm` to fix the all the bugs! You may need to write or improve on the current test conditions if you can't find the bugs.

# Classic Exercise: Shift Array

Open the file `contr-classic.c0`

Write contracts for all the functions in this file (`shift_array`, `check_arrays`) to fix the bug! Note that this is a classic infinite loop example — try to use a contract to stop it.

# Challenge Exercise: Shift Grid

Open the file `contr-chal.c0`

Write contracts for all the functions in this file (`reverse_array`, `shift_grid`, `check_grids`, `grid_from_arr`) to fix the bugs!

# Additional Resources

## Autolab Guides to Successes

- Coding with Style
- General Debugging Practices
- How to Write a Test File
- How to Debug with Print Statements
- How to Debug with Contracts

## Bootcamp

- Exercises Handout ([link](#))
- Resources Handout ([link](#))

# Thanks!

