User-defined Functions

Zachary Bennett
Software Engineer

@z_bennett_ zachbennettcodes.com

Module Layout

Declarations

Function prototypes, declarations, and header files

Best Practices

Consistency, static functions, const, no globals

Pass-by-val/ref

How to implement pass-by-reference

All prototypes are declarations but not all declarations are prototypes.

Function Declarations vs. Prototypes

Function Declaration

Any grouping of statements that declare a function

Function Prototype

A function declaration that includes parameter types

```c
// Function declaration example
void perform_side_effect();


// Function prototype example
void perform_side_effect(void);


// Prototype without named parameters
void alloc_string(int, char**);


// Prototype with named parameters
void alloc_string(int size, char** out);
```

◀ No parameters declared (this is obsolete!)

◀ Function prototype (this is good!)

◀ Function prototype without explicit names for parameters (OK, but not ideal)

◀ Function prototype with an explicit name for each parameter (best!)

Prototypes define your API (internal and external)

Prototypes - Where Do They Go?

Header File

Function prototypes that define the external API of your program/library

Top of C Source File

Function prototypes that define the internal API of your program/library

Up Next:
Demo: Function Prototypes

Demo

- Create the header file of our C library for Wired Brain Coffee
- Function prototype used to define an internal API function
- Create an external function prototype inside of our new header file

Function Best Practices

Keep them small

Focus on one thing – avoid large functions

Clearly define inputs

Do everything you can to not use global variables

Mark functions static

For internal API functions, use the “static” keyword

Use ”const”

Be explicit about read-only variables and parameters

Be consistent

Follow a coding style guide

https://github.com/mcinglis/c-style

```
static CoffeeMetric* make_coffee_metric(const int duration, const PourMode pour_mode) {
     CoffeeMetric *metric = (CoffeeMetric*)malloc(sizeof(CoffeeMetric));
    metric->duration  = duration;
    metric->pour_mode = pour_mode;


    return metric;
}
```

Best Practices

This function is small, focuses on one piece of work, clearly defines its inputs, explicitly marks its read-only parameters and variables, is clearly defined as an internal API function, and follows the appropriate style guide.

Up Next:
Demo: Best Practices

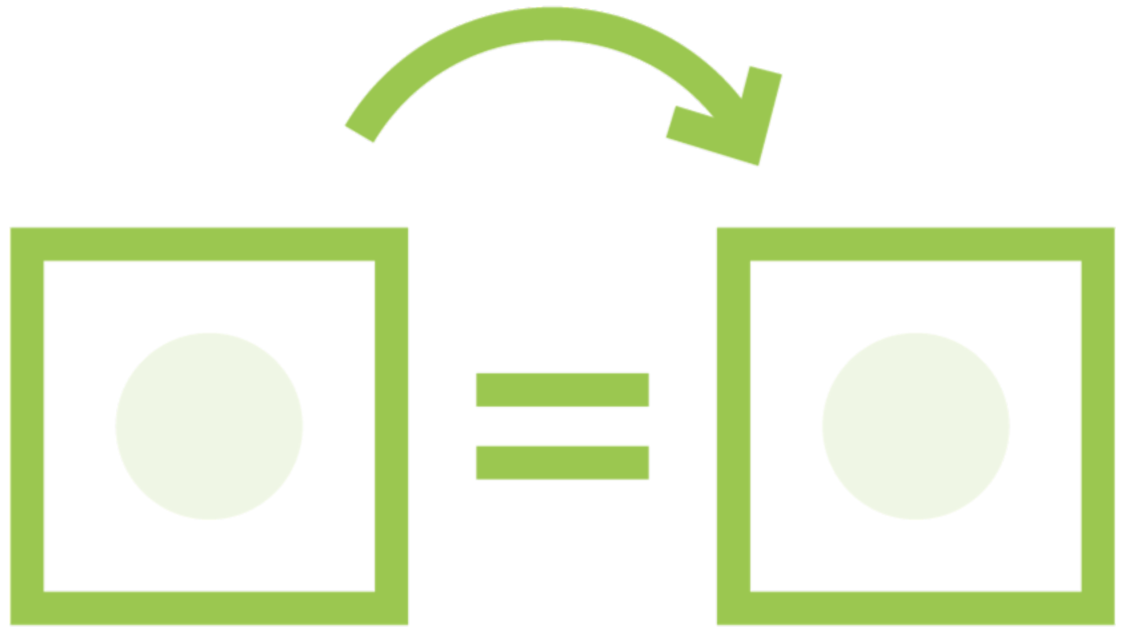
Demo

- Best practices in action
- Inspect a Wired Brain Coffee function
- Refactor this function to conform to best practices

Pass-by-value and Pass-by-reference

Pass-by-value

Function arguments are copied and used inside of function calls



Pass-by-reference

Function arguments are references to the original values passed into the function

Everything is pass-by-value in C

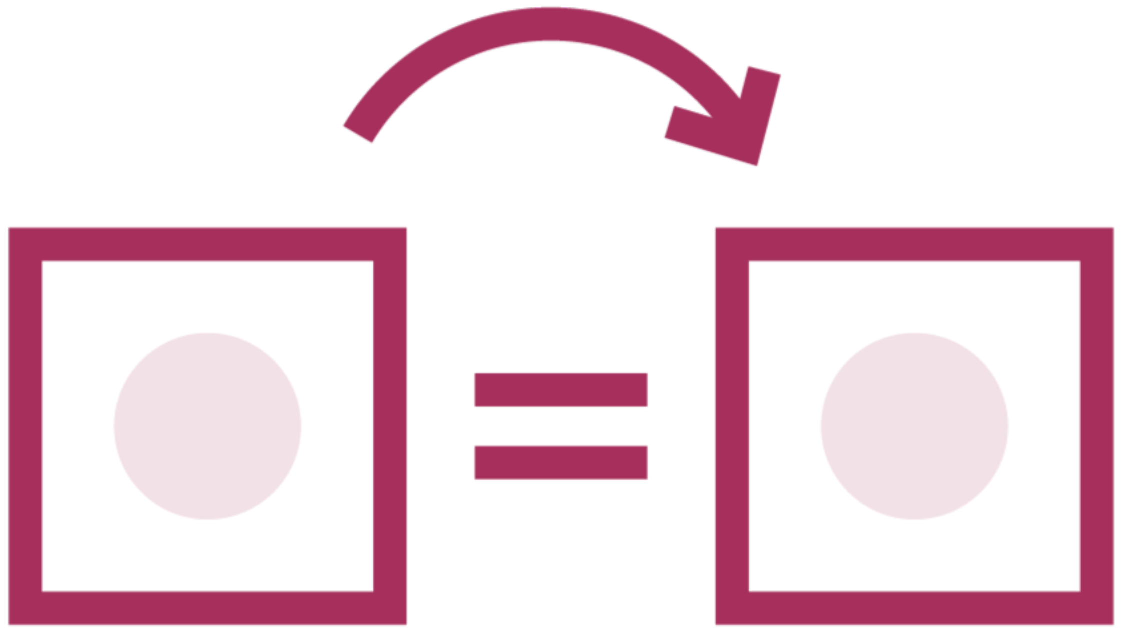
```c
int first  = 1;

int second = 2;

swap(first, second);




void swap(int a, int b) {

  int tmp = a;

  a = b;

  b = tmp;

}
```

◂ Trying to swap the values of two integer variables

◂ This will fail horribly due to pass-by-value in C

```
int first  = 1;

int second = 2;

swap(&first, &second);



void swap(int *a, int *b) {

  int tmp = *a;

  *a = *b;

  *b = tmp;

}
```

- Trying to swap the values of two integer variables via pointers
- Note: the pointers themselves are still passed by value (pointer to pointers can get around this)
- This will work great! Pass-by-reference has been implemented using pointers



- You need to optimize performance
- You need to mutate the original value of a function argument
  - Working with APIs that follow this convention

Up Next:
Demo: Value and Reference Passing

Demo

- Pass-by-value degrading performance
- Pass-by-reference performance boost

Summary

- Function declarations
  - Function prototypes are the way to go
  - External vs. internal APIs
- Function best practices
  - Keep them small
  - Clearly define inputs
- Pass-by-value vs. pass-by-reference

Up Next:
Standard Library Functions – Part One