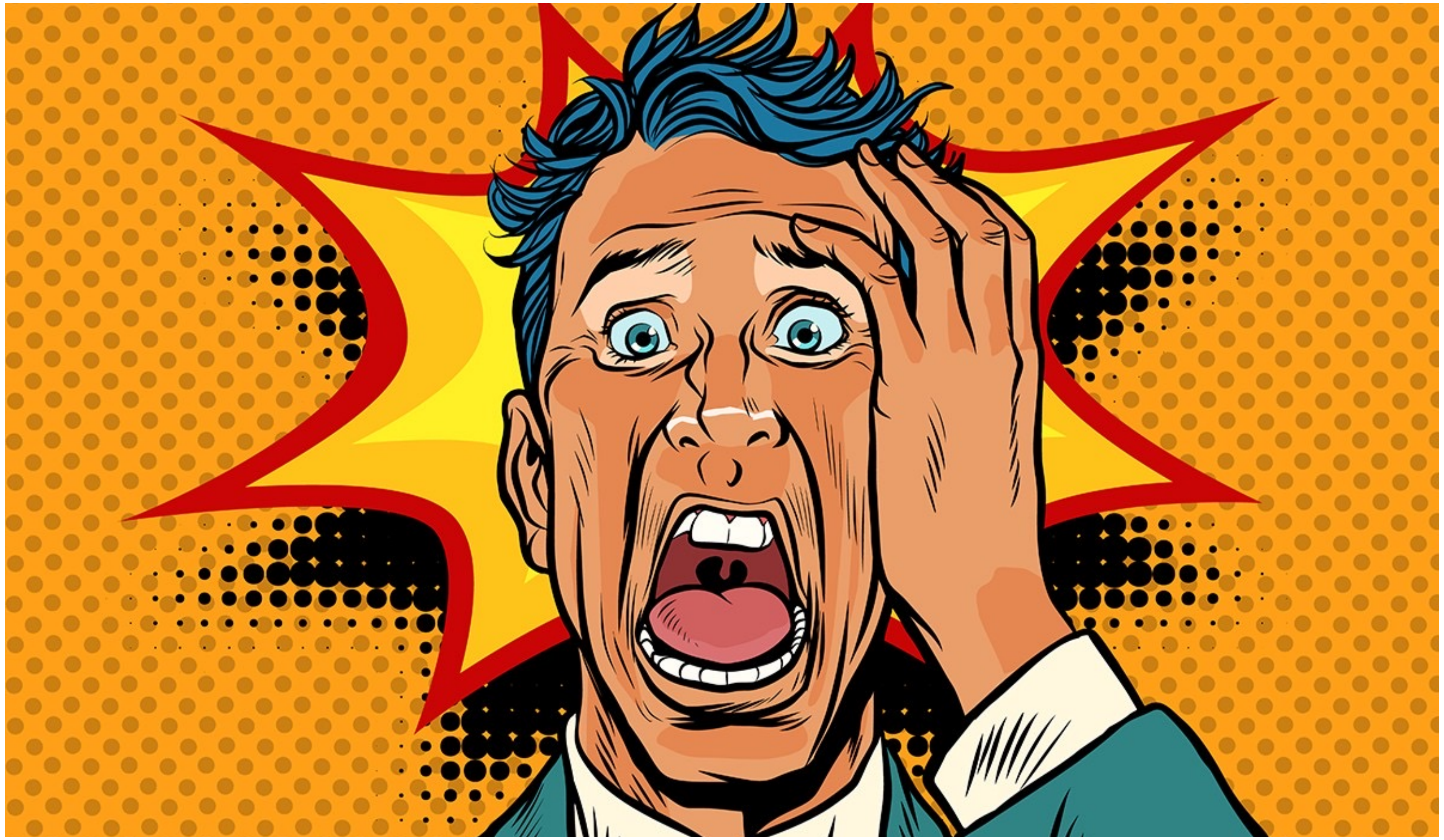


Words of Caution

- Just as assignment statements are not the same as equations in high school Algebra, functions in **C** are not the same as mathematical functions.
  - They may or may not return a value.
  - They may or may not have side-effects.
- In other languages, we might call them *procedures* or *subroutines*.
- Java aficionados call them *methods*.
  - Please, do not do that when writing in **C**.

What is a function in programming?

- A function is block of code that performs a certain task.
  - It gives a name to code that (hopefully) performs a logically consistent task.
- Functions are ***defined* exactly once**.
- Functions must be ***declared* before they are used**.
- Programs can ***declare* a function as many times as desired**.
- Programs can *call* a function as many times as desired.
- `main()`
  - Is a special function.
  - Is run when program starts.
  - All other functions are subordinate to `main()`.

Declaration

vs. Definition

Return Type

Function Name

Number of parameters and their types

```
int f(int);          // declare function f


int f(int a) {       // define function f
        return -a;
}
```

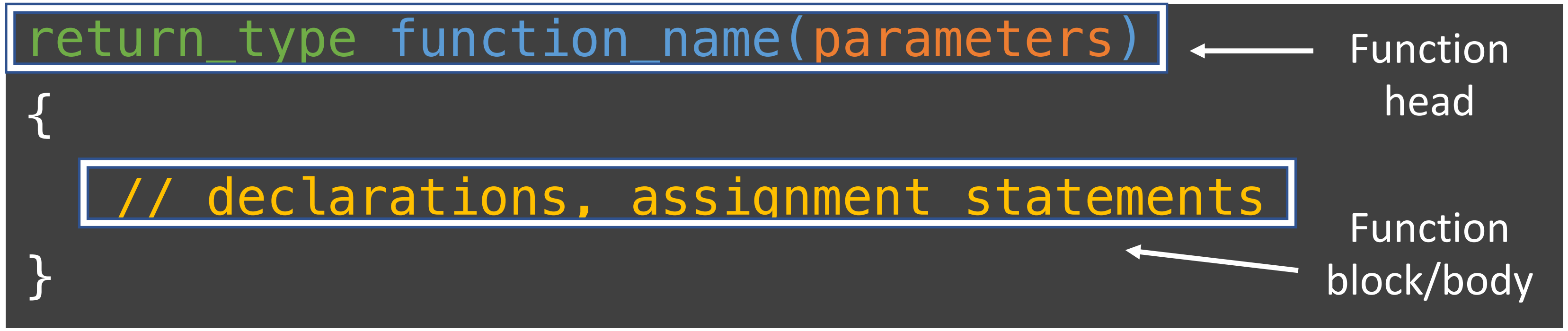
Add Function Body

Add parameter names

Why do we like functions?

- Functions should:
    - Define abstractions that are consistent and make sense logically.
    - Give names to those sequences of code.
    - Hide the implementation.
- We can use them to:
    - Refactor repeated sequences of code.
    - Simplify the code to aid understanding.
- Functions should never be:
    - Arbitrary sequences of statements.

Function Definition



```
return_type function_name(parameters)
{
    // declarations, assignment statements
}
```

- Function head
  - return_type
    - Defines the type of function’s return value
    - Return type may be `void` or any object type (except `array` type)
  - function_name()
    - Function’s name
  - parameters
    - Contained in comma-separated list of declarations
    - If function has no parameters, then this is either empty or contains the word `void`
- Function block/body
  - Declarations
    - Declared variables inside a function body are only locally known
  - Assignment statements
    - Sets and/or resets the value of a variable

Return Values

- In **C**, functions return a value.
- The value may be `void`, which means *no value*.
- You can return any scalar value (`char`, `int`, `float`, …)
- You can return a `struct` (but please don’t).
  - Okay examples: (x, y) coordinate; complex number
- You can return a pointer.
- You *cannot* return an array.

Function naming (Part 1)

- Functions have the same naming rules as variables.
- *Can't*:
  - Start with a number or any punctuation other than `_` (underscore) or `$` (dollar sign)
  - Use the same name as another function.
    - There are no *nested functions* in **C**.

- For this class, we will be using *Snake case*
  - *my_function_name*

Function naming (Part 2)

- Choose descriptive names!
- Examples (verbs)
  - sort(), free(), write()
- Examples (descriptive statements)
  - isalpha(), isdigit()
- Avoid misleading names
  - fit_virtual_pin()   // "fit" is a prefix, not a verb

Parameters — Also called Arguments

- In mathematics, when we have a function like $f(x) = x\log(x+1)$, and we write $f(2)$ we substitute 2 for x and get $2\log(2+1)$.
  - In programming languages, this is called *call-by-name*, and is *rare*.
  - **C** supports this as textual substitution in macros with the ***C** Preprocessor*.
- Most programming languages use either *call-by-value*, *call-by-reference* or both.
- **C** uses *call-by-value*, except for arrays, and only because of their relation to pointers.

Parameters

- *Formal* parameters
  - This is the name of the parameter as it is used inside of the function body.
- *Actual* parameters
  - This is the name of the value that is passed to the function.
  - The value can be copied to the formal parameter.
  - Or a reference to the actual parameter may be bound to the formal parameter.
    - In **C**, we do this by passing a pointer using call-by-value.
- *Call-by-value* means a copy of the actual parameter is placed in the formal parameter.
  - This is the only method supported by **C**.
- *Copy-in-Copy* out means that in addition to being copied in, the value is copied back out to the actual parameter.
  - **C** does not support this.

Call-by-Value

- All functions use call-by-value in C.
- Arguments passed into a function are *copied*
  - Any changes made to the parameters inside the function has no effect.
- The called function copies the values of the supplied (*actual parameters*) arguments into a new set of variables (*formal parameters*) which are pushed into the call stack.

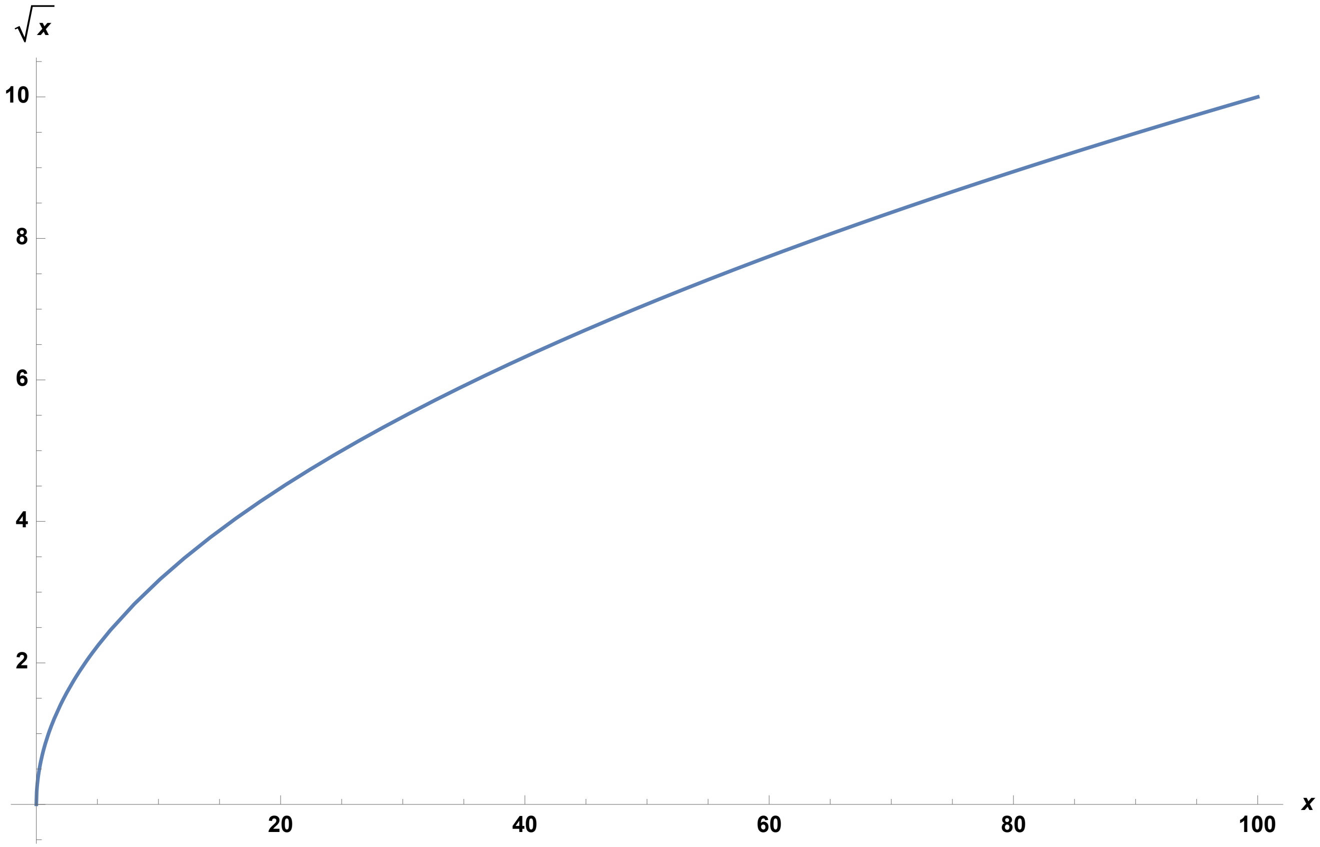
Call-by-Reference

- The references of the arguments passed into a function are *copied* meaning any changes made to the parameters inside the function has an effect on the actual values.
- Instead of passing values to the called function, references to the original variables are passed.

- **C** does not use call-by-reference
  - But you can accomplish it by passing a *pointer*.

$$\boxed{\sqrt{x}}$$

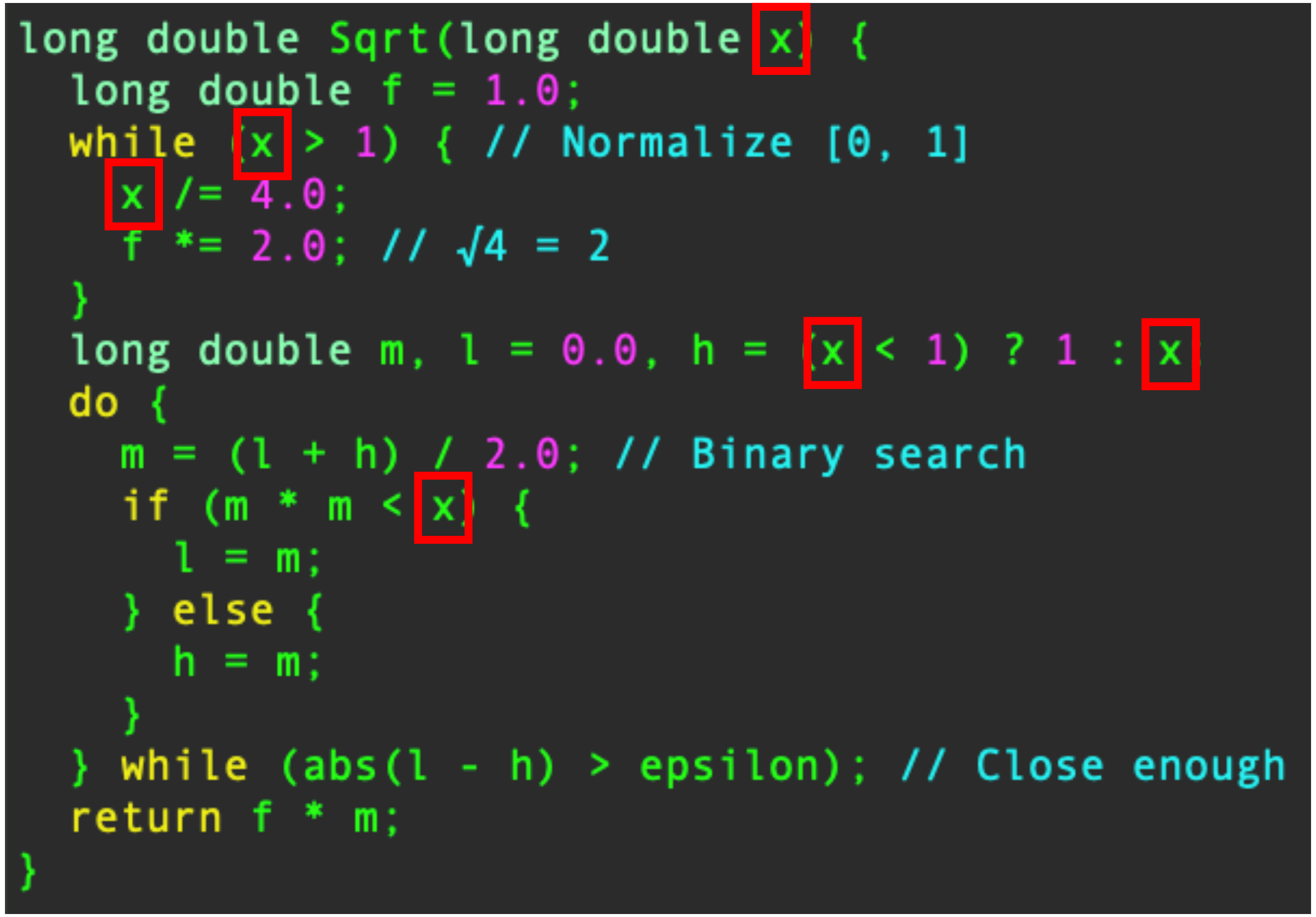
Is a well-behaved function so we can use a simple method like *bisection*.



$$\sqrt{x}$$

Call-by-value: Sqrt(2) Use value 2 for all x's.

What if "call-by-name"? Sqrt(sin(x))
(C doesn't use call-by-name. But why?)

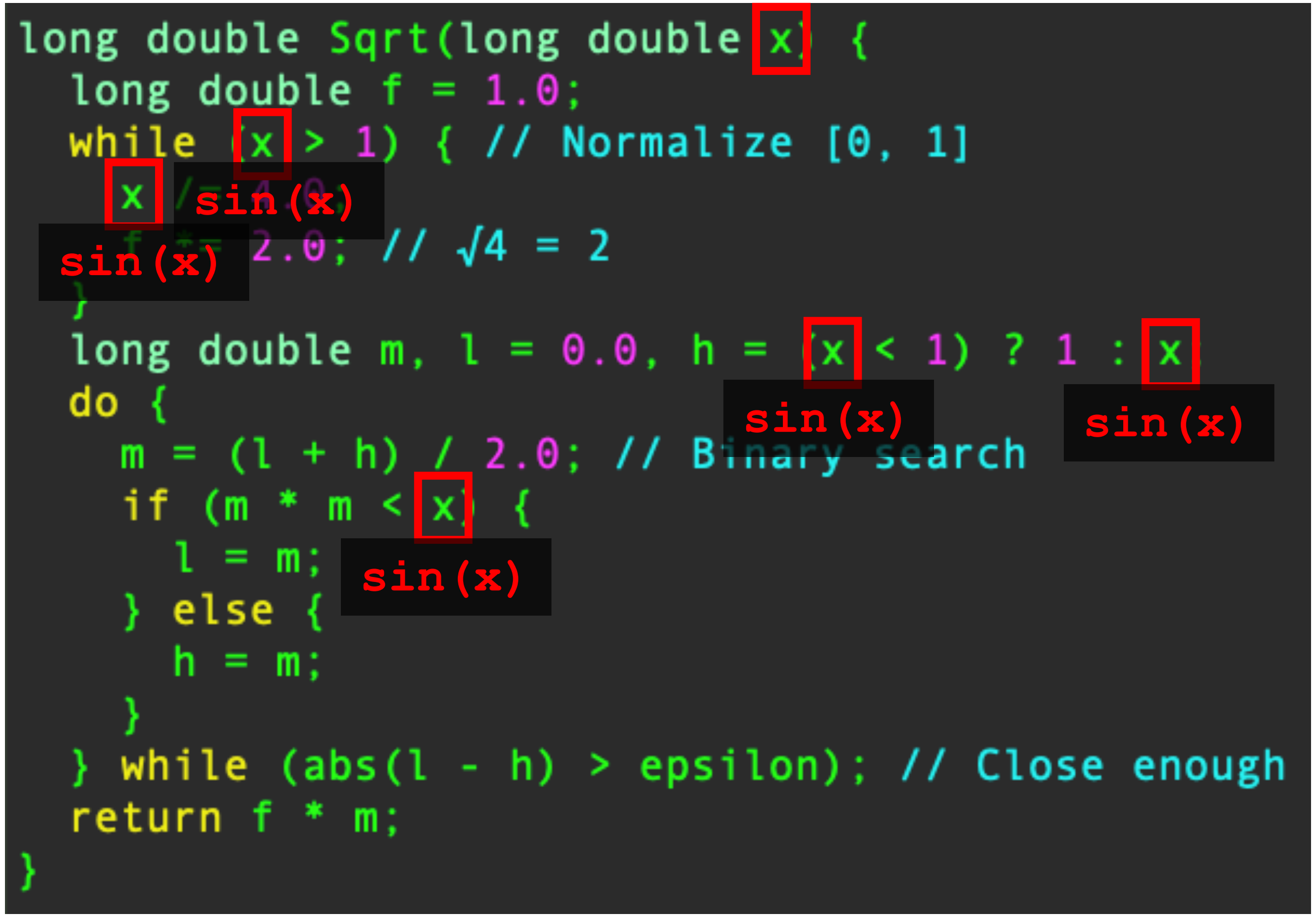


```
long double Sqrt(long double x) {
  long double f = 1.0;
  while (x > 1) { // Normalize [0, 1]
    x /= 4.0;
    f *= 2.0; // √4 = 2
  }
  long double m, l = 0.0, h = (x < 1) ? 1 : x;
  do {
    m = (l + h) / 2.0; // Binary search
    if (m * m < x) {
      l = m;
    } else {
      h = m;
    }
  } while (abs(l - h) > epsilon); // Close enough
  return f * m;
}
```

$\sqrt{x}$

Call-by-value: Sqrt(2) Use value 2 for all x's.

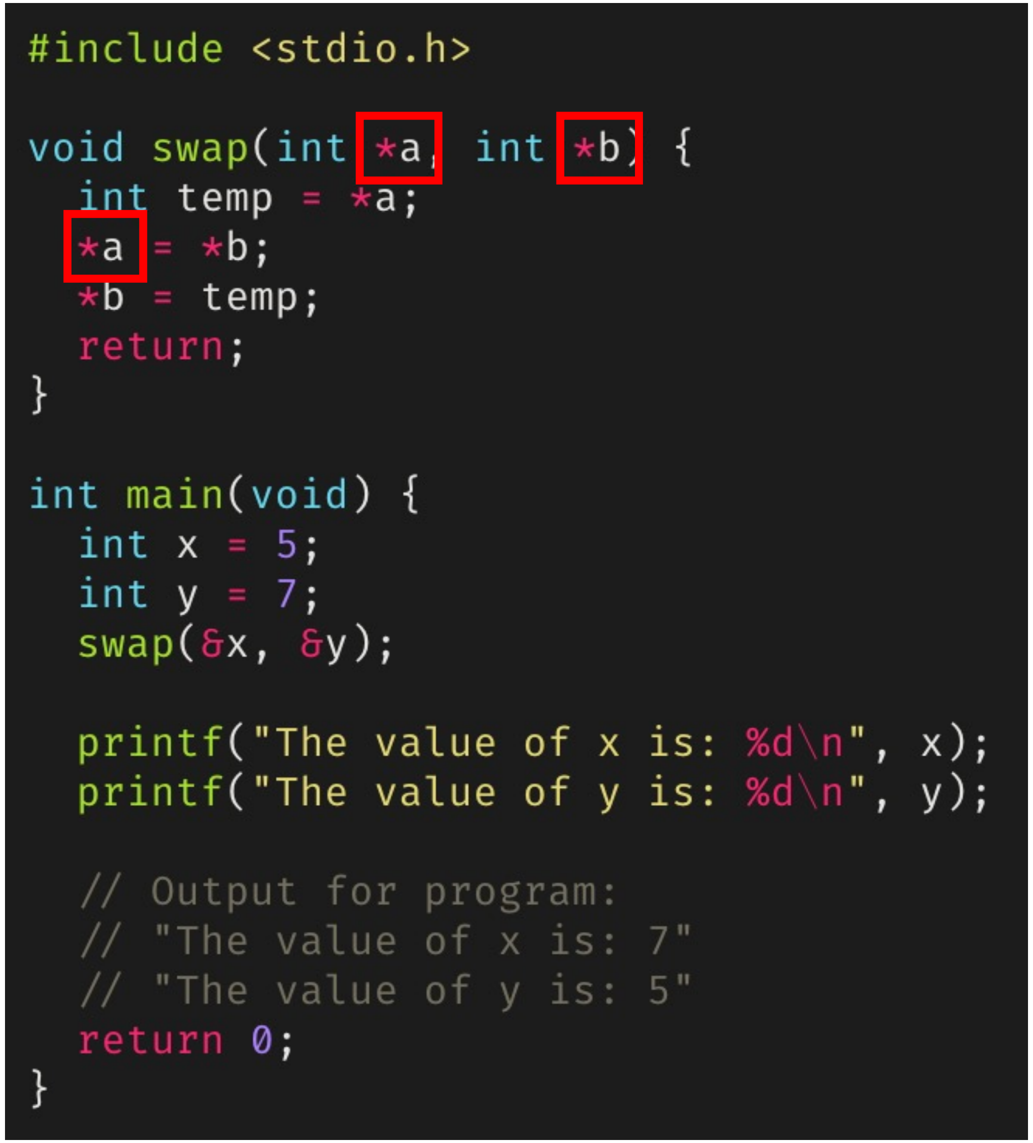
What if "call-by-name"? Sqrt(sin(x))
(C doesn't use call-by-name. But why?)



```
long double Sqrt(long double x) {
  long double f = 1.0;
  while (x > 1) { // Normalize [0, 1]
    x /= 4.0;
    f *= 2.0; // √4 = 2
  }
  long double m, l = 0.0, h = (x < 1) ? 1 : x;
  do {
    m = (l + h) / 2.0; // Binary search
    if (m * m < x) {
      l = m;
    } else {
      h = m;
    }
  } while (abs(l - h) > epsilon); // Close enough
  return f * m;
}
```

swap()

- C does not have true call-by-reference, so we use pointers.
    - Addresses, instead of values, are passed as arguments.



```c
#include <stdio.h>

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  return;
}

int main(void) {
  int x = 5;
  int y = 7;
  swap(&x, &y);

  printf("The value of x is: %d\n", x);
  printf("The value of y is: %d\n", y);

  // Output for program:
  // "The value of x is: 7"
  // "The value of y is: 5"
  return 0;
}
```

Function Prototypes

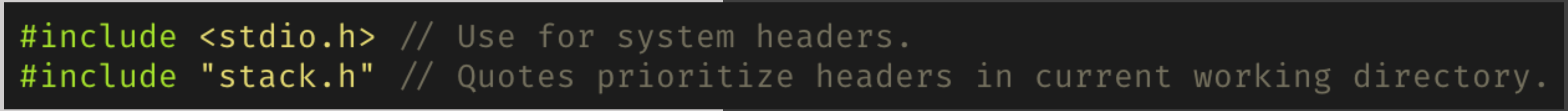
- Provides compiler with description of functions that will be used later in the program.
    - Functions in programs cannot be called unless they have either:
        - Been declared and defined prior to the function call.
        - Been prototyped at the beginning of the program.
- Syntax for Function Prototype:

```
return_type function_name(parameters);
```

- Prototypes must be declared either at the beginning of the program or in included header files, which act as interfaces.

#include

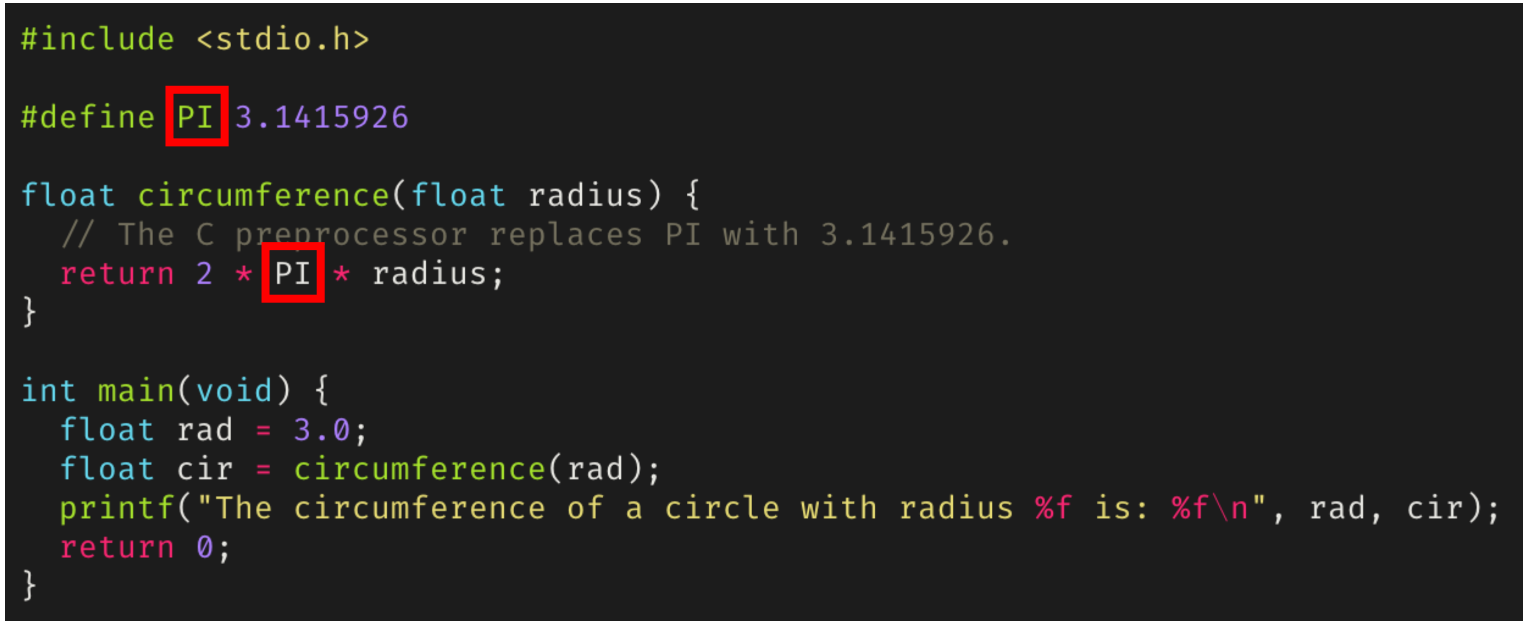
- A preprocessor directive.
  - Before compilation, **C** source files are processed by a preprocessor.
  - A preprocessor is a macro processor to transform programs before compilation.
  - Macros in **C** operate through text replacement.
- Pastes code of given file into current file.
- Used to include functions defined in other libraries.



```
#include <stdio.h> // Use for system headers.
#include "stack.h" // Quotes prioritize headers in current working directory.
```

#define

- A preprocessor directive that defines a macro for the program.
- The **C** preprocessor performs all text replacement for defined macros prior to compilation.



```c
#include <stdio.h>

#define PI 3.1415926

float circumference(float radius) {
  // The C preprocessor replaces PI with 3.1415926.
  return 2 * PI * radius;
}

int main(void) {
  float rad = 3.0;
  float cir = circumference(rad);
  printf("The circumference of a circle with radius %f is: %f\n", rad, cir);
  return 0;
}
```

Conditional Directives

- A set of preprocessor directives that uses *conditional statements* to include code selectively.
  - Uses value of conditions evaluated during compilation.
- `#ifdef` – execute statements only when `MACRO` is defined:

```
#define MACRO
#ifdef MACRO
       controlled text
#endif
```

- `#ifndef` – execute statements only when `MACRO` is undefined:

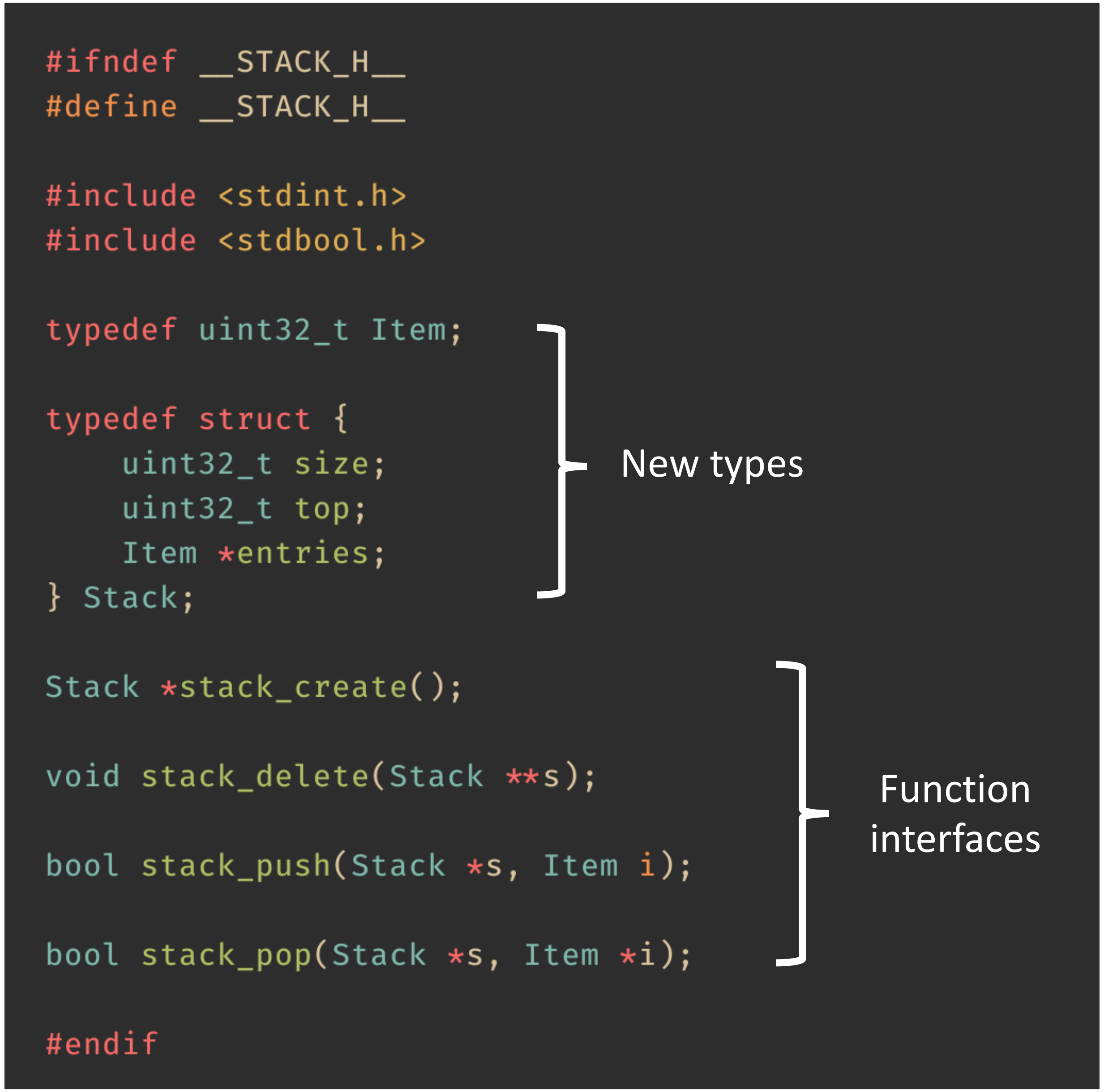
```
#ifndef macroname
       controlled text
#endif
```

Header Files

- Should only contain things that are shared between source files:
  - Function declarations
  - Macro definitions
  - Data structure and enumeration definitions
  - Global variables (see coding standards for proper usage)
  - Any `#include` directives required to compile
- Uses the file extension `.h`.
- Typically used for modules or abstract data types
  - Header files provide the function declarations so that the function implementations aren't known.
  - This allows you to have opaque data types.
- Contents of a header file must be within a header guard, implemented with the either the `#ifndef` or `#pragma once` preprocessor directive.
  - Both prevent contents of a header file from being included more than once.

Example Header File

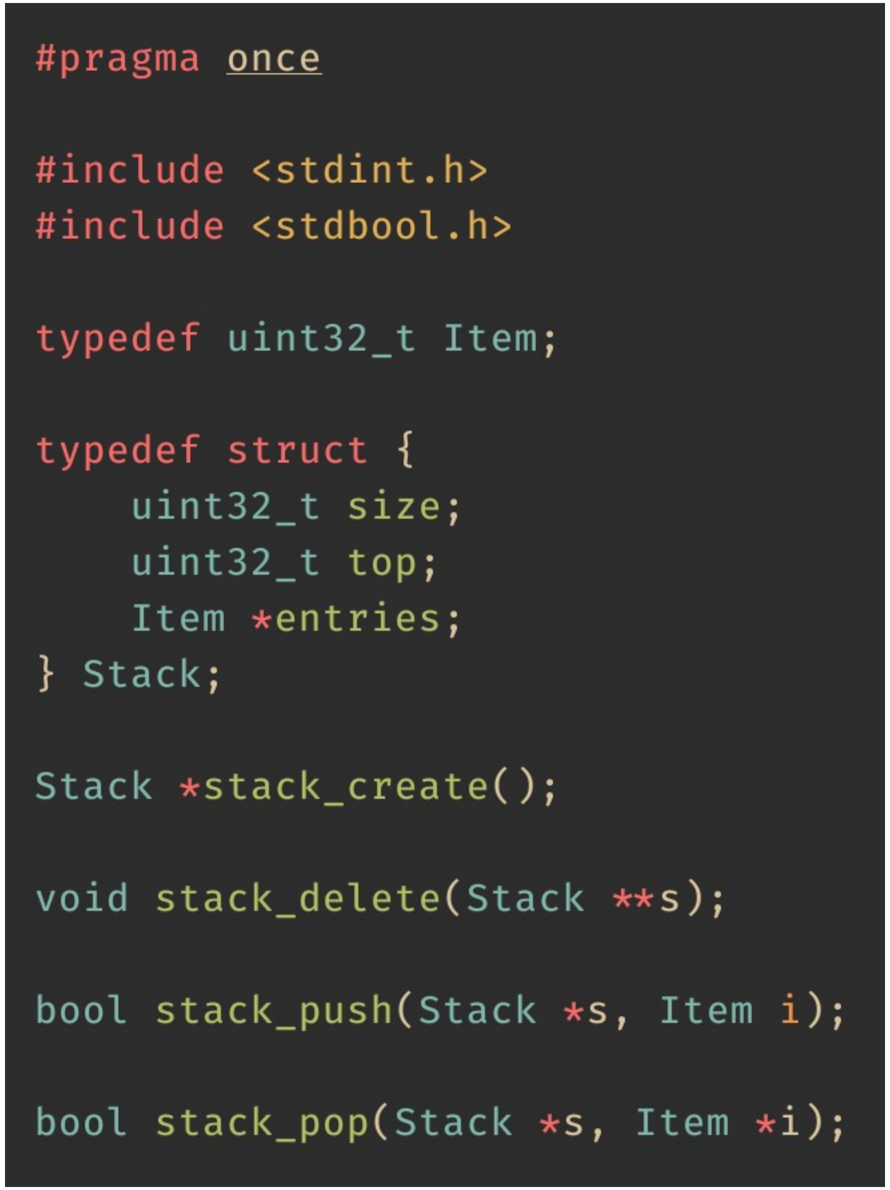
- The header file on the right is for the **stack** abstract data type.
- Note that everything is contained within a header guard using `#ifndef`.



```c
#ifndef __STACK_H__
#define __STACK_H__

#include <stdint.h>
#include <stdbool.h>

typedef uint32_t Item;

typedef struct {
    uint32_t size;
    uint32_t top;
    Item *entries;
} Stack;

Stack *stack_create();

void stack_delete(Stack **s);

bool stack_push(Stack *s, Item i);

bool stack_pop(Stack *s, Item *i);

#endif
```

Same Header File

- The header file on the right is the same as the last slide, but instead uses `#pragma once`.



```c
#pragma once

#include <stdint.h>
#include <stdbool.h>

typedef uint32_t Item;

typedef struct {
    uint32_t size;
    uint32_t top;
    Item *entries;
} Stack;

Stack *stack_create();

void stack_delete(Stack **s);

bool stack_push(Stack *s, Item i);

bool stack_pop(Stack *s, Item *i);
```

#ifndef *versus* #pragma once

- #ifndef checks a macro and will always work.
- #pragma once is a compiler directive, is an extension, and may not always work.
  - But it does work on all modern compilers.
- #pragma once is cleaner.
- #ifndef is more portable.
- Be consistent: use one, or the other!

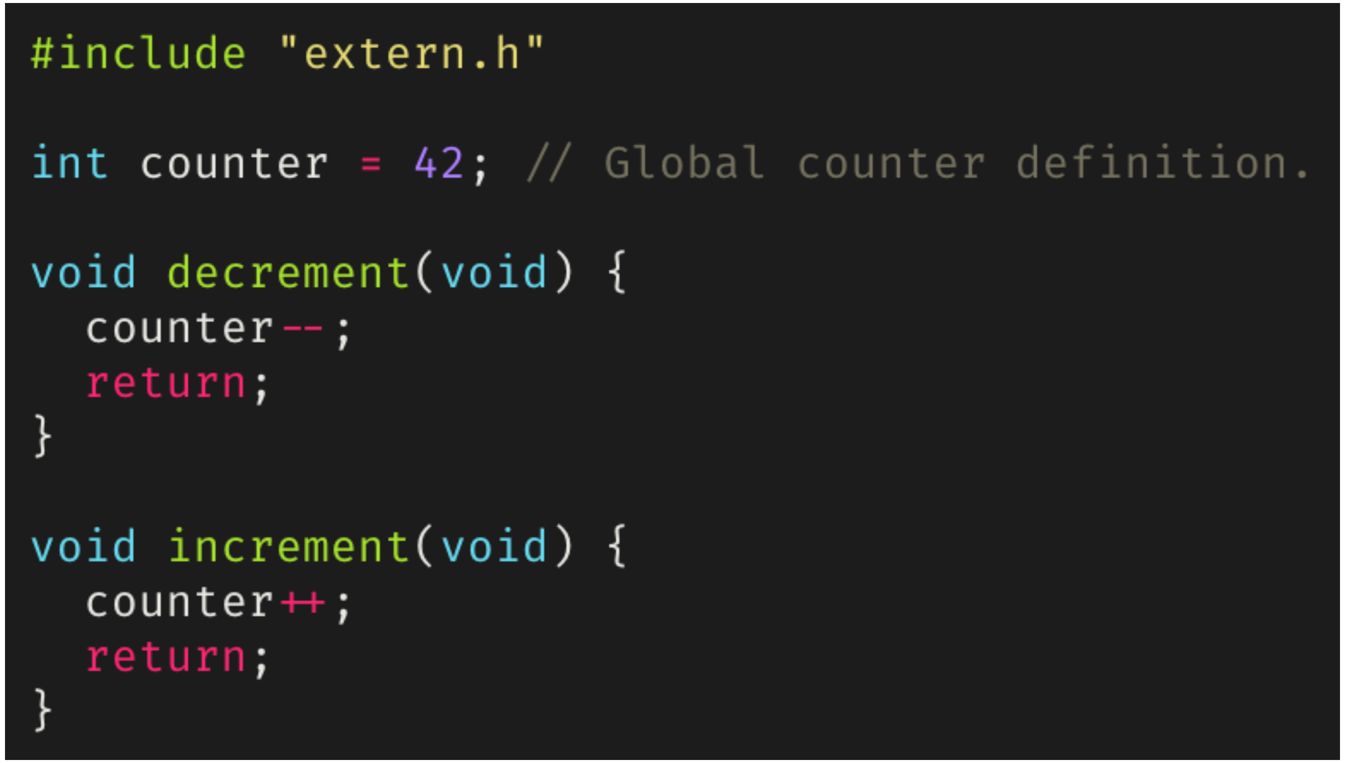
Some Standard Header Files

- `stdio.h` for input/output.
- `inttypes.h` for fixed width integer types.
- `time.h` for date/time utilities.
- `stdbool.h` for boolean types.
- `ctype.h` for functions to determine the type contained in character data.
- `math.h` common mathematical functions.

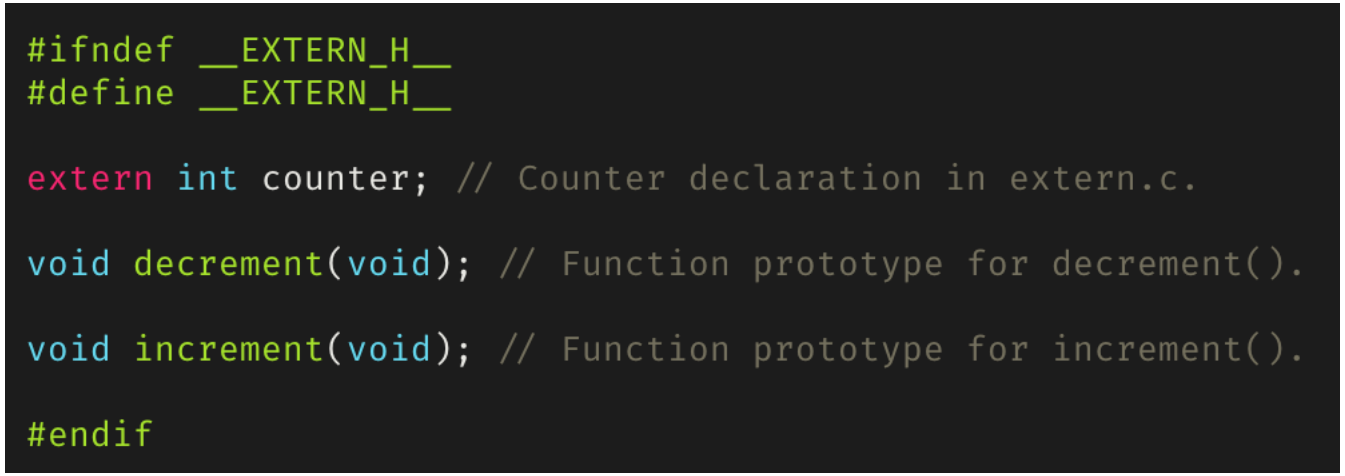
`extern`

- Extends the visibility of variables and functions such that they can be called by any program file, provided that the declaration is known.
- Functions in **C** are implicitly prepended by `extern`.
- Typically used for global variables.
- `extern` variables are declared outside of functions.
- Available until end of the execution of the program

An extern counter



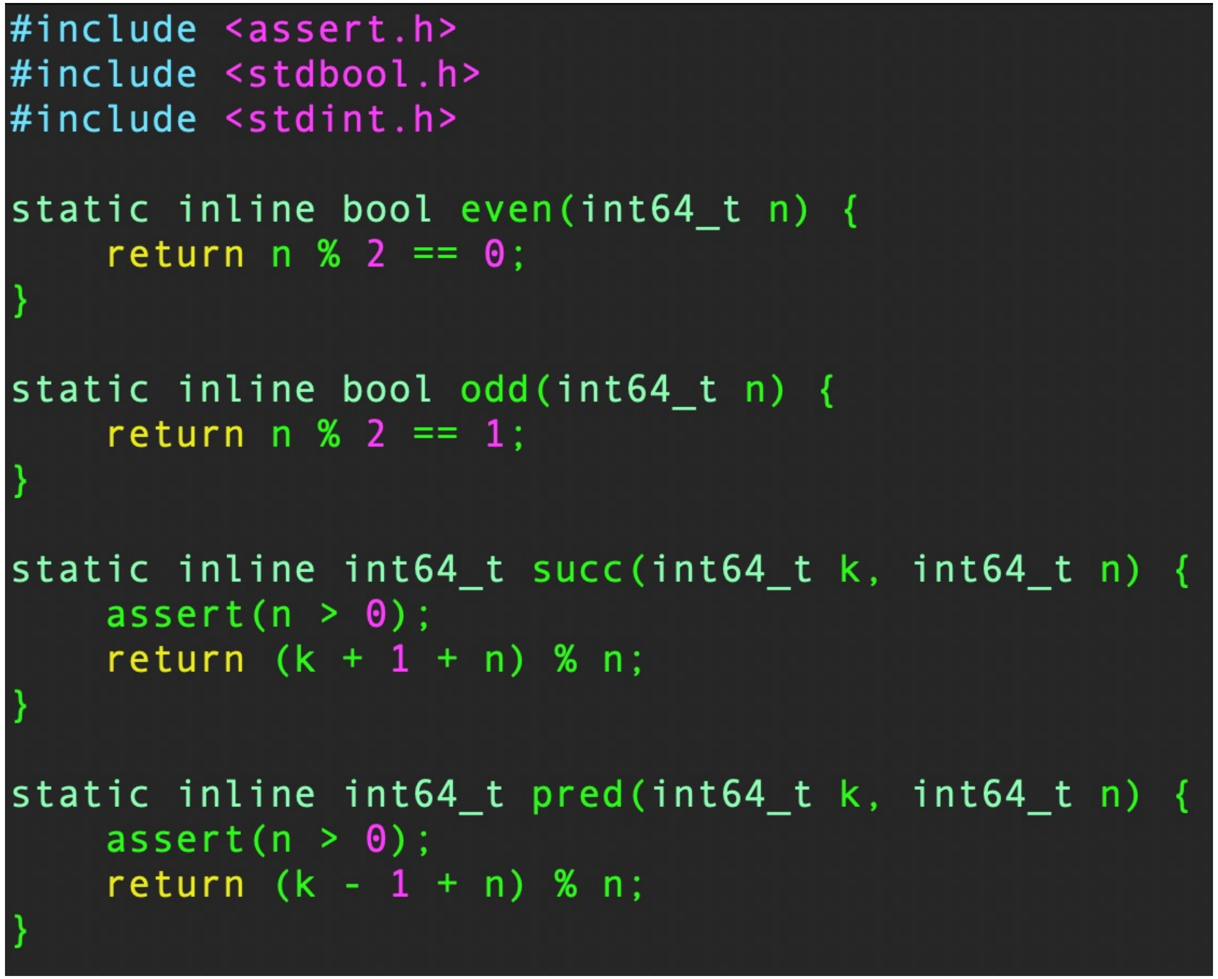
```c
#include "extern.h"

int counter = 42; // Global counter definition.

void decrement(void) {
  counter--;
  return;
}

void increment(void) {
  counter++;
  return;
}
```



```c
#ifndef __EXTERN_H__
#define __EXTERN_H__

extern int counter; // Counter declaration in extern.c.

void decrement(void); // Function prototype for decrement().

void increment(void); // Function prototype for increment().

#endif
```

`static`

- Can be declared inside and outside a function.
- Declared inside a function if the value of the variable needs to *persist* across function calls.
- Declared outside a function if the value of the variable needs to accessed by multiple functions but *only exists within the scope of the file in which it is declared*.
- Available until program finishes execution.



```c
#include <assert.h>
#include <stdbool.h>
#include <stdint.h>

static inline bool even(int64_t n) {
    return n % 2 == 0;
}

static inline bool odd(int64_t n) {
    return n % 2 == 1;
}

static inline int64_t succ(int64_t k, int64_t n) {
    assert(n > 0);
    return (k + 1 + n) % n;
}

static inline int64_t pred(int64_t k, int64_t n) {
    assert(n > 0);
    return (k - 1 + n) % n;
}
```

Utility Functions

Recursion

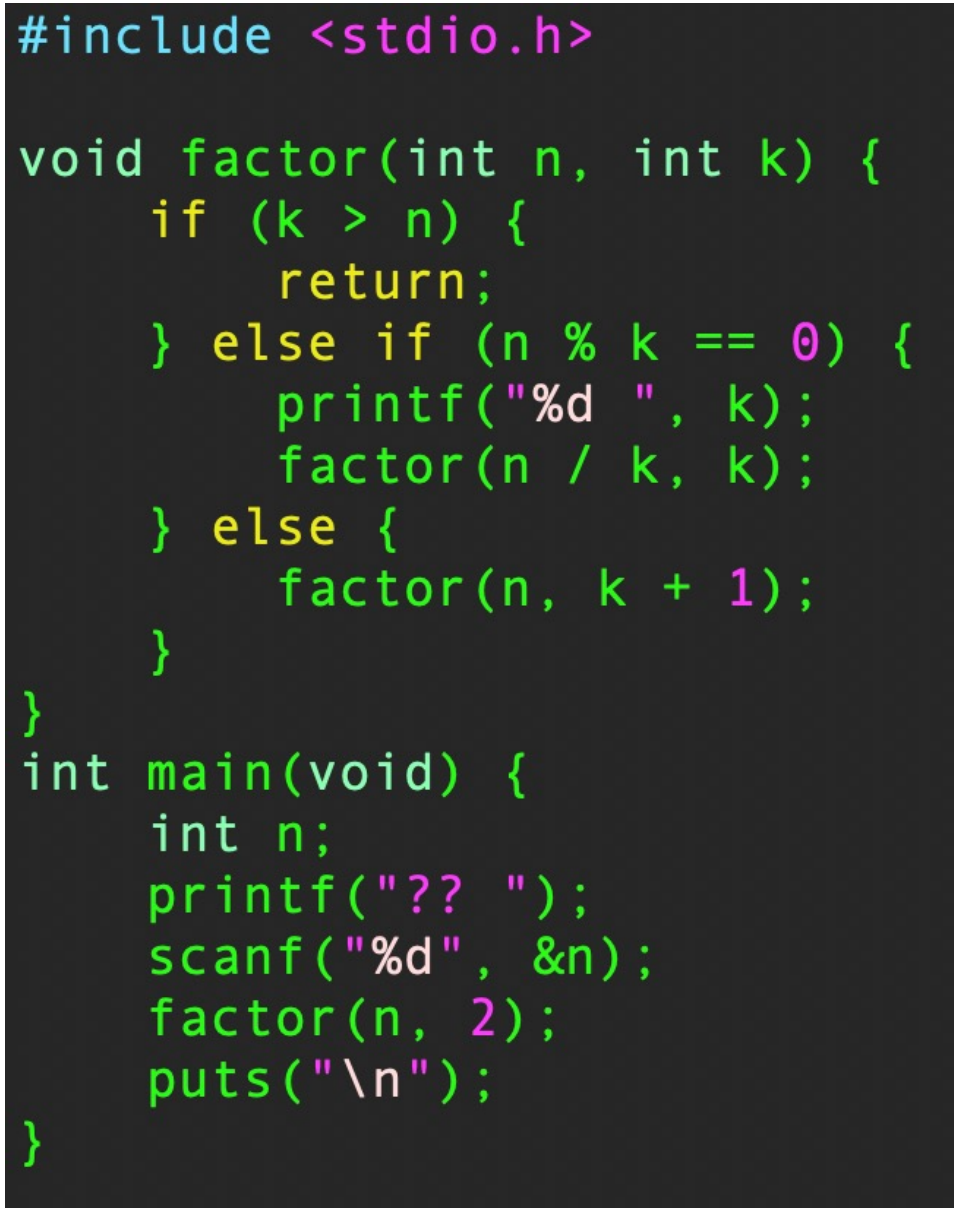
- A function may call itself! We will discuss this in detail later.
- Syntax of recursive functions:

```
void function_name(){
  function_name(); // function calls itself
}
```

- Recursive functions must always define exit conditions to prevent the function from being called an unbounded number of times.
- Recursive code is more compact and may be easier to write and understand.

Factoring (naïvely)

- Print the unique factorization of a positive `int` if it exists.
- If k is a factor of n (evenly divides it) then print k and try again with $^{n}/_{k}$.
- If k is not a factor of n, then try $k + 1$.
- Do this until $k > n$.



```c
#include <stdio.h>

void factor(int n, int k) {
    if (k > n) {
        return;
    } else if (n % k == 0) {
        printf("%d ", k);
        factor(n / k, k);
    } else {
        factor(n, k + 1);
    }
}
int main(void) {
    int n;
    printf("?? ");
    scanf("%d", &n);
    factor(n, 2);
    puts("\n");
}
```

When is this the worst algorithm you might try?

Summary

- Functions provide the ease of running a sequence of code repeatedly.
- Functions can return void, scalar values, pointers and structs (although it is advised to not go this route); they cannot return arrays.
- For this class only use *Snake case* function naming for e.g. *my_function_name.*
- Formal parameters are used inside the body of a function.
- Actual parameters are passed to a function.
- C only uses *Call-by-Value*.
  - Call-by-Reference can be performed by passing a pointer to a function.
- Prior to function calls, you need to define function prototypes:
  - Either at the beginning of a program or in header files.