Functions and Execution

CP:AMA Readings: 9.2, 9.6

CHTP Readings: 5

The primary goal of this section is to introduce what is going on "under the hood" when a function is called.

We will also introduce one of the coolest ideas in all of computer science (recursion).

Motivation

A function is a block of code that performs a specific task. This has a number of benefits:

Reusability: Once a function has been written it can be reused many times.

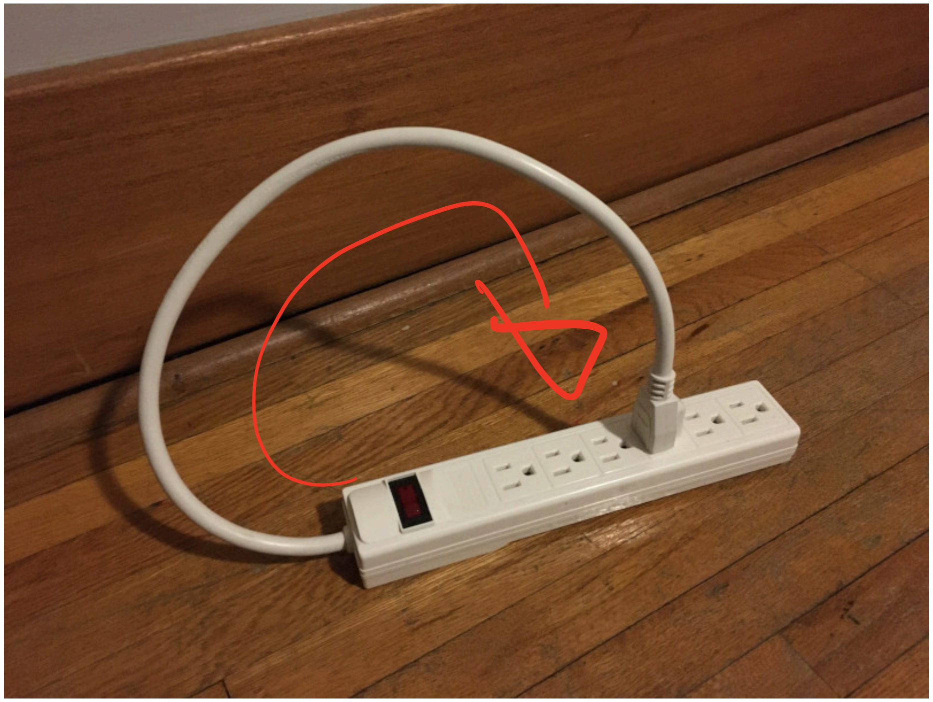
Maintainability: It is easier to test and debug individual functions instead of an entire program. It also improves the readability of the code.

Abstraction: To use a function you only need to understand **what** functionality it provides, but you do not need to understand **how** it is implemented. In other words, you only need an “abstraction” of how it works.

Recursion

A function is ***recursive*** if it calls itself.

At first, this sounds like a circular definition—how can you define something in terms of itself?



. . .Infinite power!

Structure of a recursive function

In order for recursion to work it is necessary for the recursive function to call itself on **a smaller instance of the problem**.

It is also necessary for a recursive function to be able to solve the problem in at least one case **without** making a recursive call.

Every recursive function therefore consists of a ***recursive step*** (when the function calls itself) and at least one ***base case*** (when the function solves the problem directly).

A common error is to forget the base case.

$$\underline{n!} = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

$$\downarrow$$

$$= n \cdot \underline{(n-1)!}$$

$$\mathrm{sum}(n) = n + (n-1) + (n-2) + \cdots + 2 + 1$$

$$= \underline{n + \mathrm{sum}(n-1)}$$

Recursive step

The recursive step must divide the problem into smaller subproblems, at least one of which can be solved by calling the same function.

Finally, the function must return the solution of the original problem using solutions of the subproblems.

example: summation

Say you want to sum the numbers from 0 to n. (Let this be S_n.)

If you knew the value of S_{n-1} you could find the answer, since

$+1+\cdots+n$

$$S_n = 0 + \cdots + n = (0 + \cdots + n - 1) + n = S_{n-1} + n$$

for $n > 0$. We also have that $\boxed{S_0 = 0.}$

$S_1 = 1$

Mathematically, we have

$$S_n = \begin{cases} 0 & \text{if } n = 0, \\ S_{n-1} + n & \text{if } n > 0. \end{cases}$$

```
int sum(int n) {
        if(n ≤ 1)  //base case
            return n;

        return sum(n-1) +n;
}
```

n-2

n-3

⋮

$\boxed{1}$

example: summation (code)

```
// sum(n) returns the sum of numbers from 0 to n
// requires: n >= 0
int sum(int n) {
  if (n == 0) {
    return 0;                // base case
  }
  return sum(n - 1) + n;     // recursive step
}
```

```
sum(0) => 0
sum(1) => 1
sum(2) => 3
sum(3) => 6
sum(4) => 10
```

Multiple recursive calls

A function can call itself more than once so long as each recursive call will eventually converge on some base case.

As an example, we will consider the ***Fibonacci sequence***

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

where the nth term F_n is given by

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \boxed{F_{n-1} + F_{n-2}} & \text{if } n > 1. \end{cases}$$

n: 0, 1, 2, 3, 4, 5, 6

F: 0, 1, 1, 2, 3, 5, 8 - - -

```
int Fib(int n){
    if(n==0)
        return 0;
    if(n==1)
        return 1;
    ~~if(n==2)
        return 1;~~
    return Fib(n-1) + Fib(n-2);
}
```

```
n <= 1
    return n;
```

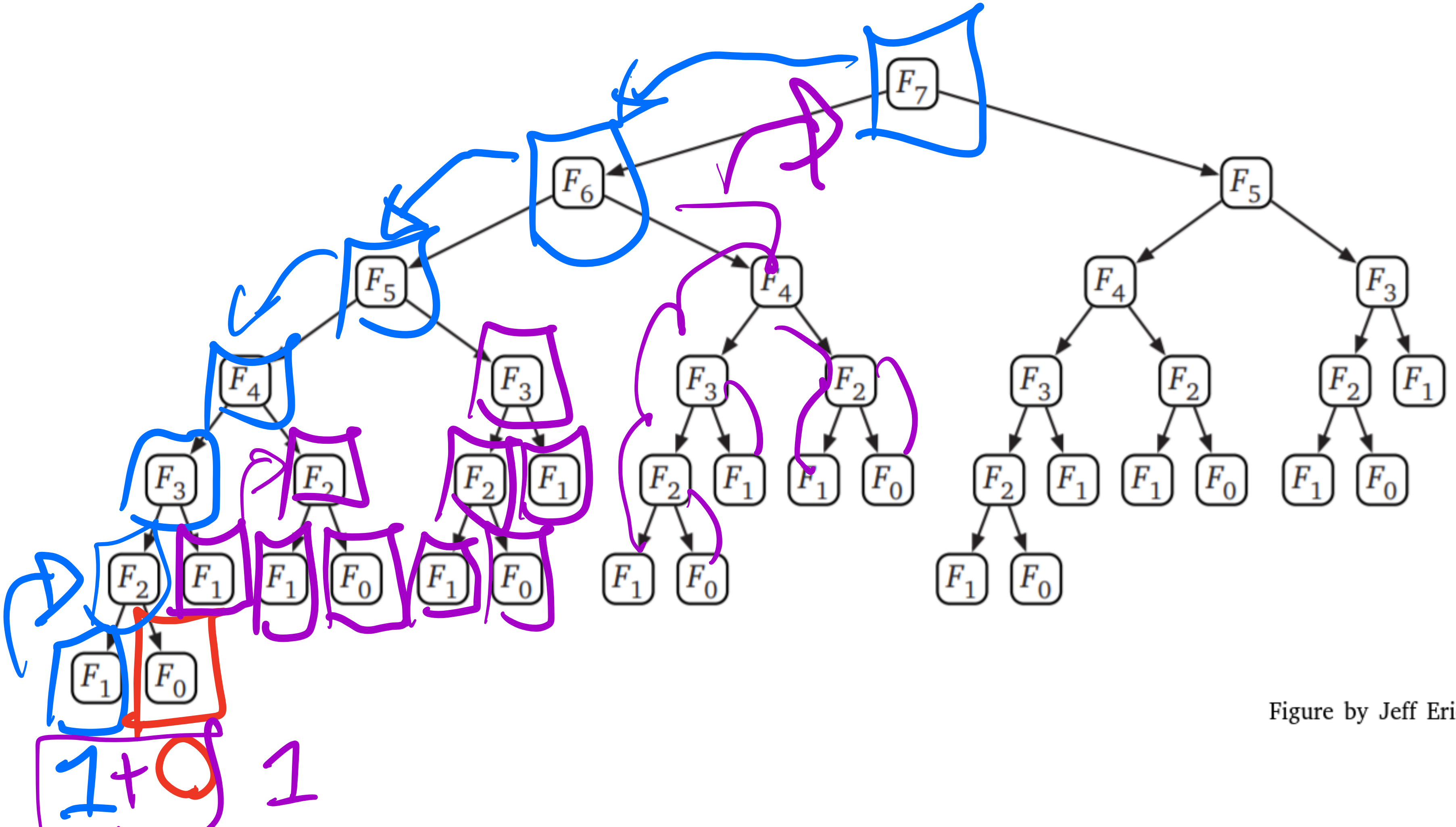
example: Fibonacci (code)

The recursive mathematical definition of the Fibonacci sequence nicely translates into working C code:

```c
// fibonacci(n) returns the nth Fibonacci number
// requires: n >= 0
int fibonacci(int n) {
  if (n == 0) {
    return 0;          // base case
  }
  if (n == 1) {
    return 1;          // base case
  }
  // recursive step
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Calling `fibonacci` on even small n leads to a large number of recursive calls.

This figure shows computing F_7 with each arrow representing a recursive call:



F_7
F_6 F_5
F_5 F_4 F_4 F_3
F_4 F_3 F_3 F_2 F_3 F_2 F_2 F_1
F_3 F_2 F_2 F_1 F_2 F_1 F_1 F_0 F_2 F_1 F_1 F_0 F_1 F_0
F_2 F_1 F_1 F_0 F_1 F_0 F_1 F_0 F_1 F_0
F_1 F_0
1+0
1

Figure by Jeff Erickson

Pros and cons of recursion

Recursion is an alternative to loops (iteration). Any problem that can be solved using recursion can be solved using iteration and vice versa.

Recursion allows short and simple solutions to some complex problems—though not all problems can *naturally* be solved using recursion.

Iterative solutions tend to be more efficient than recursive solutions, but the overhead can be minimized (or removed) with carefulness.

Mutual recursion

C99 requires a function to be declared **before** it is used.

One way of doing this is to order your functions so that if `foo` calls `bar`, then `bar` is defined first.

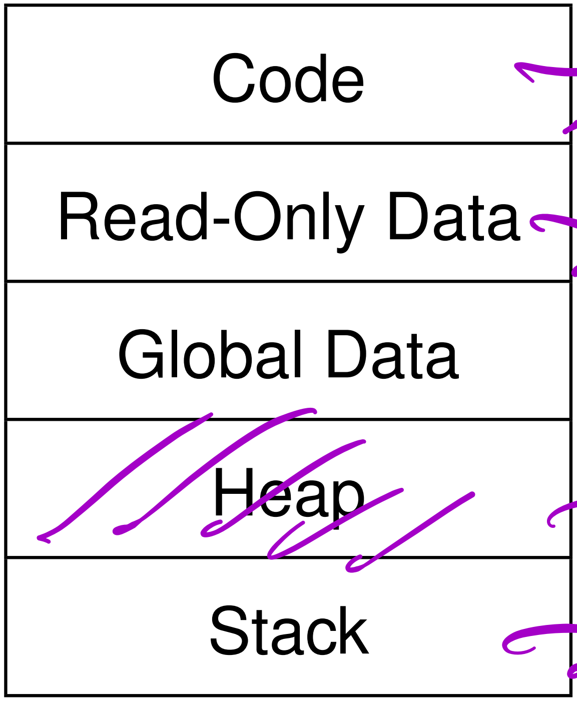
What if `foo` calls `bar` but also `bar` calls `foo`? This is a “catch-22” situation. . .

This is known as mutual recursion. In C, you can use a ***function prototype*** to **declare** function’s name, parameters, and return type. The actual function definition can come later.

```
  int foo(int n);       // Prototype of foo
  int bar(int n);       // Prototype of bar
```

Sections of memory

In these notes we model five ***sections*** (or “regions”) of memory:



Code

Read-Only Data

Global Data

~~Heap~~

Stack

for code

Const int x = 3;

global variables

later

variables in functions

return address

function information

Other courses may use alternative names.

The **heap** section will be introduced in week 9 (Section 8).

Sections are combined into memory ***segments***, which are recognized by the hardware (processor).

When you try to access memory outside of a segment, a **segmentation fault** occurs.

Temporary results

When evaluating C expressions, the intermediate results must be *temporarily* stored.

```
a = f(3) + g(4) - 5;
```

In the above expression, C must temporarily store the value returned from `f(3)` “somewhere” before calling `g`.

In this course, we are not concerned with this “temporary” storage.

The code section

#include<..>
int main(){}

When you program, you write ***source code*** in a text editor using ASCII characters that are “human readable”.

To “run” a C program, the *source code* must first be converted into ***machine code*** that is “machine readable”.

This machine code is then placed into the **code section** of memory where it can be executed.

gcc -Wall code.c

Converting source code into machine code is known as ***compiling***.

The read-only & global data sections

Earlier we described how C "reserves space" in memory for a variable definition. For example:

```
int n = 0;
```

The location of memory depends on whether the variable is **global** or **local**.

First, we discuss global variables.

All global variables are placed in either the **read-only data** section (**constants**) or the **global data** section (**mutable variables**).

Global variables are available throughout the entire execution of the program, and the space for the global variables is reserved **before** the program begins execution.

- First, the code from the entire program is scanned and all global variables are identified.
- Next, space for each global variable is reserved.
- Finally, the memory is properly initialized.
- This happens **before the `main` function is called**.

The read-only and global memory sections are created and initialized at compile time.

example: read-only & global data

```
#include<stdio.h>
// global

int gmv = 13;
int n = 0;
int k;
// read-only

const int c = 42;

int main() {
    int m;

    return 0;
}
```

Memory diagram:

Global	
gmv	13
n	0
k	0

Read-Only	
c	42

Aside

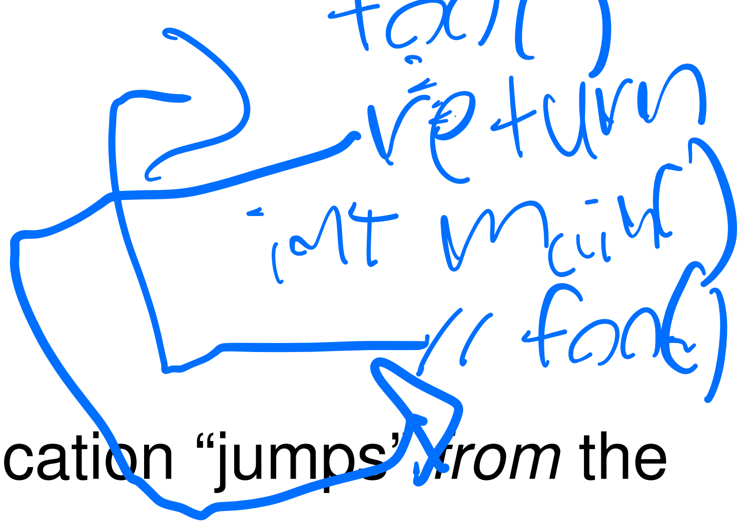
char value of 0 = '\0'

Arrays

int a[3];

[0,0,0]

Function calls (revisited)



Recall the control flow for function calls:

- When a function is called, the program location “jumps” *from* the current location *to* the start of the function

- `return` changes the program location to go *back* to the **most recent** calling function (where it “jumped from”)

- C needs to track where it “jumped from” so it knows where to `return` to

We model function calls with a ***stack***, and store the information in the **stack section** of memory.

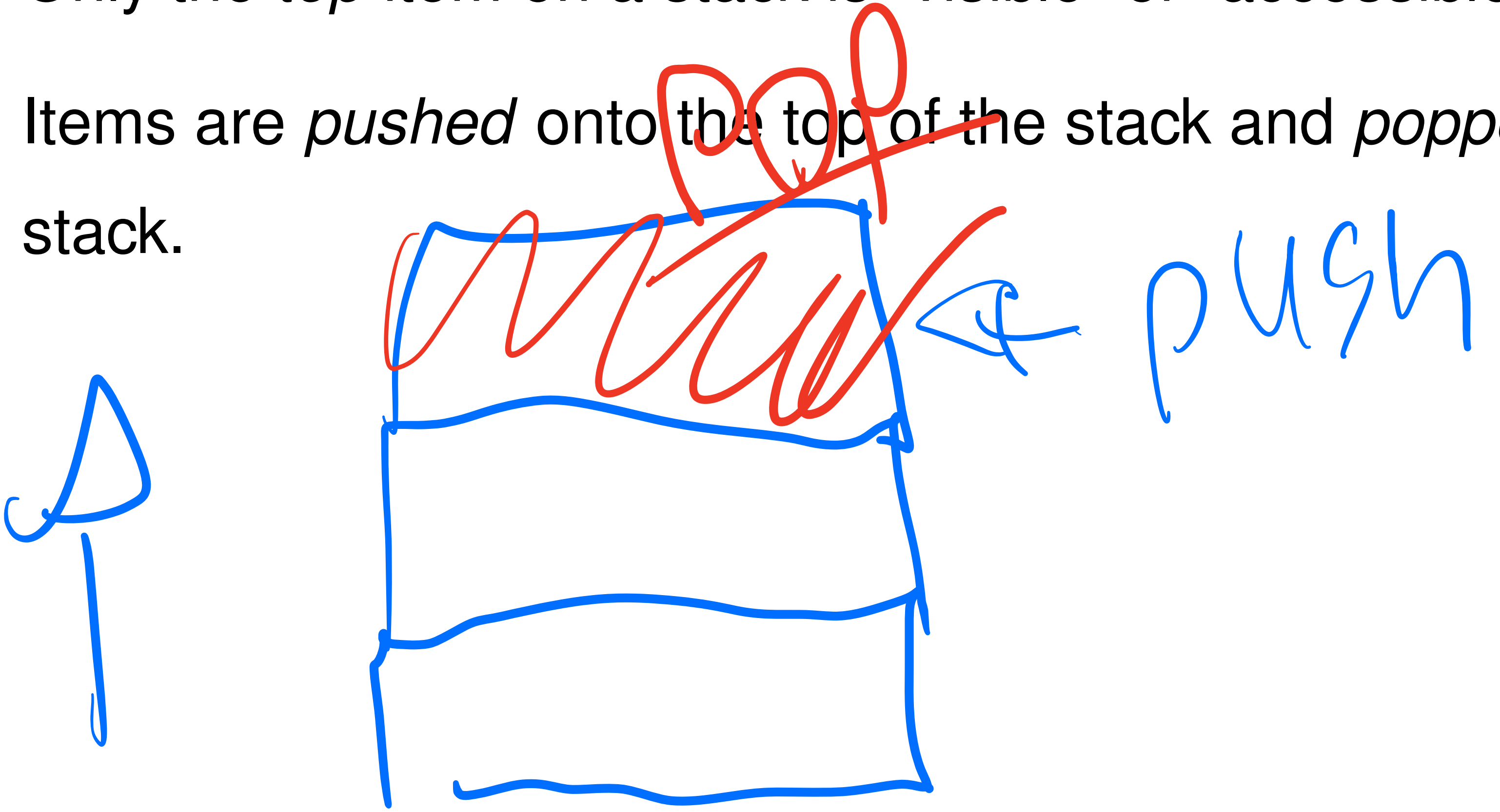
Stacks

A **stack** in computer science is similar to a physical stack where items are “stacked” on top of each other.

For example, a stack of papers or a stack of plates.

Only the *top* item on a stack is “visible” or “accessible”.

Items are *pushed* onto the top of the stack and *popped* off of the stack.



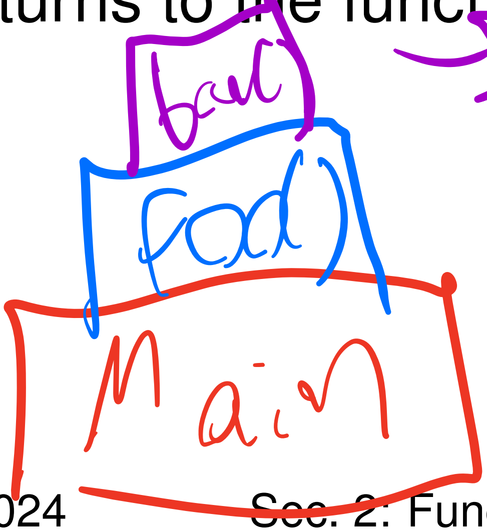
The call stack

Whenever a function is called, we can imagine that it is *pushed* onto a stack, and it is now on the *top* of the stack.

If another function is called, it is then *pushed* so it is now on *top*.

The call stack illustrates the “history” or “sequence” of function calls that led us to the current function.

When a function `return`s, it is *popped* off of the stack, and the control flow returns to the function now on *top*.



→ pops when func-returns

example: call stack

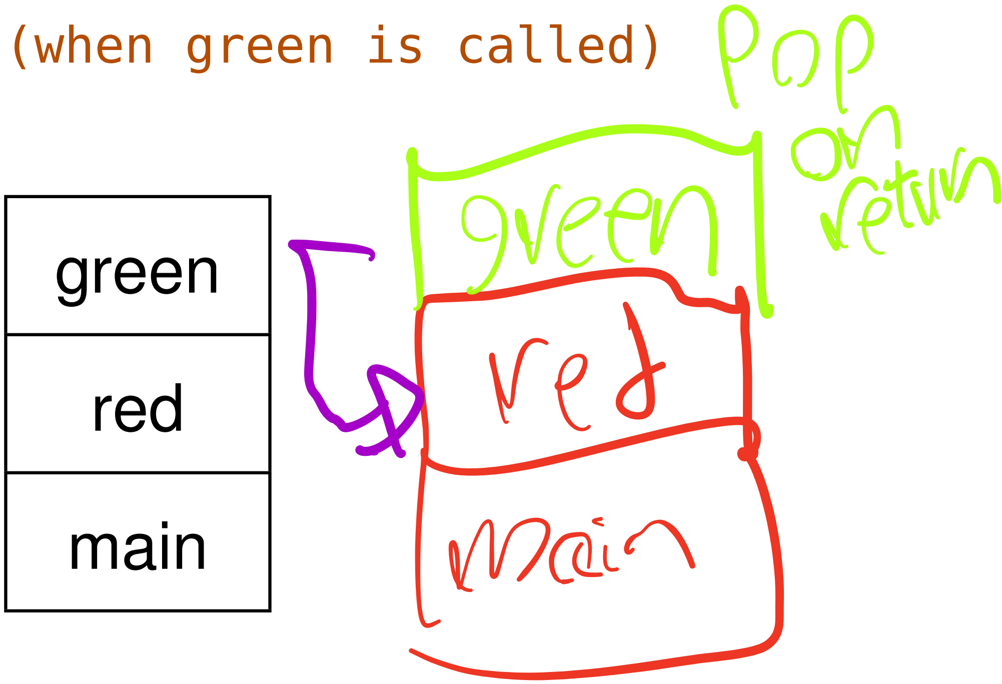
```
void blue(void) {
  return;
}

void green(void) {
  ⇒return;
}

void red(void) {
  green();
  blue();
  return;
}

int main(void) {
  red();
}
```

Call Stack
(when green is called)



When green returns:
* green is popped
* red is now on top
* control flow returns to red

example: call stack 2

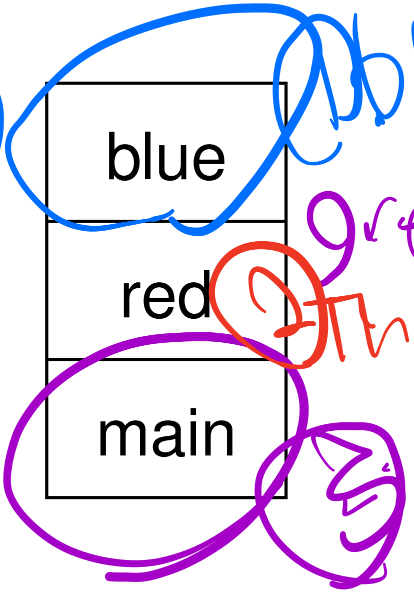
```
1 void blue(void) {
    ⇒return;
  }

  void green(void) {
    return;
  }

  void red(void) {
    green();
    blue();
    return; 4
  }

  int main(void) {
    red(); 2
    return 0;
3 }
```

Call Stack
(when blue is called)



blue gets popped

green was popped

then red gets popped

main pops

green does not appear because it was previously popped before blue was called

The return address

When C encounters a `return`, it needs to know: “where was the program location **before** this function was called?”

In other words, it needs to “remember” the program location to “jump back to” when `return`ing.

This location is known as the ***return address***.

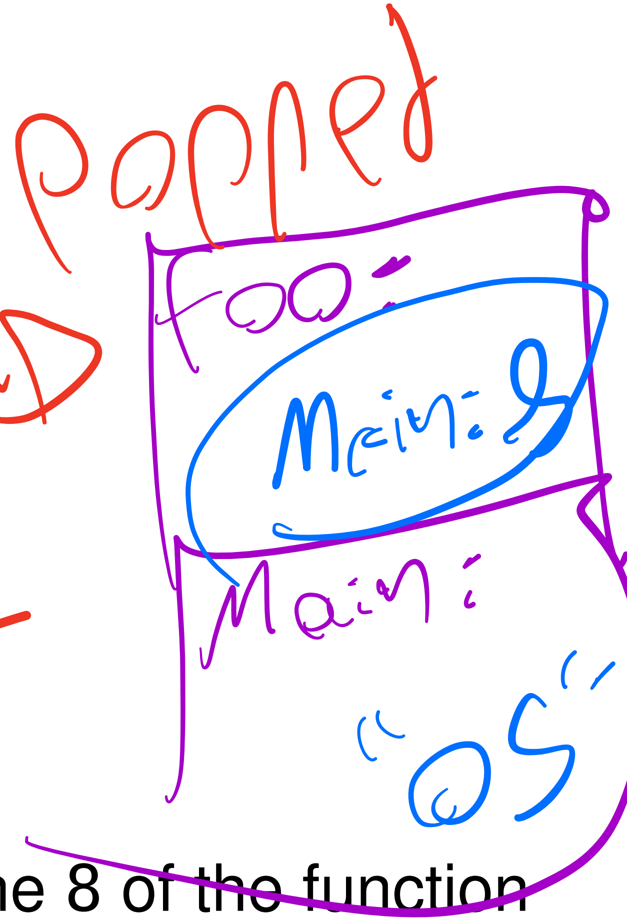
In this course, we use the name of the calling function *and a line number* (or an arrow) to represent the return address.

The *operating system* calls the `main` function, so that is shown as `"OS"`.

example: return address

```
1  void foo(void) {
2    printf("inside foo\n");
3    return;
4  }
5
6  int main(void) {
7    printf("inside main\n");
8    foo();
9    printf("back from foo\n");
10 }
```

foo: 9



When `foo` is called, the program location is on line 8 of the function `main` so we would record the **return address** as:

```
main: 8
```

Stack frames

The “entries” pushed onto the *call stack* are known as ***stack frames***.

Each function call creates a *stack frame* (or a “*frame* of reference”).

Each *stack frame* contains:

- the **argument values**
- all **local variables** (both mutable variables and constants) that appear within the function *block* (including any sub-blocks)
- the **return address** (the program location in the *calling* function to *return to*)

example: stack frames

```
               2
 1  int pow4(int j) {
 2    printf("inside pow4\n");
 3    int k = j * j;  4
 4    // DRAW DIAGRAM
 5    return k * k;
 6  }          16
 7
 8  int main(void) {
 9    printf("inside main\n");
10    int i = 1;
11    printf("%d\n", pow4(i + i));
12  }          16           2
```

```
============================
pow4:
  j: 2
  k: 4
  return address: main:11
============================
main:
  i: 1
  return address: OS
             R/A: OS
============================
```

Before a function can be called, all of the **arguments must be values**, *e.g.*, since `i` has the value 1 the expression `i + i` becomes 2 in the code above before `pow4(i + i)` is called.

Pass by value

foo(x);

C **makes a copy** of each argument value and **places the copy in the stack frame**.

This is known as the “pass by value” convention.

Changes made to the arguments inside a function will only affect the *copies* in the stack frame and not the original argument values.

This is in contrast to the “pass by reference” convention where changes to arguments inside a function *will* affect the values of the original arguments.

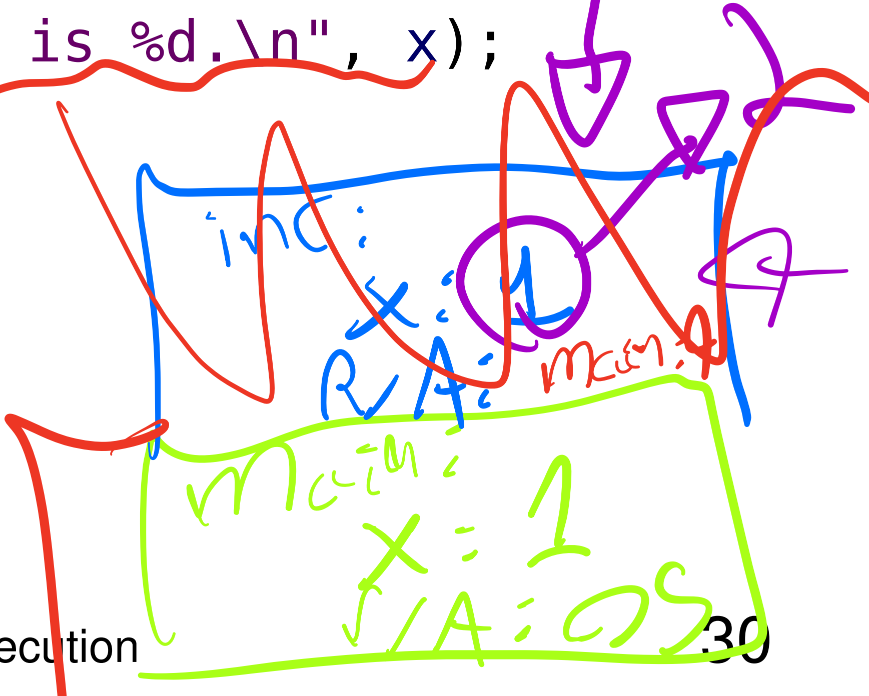
We will see how C simulates pass by reference in Section 3.

Example: pass by value

```
1  void inc(int x) {
2    ++x;
3    printf("The value of x in  inc is %d.\n", x);
4  } return;
5  }

6  int main(void) {
7    int x = 1;
8    printf("The value of x in main is %d.\n", x);
9    inc(x);
10   printf("The value of x in main is %d.\n", x);
11 }
```

```
The value of x in main is 1.
The value of x in  inc is 2.
The value of x in main is 1.
```



Whereas space for a *global* variable is reserved *before* the program begins execution, space for a *local* variable is only reserved **when the function is called**.

The space is reserved within the newly created stack frame.

When the function `return`s, the variable (and the entire frame) is popped and effectively “disappears”.

Calling a function

We can now model all of the **control flow** when a function is called:

- a *stack frame* is created (“pushed” onto the Stack)
- the current program location is placed in the stack frame as the *return address*
- a **copy** of each of the arguments is placed in the stack frame
- the program location is changed to the start of the new function
- the initial values of local variables are set when their definition is encountered

return

When a function `return`s:

- the current program location is changed back to the *return address* (which is retrieved from the stack frame)
- the stack frame is removed ("popped" from the Stack memory area)

The return **value** (for non-`void` functions) is stored in a *temporary* memory area we are not discussing in these notes.

Return vs. return address

Beginners often confuse the return address and `return` statements.

Remember, the return address is a **location from the calling function**.

It has **nothing** to do with the **location of** any `return` statement(s), or if one does not exist (*e.g.*, a `void` function).

There is **always** one (and only one) return address in a stack frame.

Modelling recursion in C

Now that we understand how stack frames are used, we can see how recursion works in C.

In C, each recursive call is simply a new *stack frame* with a separate frame of reference.

The only unusual aspect of recursion is that the *return address* is a location within the same function.

In this example, we also see control flow with the `if` statement.

example: recursion

```
1  int sum(int n) {
2    ⇒if (n == 0) {
3       return 0;
4     } else {
5       return sum(n - 1) + n;
6     }
7  }
8
9  int main(void) {
10   int a = sum(2);
11   //...
12 }
```

0

1 + 2

3

```
==============================
sum:
  n: 0
  return address: sum:5
==============================
sum:
  n: 1
  return address: sum:5
==============================
sum:
  n: 2
  return address: main:10
==============================
main:
  a: ???
  return address: OS
```

popped

3

N/A

main:
a: 3
R/A: OS

Stack section

The *call stack* is stored in the ***stack section***, the fourth section of our memory model. We refer to this section as “the stack”.

In practice, the “bottom” of the stack (*i.e.*, where the `main` stack frame is placed) is placed at the *highest* available memory address. Each additional stack frame is then placed at increasingly *lower* addresses. The stack “grows” toward lower addresses.

If the stack grows too large, it can “collide” with other sections of memory. This is called *“stack overflow”* and can occur with very deep (or infinite) recursion.

Uninitialized memory

In most situations, mutable variables *should* be initialized, but C allows variable definitions without any initialization.

```
int i;
```

Uninitialized
No default

on
Stack

For **global** variables, C automatically initializes the variable to be zero, but it is good style to explicitly initialize global variables.

```
int g = 0;
```

int g;

global data
default is 0

Local variables are preferred over global variables. Don't use global variables at all in this course.

A **local** variable (on the *stack*) that is uninitialized has an **arbitrary** initial value.

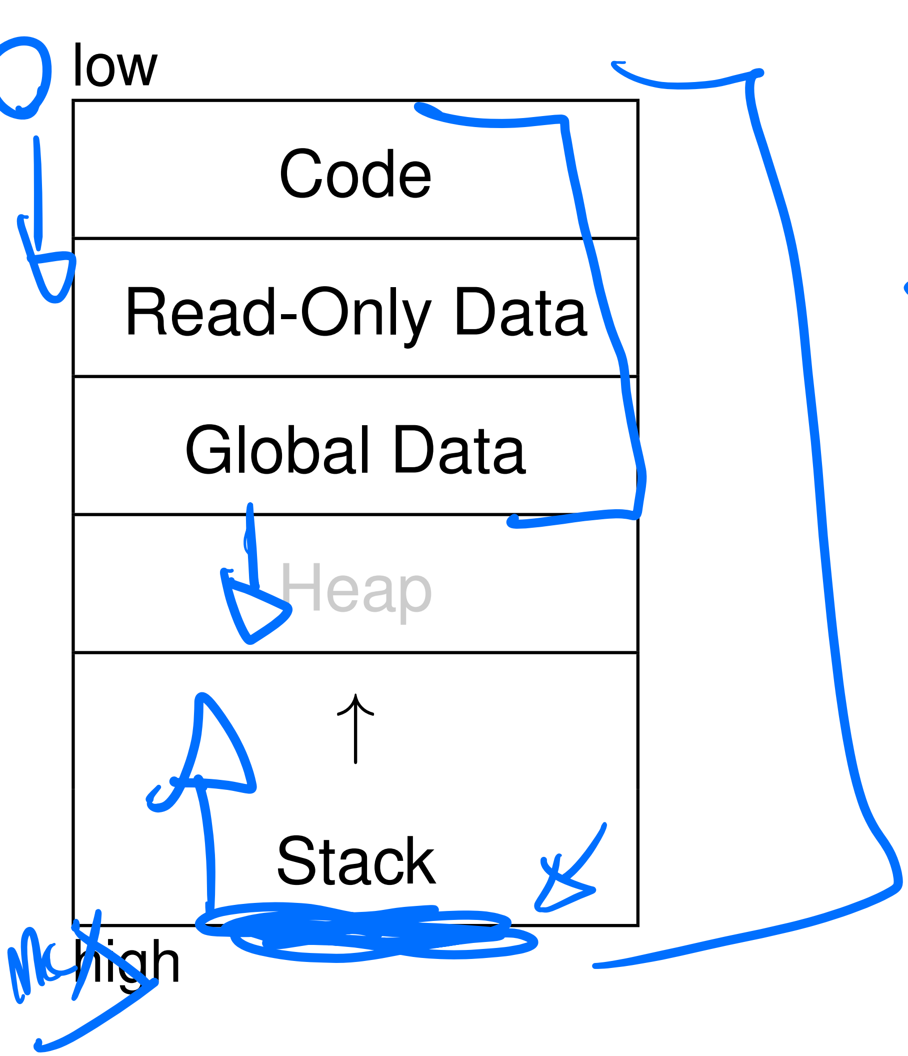
```
void mystery(void) {
  int k;
  printf("the value of k is: %d\n", k);
}
```

Your compiler may give you a warning if your code uses the value of an uninitialized variable.

-Wall

> In the example above, the value of k will likely be a leftover value from a previous stack frame.

Memory sections (so far)


low

Code

Read-Only Data

Global Data

Heap

↑

Stack

high

RAM

Scope vs. memory

Just because a variable exists in memory, it does not mean that it is *in scope*.

Scope is part of the C syntax and determines when a variable is “visible” or “accessible”.

Memory is part of our C model (which closely matches how it is implemented in practice).

```
{
    int x = 0;
}
```

end of scope

x = 2; //Error

example: snapshot and scope

```
1  int foo(void) {
2    // SNAPSHOT HERE (at line 2)
3    // 5 variables are in memory,
4    // but none are in scope
5    int a = 1;
6    {
7      int b = 2;
8    }
9    return a;
10 }
11
12 const int c = 3;
13 int d = 4;
14
15 int main(void) {
16   int e = 5;
17   foo();
18 }
```

```
READ-ONLY DATA:
c: 3
GLOBAL DATA:
d: 4

STACK:
=========================
foo:
  a: ???
  b: ???
  return address: main:17
=========================
main:
  e: 5
  return address: OS
```

Model

We now have the tools to model the behaviour of a C program.

At any moment of execution, a program is in a specific *state*, which is the combination of:

- the current *program location*, and
- the current contents of the *memory*.

To properly interpret a program's behaviour, we must keep track of the program location and all of the memory contents.

Goals of this Section

At the end of this section, you should be able to:

- understand recursion and write recursive functions in C
- describe the 4 areas of memory seen so far: code, read-only data, global data and the stack
- identify which section of memory an identifier belongs to
- explain a stack frame and its components (return address, parameters, local variables)
- explain how C makes copies of arguments for the stack frame