CSE 374 Programming Concepts and Tools

Lecture 07 - Intro to C

Today

A new programming language, **C**!

Today we'll cover the basics of C:

- The C language
- A "Hello World" example
- Variables, pointers, arrays, and strings
- Call traces and the stack
- Getting familiar with the documentation

Discussion

Turn and talk:

What comes to mind when you think about the C programming language? Any particular terms or even some stereotypes?

C Language

The C Programming Language

Invented to rewrite the Unix OS (ancestor of Linux/MacOS)

A “low level” language gives the developer the ability to work directly with memory and processes

- Low level means it sits closer to binary, the language the CPU uses.
- Java is a “high level” language, compiles to bytecode, has a garbage collector that manages memory for you

Useful for software that requires low-level OS interaction

- Robotics, mobile, high performance software, microcontrollers (using Arduino C)

Ancestor of most modern languages: Java, C++, C#

C vs Java

C

- low level
- *user* responsible for memory
- “functions”
- No classes - NOT object oriented
- compiled
- conditional control flow
- modern syntax (human readable)
- small standard library

portable

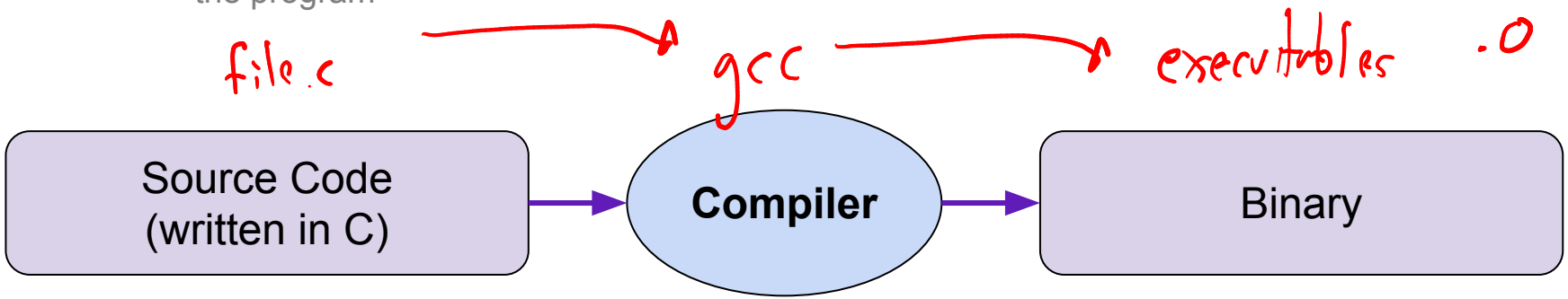
Java

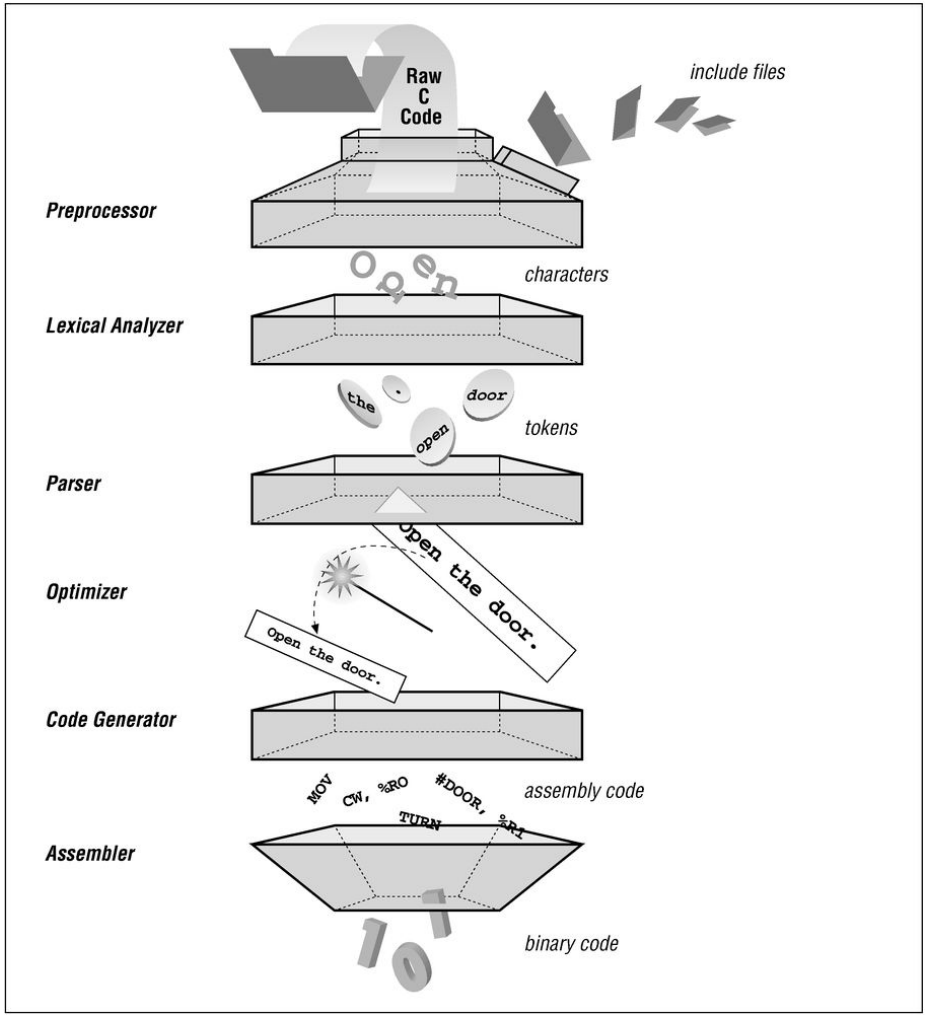
- high level
- memory managed (garbage collection)
- “methods”
- classes define objects
- compiled
- conditional control flow
- modern syntax (human readable)
- large standard library, HUGE extended libraries

How C Works

You compile your C programs into binary, which is the only language that your CPU understands.

- We call them "binary" programs because they're composed of bits: 0s and 1s!
- Bash and Python are *interpreted*.
  - Instead of being turned into machine instructions and then being run, an interpreter evaluates the program




Raw
C
Code
include files
Preprocessor
characters
Lexical Analyzer
tokens
Parser
Optimizer
Open the door.
Code Generator
assembly code
MOV
CX,
%RO
#DOOR,
%R1
TURN
Assembler
binary code

Using GCC to compile

GCC (GNU Compiler Collection) is the compiler we will use

- Translates C into an executable binary (or assembly with `-s`).

Only 1 file

Compile a C program like this

```
gcc [options] -o <output exe> <c file>
```

- Ex: `gcc -o hello hello.c`
- Conventionally the executable has the same name as the file

./hello

Always use these options: `-g -Wall -std=c11`

- `-g` enables debugging
- `-Wall` checks for all warnings
- `-std=c11` uses the 2011 C standard, what we will use for the class

Full command: `gcc -g -Wall -std=c11 -o hello hello.c`

$1 $2

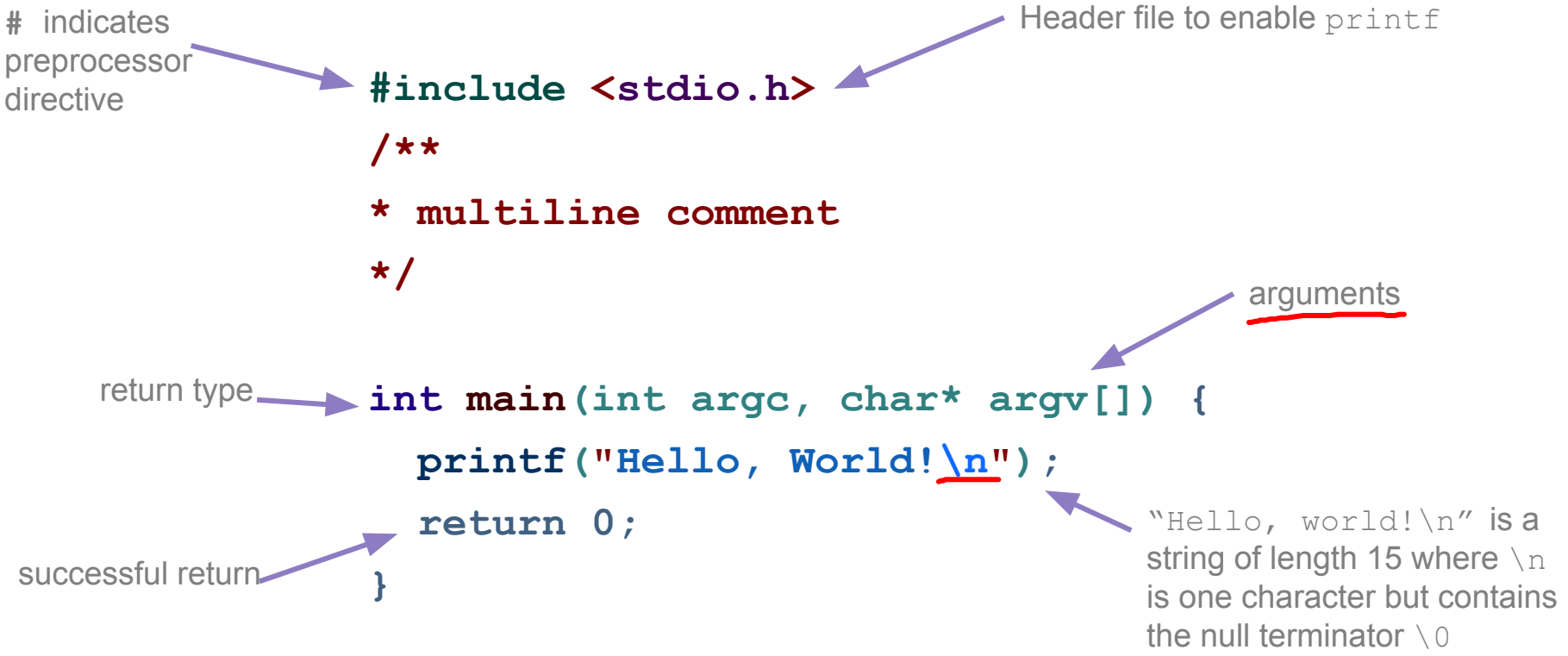
C: Hello World

Hello World

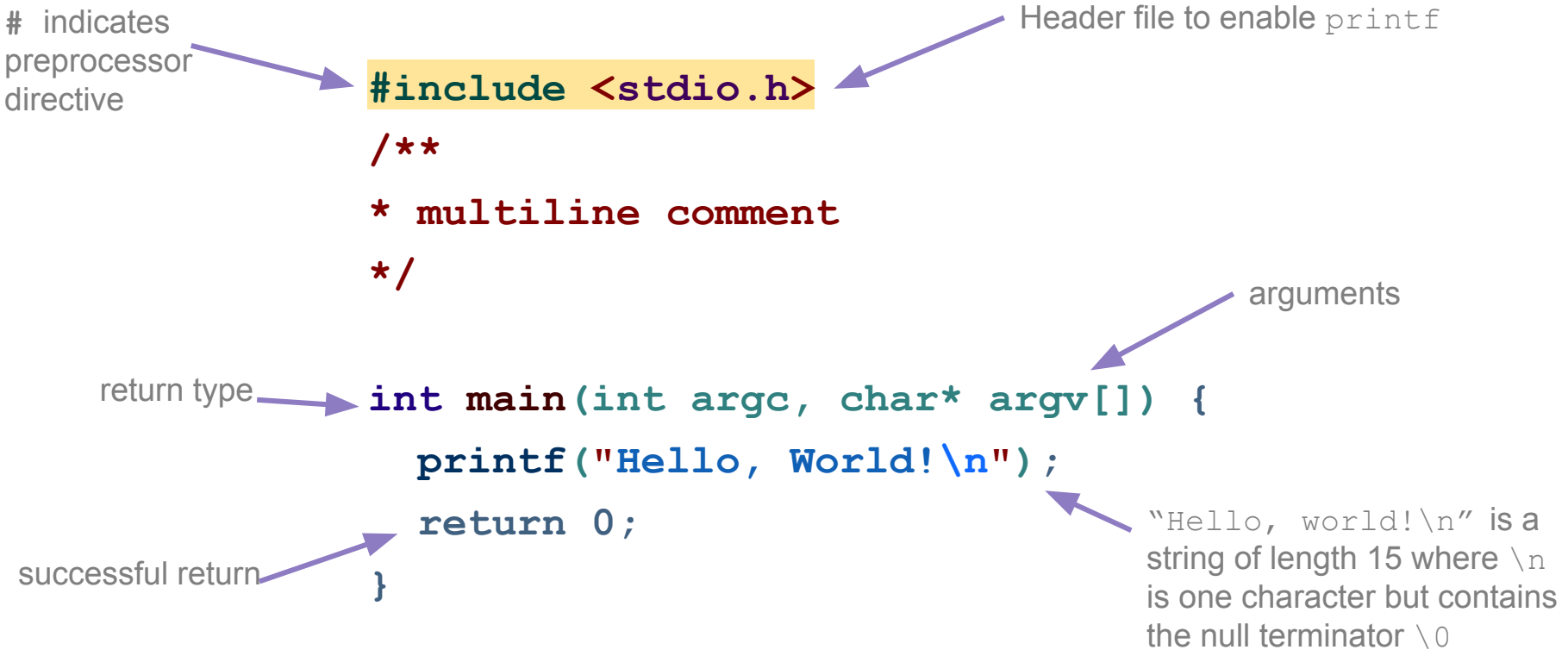
```c
#include <stdio.h>
/**
* multiline comment
*/
// Single line comment
int main(int argc, char* argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

Demo: Hello World

Hello World


# indicates preprocessor directive
Header file to enable printf
#include <stdio.h>
/**
* multiline comment
*/
arguments
return type
int main(int argc, char* argv[]) {
printf("Hello, World!\n");
return 0;
successful return
}
"Hello, world!\n" is a string of length 15 where \n is one character but contains the null terminator \0

Hello World


# indicates preprocessor directive
Header file to enable printf
#include <stdio.h>
/**
* multiline comment
*/
arguments
return type
int main(int argc, char* argv[]) {
printf("Hello, World!\n");
return 0;
successful return
}
"Hello, world!\n" is a string of length 15 where \n is one character but contains the null terminator \0

#include

Provides access to code in another file, similar to Java import statements

Inserts the code of another file into the current C file

- Usually used with `.h` files (header files), which contain information that is shared between a *module* (collection of functions that perform related tasks)

```
#include <some_standard_library.h> // standard libraries
```

- Searches for `some_standard_library.h` in “system include” directories, and includes that file

```
#include “my_library.h” // developer files
```

- Searches current directory – this lets developers break project into smaller files

What kind of code is #include?

Not normal code – preprocessor directive

- “instructions for the C preprocessor”

Executed by preprocessor (part of compiler’s job)

- More on this later

`stdio.h` is a special header file that is always available

`#include <stdio.h>` will make all of its functions globally visible

- Tells the preprocessor to take the file `stdio.h` (Standard I/O) and insert it in the program
- Provides foundational set of input and output functions: `printf`, `fgets`
- Other useful standard libraries: `stdlib`, `math`, `assert`, `string`

C Preprocessor

Before compiling your code, gcc runs the C preprocessor on it

- Removes comments (both `//` and `/* */` style)
- Handles preprocessor directives (the lines that start with `#`)

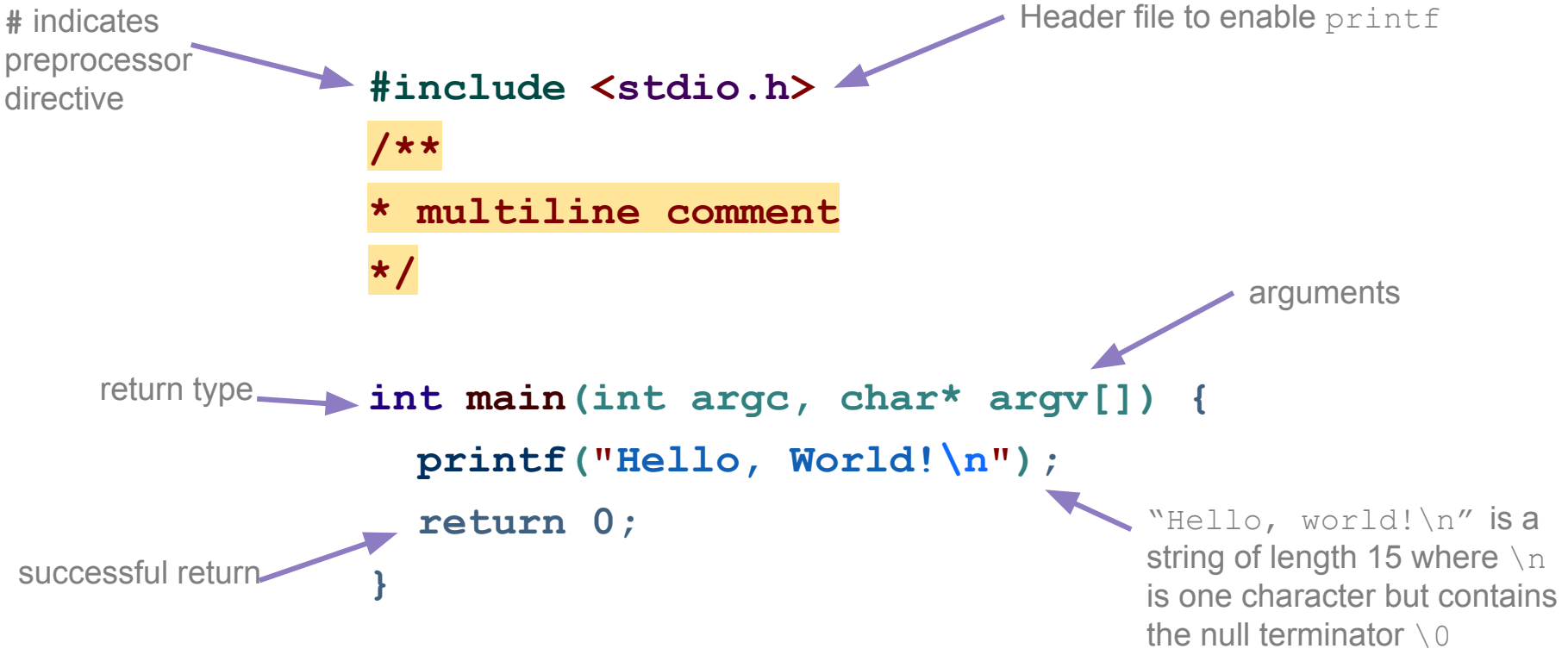
`#include <somefile.h>`

- Literally pastes the standard library file somefile.h into your C code
- C equivalent of import statements in Java

`#define MY_MACRO INSERT_THIS_TEXT`

- Find and replaces MY_MACRO with INSERT_THIS_TEXT in your code
  - Again, pure text replacement.
  - Note, **no semicolon**!
- Ex: **`#define MAX_LENGTH 500`**

Hello World: Comments



Comments in C

```
/* long form, multiline comments */
```

- Provide detailed explanations or information
- Uses:
  - Documenting function prototypes (usually start with `/**` to alert IDE that it's documentation)
  - Disabling blocks of code

```
// shorter, 1-line comments
```

- Add short explanations
- Uses:
  - Debugging and troubleshooting
    - Delta debugging
  - TODOs and future improvements, like `// TODO: Move to the math functions file`

Hello World


# indicates preprocessor directive
Header file to enable printf
#include <stdio.h>
/**
* multiline comment
*/
arguments
return type
int main(int argc, char* argv[]) {
printf("Hello, World!\n");
return 0;
successful return
}
"Hello, world!\n" is a string of length 15 where \n is one character but contains the null terminator \0

Main function

Main is the first thing run once it the program starts executing

- May call other functions, library functions, etc.

```
int main(int argc, char* argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

- `argc` stores the number of arguments (including the program name) (#? in bash!)
- `argv` is the array of inputs from the command line ($@ in bash!)

Expect a return of 0 for successful execution or 1 for failure (again, just like bash!)

Arguments for `main`

`char* argv[]`

In English: `argv` is expected to be an array of strings (command-line arguments) where each element is a pointer to a null-terminated character array (aka string).

- **`char`**: character datatype
- **`char*`**: a pointer to a character or a sequence of characters ending in **`\0`** (null terminator), which can be interpreted as a **string**.

`int argc`

In English: number of pointers stored in **`argv`**

- In C, array lengths must be sent as separate arguments
- Also access values with **`argv[0], argv[1], …, argv[argc-1]`**
  - **`argv[0]`** is the name of the program being run (like bash **`$0`**!)

Note: The **`argc`** and **`argv`** parameters are *not required* in the **`main`** function unless your program needs to handle command-line arguments.

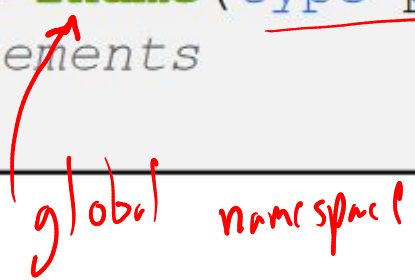
Functions

C programs are broken up into functions

- Functions are the main building blocks of C code
- Named portions of code that can be referenced by code elsewhere
- Similar to methods in java

Syntax is almost exactly the same as Java methods

```
returnType fname(type param1, …, type paramN) {
    // statements
}
```



Function Declaration vs. Definition

In C, a function can be declared before it is defined.

Declaration – specifies the function name, return type and parameters

```
int square (int n);
```

- The function signature ends in ;
- Similar to interface declarations of methods in Java
- Exist so you can call a function before you fully define it

stored in a .h file

Definition – declaration plus the code to run

```
int square (int n) {
    return n * n;
}
```

- You will get an error if a function is used but not defined (Java equivalent of “symbol not found”)

.c

Function Prototypes

In C, you cannot call a function before you declare it!

- It is okay to call a function that is defined later in the file, as long as it has already been declared

Style best practices:

- Put function declarations (aka function prototypes) at the top of the file, before any definitions
- The `main` function should be the first function being defined
  - Doesn't need a declaration because typically your other functions will not run main
- Comment what a function does right above its declaration

```
/* Compute the area of a triangle */
float triangle(float width, float height);
```

Functions that don't take parameters

In C, you need to do the following to declare a function with no parameters:

```
<return type> <function name>(void);
```

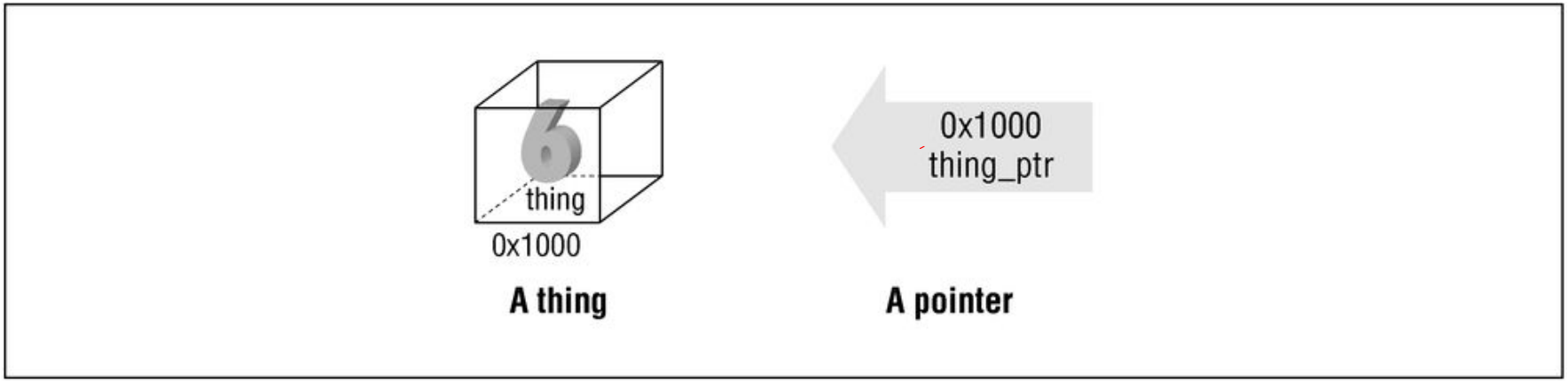
After declaring it like this, you can define it how you would normally in Java:

```
<return type> <function name>() {

  // function code…

}
```

Variables, Pointers, Arrays, and Strings

Quick view of Pointers

The address of **`thing`** is **`0x1000`**. Addresses are automatically assigned by the C compiler to every variable. Our pointer (**`thing_ptr`**) points to the variable **`thing`**. Pointers are also called “address variables” because they contain the addresses of other variables.

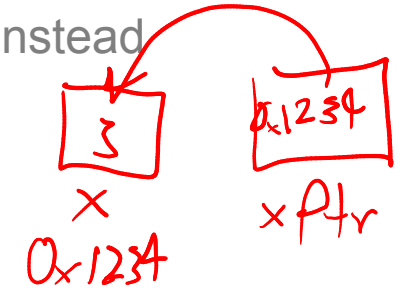


Variables

C variable types: int, char, double, arrays, pointer (details)

- No booleans, use int values of **nonZero=true** and **0=false** instead
  - ⚠️ WARNING: opposite of bash !!! ⚠️

<type> <name> = <value>;



- Left side gives locations, right side evaluates to values

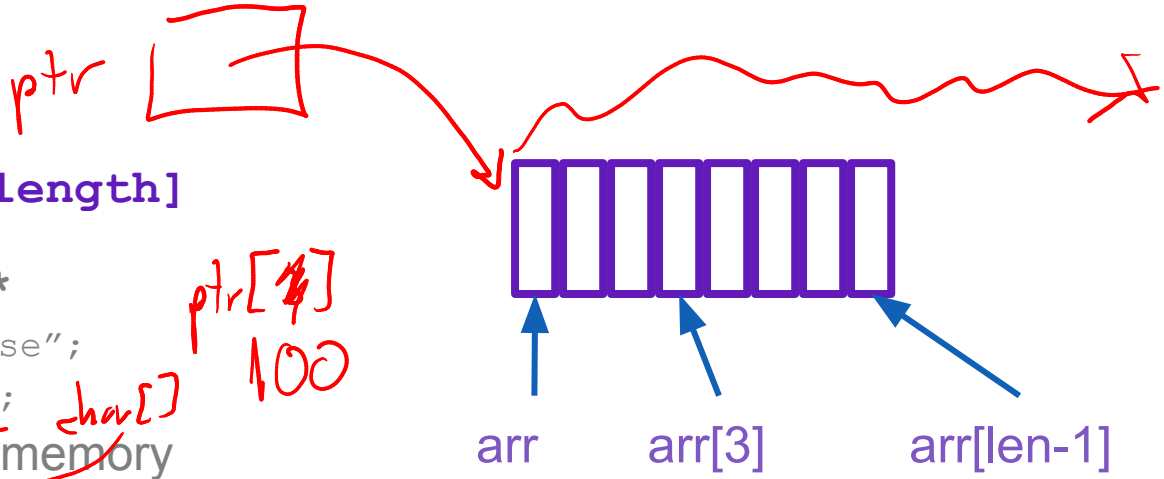
```
int x = 1;          // stores value 1 at location labeled x
char c = 'a';       // stores value a at location labeled c
double d = 2.5;     // stores value 2.5 at location labeled d
int* xPtr = &x;     // stores address (&) of location x at xPtr
x = 2;              // stores value 2 at location x
*xPtr = 3;          // stores value 3 at the address (*) in xPtr
```

"pointer"

"address of"

More on * and & next lecture!

Arrays in C



<datatype> <name>[length]

- Type: **<datatype>***
  - `char arr[] = "cse";`
  - `char* ptr = arr;`
- Contiguous block of memory
- Stores the location in memory of the first value

Arrays must be declared with a known length (so compiler can allocate space)

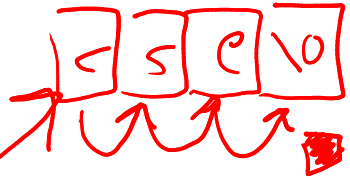
- This size is not stored like in Java, you have to save length as a separate variable

No default values, arrays will hold whatever was in that spot before you declared it so accessing those indices before initializing can cause errors

Strings in C

All three of these are equivalent ways of defining a string in C:

```
char s1[] = {'c', 's', 'e', '\0'};
char s2[] = "cse";
char* s3 = "cse";
```



There are no “string” in C, only arrays of characters

- “null terminated (`\0`) array of characters”

`char*` is another way to refer to strings in C

- Technically is a pointer to the first char in the series of chars for the string

Strings cannot be concatenated in C

- `printf(“hello, “ + myName + “\n”); // will not work`

Strings in C

```
int main(int argc, char* argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

[H, e, l, l, o, ",", " ", W, o, r, l, d, !, \n, \0]

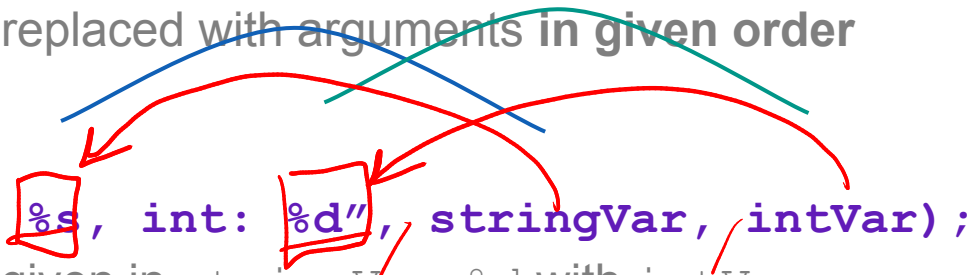
- Strings terminate with `\0` so their length can be determined
- Therefore, it is a string of length 15 (\n is one character, and contains \0)
- Fortunately, functions like **`strlen`** report the length as 14 (excluding the \0).

Printf: print format function

Sends string literals to stdout based on given string with format tags

- Format tags are stand-ins for where things should be inserted into the string literal
- `%s` – string with null termination, `%d` – int, `%f` – float
- Number of format tags should match number of arguments
  - Format tags will be replaced with arguments **in given order**

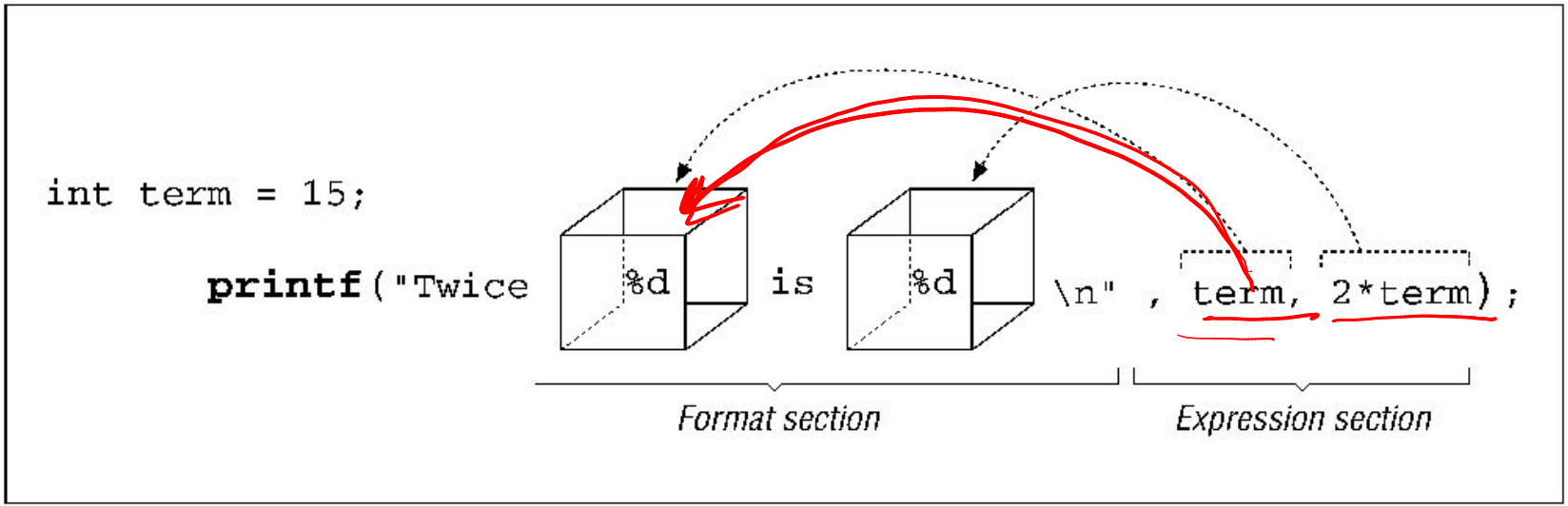
Defined in **`stdio.h`**



```
printf("string: %s, int: %d", stringVar, intVar);
```

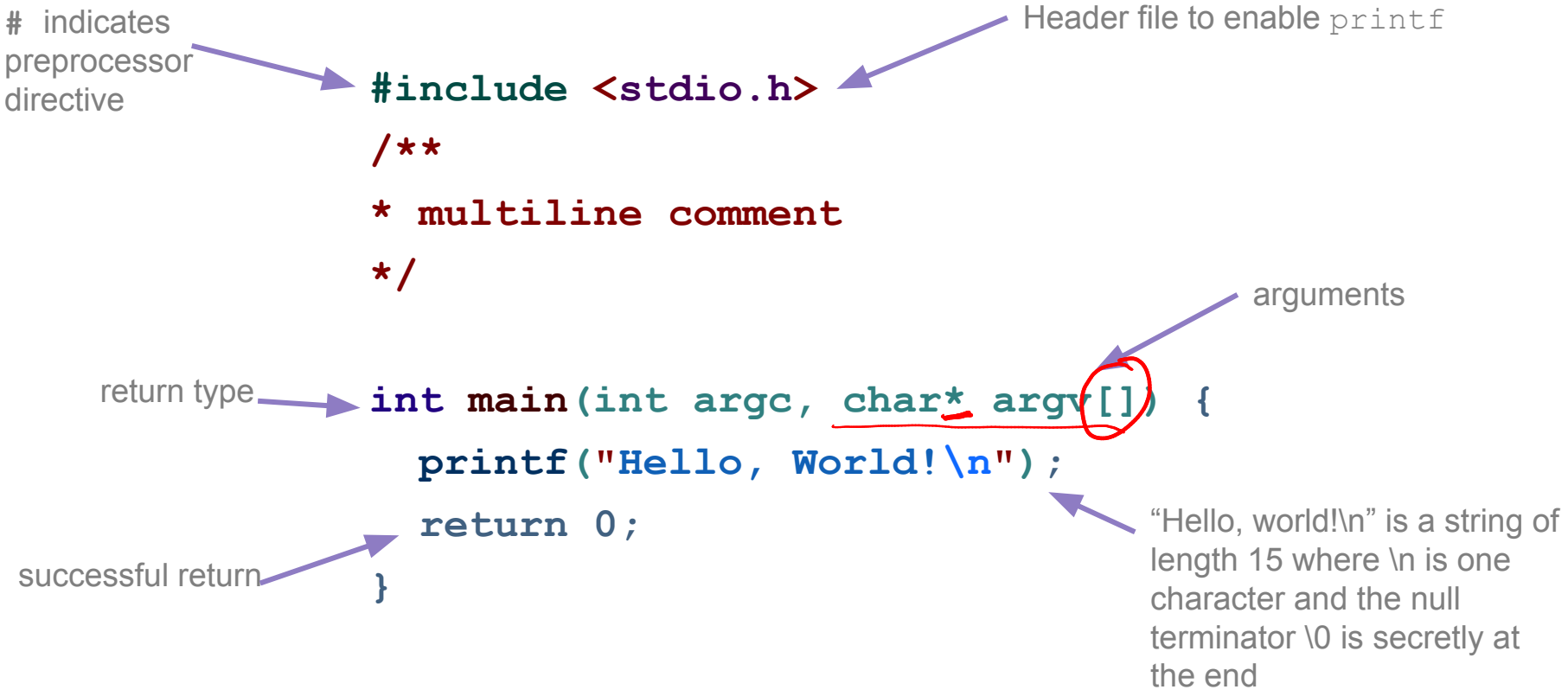
- Replaces `%s` with value given in `stringVar`, `%d` with `intVar`
- **`printf("hello, %s\n", myName);`**

Printf: print format function



This statement prints: “Twice 15 is 30”.

One more look…



Polling Time!

Please answer in the LP7 assignment on Gradescope!

What is the output printed from running this program?

```
#include <stdio.h>

int main() {
    int year = 2025;
    char* quarter = "Summer";
    char* course = "CSE 374";
    printf("I am taking %s during %s %d.\n",
           course, quarter, year);
    return 0;
}
```

How many characters are in the `course` variable?

```
char* course = "CSE 374";
```

- 6
- 7
- 8
- 9

What will `strlen(course)` return?

- 6
- 7
- 8
- 9

Variable Scope

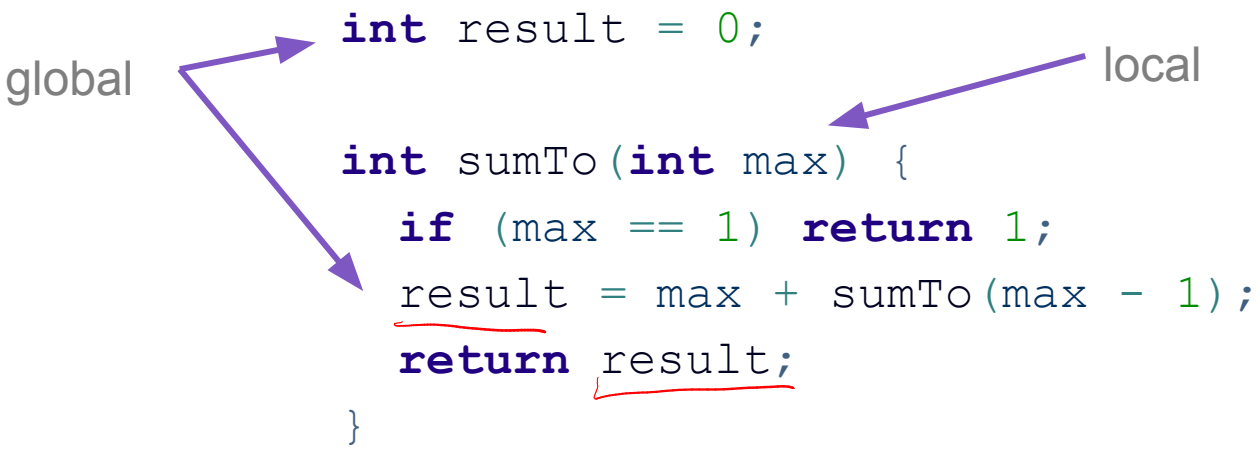
Variables defined **inside of blocks { … }** are **local** variables

- Local variables can only be accessed from inside their defining block
- Blocks include:
  - Functions, if statement branches, while loop bodies, for loop bodies, etc.
  - Best practice: inside of functions, declare all the local variables you'll use at the top
- May have multiple instances (due to recursion)

Variables defined **outside of functions** are **global** variables

- Global variables can be accessed at any time from any function
- Will only ever be a single instance of a global variable
- Should be avoided if possible for encapsulation

sumTo


global
local
int result = 0;
int sumTo(int max) {
if (max == 1) return 1;
result = max + sumTo(max - 1);
return result;
}

Polling Time!

Please answer in the LP7 assignment on Gradescope!

How many **instances** of each variable are created when we call **`sumTo(3)`**?

```
int result = 0;


int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

Number of instances of **`result`**:

- 1
- 2
- 3

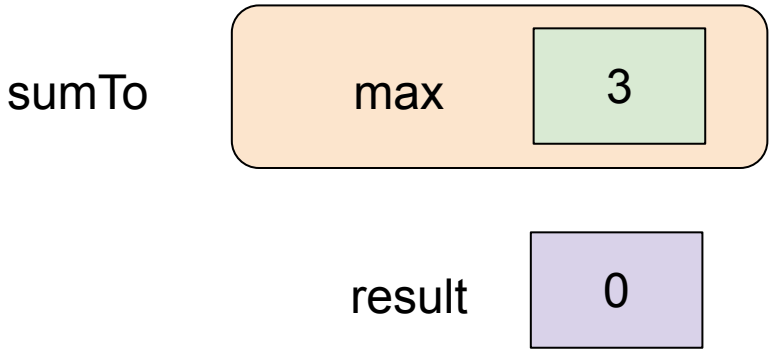
Number of instances of **`max`**:

- 1
- 2
- 3

Call Traces and the Stack

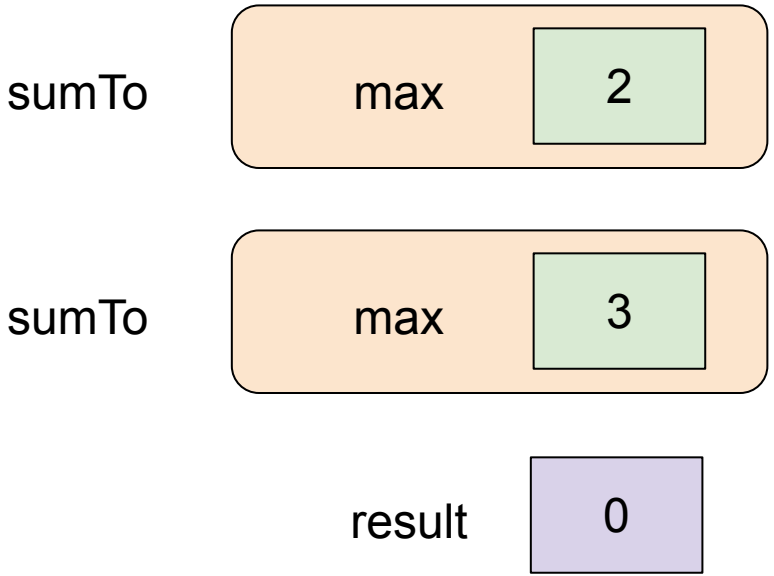
Call Trace: `sumTo(3)`

```
int result = 0;

                  =3
int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```



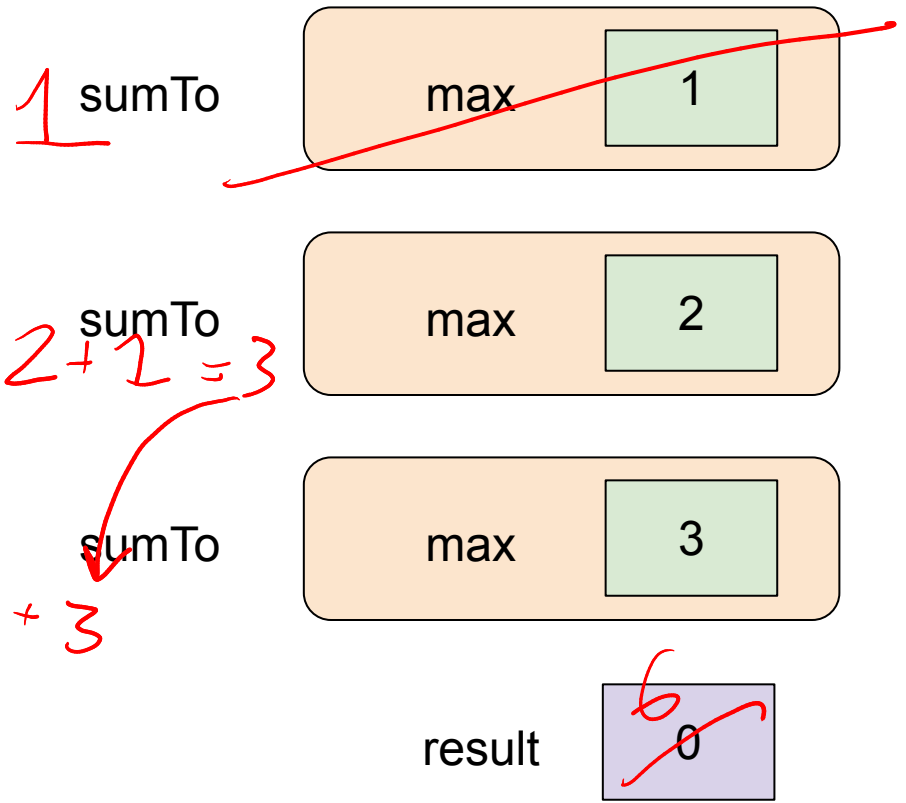
Call Trace: `sumTo(3)`

```
int result = 0;

                    =2
int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```



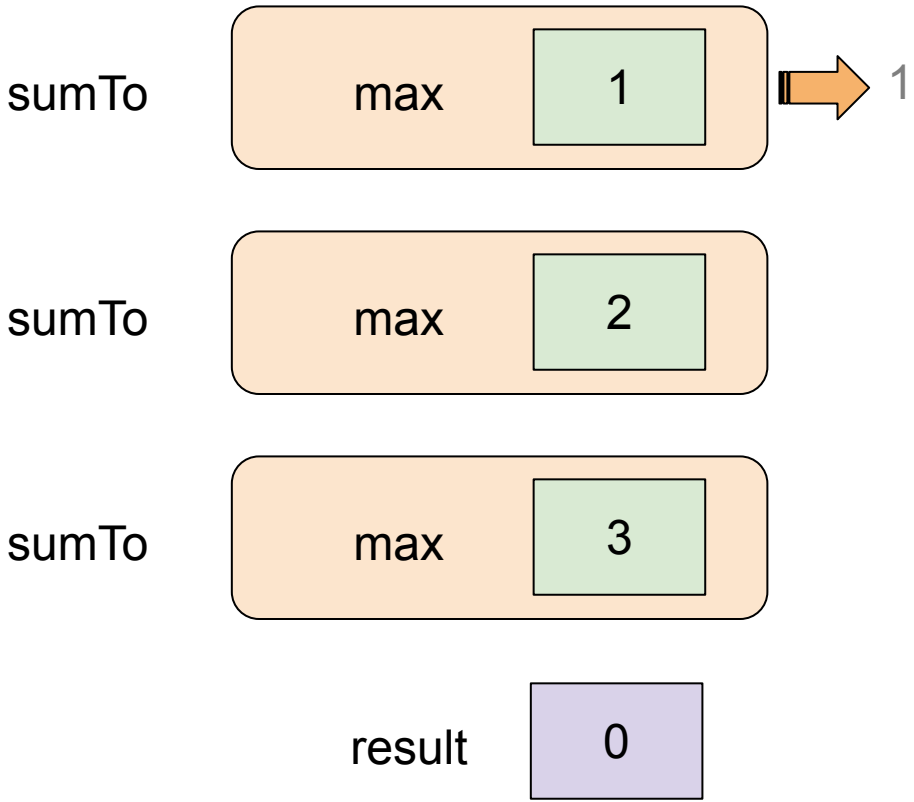
Call Trace: `sumTo(3)`

```
int result = 0;

int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```



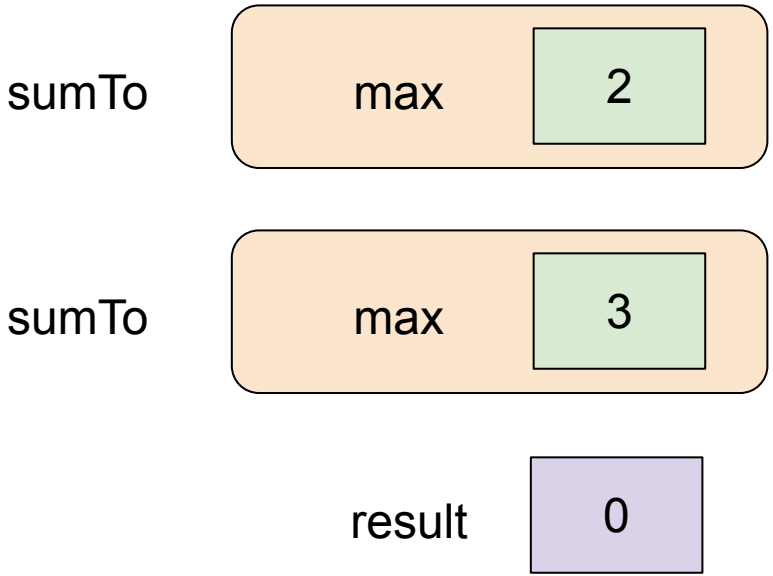
Call Trace: `sumTo(3)`

```
int result = 0;

                  =1
int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```



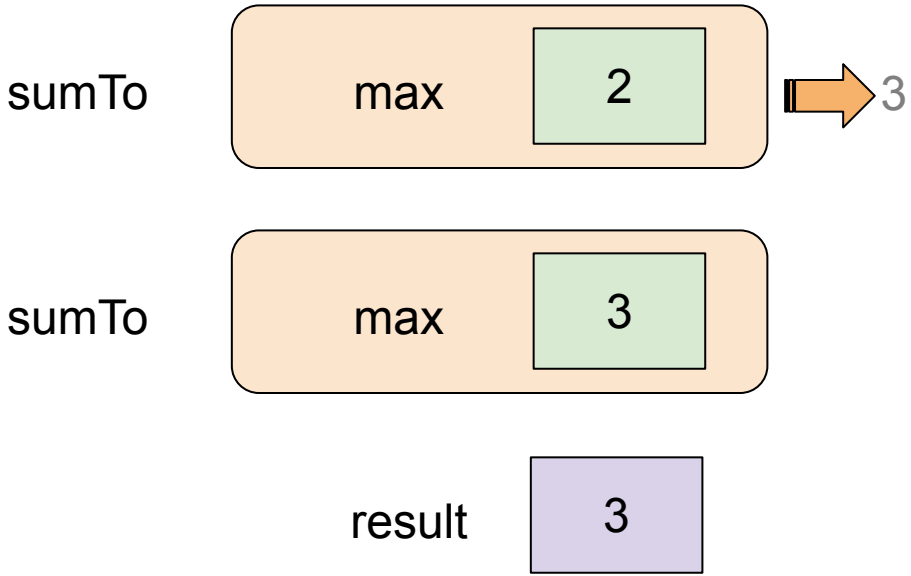
Call Trace: `sumTo(3)`

```
int result = 0;

                  =2
int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;          1
}
```



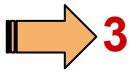
Call Trace: `sumTo(3)`

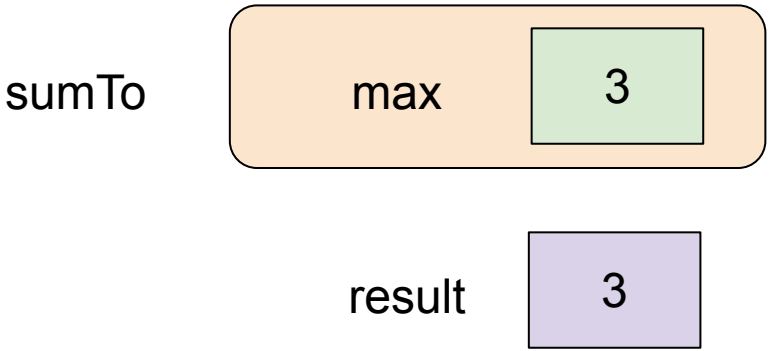
```
int result = 0;

                  =2
int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;          1
}
```



Call Trace: `sumTo(3)`

```
int result = 0;

                  =3
int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

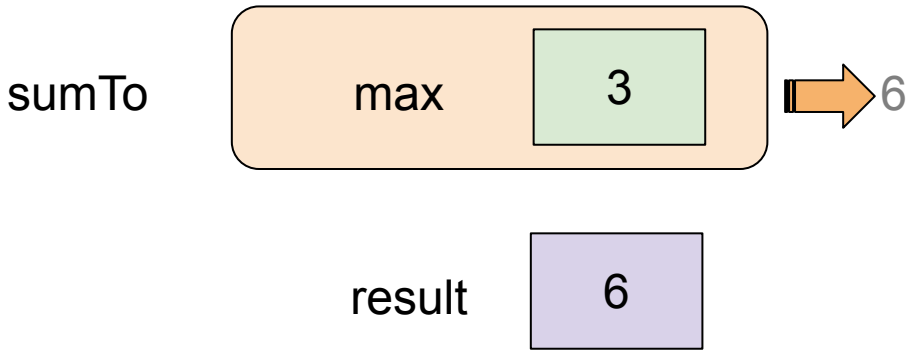




Call Trace: `sumTo(3)`

```
int result = 0;

                  =3
int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

3



Call Trace: `sumTo(3)`

```
int result = 0;

int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

result **6**

Instances of Global vs. Local Variables

There will only ever be one instance of a global variable

- Stored in the global data section (not the stack nor the heap)

There may be multiple instances of a local variable

- This mostly happens during recursion

How can we create multiple, separate instances of the same variable?

- Answer: We need a **space in memory** that can expand

The Stack

The "stack" is an area of memory that holds the local variables

Similar idea to the stack data structure (LIFO), but for local variables

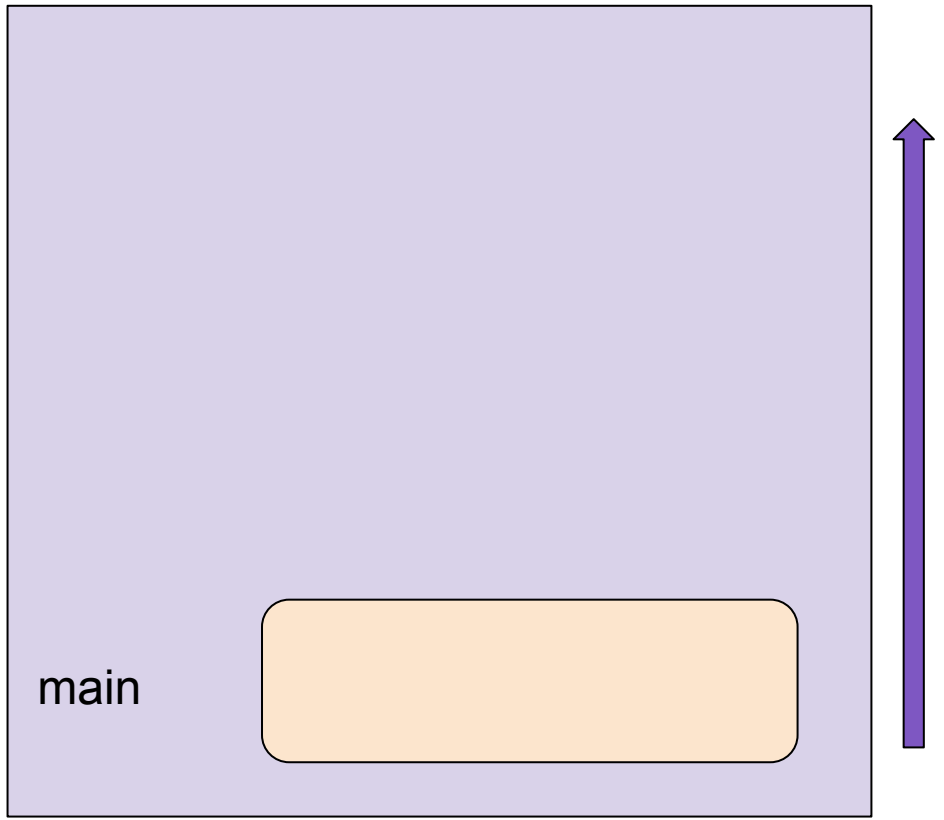
When we call a function, it **allocates** memory on the stack for those local variables

- Size of memory depends on the data type (e.g. int, char).
- If a recursion goes wrong… **Stack overflow**!

When that function returns, it **deallocates** its space on the stack

Stack: `sumTo(3)`

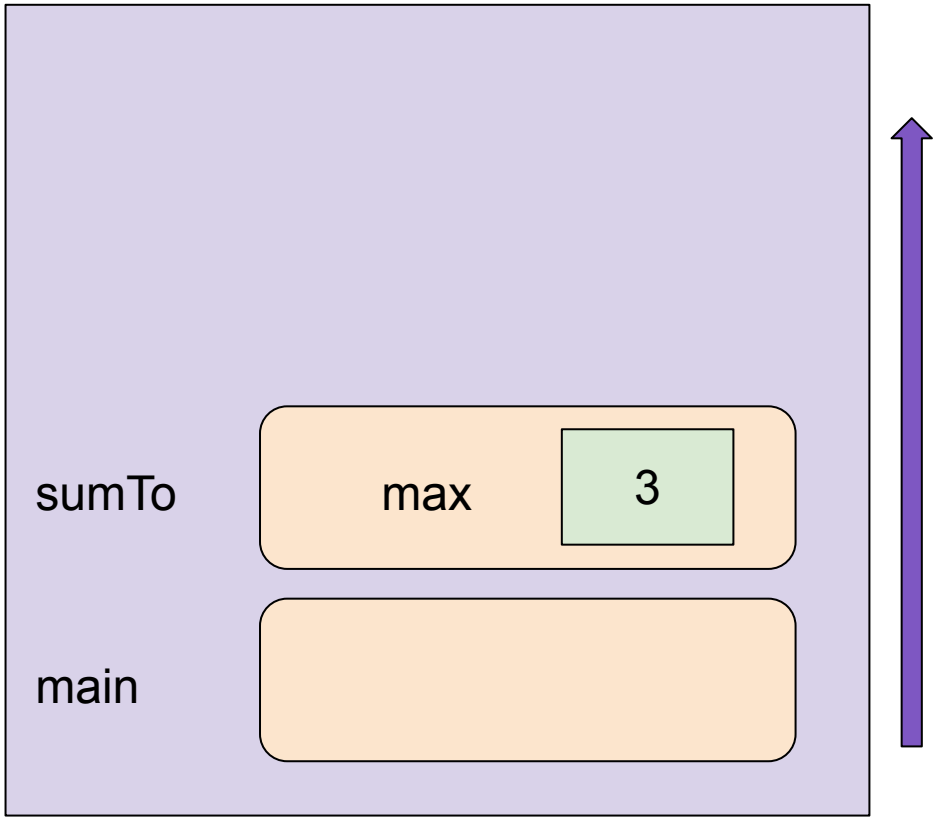
Stack



Stack: `sumTo(3)`

```
int result = 0;

int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

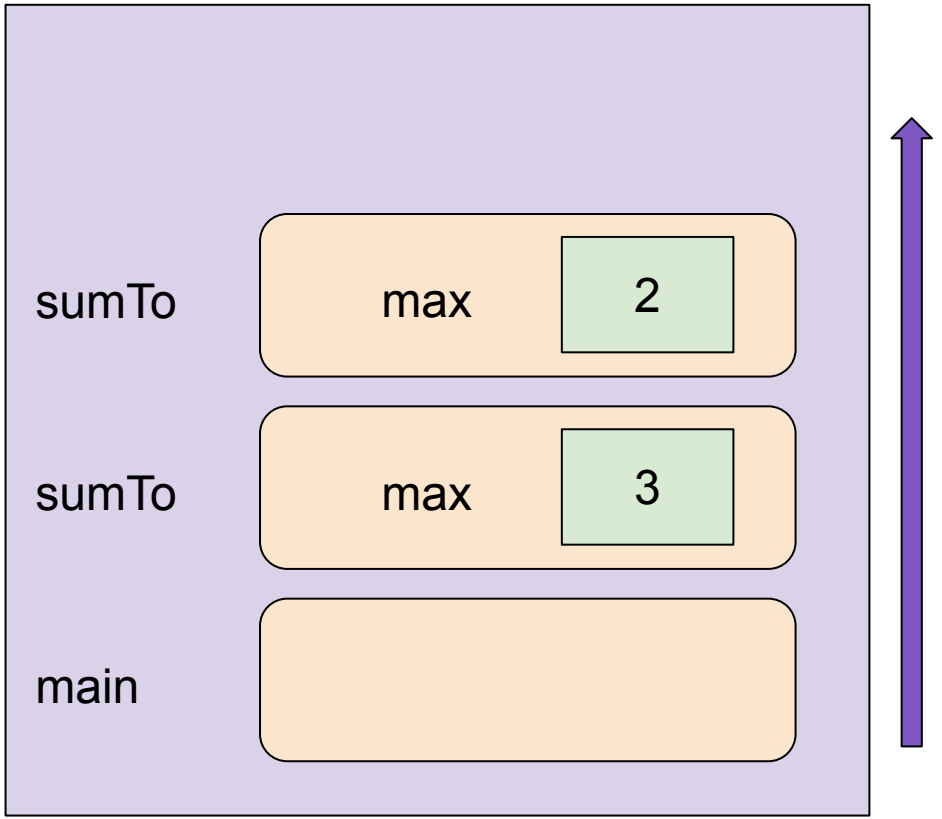
Stack



Stack: `sumTo(3)`

```
int result = 0;

int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

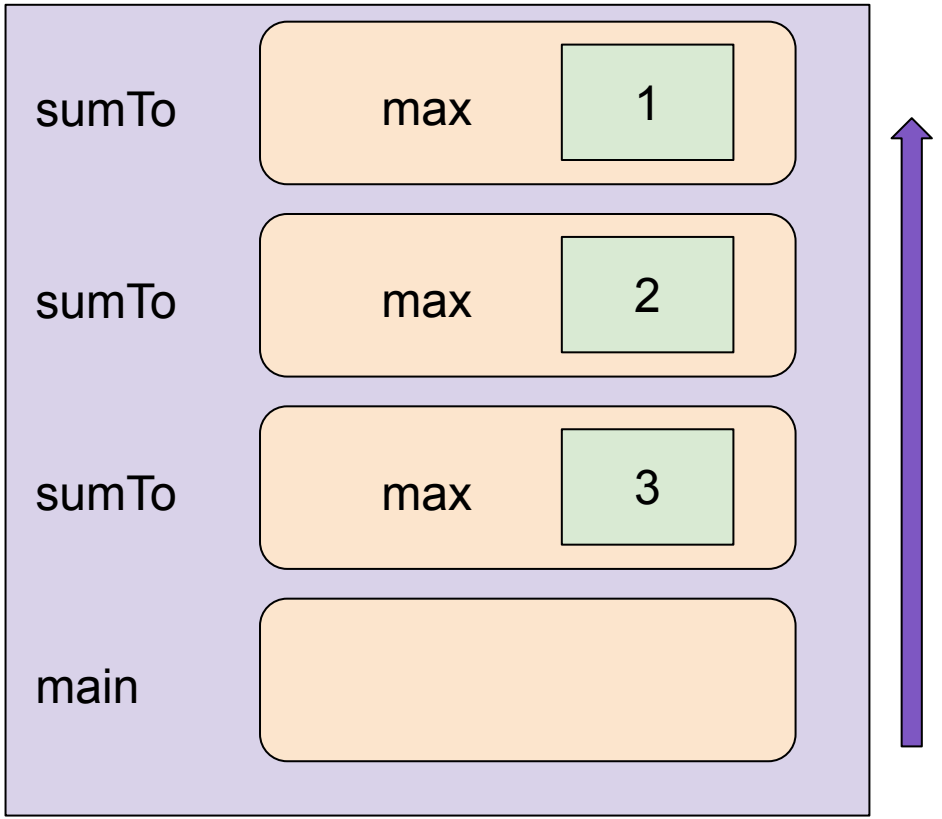
Stack



Stack: `sumTo(3)`

```
int result = 0;

int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

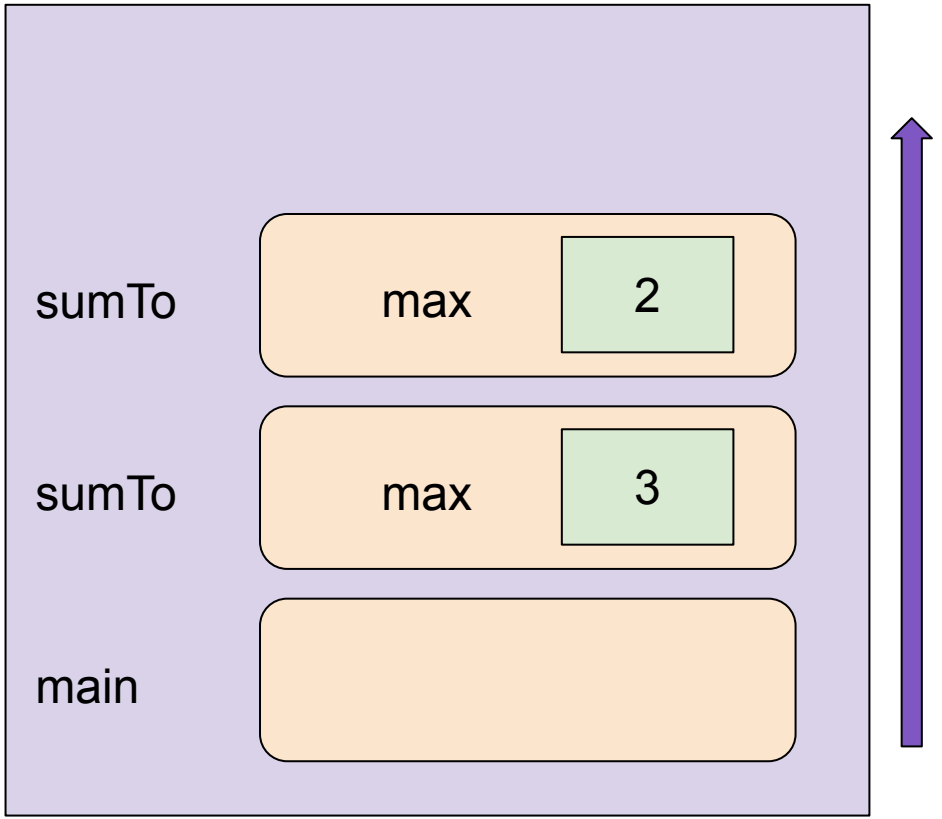
Stack



Stack: `sumTo(3)`

```
int result = 0;

int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

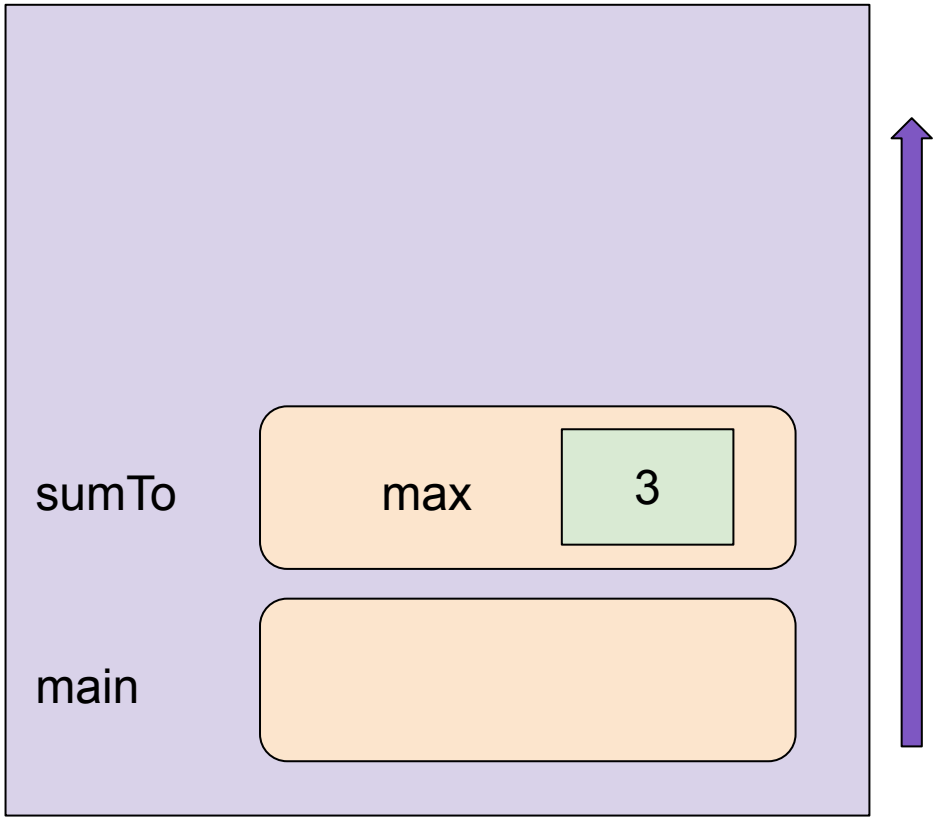
Stack



Stack: `sumTo(3)`

```
int result = 0;

int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

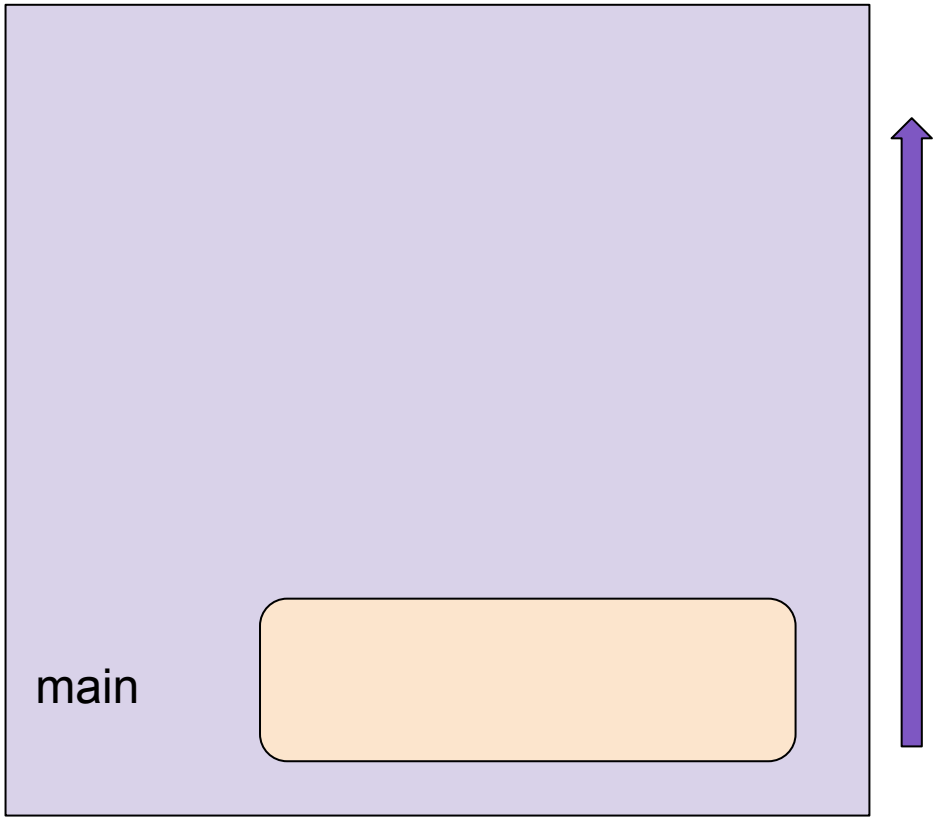
Stack



Stack: `sumTo(3)`

```
int result = 0;

int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

Stack



Lifetimes

When a function returns, it deallocates its stack frame, destroying the local variables

Local variables only "live" until the end of the function

- When the function ends, the variables “die”

Global variables live until the end of the program

Things to keep in mind…

Because Java is a descendent of C, much syntax is shared

- `if`, `for`, and `while` all look like they do in Java
- This does not mean however that C always acts the same as Java

C does not really have booleans, instead it uses `int`

- **0** means **false**, **nonzero** means **true** (**opposite of bash!**)

Getting comfortable with reading documentation

- Reading documentation is a part of becoming an independent programmer – I do it all the time :)
- Lots of ways to view documentation – man pages, Google, etc.
- Can be jargon-y – if the docs are confusing, ask in OH or on Ed!

Polling Time!

Please answer in the LP7 assignment on Gradescope!

Which of the string comparisons results in "**true**" (nonzero) when **`string1`** *equals* **`string2`**?

- `if (strncmp(string1, string2, n))`
- `if (strncmp(string1, string2, n) == 0)`

Tip: Check out the reference page for **`strncmp`**:
https://cplusplus.com/reference/cstring/strncmp/

Recall: What value is "true" in C? And what value is "false"?

In summary...

Anatomy of a C Program

```
// includes for functions & types defined elsewhere
#include <stdio.h>
#include “localstuff.h”
// symbolic constants
#define MAGIC 42
// global variables (if any, please avoid using)
int x = 1;
// function  declarations (aka prototypes)
void do_this(char s, int m);
// function definitions
int main(int argc, char* argv[]) {...}
void do_this(char s, int m) {...}
```

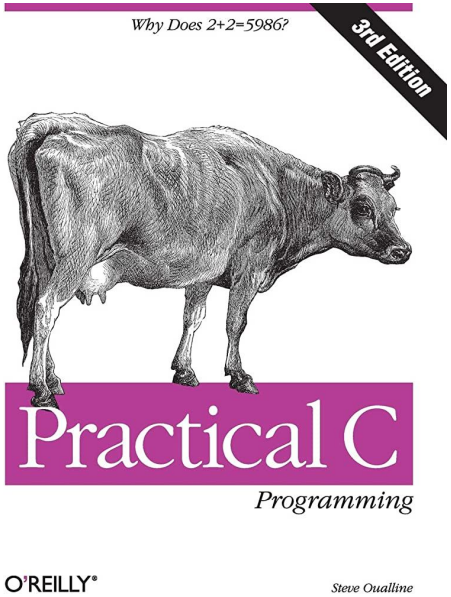
Useful C Libraries

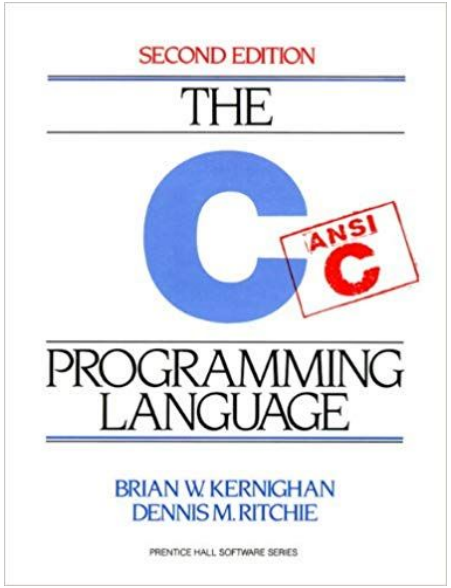
We won't be going over these in detail in class – you will be exploring them in HW3

- I/O: `stdio.h`
- String manipulation: `string.h`
- Various properties about characters (is a number, is a letter, etc.): `ctype.h`
- General utility functions (memory allocation, string conversion, etc.): `stdlib.h`

C Resources

- http://www.cplusplus.com/
  - C++ and C reference
- Practical C Programming
  - Available online using UW library
- The C Programming Language
  - Available on Kindle and in the UW library
- Essential C - Stanford pdf
- Google :)





Reminders

EX5 and EX6 due Friday 7/10 **@10:49am**

HW2: Bash Scripts is **due Friday @11:59pm**

- Come to any of our office hours for help!

Exam 1 will be on Monday, starting an hour earlier than normal (9:50)

PCAR 291, not PCAR 395

Can bring one double-sided handwritten cheat sheet, in addition to the reference sheet

Review session on Friday from 1-3 PM in CSE2 G04

Bonus Slides 

Implementing `echo`

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  for (int i = 1; i < argc; i++) {
    printf("%s ", argv[i]);
  }
  printf("\n");
  return 0;
}
```