
This is CS50.

quiz 0 details

wed oct 13, 1pm

see handout for locations

covers weeks 0 through 5

closed book

bring a 8.5” × 11”, 2-sided cheat sheet

75 minutes

15% of final grade

resources

old quizzes + solutions

lecture slides

lecture videos + transcripts

source code

scribe notes

section videos

pset specs

office hours

topics

review

Part 0

Scott Crouch

Binary Numbers

- Base-2 Representation
- The memory of your computer is contained in bits that are either

1 or 0

Binary Numbers

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Maximum 8-digit binary value?

11111111 = 255 or $2^8 - 1$

Some Practice

What is 122 in Binary?

01111010

What is 00011001 in Decimal?

25

Binary Addition

0 + 1 = 1, 0 + 0 = 0, 1 + 0 = 1

1 + 1 = 10, but carry the 1

Example:

```
  1 1 1   1 1 1
    00110110
+   01010111
-----------
   10001101
```

Hexadecimal

Base 16 with 16 distinct symbols

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Each digit is a nibble or 4 bits

0001 = 0x1

1111 = 0xF

00011111 = 0x1F

10101111 = 0xAF

ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

ASCII Again

- Encoding scheme for characters
- For arithmetical operations, you can use the ASCII char.

```
//sets b to ‘B’
char b = ‘A’ + 1;

//sets e to 101
int e = ‘d’ + 1;
```

GCC and Compilers

GNU C Compiler (aka GNU Compiler Collection)

Compiling Commands:

```
gcc <program_name>.c
      produces a.out executable file

gcc –o <program_name>
<program_name>.c
      produces an executable file
 with the name of your file
```

Common Compiling Errors and Warnings

undefined reference to function “GetString”

forgot to link in cs50 library (-lcs50)

implicit declaration of built in function ‘sqrt’

forgot to #include <math.h>

control reaches end of non-void function

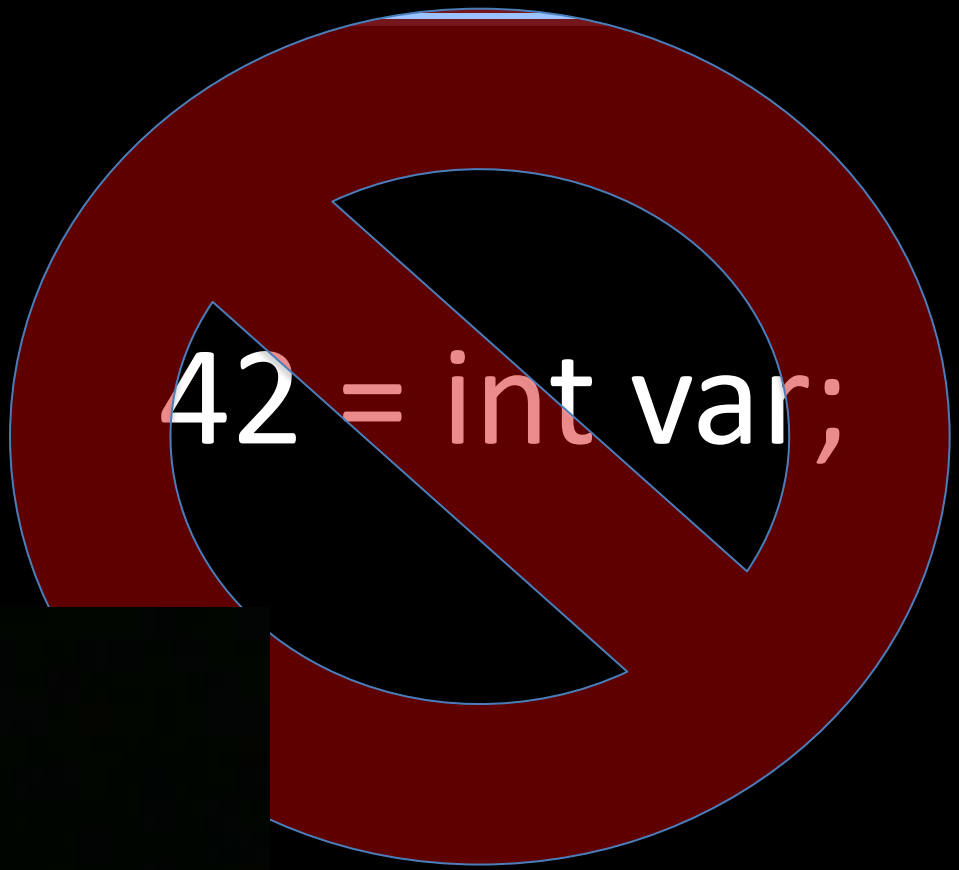
one of your non-void functions did not return a value.

Variables

Allow us to store information about the *state* of a program so we can access/change this information at a later time.

```
int var1 = 5;              //declares an integer with
                           value 5
var1++;                    //increments var1

printf("%d", var1);        //prints out 6
```

Be Careful!!



Types

Some types in C:

int: 4 bytes goes from -2^{31} -> $2^{31} - 1$

float: 4 bytes (7-digit precision)

double: 8 bytes (15-digit precision)

char: 1 byte goes from -2^{7} -> $2^{7} – 1$

long: 4 bytes goes from -2^{31} -> $2^{31} – 1$

long long: 8 bytes goes from -2^{63} -> $2^{63} – 1$

Signed vs. Unsigned?

Note: The sizes above are machine dependent, not C-Dependent, however these sizes are the most common.

Type Casting

Useful if you want to go between types:

Syntax:

```
int oldNum = 42;
float newNum = (float) oldNum;

char c = (char) oldNum;
```

Conditionals

Based off Booleans or Predicates: A statement that returns true or false which must first be fulfilled for the code to executed.

Represented with if, else if and else statements.

if, else if, else

```
int num = GetInt();

if (num > 0)
  printf(“You entered a positive number!”);
else if (num < 0)
  printf(“You entered a negative number!”);
else
  printf(“You entered zero”);
```

The Ternary Operator (aka The Sexy Statement)

Condense if and else into a 1 line statement!

Example:

```
int num = GetInt();
string statement = (num < 0) ? “Error” : “Valid”;
```

Syntax:

<variable_name> = (<condition>) ? <if true then> : <else then>;

For loops

```
for (<counter(s) initialization>; <condition(s)>;
   <change counter(s)>
{
    //your code here
}

Example:

int i, j;

for (i = 0, j = 0; i < 3 && j < 10; i++, j+= 2)
{
      printf("\ni:%d, j: %d", i, j);
}
```

While and Do-While Loops

```
while (<condition>)
{

   //do this

}
```

This loop checks then evaluates.

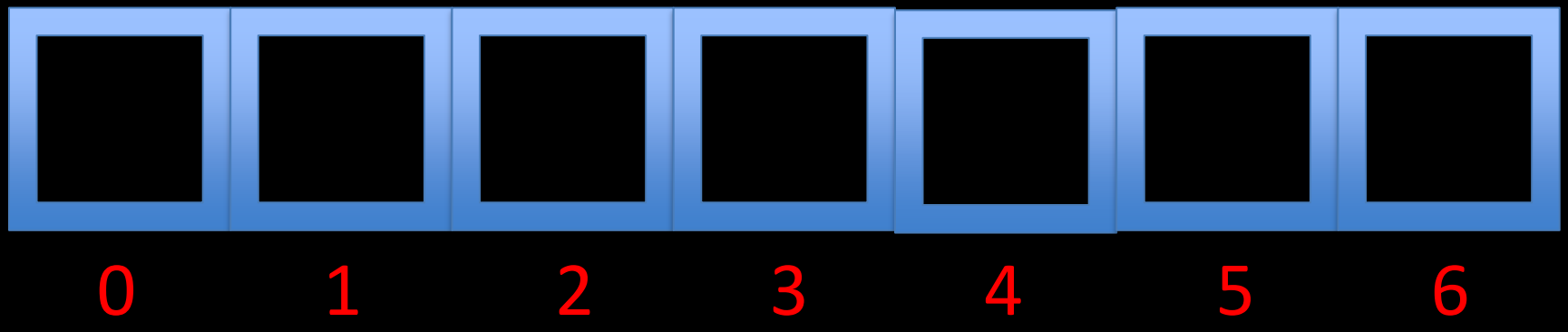
```
do
{

   //do this

} while (<condition>)
```

This loop evaluates then checks.

Arrays

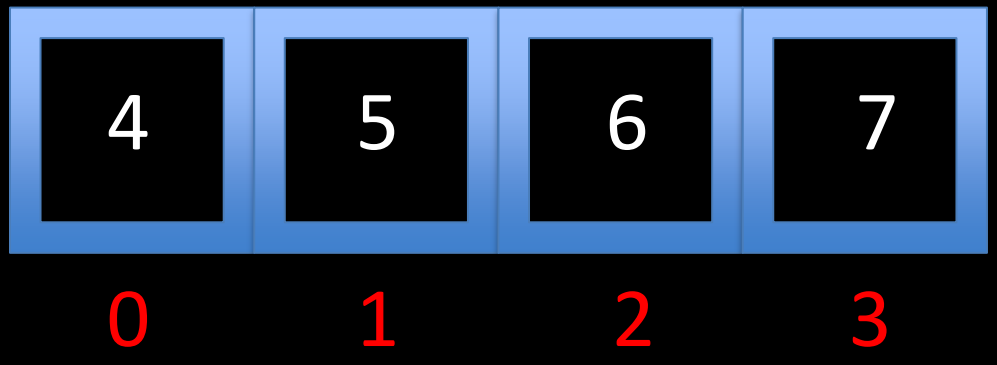
Simple data structure for storing objects of the same type.

Imagine them as lined up mailboxes, each with its own distinct number or shall we say index!



Declaring and Initializing Arrays

```
//declare an array of integers of length and fill it
int myArray[] = {4, 5, 6, 7};

//change the value in the 2nd slot to be 3
myArray[2] = 3;
```



Using Loops with Arrays

Loops can be used to iterate through arrays!

```
int myArray[4];

for (int i = 0; i < 4; i++)
  myArray[i] = i;
```

Part 1

Josh Bolduc

libraries

Standard Library

- `printf`
- `...`

Math Library

- `round`
- `...`

CS50 Library

- `GetChar`
- `GetDouble`
- `GetFloat`
- `GetInt`
- `GetLongLong`
- `GetString`

```
#include <cs50.h>
```

```
gcc foo.c -lcs50
```

functions

```
int
main(void)
{

    <do stuff>
    return 0;
}
```

```
return-type
name([arg-type arg-name, ...])
{

    <do stuff>
    return value;
}
```

$$f(x) = x^2 + 4x$$

$$f(x) = x^2 + 4x$$

$$f(2) = (2)^2 + 4(2)$$

$$f(2) = 4 + 8$$

$$f(2) = 12$$

```
int
foo(int x)
{
    return x*x + 4*x;
}
```

command-line args

```
int
main(int argc, char *argv[])
{

    <do stuff>
    return 0;
}
```

```
./programname cmd line args
```

```
./programname cmd line args

argc =
```

```
./programname cmd line args

argc = 4
```

```
./programname cmd line args

argc = 4

argv[0] =
argv[1] =
argv[2] =
argv[3] =
```

```
./programname cmd line args

argc = 4

argv[0] = "./programname"
argv[1] = "cmd"
argv[2] = "line"
argv[3] = "args"
```

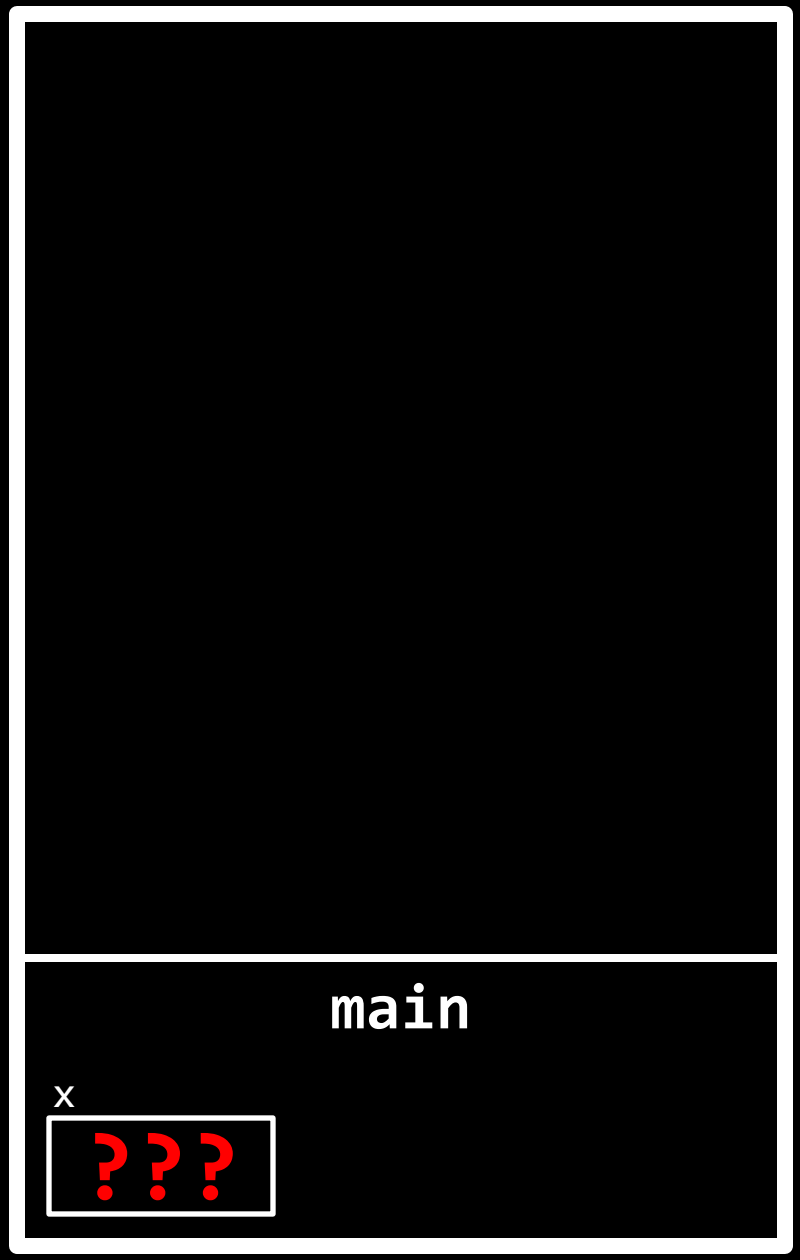
scope

```
// Swaps a and b. (lol jk)
void
swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

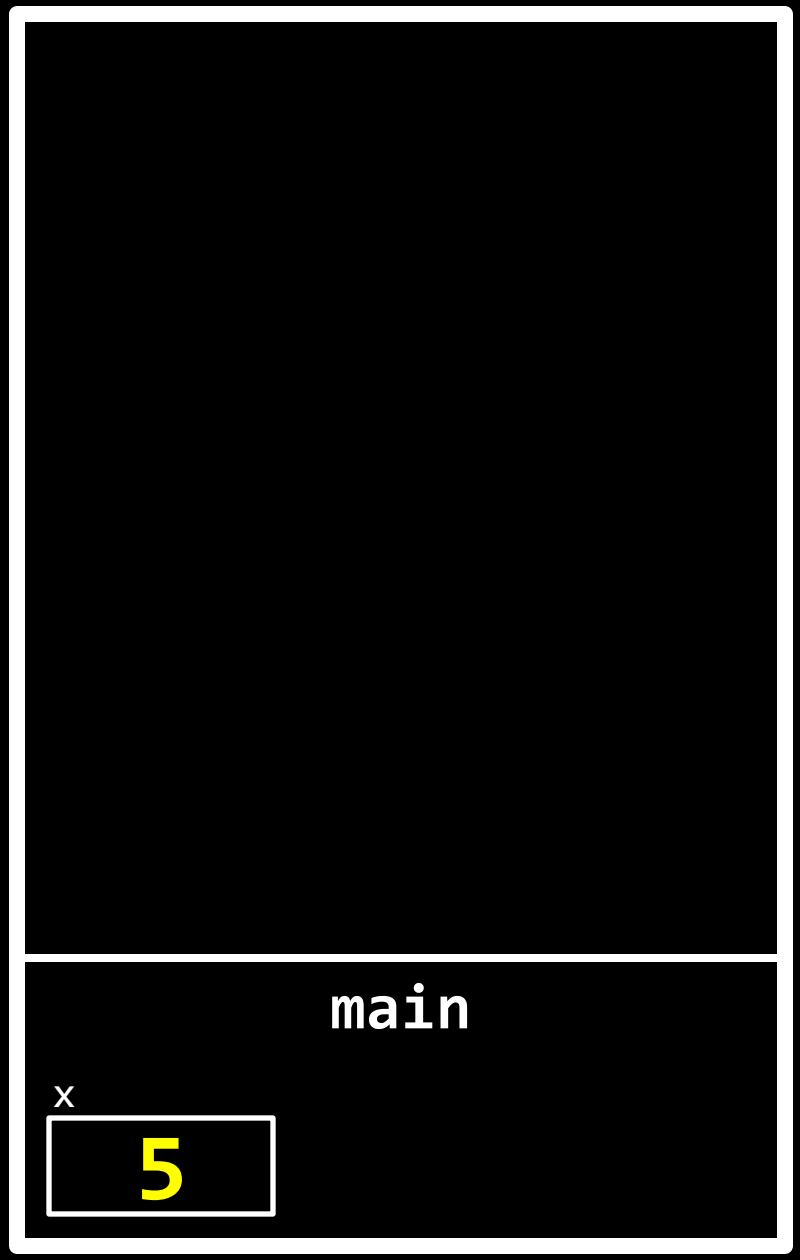
```
// Swaps a and b. (srsly)
void
swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

frames

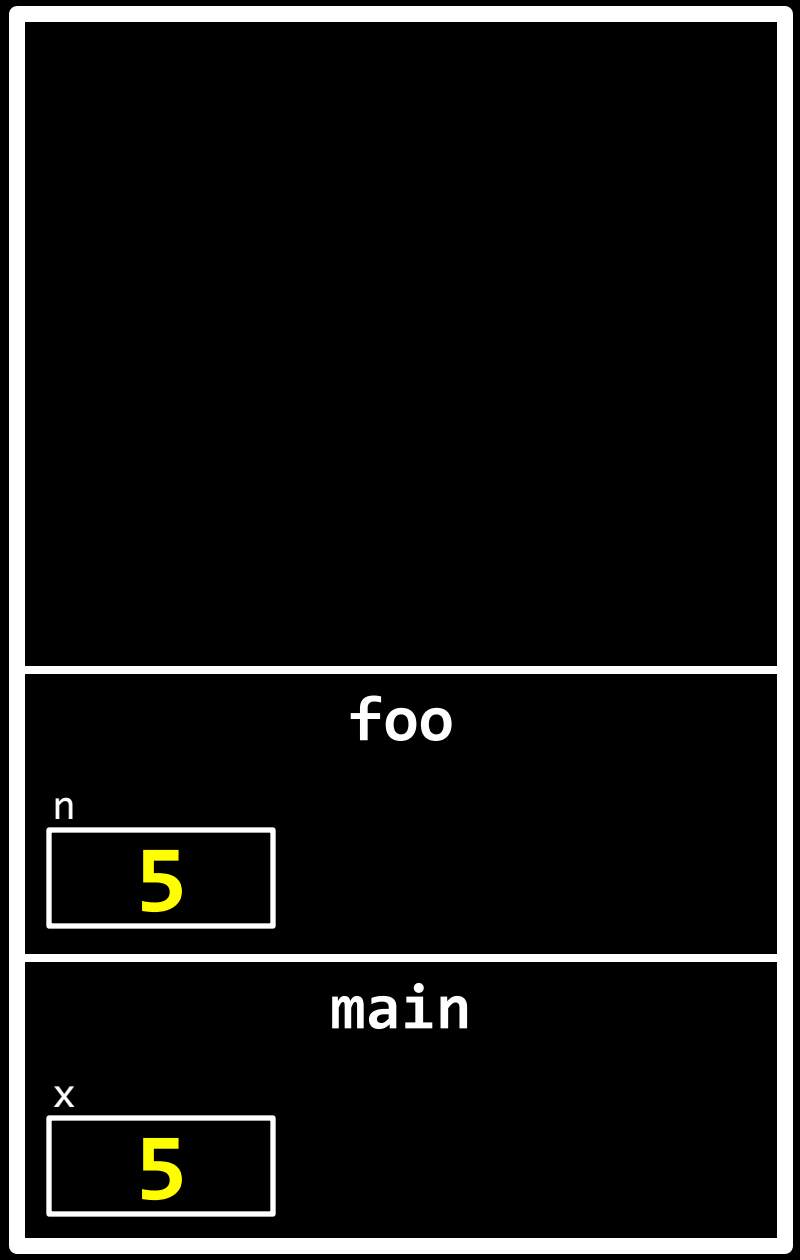
```
int
bar(int i)
{
    return i + 1;
}

int
foo(int n)
{
    return bar(n) * 2;
}

int
main(void)
{
    int x = 5;
    x = foo(x);
    return 0;
}
```



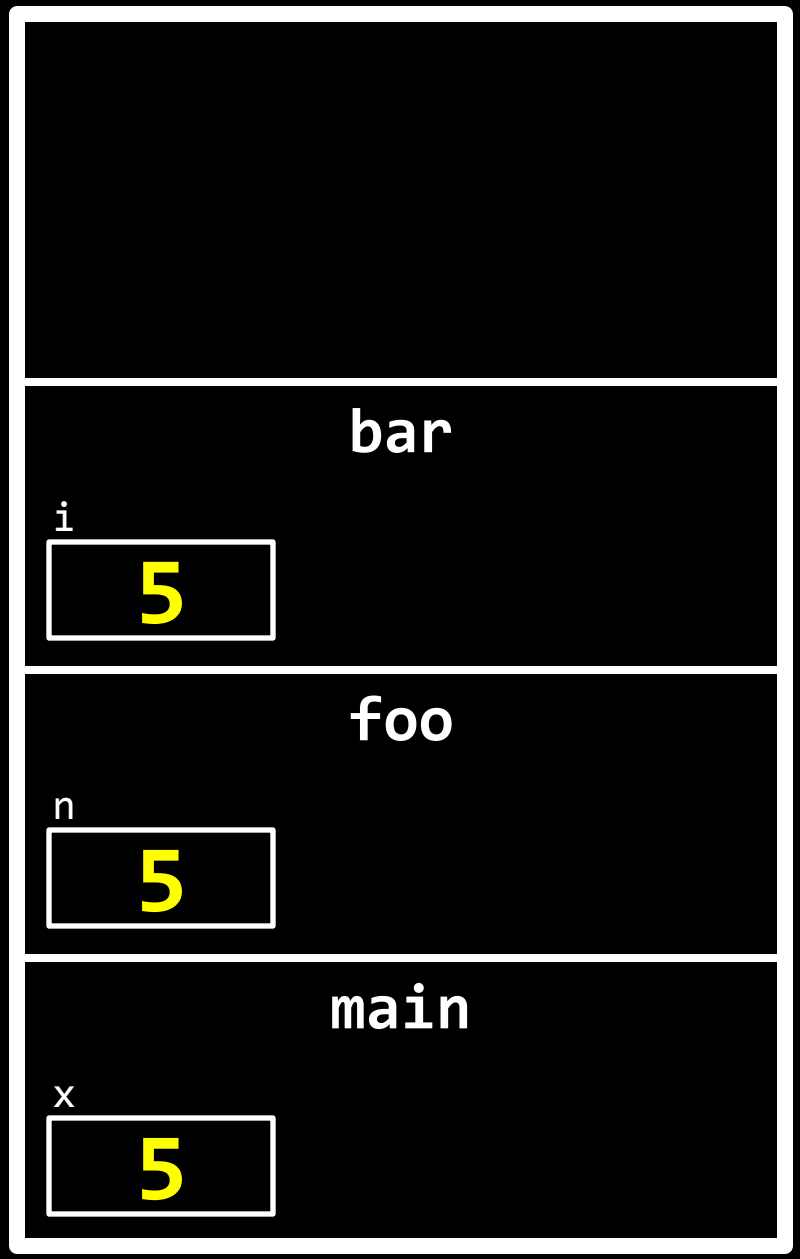
```
int
bar(int i)
{
    return i + 1;
}

int
foo(int n)
{
    return bar(n) * 2;
}

int
main(void)
{
    int x = 5;
    x = foo(x);
    return 0;
}
```



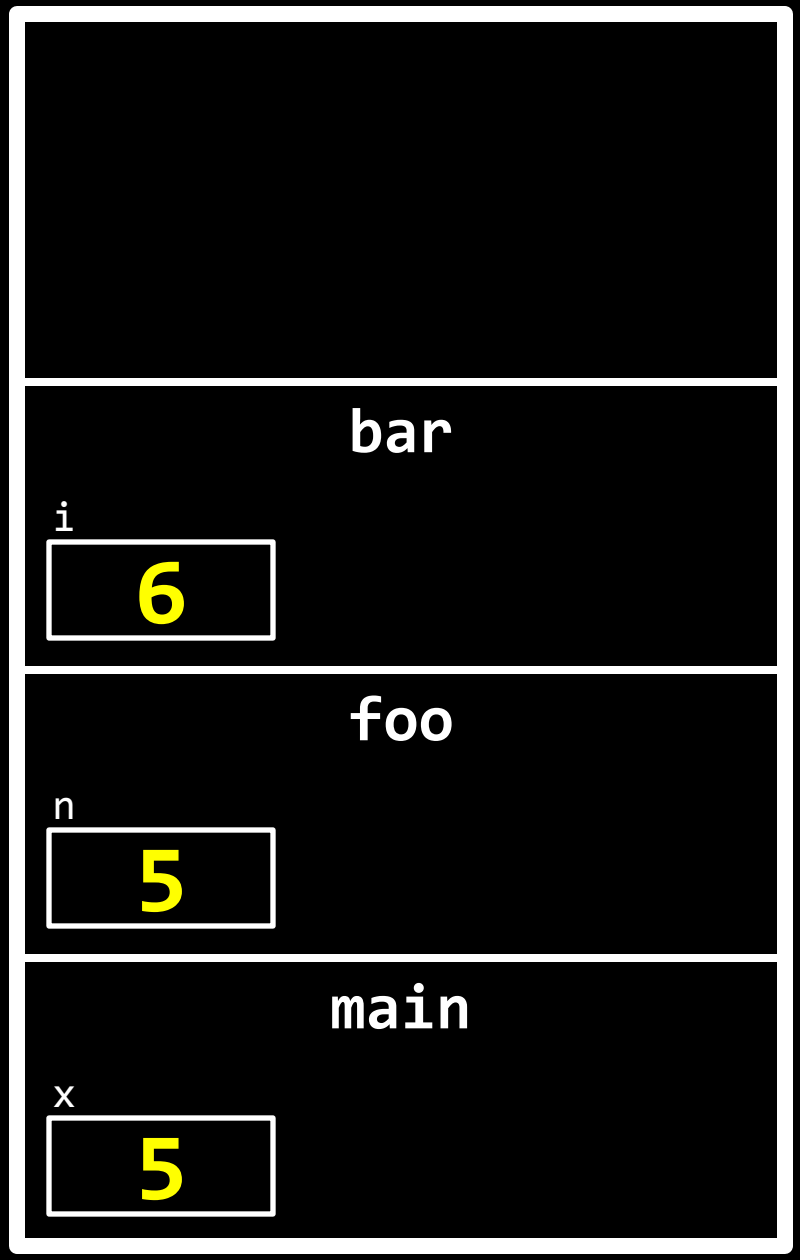
```
int
bar(int i)
{
    return i + 1;
}

int
foo(int n)
{
    return bar(n) * 2;
}

int
main(void)
{
    int x = 5;
    x = foo(x);
    return 0;
}
```



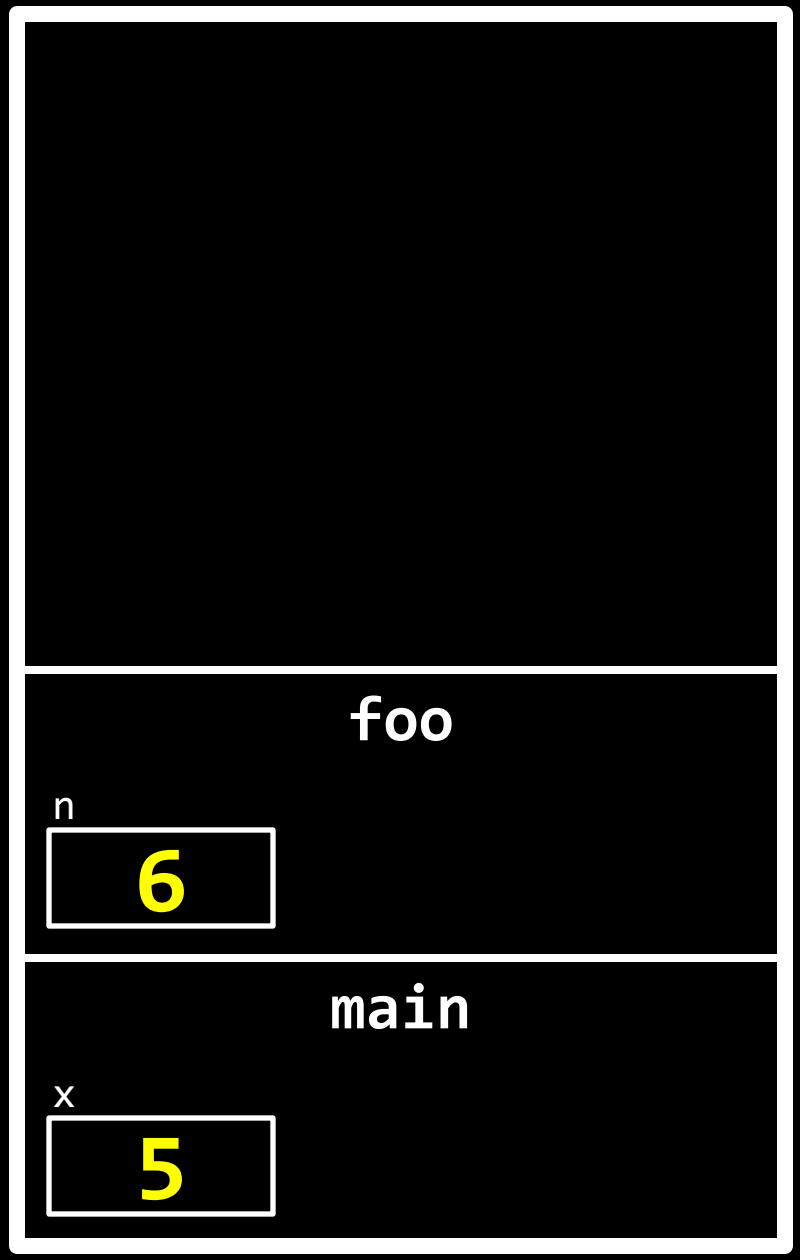
```
int
bar(int i)
{
    return i + 1;
}

int
foo(int n)
{
    return bar(n) * 2;
}

int
main(void)
{
    int x = 5;
    x = foo(x);
    return 0;
}
```



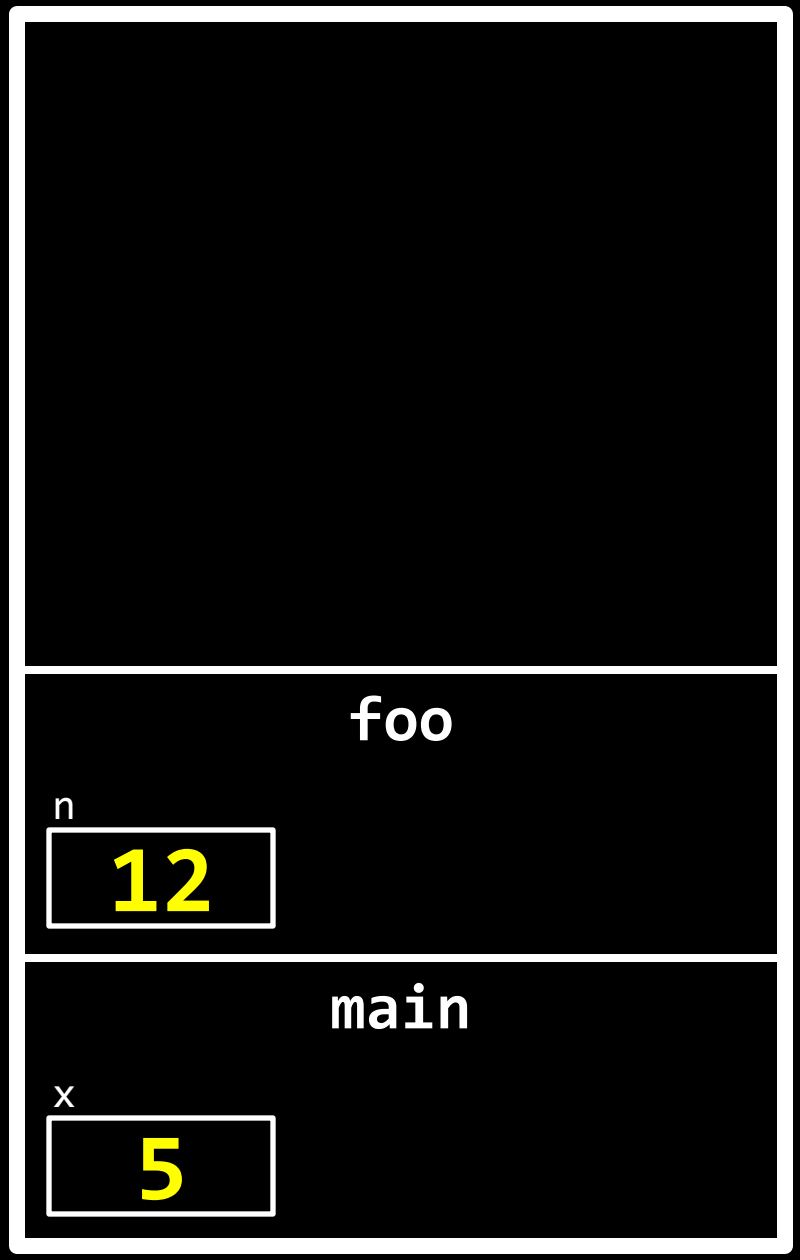
```
int
bar(int i)
{
    return i + 1;
}

int
foo(int n)
{
    return bar(n) * 2;
}

int
main(void)
{
    int x = 5;
    x = foo(x);
    return 0;
}
```



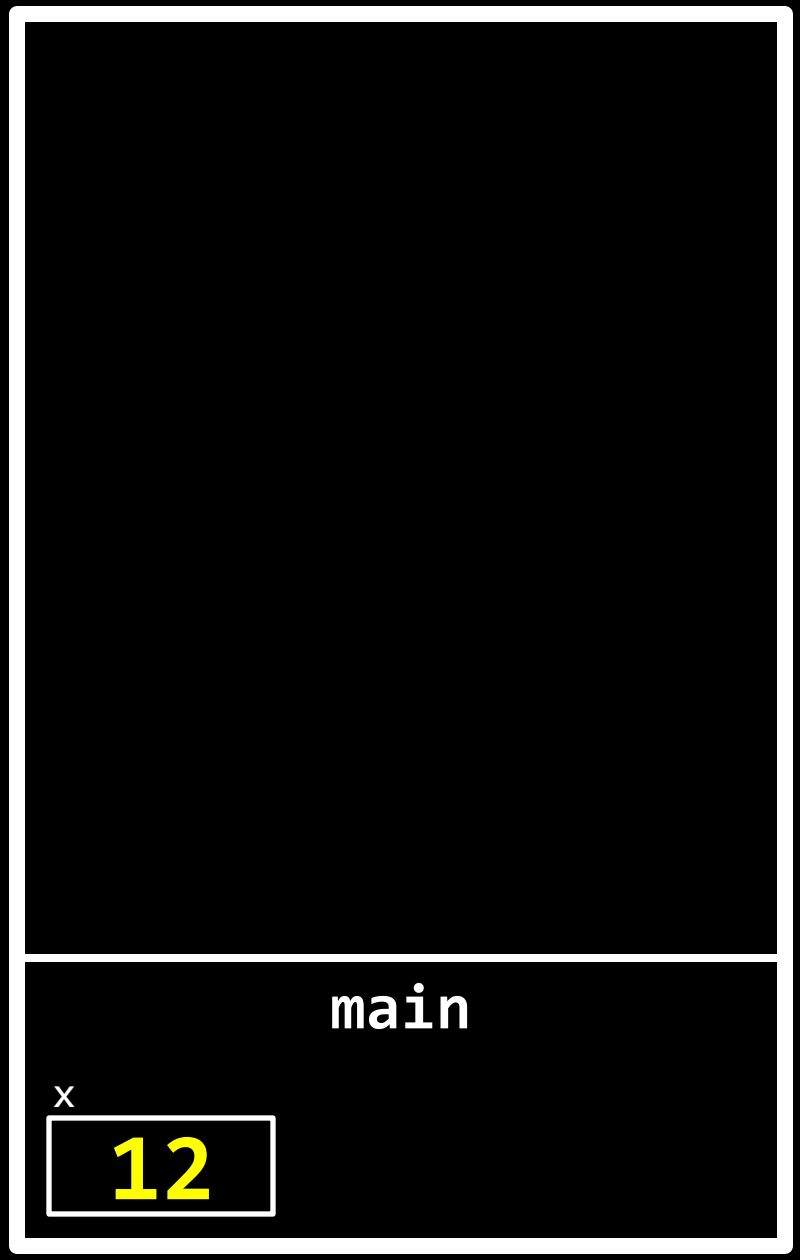
```
int
bar(int i)
{
    return i + 1;
}

int
foo(int n)
{
    return bar(n) * 2;
}

int
main(void)
{
    int x = 5;
    x = foo(x);
    return 0;
}
```

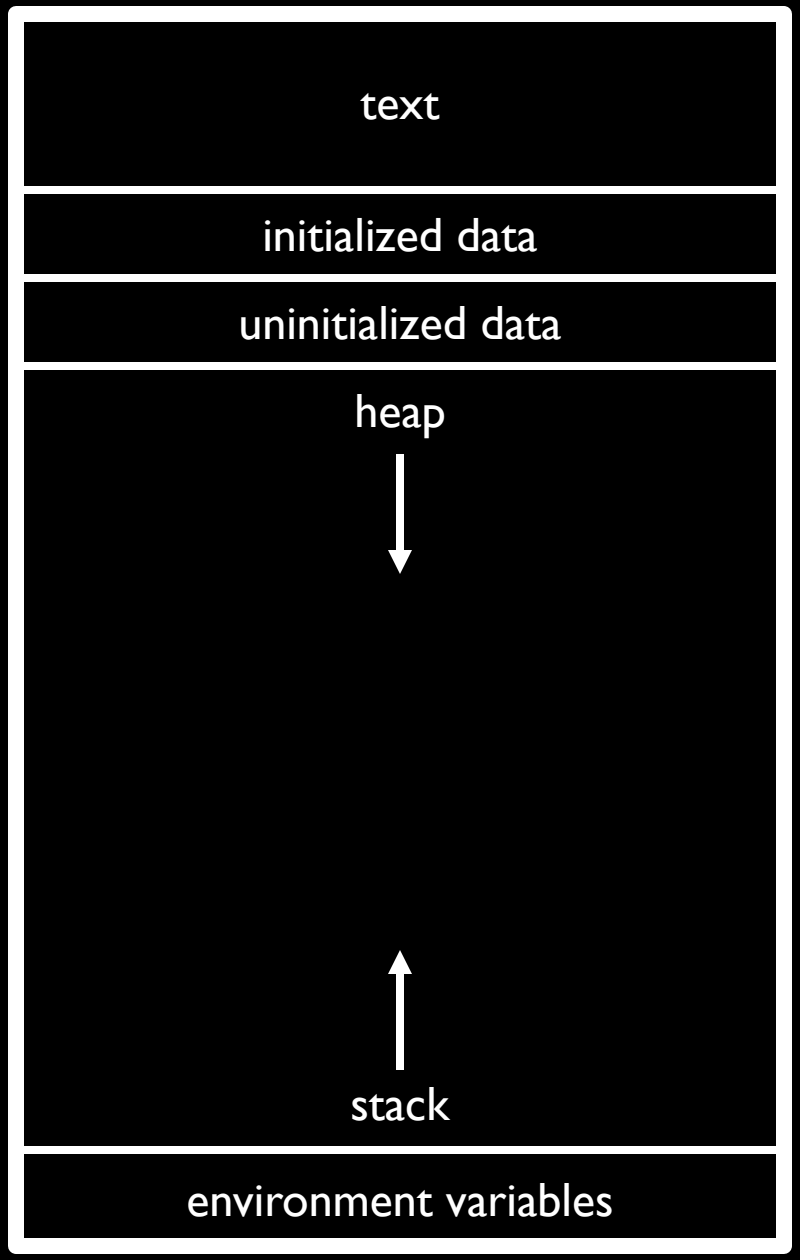


```
int
bar(int i)
{
    return i + 1;
}

int
foo(int n)
{
    return bar(n) * 2;
}

int
main(void)
{
    int x = 5;
    x = foo(x);
    return 0;
}
```



```
int
bar(int i)
{
    return i + 1;
}

int
foo(int n)
{
    return bar(n) * 2;
}

int
main(void)
{
    int x = 5;
    x = foo(x);
    return 0;
}
```




text
initialized data
uninitialized data
heap
stack
environment variables

Part 2

Rose Cao

topics = topics -> next;

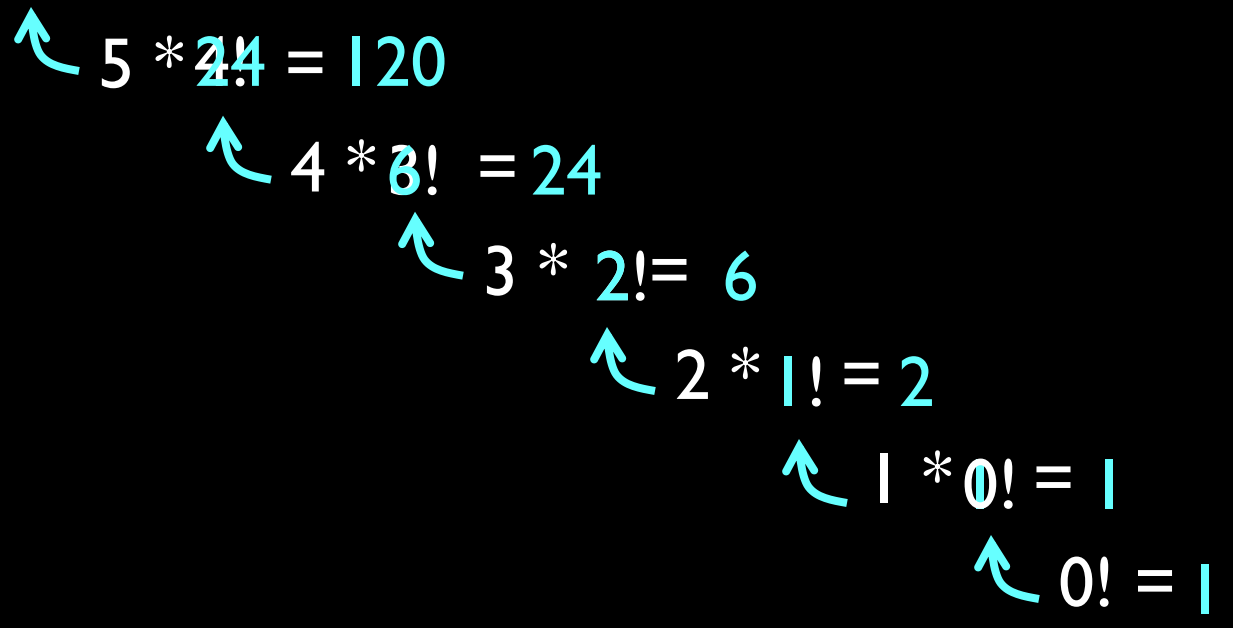
(Hi! I'm Rose,
for part 2. =))

- Recursion
- Search
- Sort
- Asymptotic Notation

Recursive Functions

(as opposed to iterative)

- When a program repetitively calls on itself
- Performs a task by repeating smaller, similar tasks
- Ex: 5! = 120



- Needs a base case to know when to stop!

A more interesting example:

Print the characters of a string.
(recursively, since you know the iterative version already)

```
void print_chars(char str[], int spot)
{
    // Base case: at end of string
    if (str[spot] == '\0')
        return;
    else
    {
        // Print the character
        printf("%c\n", str[spot]);

        // Call print_chars with next spot
        print_chars(str, spot + 1);
    }
}
```

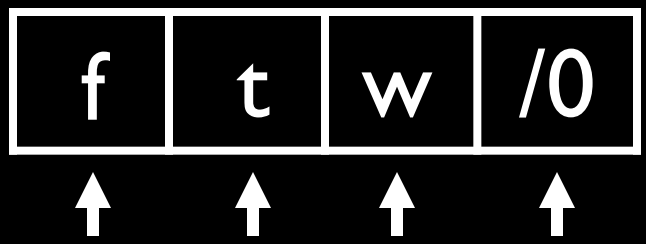
Recursive part!

How it happens:

Somewhere in main():

```
… print_chars(str, 0); …
```

with str:



Printed:

ftw

spot = 0

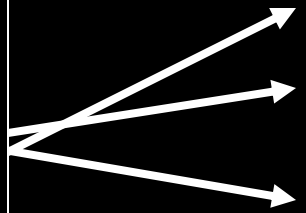
spot = 1

spot = 2

spot = 3

```
void
print_chars(char str[], int spot)
{
    // Base case: at end of string
    if (str[spot] == '\0')
        return;
    else
    {
        // Print the character
        printf("%c", str[spot]);
        // Call print_chars with next spot
        print_chars(str, spot + 1);
    }
}
```

Yes! (finally!)



- print_chars(str, 3)
- print_chars(str, 2)
- print_chars(str, 1)
- print_chars(str, 0)
- main()

Done with print_chars()!

main() goes on its merry way.

Fun Fact:

If you switch the two lines in else{}, you print the string backwards!

(Do you see why?)

```
void print_chars(char str[], int spot)
{
    // Base case: at end of string
    if (str[spot] == '\0')
        return;
    else
    {
        // Call print_chars with next spot
        print_chars(str, spot + 1);

        // Print the character
        printf("%c\n", str[spot]);
    }
}
```

Will call itself before printing, stacking frames, and will print when the frames are returning!

On the return:

spot = 0

spot = 1

spot = 2

spot = 3

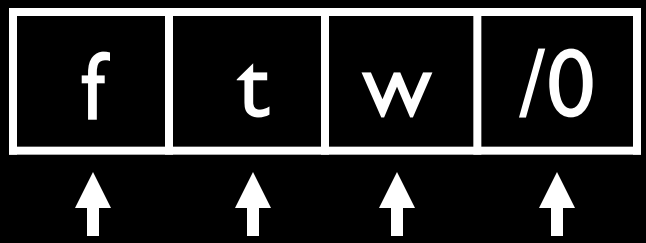
```
void
print_chars(char str[], int spot)
{
    // Base case: at end of string
    if (str[spot] == '\0')
        return;
    else
    {
        // Call print_chars with next spot
        print_chars(str, spot + 1);
        // Print the character
        printf("%c", str[spot]);}
    }
}
```

Done!

Somewhere in main():

… `print_chars(str, 0);` …

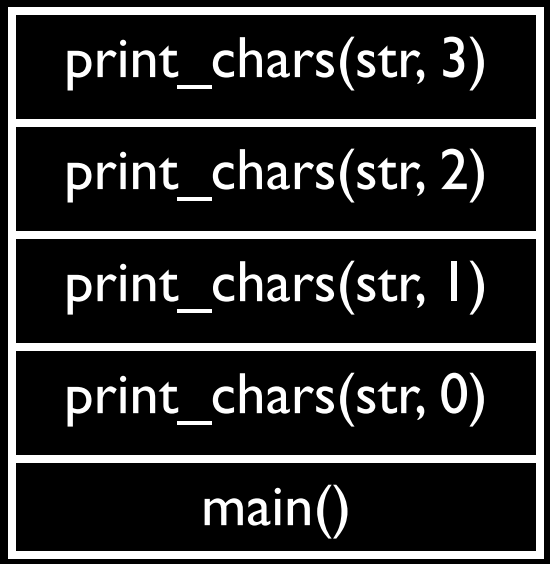
with str:



Printed:

wtf ?!

(ftw != wtf….)



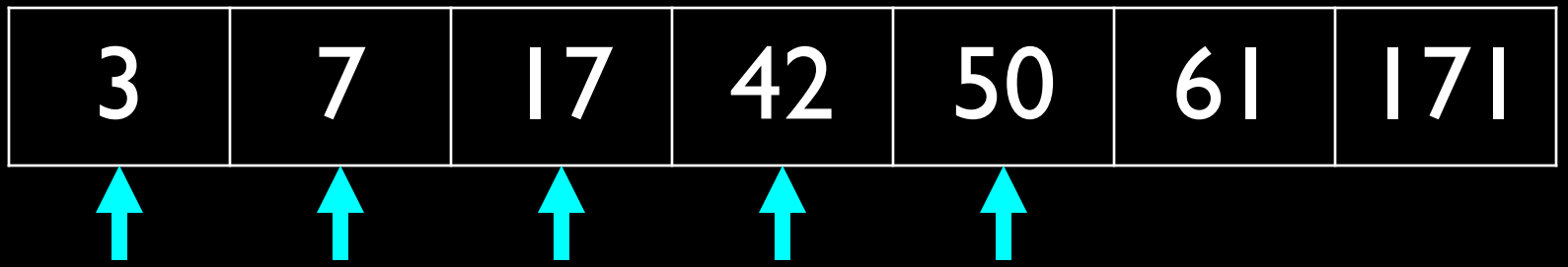
Quiz[Recursion]

- "What's this do?"
  - Think about call stack
    - Draw it out if need be
  - Remember where execution stopped on prev instance
    - (e.g. at the recursive call)
- "Write one."
  - What's repeating? What's changing?
  - Base case!

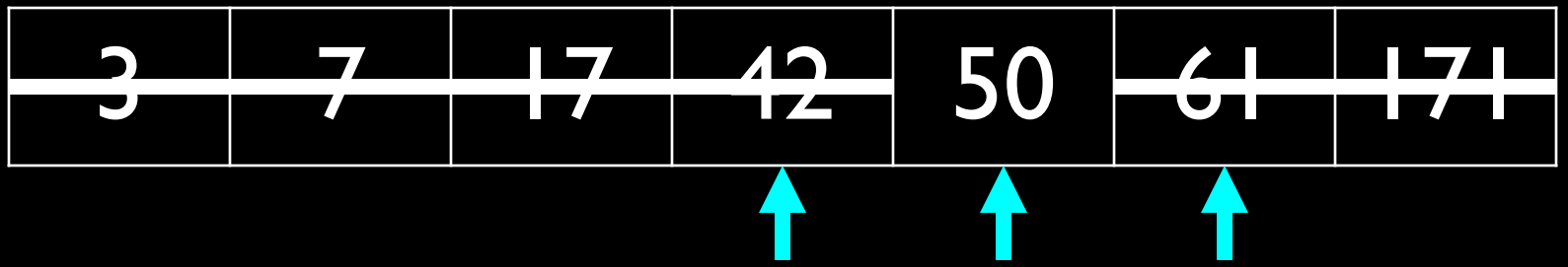
Searching

(for the number 50)

- Linear: $O(n)$, $\Omega(1)$



- Binary: $O(\log n)$, $\Omega(1)$

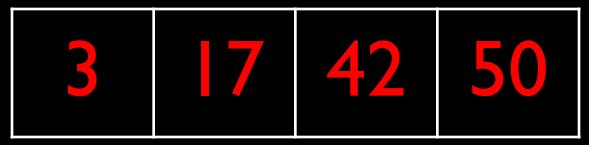


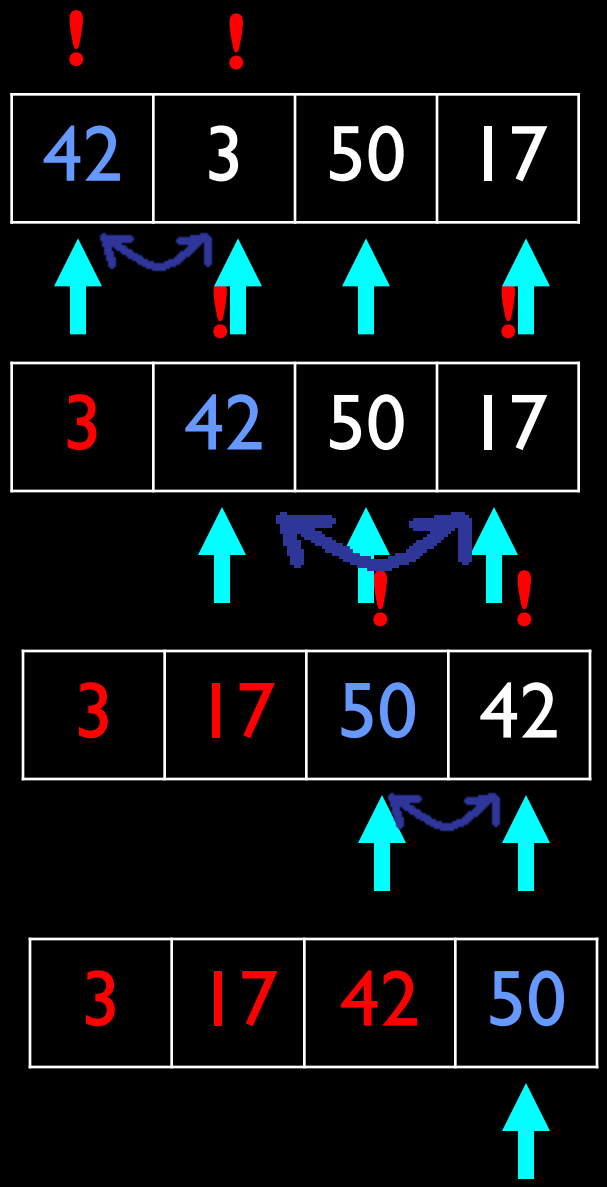
Note: list needs to be pre-sorted for binary search—but it's worth it!

Selection Sort

- $O(n^2)$, $\Omega(n^2)$

1. Look for smallest # in unsorted part
2. Switch it with the first slot of the unsorted
3. Repeat until all sorted

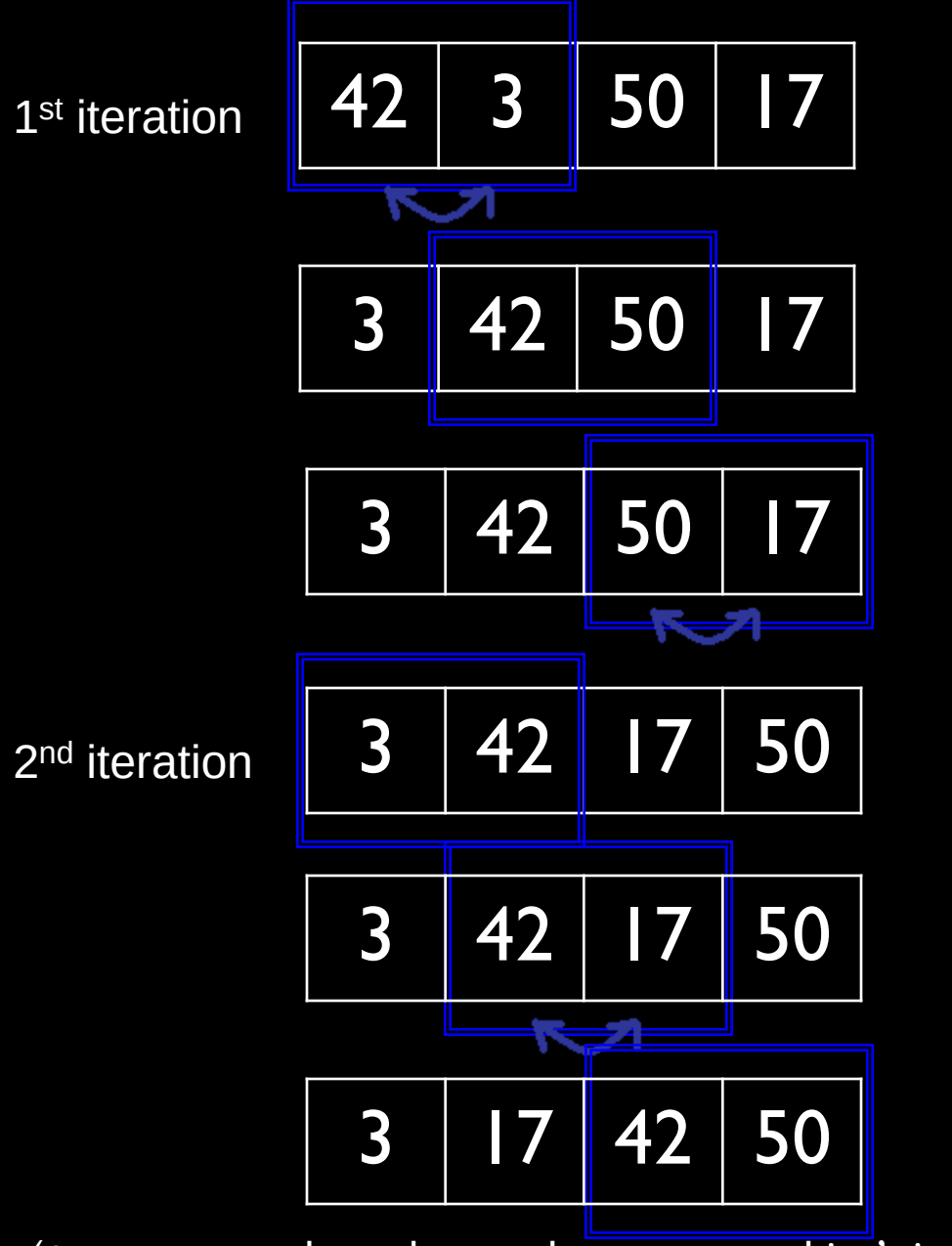




Bubble Sort

- $O(n^2)$, $\Omega(n)$

1. Go down list
   - If two adjacent #'s are out of order, swap 'em
2. When at end of list, repeat if swaps were made



(+ once more through to make sure everything's in order, that there aren't any swaps)

Merge Sort

- $O(n \log n)$, $\Omega(n \log n)$
- Recursive!

mSort (list of *n*)

→ If *n* <2, return.

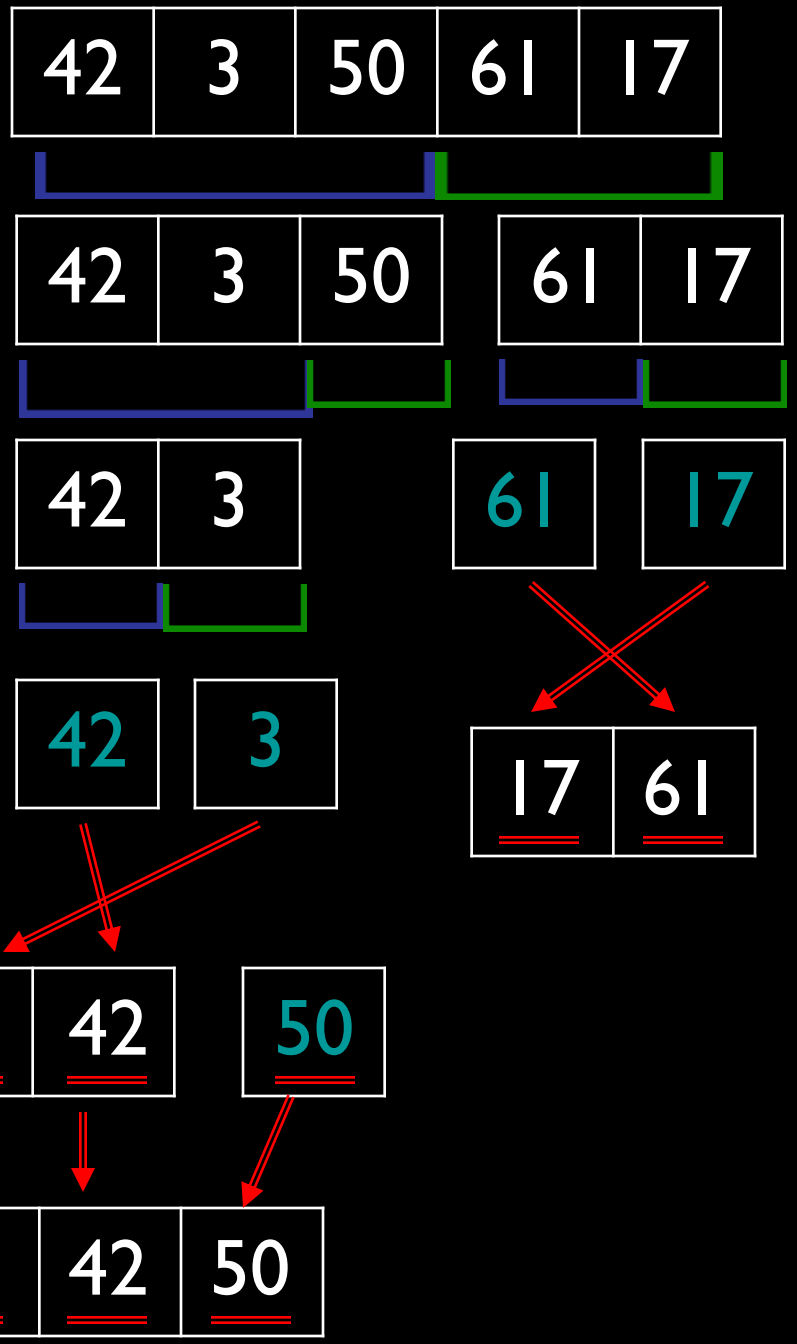
Else

→ **mSort** (left half).

→ **mSort** (right half).

→ Merge sorted

3	17	42	50	61



Asymptotic Notation

- Algorithm Efficiency
- **O(n) – upper bound**
- Ω(n) – lower bound
- Θ(n) – *O*(n) == Ω(n)

	O(n)	Assumptions
Linear Search	n	
Binary Search	Log(n)	Input is a sorted list
Selection Sort	n^2	
Insertion Sort	n^2	
Bubble Sort	n^2	
Merge Sort	nlog(n)	

(n = size of input)

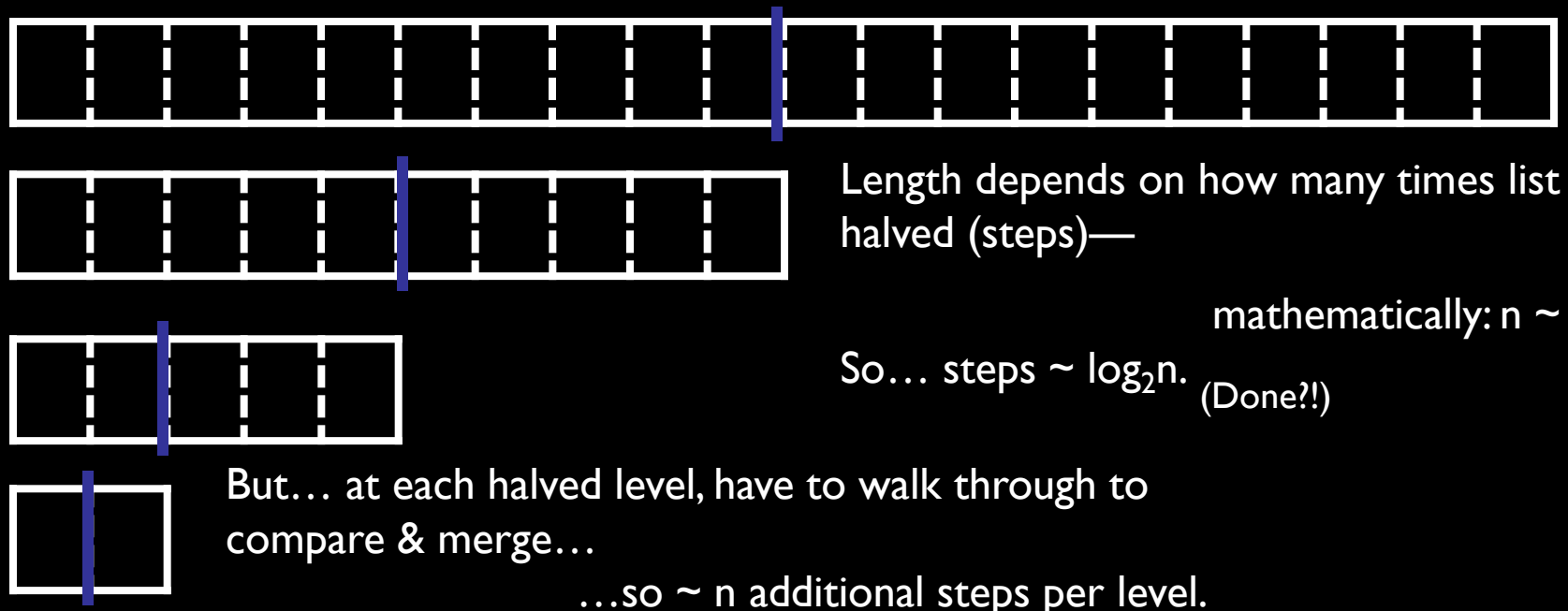
(best) Constant < Log < Linear < Quadratic / Polynomial < Exponential < Factorial (worst)

Quiz[AsymptoticNotation]

- Just memorize or cheatsheet it.
- Or…walk though algorithm & think about math.

(Ew.)

Ex: Merge Sort, list length *n*



Length depends on how many times list was halved (steps)—

mathematically: $n \sim 2^{steps}$

So… steps ~ $\log_2 n$. (Done?!)

But… at each halved level, have to walk through to compare & merge…

…so ~ n additional steps per level.

steps ~ $n * \log_2 n$

Part 3

Tian Feng

Fun With Pointers

Notation:

- &var returns the address of var
  - &tian == eliot
- *ptr returns the value referenced by ptr
  - *eliot == tian

Pointer Arithmetic

- Move around addresses
  - Incidentally, array[i] = *(array + i), we'll discuss this more later in the semester

Malloc and Heaps

Malloc

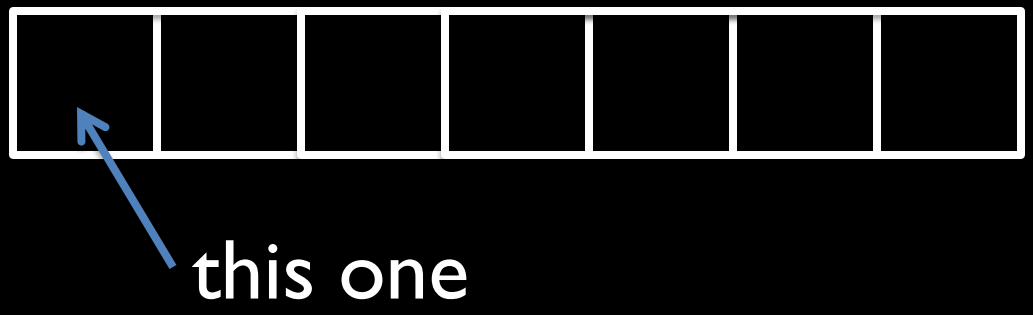
- Dynamic memory allocation
- Syntax:
  - type *array = malloc(size);
- Memory created on the heap
- Used in conjunction with free()
  - free(array);

Heap

- Dynamically allocated memory with variable length duration
- Exists until released by user

Arrays and Strings

- The name of an array is just a pointer to the first value in the array



- Strings are just arrays of chars
  - End with '\0', the null character
  - Thus the name of a string is a reference to the location of the first char of the string

Structs and Typedef

Structs: Custom object of aggregated data types

- struct name
  {
  _____;
  };

When referencing data in a struct:

- Struct.field
  - tian.name
- Ptr->field
  - eliot->name

Typedef: Custom naming of data types/objects

- typedef _____ name;

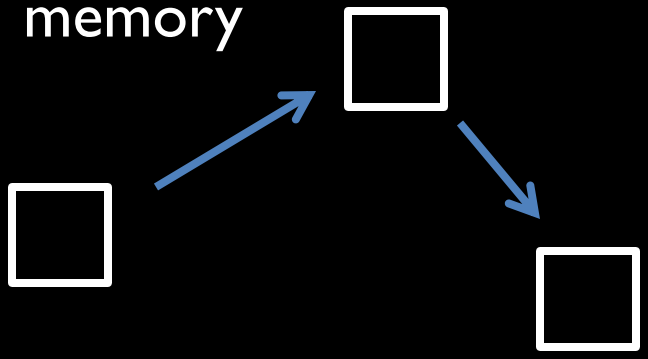
When using typedef and structs in conjunction:

- typedef struct
  {
  int id;
  char name[30];
  } student;

Linked Lists vs. Arrays

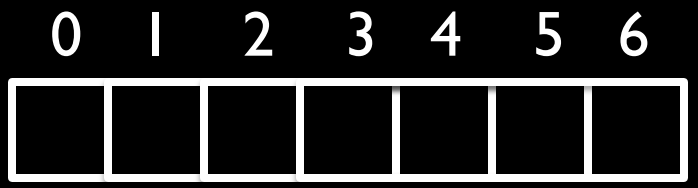
Linked Lists

- Organized collections
- Traversable
- Variable Length
- Variable Order
- Non Index-able
- Non-contiguous blocks of memory



Arrays

- Organized collections
- Traversable
- Fixed Length
- Fixed Order
- Index-able
- Contiguous blocks of memory



Linked Lists vs. Arrays Pros & Cons

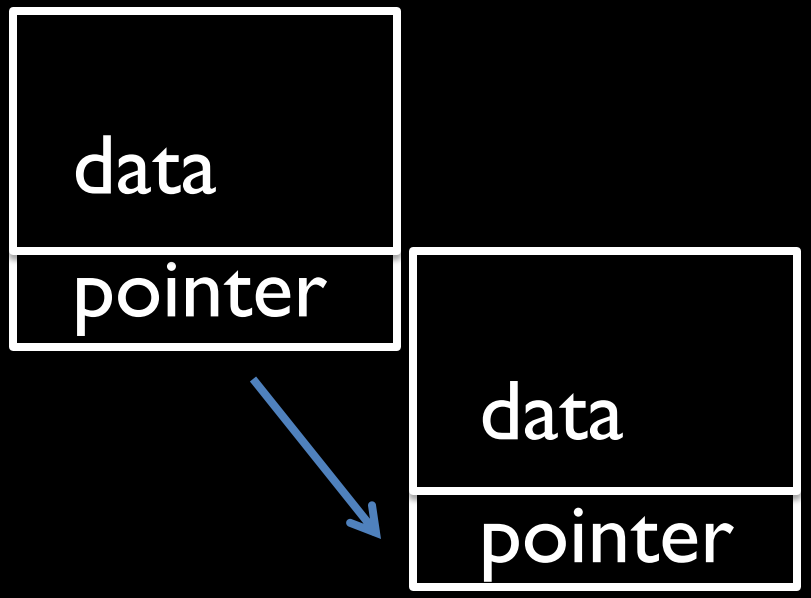
Linked Lists

- Pros
  - Arbitrary insertions and deletions
  - Easy to reorder
  - No need for a contiguous block of memory
- Cons
  - Linear time access
  - Overhead for pointer data

Arrays

- Cons
  - Need to realloc memory and transfer array
  - Need to "bump" every other value
  - Ahh! Contiguous block
- Pros
  - Constant time access (index)
  - Minimal storage overhead

Linked List Construction




```
typedef struct _node
{
        struct _node *next;
        _____________;
} student;
```

Stacks and Queues

Stacks

- LIFO
  - "last in first out"
- Real life applications:
  - Box of saltines
- Like the stack memory construct

Queues

- FIFO
  - "first in first out"
- Real life applications:
  - Lines in restaurants
  - Printer queues

File I/O

File related calls:

- fopen and fclose
  - Open and close file
- Fgetc
  - Gets a char from the file
- fprintf
  - Prints in file in stated format

questions?