Quiz 0 Review!

Part 0

Lexi Ross

Logistics

Quiz takes place on Wednesday 10/10 in lieu of lecture

(See http://cdn.cs50.net/2012/fall/quizzes/0/about0.pdf for details)

Bits ‘n’ Bytes

- A bit = 0 or 1
- A byte = 8 bits
- 11010011
  - How many bits?
  - How many bytes?
  - Convert it to hexadecimal!
  - Conver it to decimal!

Bits 'n' Bytes

- A bit = 0 or 1
- A byte = 8 bits
- 11010011
  - How many bits? 8
  - How many bytes? 1
  - Convert it to hexadecimal! 0xD3
  - Conver it to decimal! 211

Ye Olde ASCII Table

INT	CHAR		INT	CHAR	INT	CHAR	INT	CHAR
0	NUL	(null)	32	SPACE	64	@	96	`
1	SOH	(start of heading)	33	!	65	A	97	a
2	STX	(start of text)	34	"	66	B	98	b
3	ETX	(end of text)	35	#	67	C	99	c
4	EOT	(end of transmission)	36	$	68	D	100	d
5	ENQ	(enquiry)	37	%	69	E	101	e
6	ACK	(acknowledge)	38	&	70	F	102	f
7	BEL	(bell)	39	'	71	G	103	g
8	BS	(backspace)	40	(	72	H	104	h
9	HT	(horizontal tab)	41	)	73	I	105	i
10	LF	(line feed)	42	*	74	J	106	j
11	VT	(vertical tab)	43	+	75	K	107	k
12	FF	(form feed)	44	,	76	L	108	l
13	CR	(carriage return)	45	-	77	M	109	m
14	SO	(shift out)	46	.	78	N	110	n
15	SI	(shift in)	47	/	79	O	111	o
16	DLE	(data link escape)	48	0	80	P	112	p
17	DC1	(device control 1)	49	1	81	Q	113	q
18	DC2	(device control 2)	50	2	82	R	114	r
19	DC3	(device control 3)	51	3	83	S	115	s
20	DC4	(device control 4)	52	4	84	T	116	t
21	NAK	(negative acknowledge)	53	5	85	U	117	u
22	SYN	(synchronous idle)	54	6	86	V	118	v
23	ETB	(end of transmission block)	55	7	87	W	119	w
24	CAN	(cancel)	56	8	88	X	120	x
25	EM	(end of medium)	57	9	89	Y	121	y
26	SUB	(substitute)	58	:	90	Z	122	z
27	ESC	(escape)	59	;	91	[	123	{
28	FS	(file separator)	60	<	92	\	124	|
29	GS	(group separator)	61	=	93	]	125	}
30	RS	(record separator)	62	>	94	^	126	~
31	US	(unit separator)	63	?	95	_	127	DEL

ASCII Math

- ‘P’ + 1?

- ‘5’ ≠ 5
  - How would we transform one to the other?

ASCII Math

- ‘P’ + 1?
  - ‘Q’

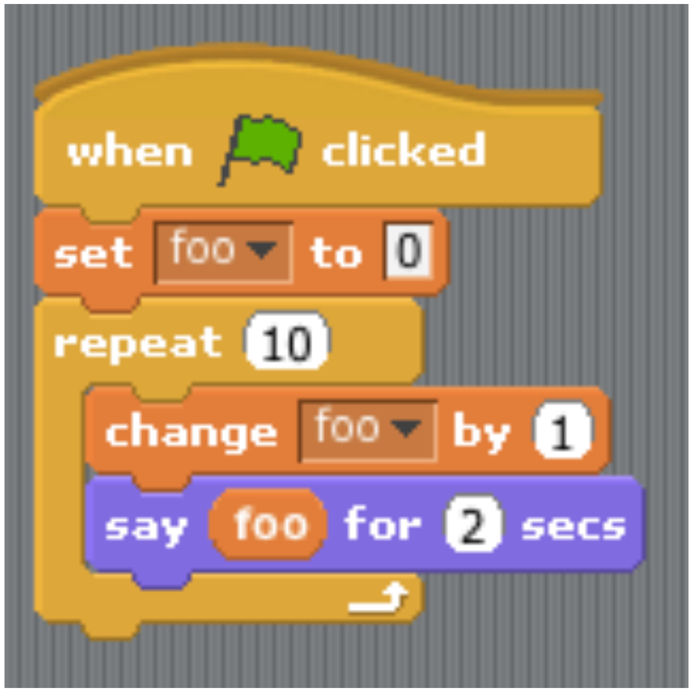
- ‘5’ ≠ 5
  - How would we transform one to the other?
    - ‘5’ – ‘0’ = 5
    - 5 + ‘0’ = ‘5’

Algorithms and Programming

- An algorithm is a specific set of instructions (like a recipe) for how to perform a certain task
  - *E.g.,* Checking whether a number is even or odd
  - *E.g.,* Binary search
- Programming is the act of converting an algorithm into code that a computer can understand

Algorithms and Programming

```
int foo = 0;

for (int i = 0; i < 10; i++)
{
   foo++;
   printf(“Foo: %d\n”, foo);
}
```



Boolean expressions and conditions

- `bool isSet = false;`

-
  ```
  int x = 5;
  if (x <= 5)
     printf("x is no more than 5!\n");
  ```
  - What will the above code print?
  - What is the condition?
- Operators: &&, ||, !, ==, <=, >=, <, >

Loops

When is each of the following structures *most* appropriate to use?

- for?
- while?
- do while?

Loops

When is each of the following structures *most* appropriate to use?

- for?
  - We already know how many times we want to iterate through our loop (could also use while)
- while?
  - We're not sure how **many** times we want our loop to run, but there is some condition that needs to be true for our loop to keep running
- do while?
  - Similar to while, but we want the code in our loop to run **at least once**

Loops

Each loop needs an initialization, a condition, and an update.

- for?

  -
    ```
    for (initialization; condition; update)
    {
      // do this
    }
    ```
- while?

  -
    ```
    initialization
    while (condition)
    {
      // do this
      // update
    }
    ```
- do while?

  -
    ```
    initialization
    do
    {
      // do this
      // update
    }
    while (condition);
    ```

Statements and Variables

```
int bar;
// What is bar’s value now?

bar = 42;
int baz = bar + 1;
baz++;
```

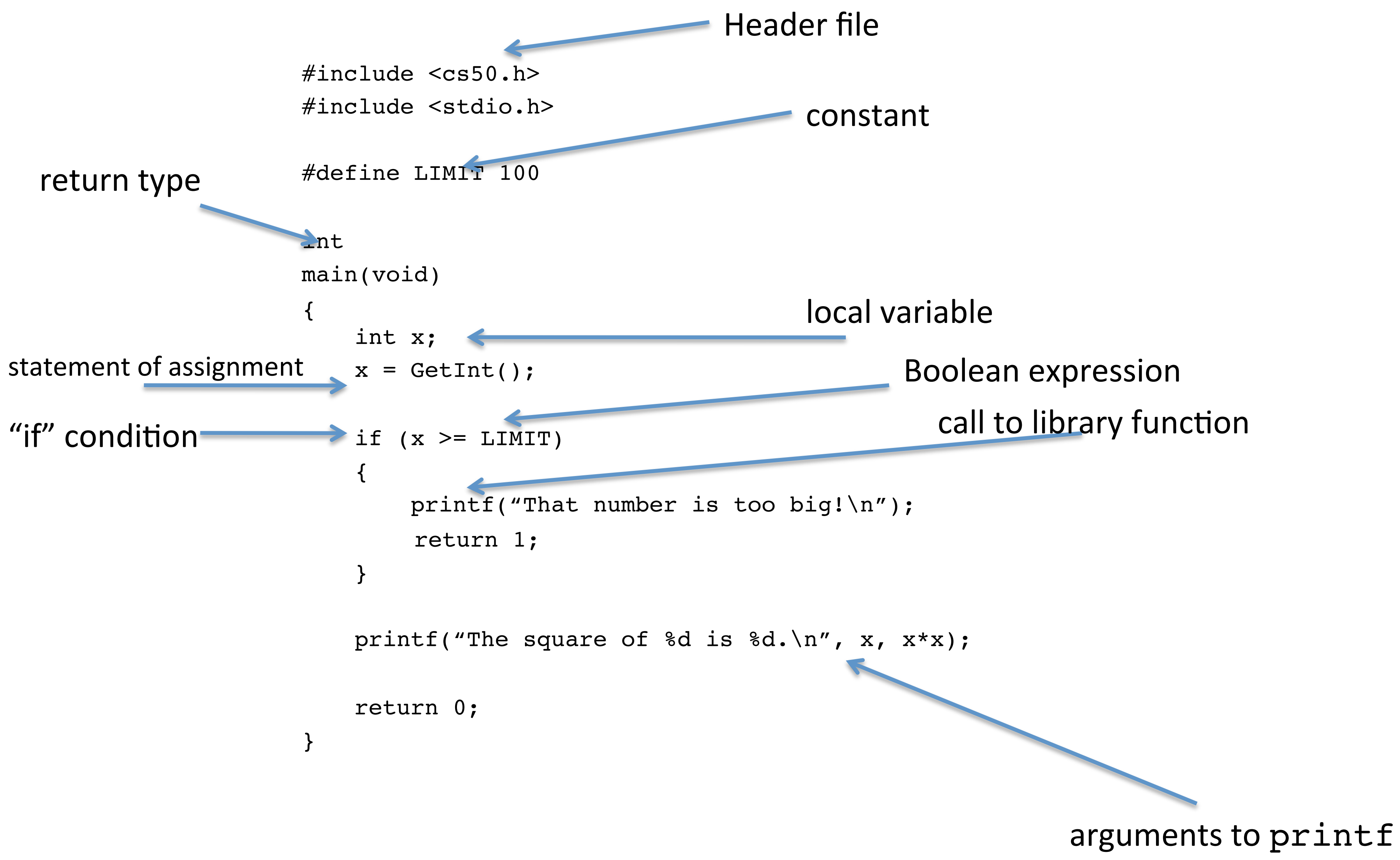
Threads and Events

- **Threads** refer to the concept of multiple sequences of code executing at the same time
  - Your computer isn't actually doing multiple things at once unless it has a multicore processor
  - *E.g.,* In Scratch, multiple sprites can execute scripts at the same time
- **Events** refers to concept of different elements of your code "communicating" with each other
  - In Scratch, this corresponds to the **Broadcast/When I Hear** blocks

A simple C program…

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

#define LIMIT 100

int
main(void)
{
    int x;
    x = GetInt();

    if (x >= LIMIT)
    {
        printf("That number is too big!\n");
        return 1;
    }

    printf("The square of %d is %d.\n", x, x*x);

    return 0;
}
```

…with lots of elements!



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

#define LIMIT 100

int
main(void)
{
    int x;
    x = GetInt();

    if (x >= LIMIT)
    {
        printf("That number is too big!\n");
        return 1;
    }

    printf("The square of %d is %d.\n", x, x*x);

    return 0;
}
```

MyFunnyPics.net

Week 1 and 2.1!

Lucas Freitas

C and compilers

- You
  - translate your ideas into language that your **compiler** can understand
- Your computer is dumb
  - It only understands 0s and 1s (binary code!)
- Compiler
  - translates your C code into language that your **computer** can understand (object code)
  - we use a compiler called **clang**

clang

clang

- Compiling a program
  - `clang program.c`
  - `clang -o program program.c`
  - `make program`

  `-lcs50, -lm`
- Running a program
  - `./a.out`
  - `./program`

Data types

Data types

Data size depends on the machine architecture. For 32-bit machines (the appliance for instance):

- int (4 bytes)
- char (1 byte)
- float (4 bytes)
- double (8 bytes)
- long (4 bytes)
- long long (8 bytes)
- string (char *) – size?

Casting

-
  ```
  int x = 3;
    printf("%d", x/2);
  ```

-
  ```
  int x = 3;
    float y = x;
    printf(”%.2f", y/2.);
  ```

-
  ```
  float x = 3.14;
    printf("%.2f", x);
    x = (int)x;
    printf("%.2f", x);
  ```

-
  ```
  int x = 65;
    char c = x;
    printf("%c", c);
  ```

Math operators

- `+`
- `-`
- `*`
- `/`
- `%`

```
10 % 3 == 1
```

Math operators

- Be careful when combining integer * and /
  - What is (3 / 2) * 2?
- precedence

Useful shortcuts

- `i += 1; or i++;`
  - same as i = i + 1;
- `i -= 1; or i--;`
  - same as i = i - 1;
- `*=`
  - i *= 2 is the same as i = i * 2;
- `/=`
  - i /= 2 is the same as i = i / 2;

Functions

- main **is** a function
- printf, GetInt, touppper
  - implemented in other libraries, included in the code
- you can also write your own functions!
  - draw, scramble, lookup
- saves code!

Format

```
// this comment explains the
// function of this function
returnType functionName (parameters)
{
    // doThis
    return something;
}
```

Example

- Sum all numbers in array of integers

```
int sumArray (int nums[],int length)
{
   int sum = 0;
   for (int i = 0; i < length; i++)
      sum += nums[i];
   return sum;
}
```

Conditions

- if
- else
- else if
- What is the difference between these two codes:

```
if (x > 0)
  printf (“positive!
\n”);
if (x ==0)
  printf (“zero!\n”);
if (x < 0)
  printf (“negative!
\n”);
```

```
if (x > 0)
  printf (“positive!
\n”);
else if (x ==0)
  printf (“zero!\n”);
else
  printf (“negative!
\n”);
```

AND and OR (Boolean expressions)

- &&
- ||
- if (condition1 && condition2 || condition3)
  - confusing and hard to understand the precedence
  - use parentheses for more than 2 conditions!
- if (condition1 && (condition2 || condition3))
- if ((condition1 && condition2) || condition3)

Switches

```
switch (expression)
{
   case value1:
      // do this
      break;
   case value2:
      // do that
      // break;
   …
   default:
      // do something else (optional)
}
```

quiz0

Tommy MacWilliam

tmacwilliam@cs.harvard.edu

October 7, 2012

Topics

- crypto
- scope
- arrays
- command-line arguments
- searching
- sorting
- asymptotic notation
- recursion

Variable scope

- global variables: accessible by all functions
  - defined outside of `main`
- local variables: accessible by a single block
  - defined within a block, only accessible in that block

Variable scope

```
int x = 5;
void f(void)
{
    int y = 6;
    x++;
}
void g(void)
{
    int y = 8;
    x--;
}
```

Arrays

- list of elements of the same type
- elements accessed by their **index** (aka position)
  - index starts at 0!
- `int array[3] = {1, 2, 3};`
- `array[1] = 4;`

Multi-dimensional Arrays

- can also have arrays of arrays!
- multi-dimensional array creates a grid instead of a list
- needs multiple indices: `int grid[3][5];`
  - 3 rows, 5 columns

Multi-dimensional Arrays

```
int grid[2][3] = {{1, 2, 3}, {4, 5, 6}};
grid[1][2] = 6;
```

		Columns		
		0	**1**	**2**
Rows	**0**	1	2	3
	1	4	5	6

Passing Arrays to Functions

- `ints`, `chars`, `floats`, etc. are passed by **value**
  - contents CANNOT be changed by the function they're passed to (unless we use pointers!)
- arrays (of any type) are passed by **reference**
  - contents CAN be changed by the function they're passed to

main

- main is a **function** that can take 2 arguments
  - `argc`: number of arguments given
  - `argv[]`: array of arguments

Arguments

- `./this is cs 50`
  - `argc == 4`
  - `argv[0] == "./this"`
  - `argv[1] == "is"`
  - `argv[2] == "cs"`
  - `argv[3] == "50"`
- `"50" != 50`
  - `atoi("50") == 50`

Big O

- O: worst-case runtime
  - given the worst possible scenario, how fast can we solve a problem?
    - e.g. array is in descending order, we want it in ascending order
  - upper bound on runtime

Omega

- Ω: best-case runtime
  - given the best possible scenario, how fast can we solve a problem?
    - e.g. array is already sorted
  - lower bound on runtime

Common Running Times

- in ascending order:
  - $O(1)$: constant
  - $O(\log n)$: logarithmic
  - $O(n)$: linear
  - $O(n \log n)$: linearithmic
  - $O(n^c)$: polynomial
  - $O(c^n)$: exponential
  - $O(n!)$: factorial

Comparing Running Times

- $O(n)$, $O(2n)$, and $O(5n + 3)$ are all asymptotically equivalent: $O(n)$
  - constants drop out, because n dominates
- similarly, $O(n^3 + 2n^2)$ and $O(n^3)$ are equivalent
  - n^3 dominates n^2
- however, $O(n^3) > O(n^2)$
  - 2 and 3 are not factors here, they're exponents

Linear Search

- implementation: iterate through each element of the list, looking for it
- runtime: $O(n)$, $\Omega(1)$
- does not require list to be sorted

Binary Search

- implementation: keep looking at middle elements
  - start at middle of list
  - if too high, forget right half and search of left half
  - if too low, forget left half and search of right half
- runtime: $O(\log n)$, $\Omega(1)$
- requires list to be sorted

Binary Search

```
while items remain to search
  if middle item matches
    return true
  if middle is less
    exclude middle and earlier items
  if middle is more
    exclude middle and later items
return false
```

Binary Search

50 61 121 124 143 161 164 171 175 182

Binary Search

164 171 175 182

Binary Search

164 171

Binary Search

164

Bubble Sort

- implementation: if adjacent elements are out of place, switch them
  - repeat until no swaps are made
- runtime: $O(n^2)$, $\Omega(n)$

Bubble Sort

```
do
  swapped = false
  for i = 0 to n - 2
    if array[i] > array[i + 1]
      swap array[i] and array[i + 1]
      swapped = true
while elements have been swapped
```

Bubble Sort

5 0 1 6 4

Bubble Sort

0 5 1 6 4

Bubble Sort

0 1 5 6 4

Bubble Sort

0 1 5 6 4

Bubble Sort

0 1 5 4 6

Bubble Sort

0 1 5 4 6

Bubble Sort

0 1 5 4 6

Bubble Sort

0 1 4 5 6

Bubble Sort

0 1 4 5 6

Selection Sort

- implementation: start at beginning of list, find smallest element
  - swap first element with smallest element
  - go to second element, treat that as the new first element, continue
    - because everything to the left is already sorted
- runtime: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$

Selection Sort

```
for i = 0 to n - 2
  min = i
  for j = i + 1 to n - 1
    if array[j] < array[min]
      min = j
  if array[min] != array[i]
    swap array[min] and array[i]
```

Selection Sort

5 0 1 6 4

Selection Sort

0 5 1 6 4

Selection Sort

0 1 5 6 4

Selection Sort

0 1 4 6 5

Selection Sort

0 1 4 5 6

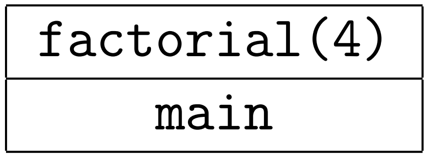
Recursion

- base case: when function should stop calling itself
  - without a base case, function would call itself forever!
- recursive case: function calls itself, probably using different arguments

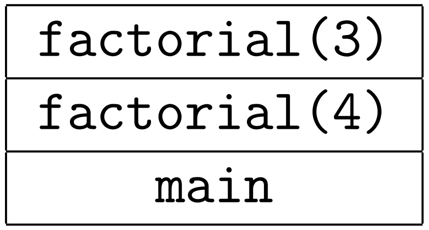
Recursion

```
int factorial(int n)
{
    if (n <= 1)
        return 1;

    return n * factorial(n - 1);
}
```

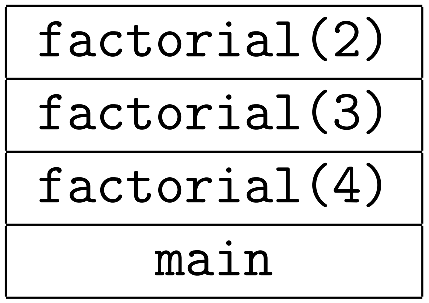
Recursion and the Stack



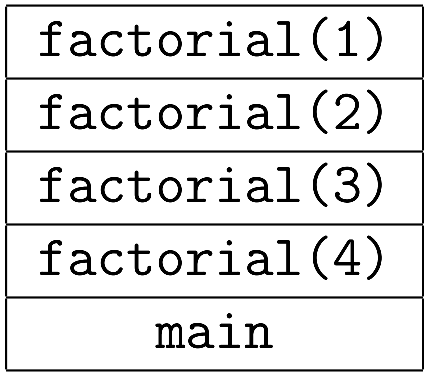
Recursion and the Stack



Recursion and the Stack



Recursion and the Stack



CS50

This is CS50. (Quiz 0 Review)

o hai!

Joseph Ong

CS50: Quiz 0

Merge Sort

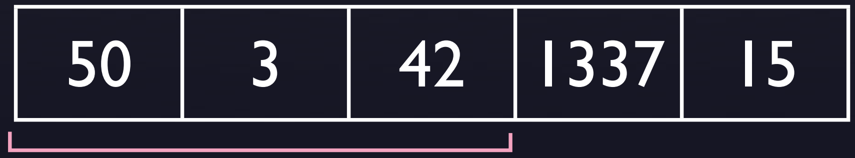
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```

50	3	42	1337	15

Merge Sort

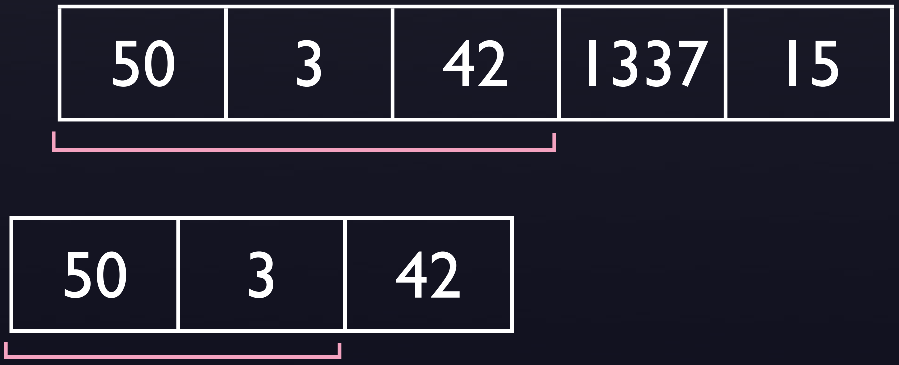
```
mSort (list of n numbers)
    if n < 2
        return;
    else
  →     mSort left half;
        mSort right half;
        merge sorted halves;
```



50	3	42	1337	15

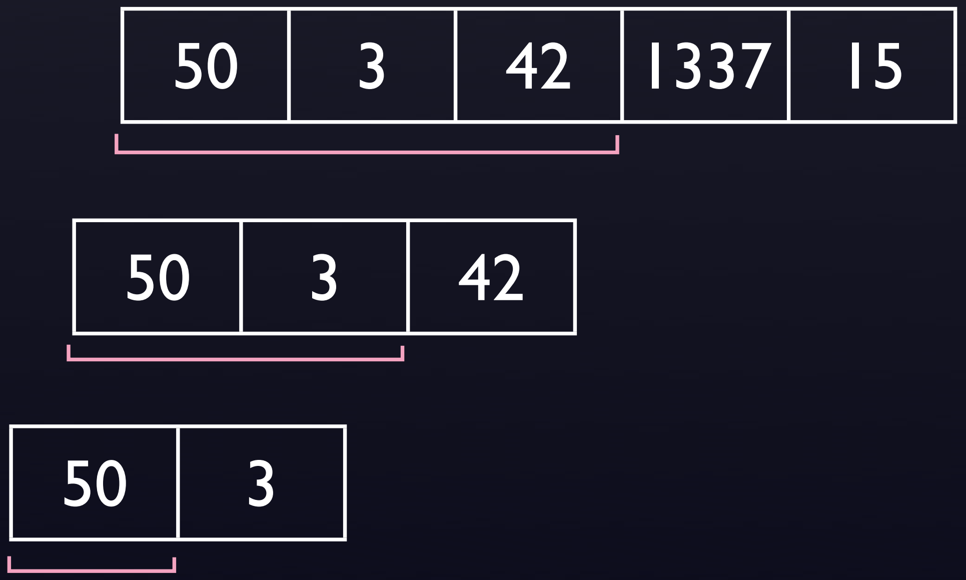
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
  →     mSort left half;
        mSort right half;
        merge sorted halves;
```



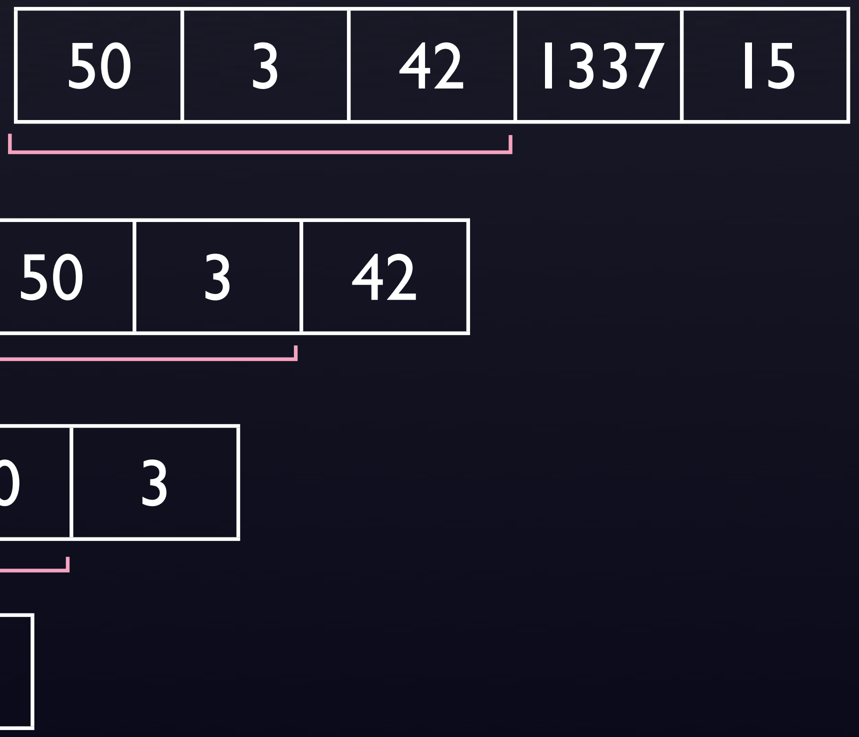
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
  ⟶     mSort left half;
        mSort right half;
        merge sorted halves;
```



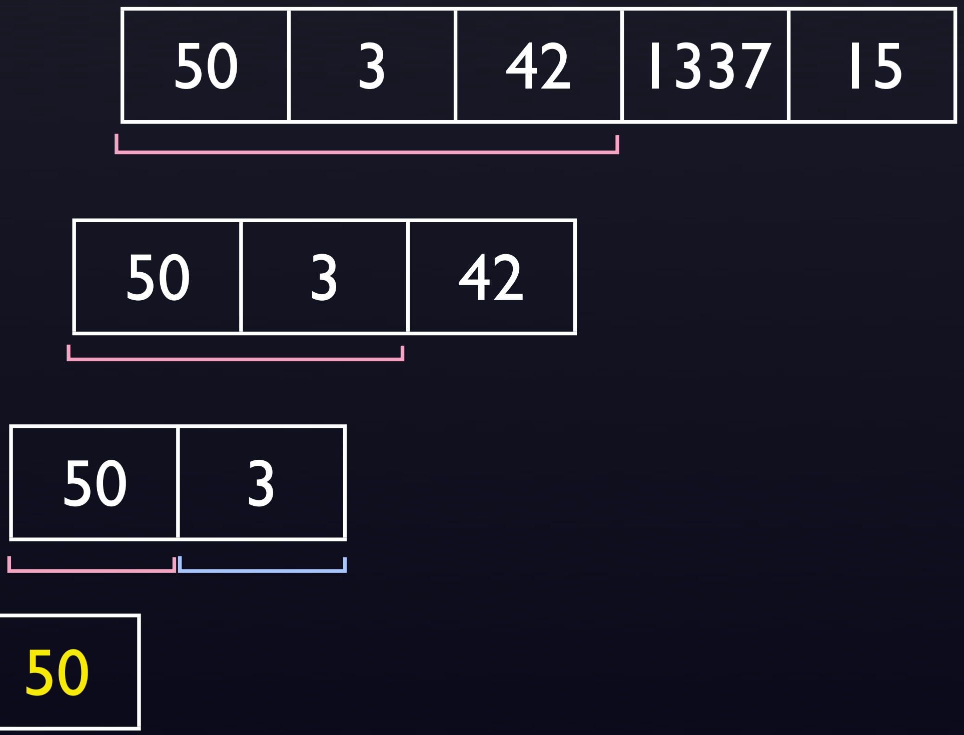
Merge Sort

```
mSort (list of n numbers)
    if n < 2
  ⟶    return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



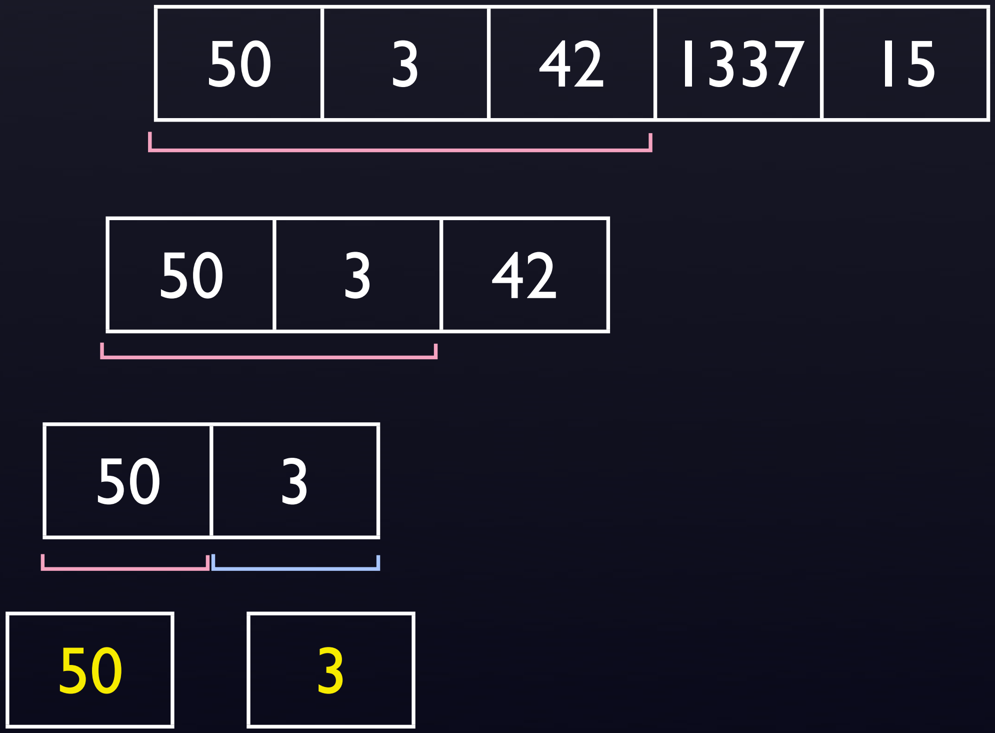
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
  →     mSort right half;
        merge sorted halves;
```



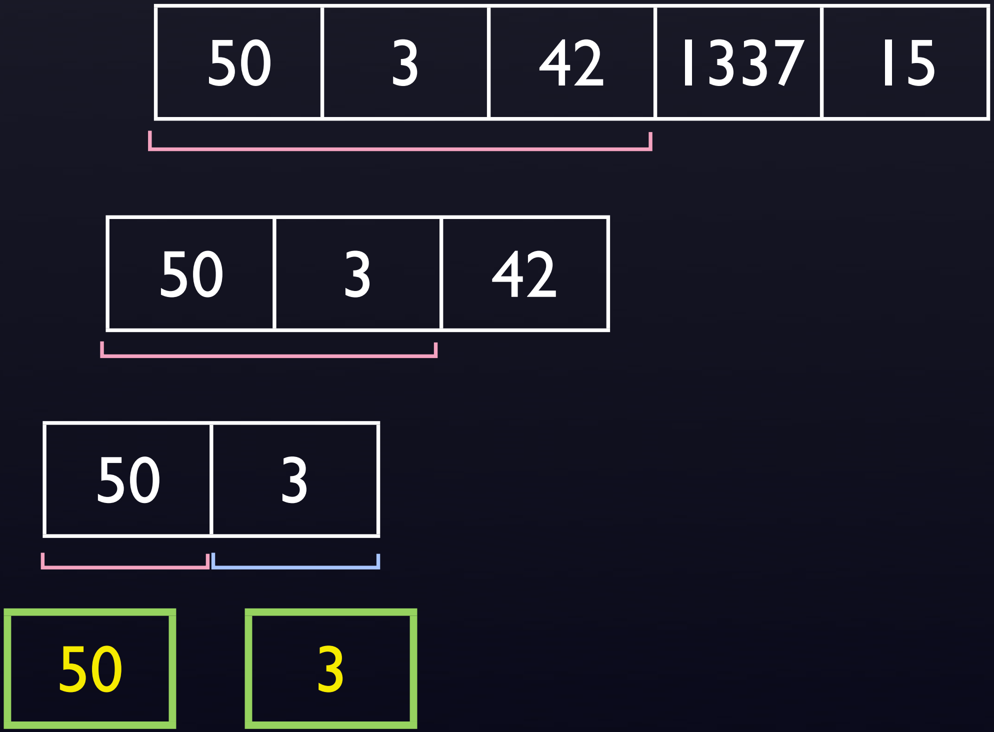
Merge Sort

```
mSort (list of n numbers)
    if n < 2
  →     return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



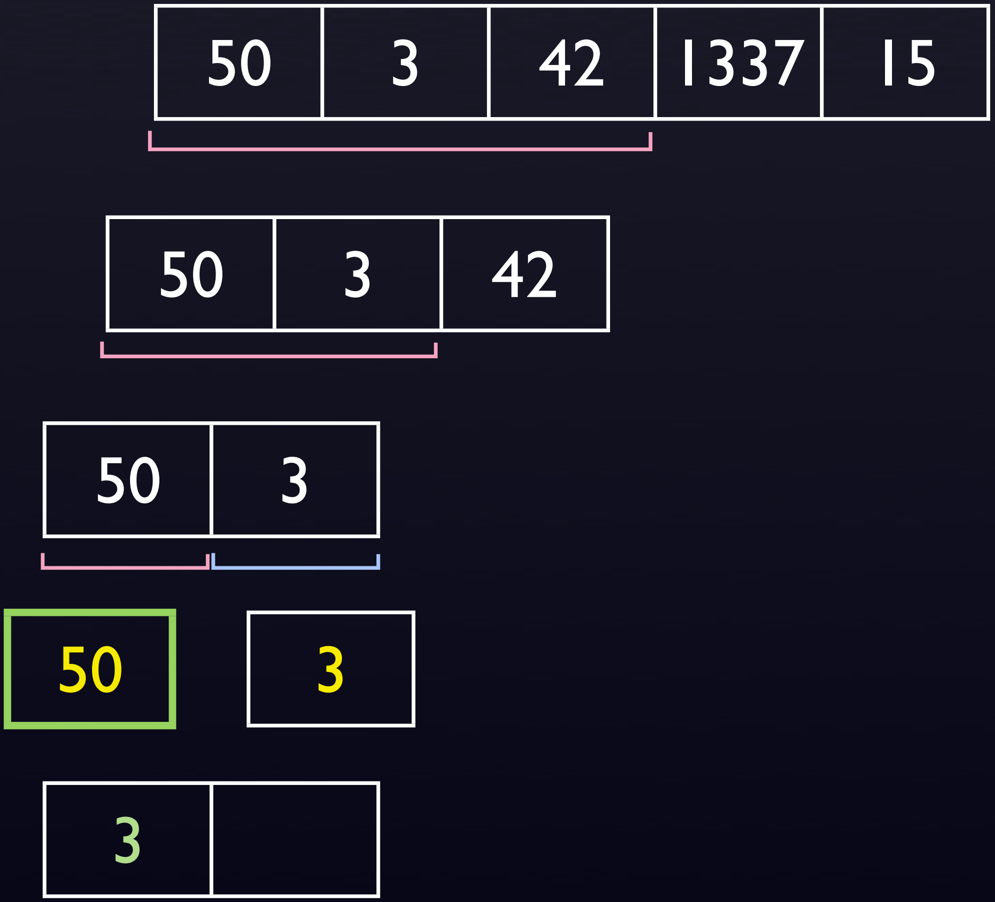
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  ⟶     merge sorted halves;
```



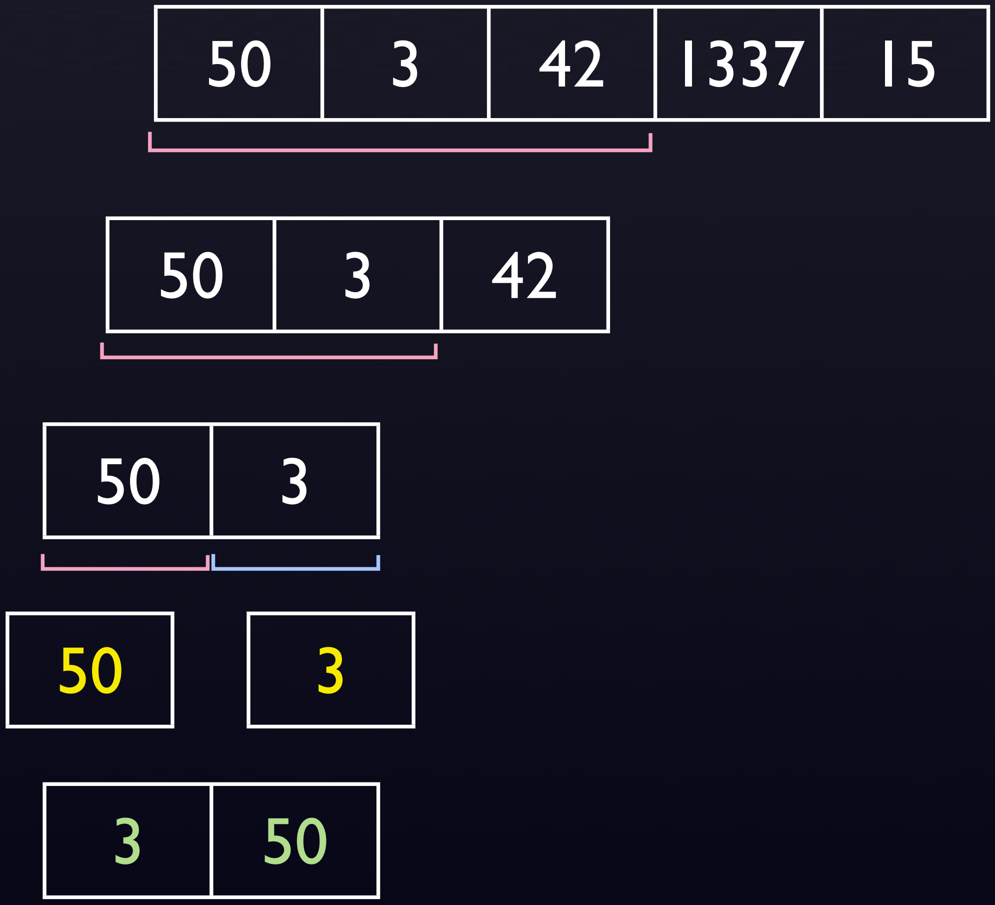
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  ⟶    merge sorted halves;
```



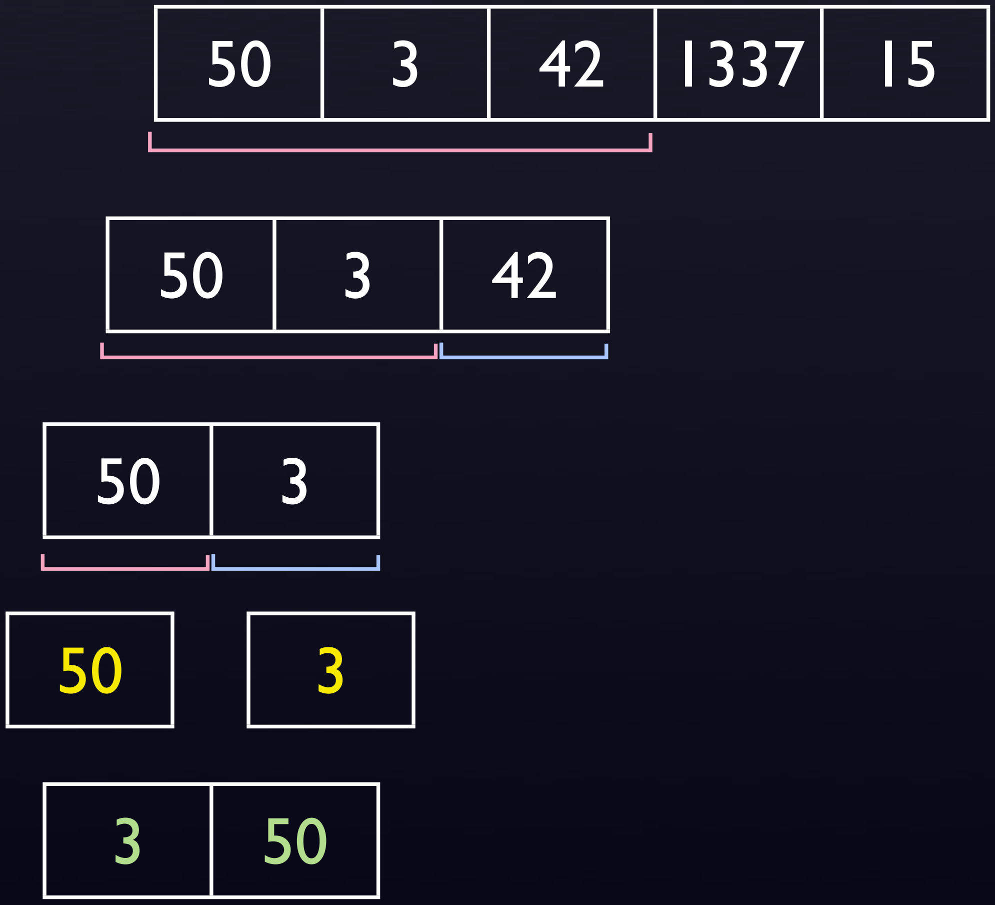
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



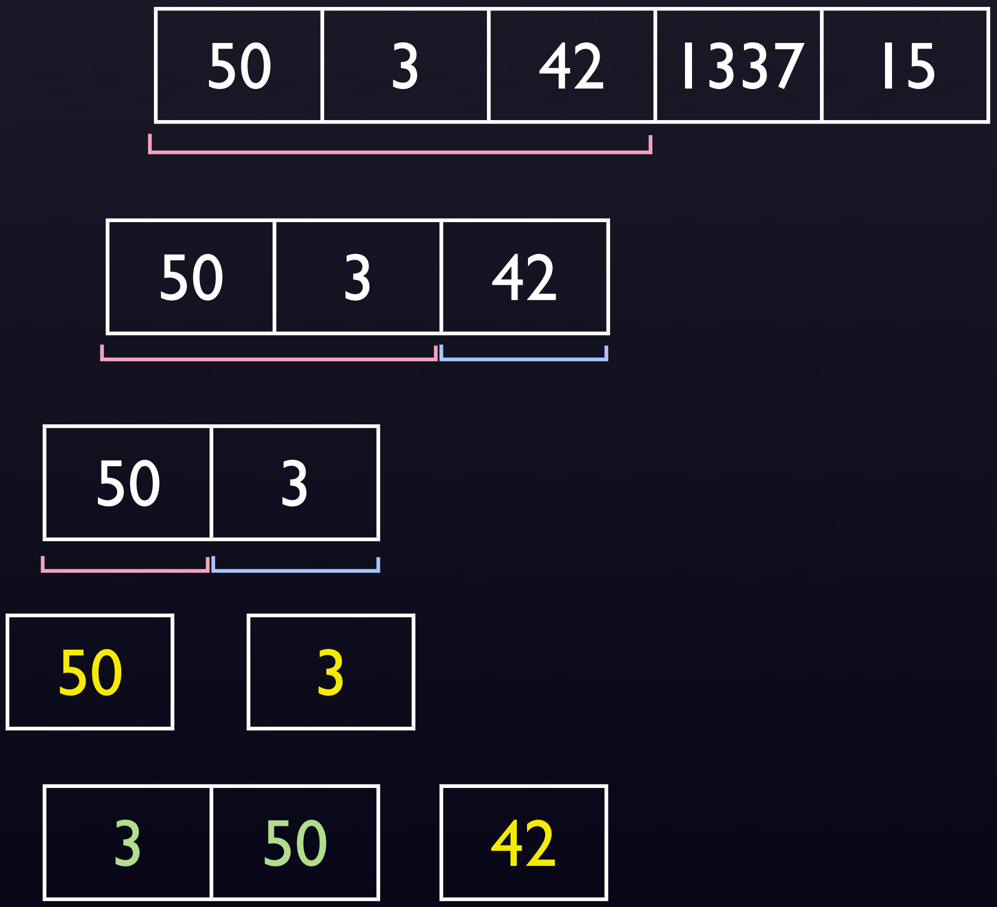
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
  →     mSort right half;
        merge sorted halves;
```



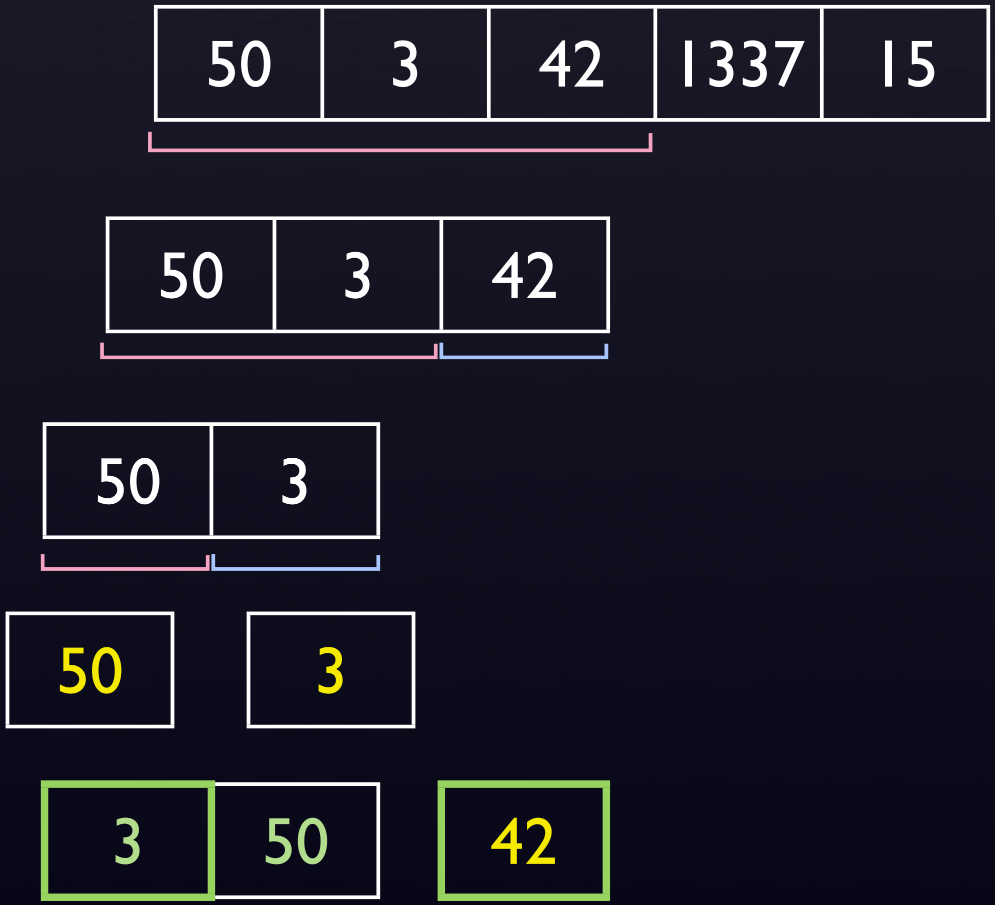
Merge Sort

```
mSort (list of n numbers)
    if n < 2
  ⟶ return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



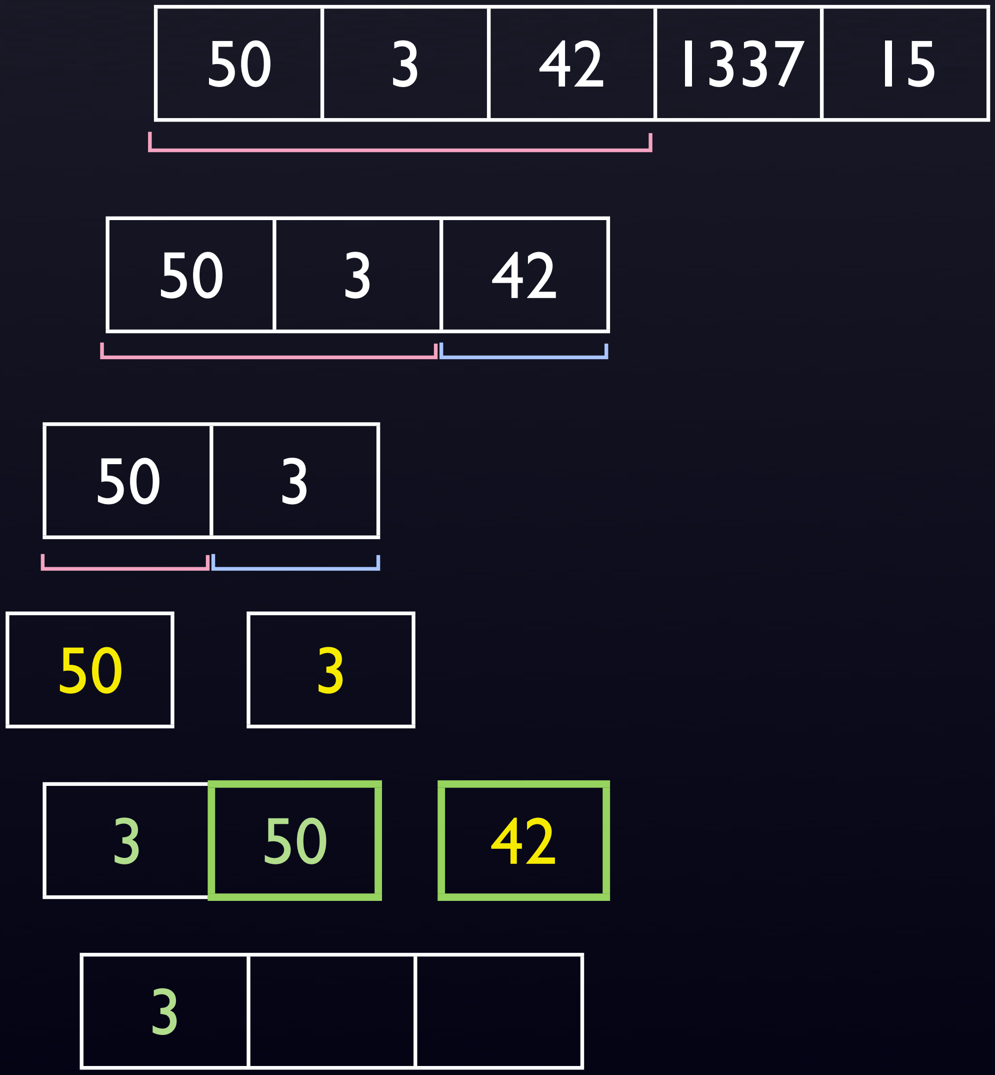
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
    →   merge sorted halves;
```



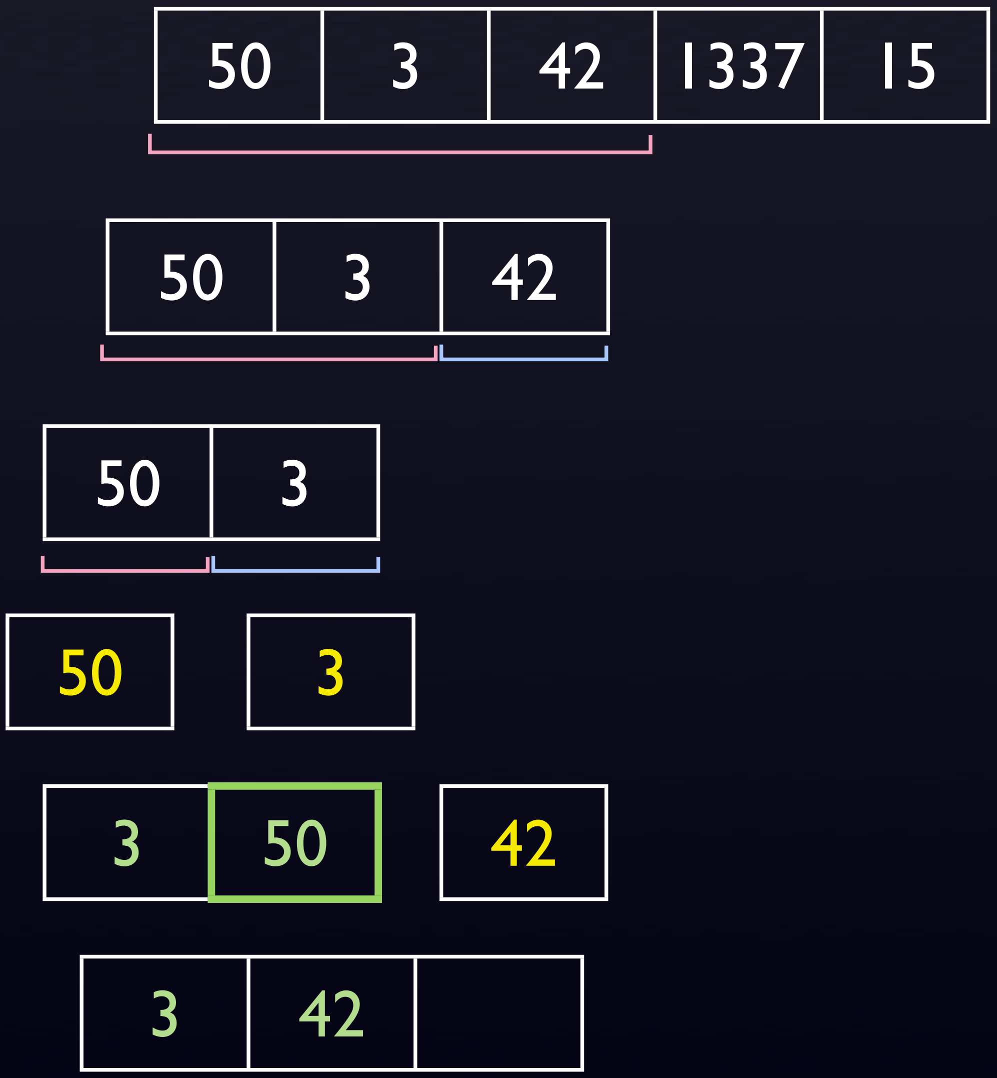
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



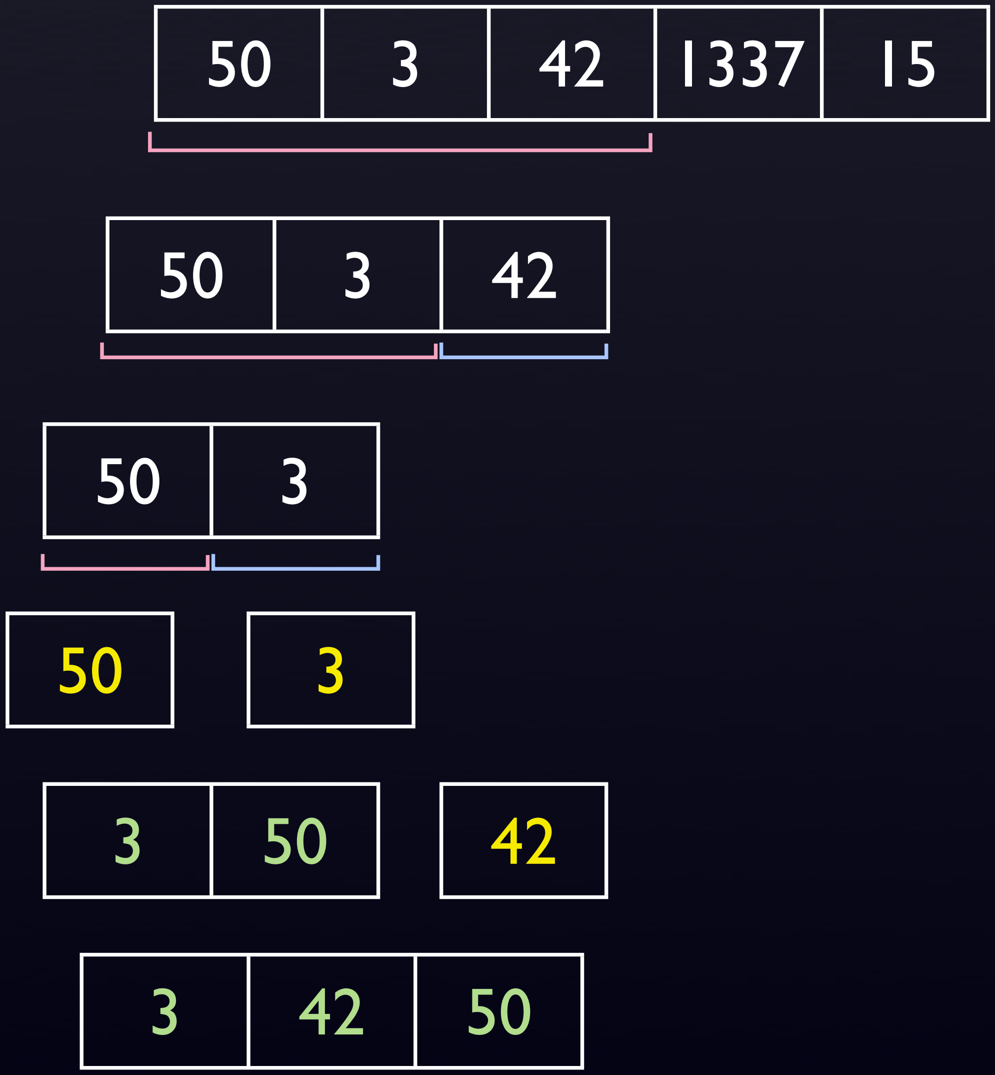
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  ───→  merge sorted halves;
```



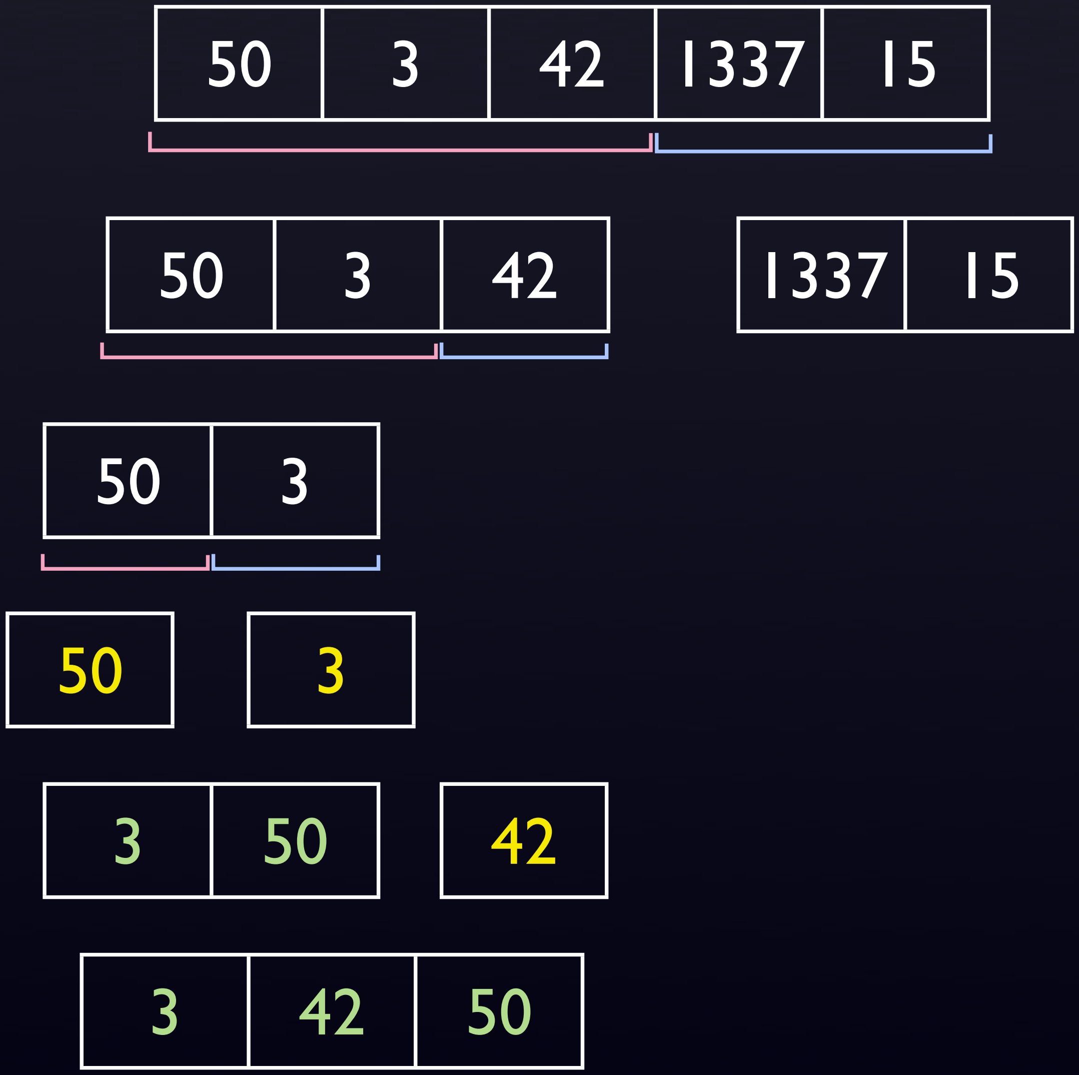
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



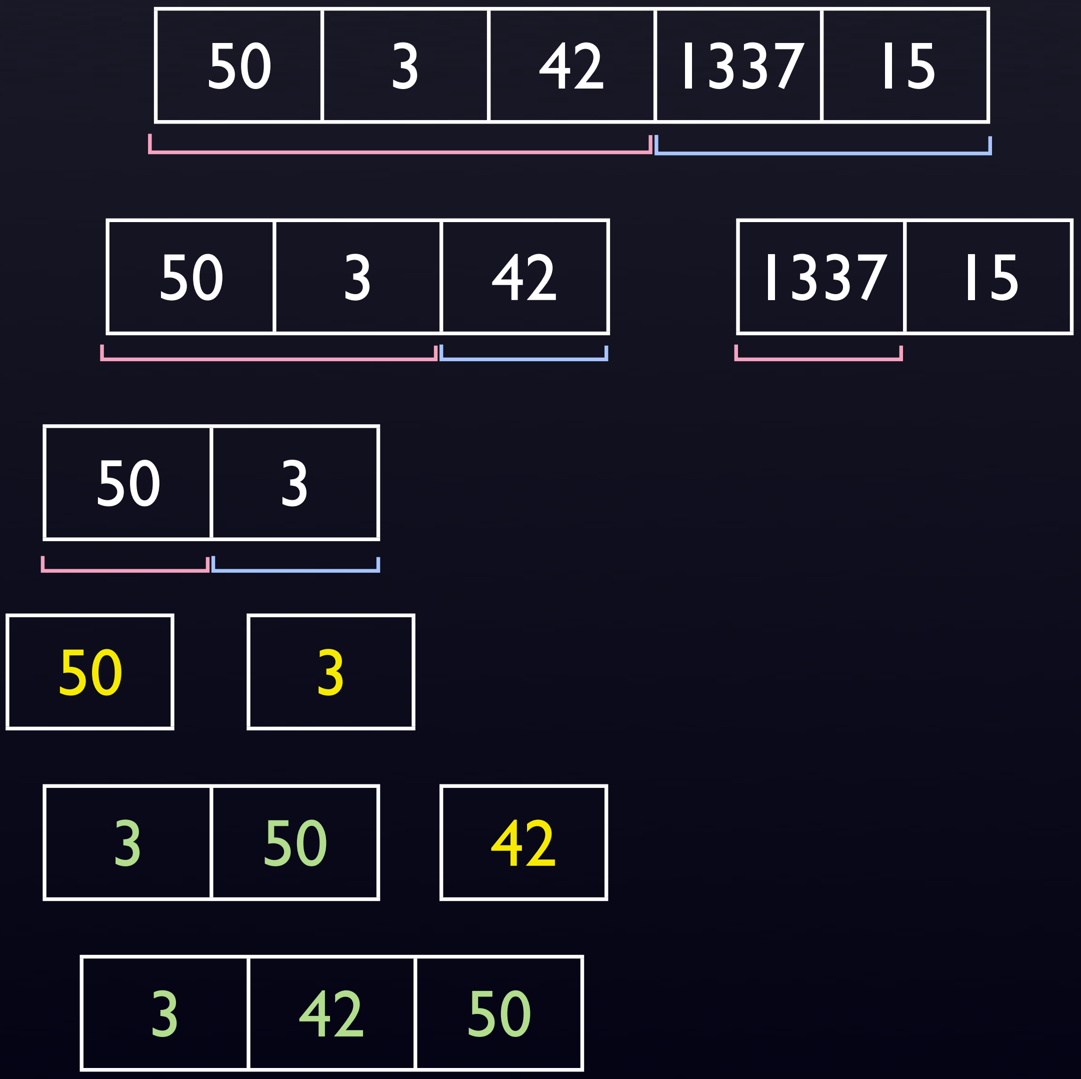
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
  →     mSort right half;
        merge sorted halves;
```



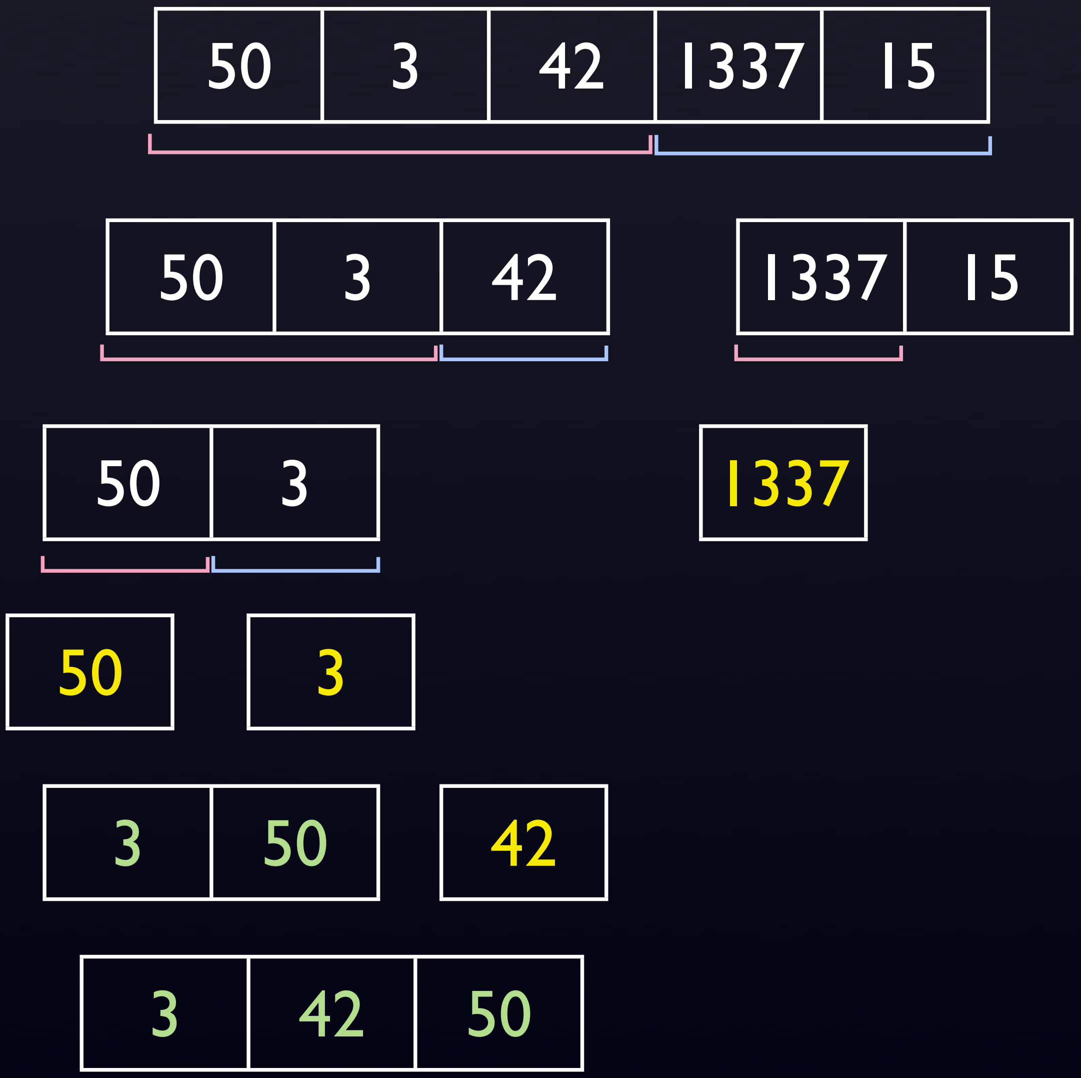
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
  →     mSort left half;
        mSort right half;
        merge sorted halves;
```



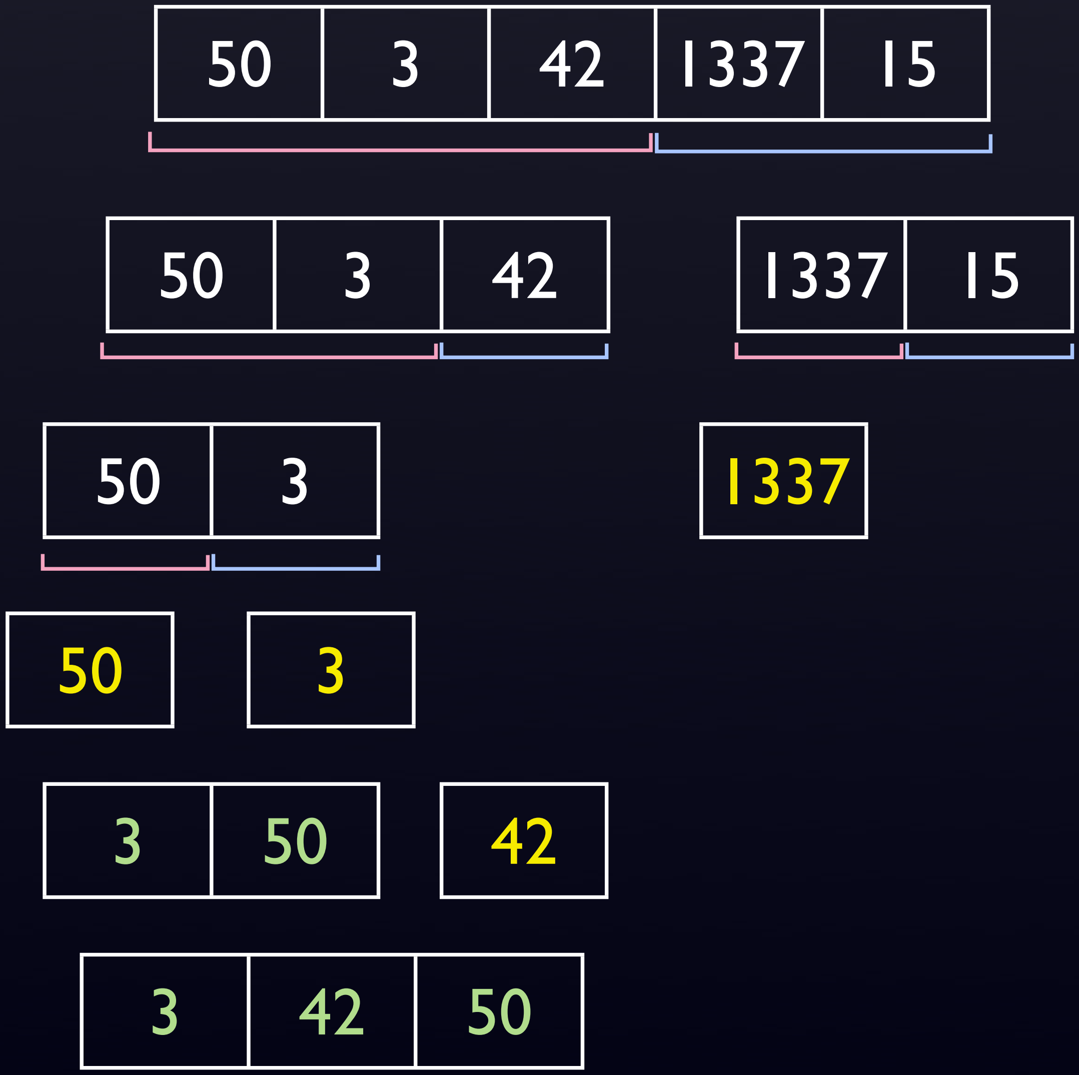
Merge Sort

```
mSort (list of n numbers)
    if n < 2
  ⟶ return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



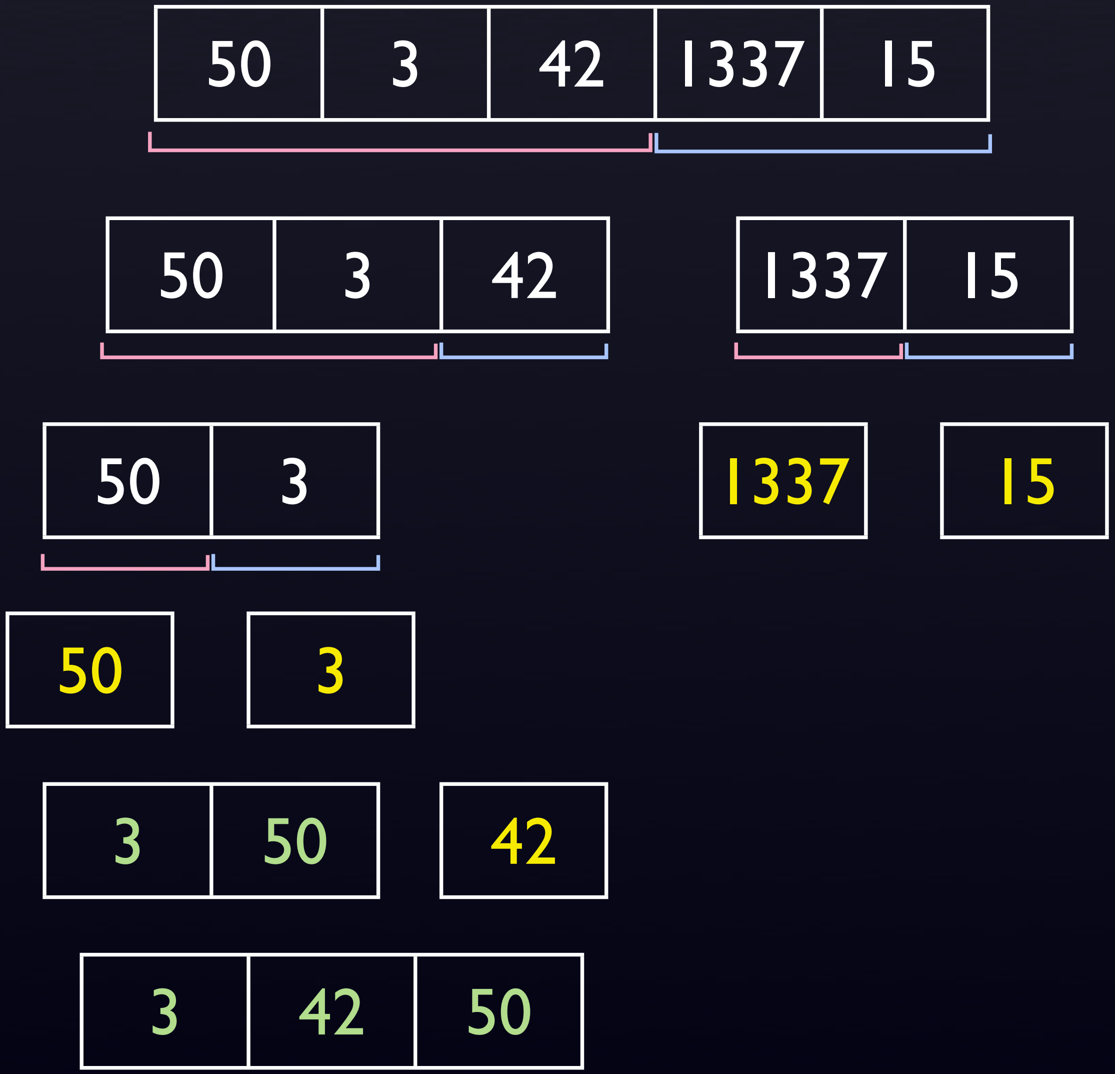
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
  →     mSort right half;
        merge sorted halves;
```



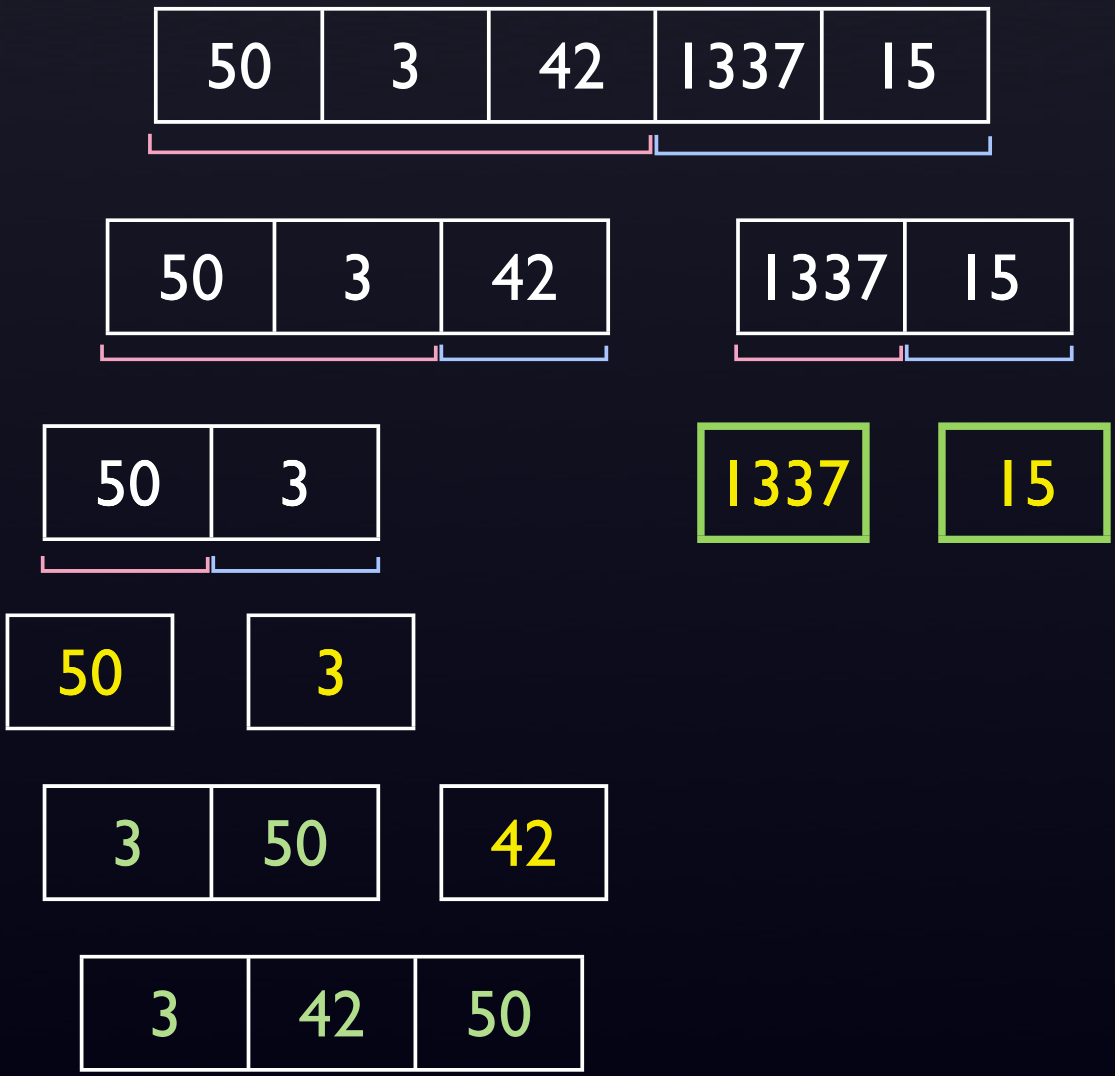
Merge Sort

```
mSort (list of n numbers)
    if n < 2
  ⟶     return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



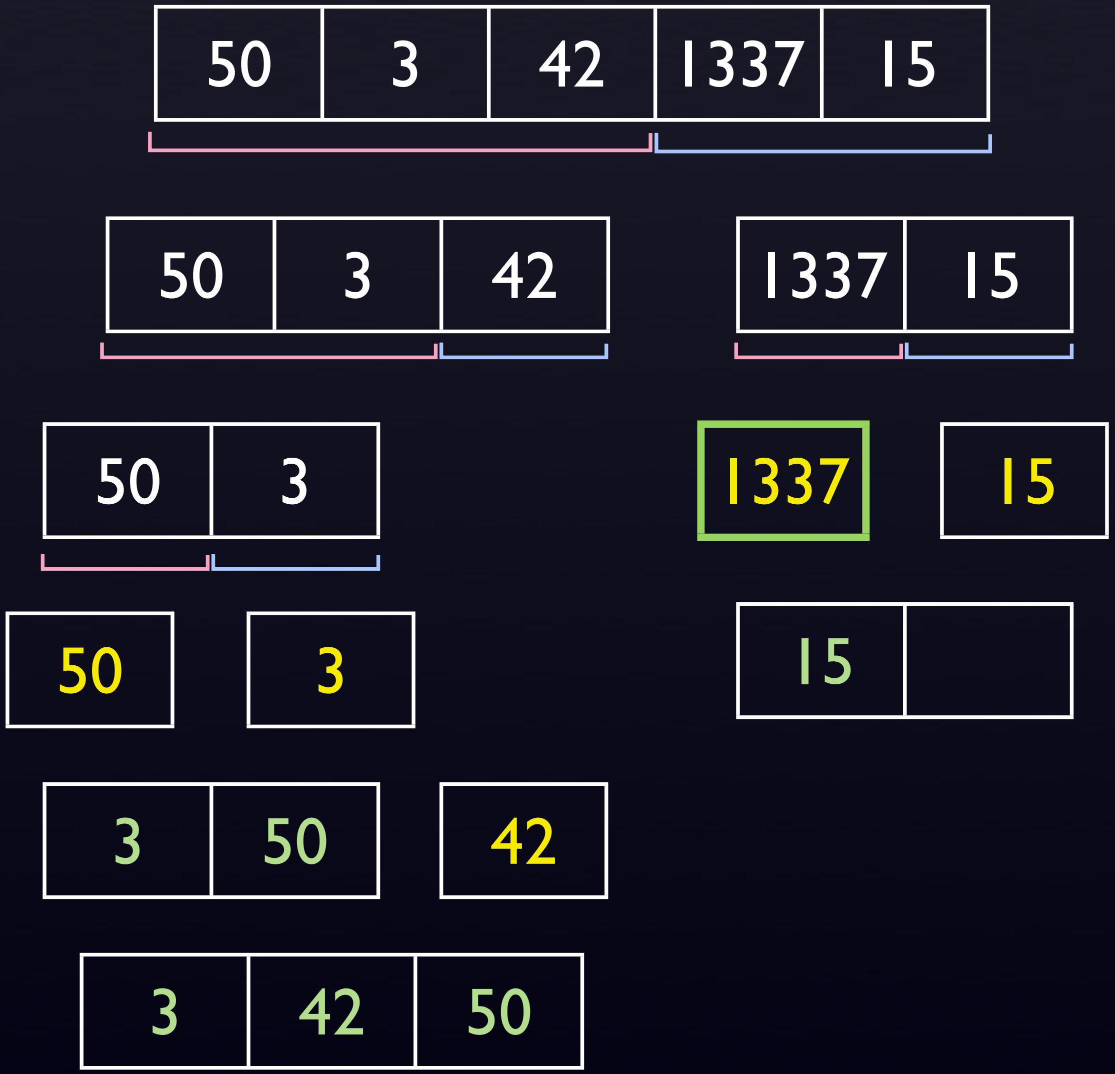
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
 →      merge sorted halves;
```



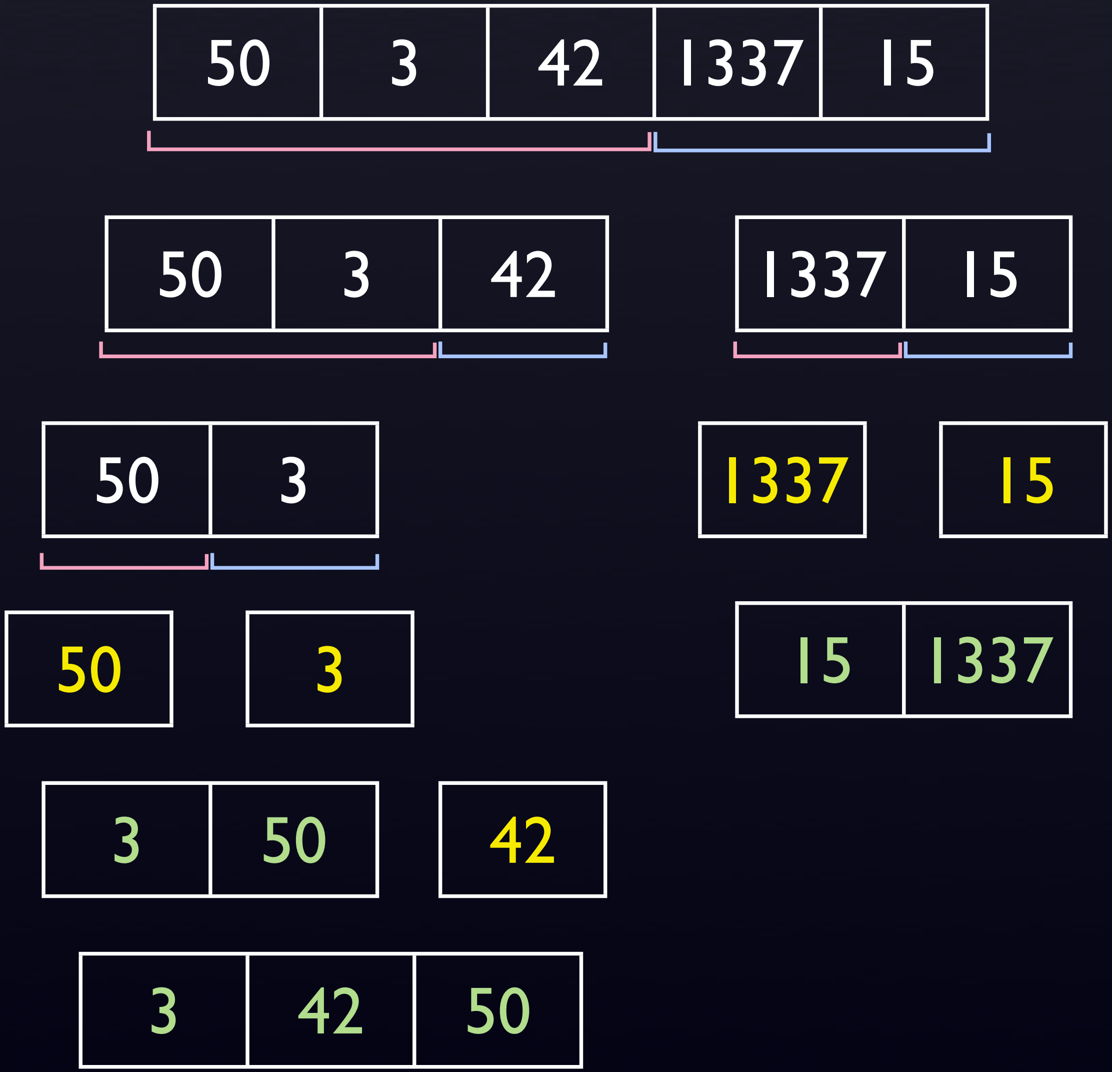
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



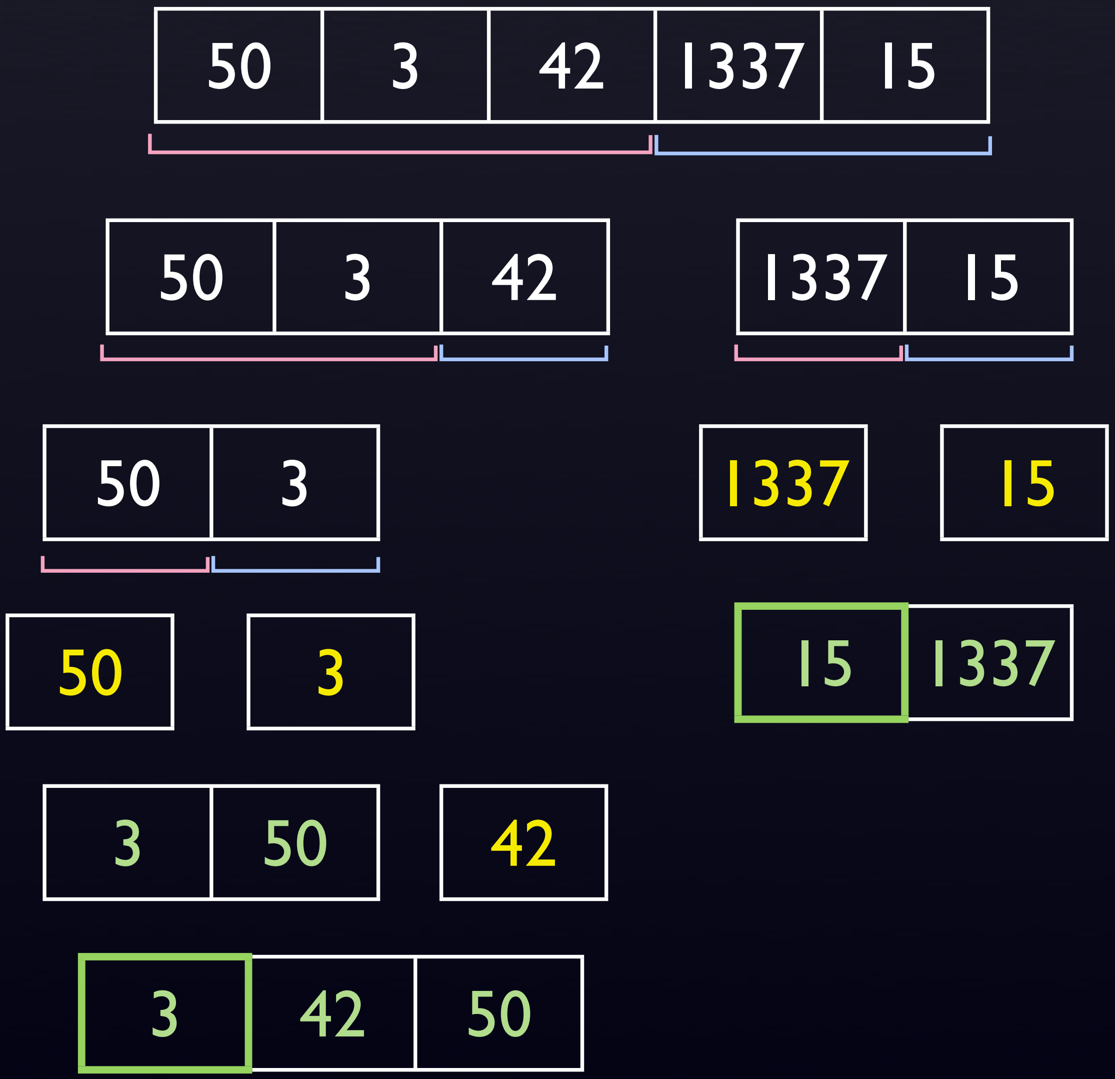
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



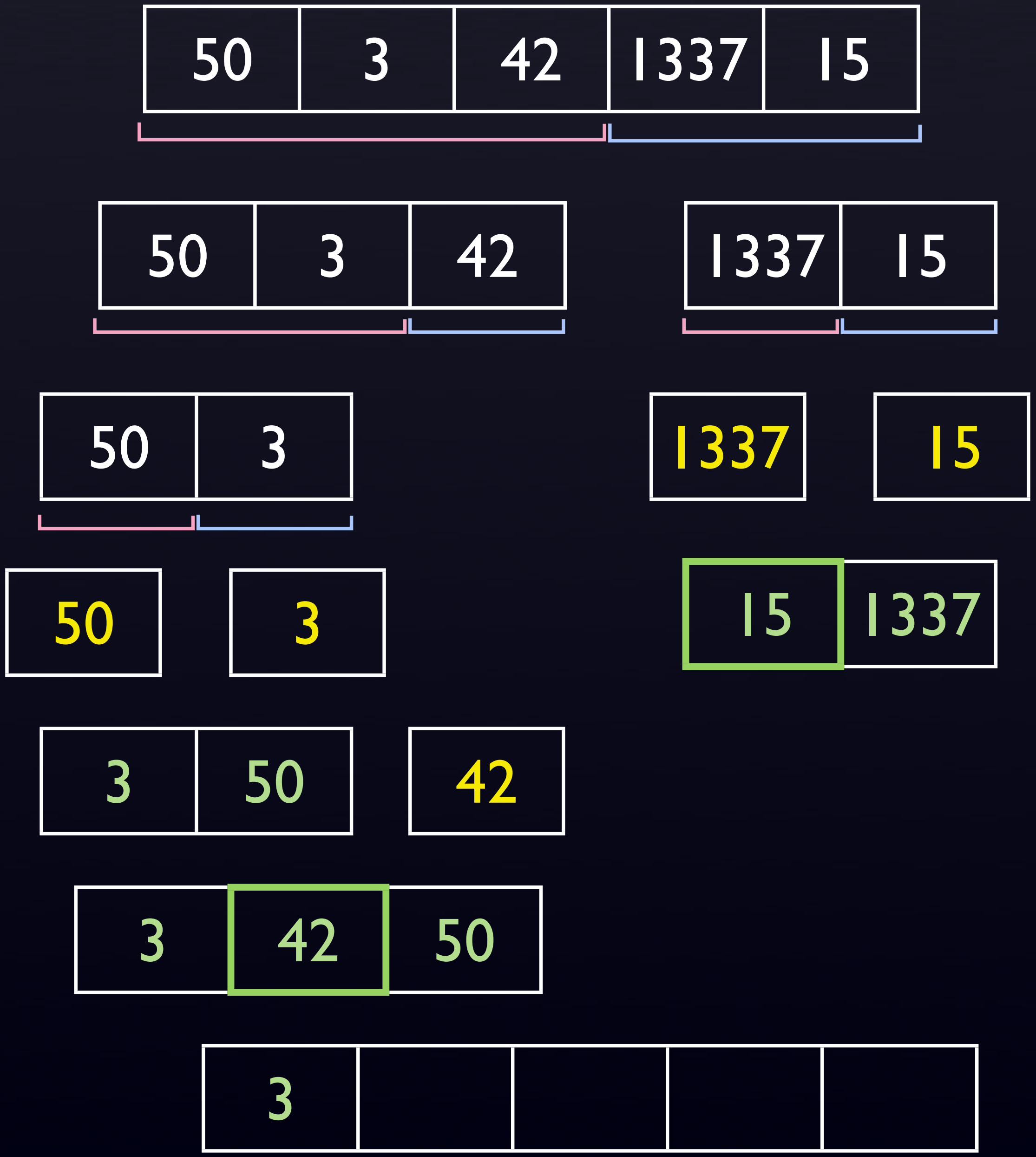
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



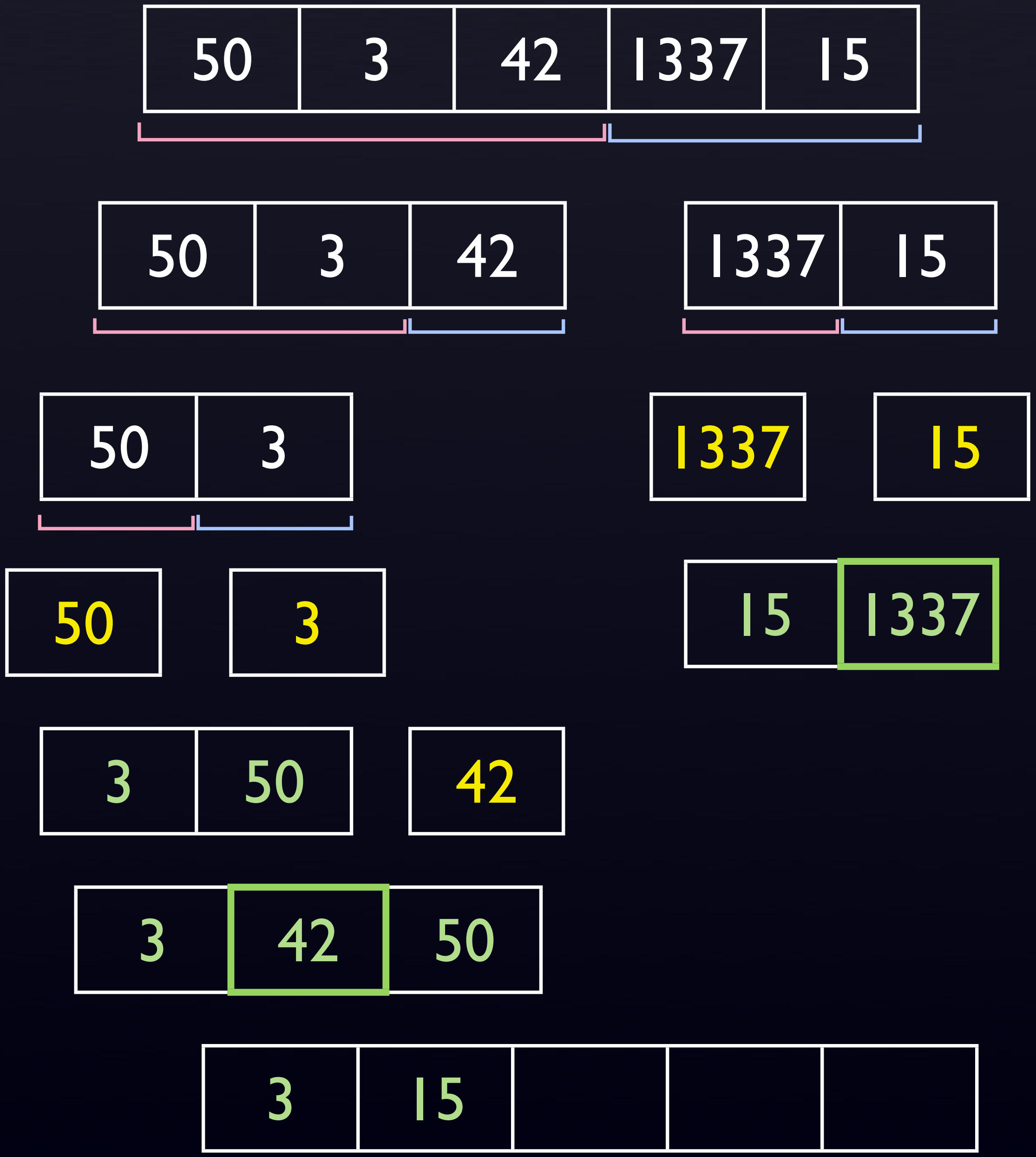
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  ───→  merge sorted halves;
```



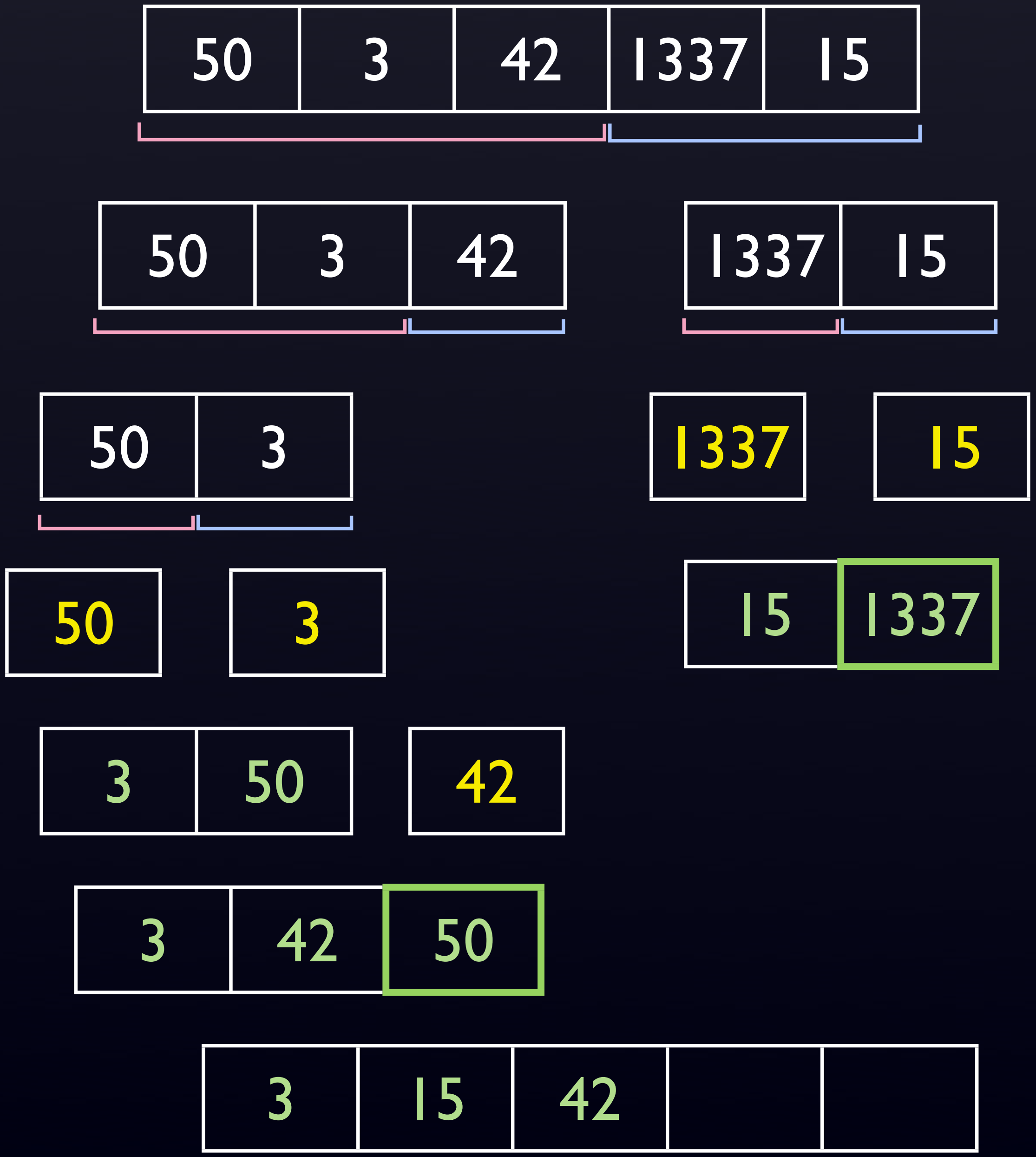
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



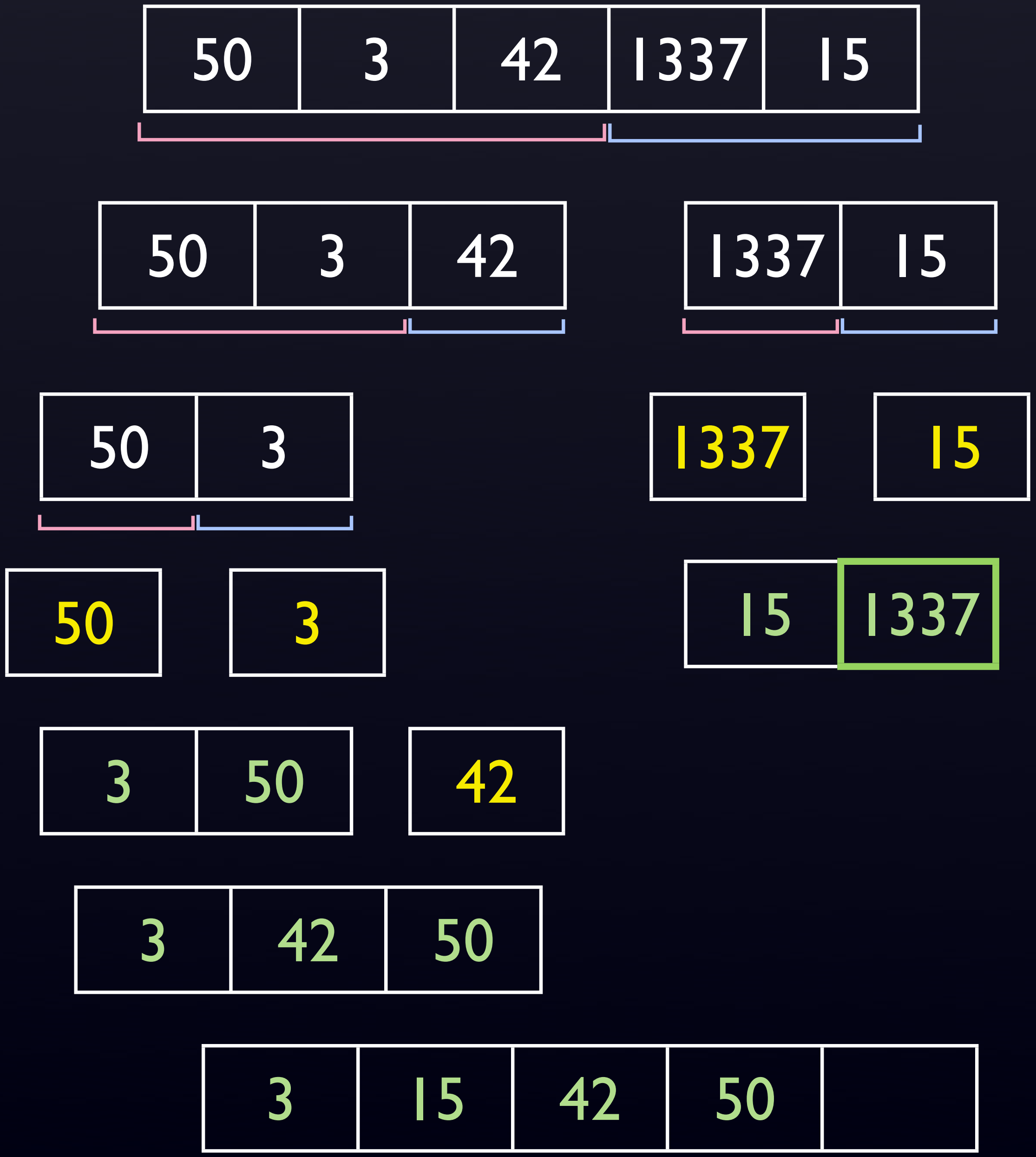
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  ⟶    merge sorted halves;
```



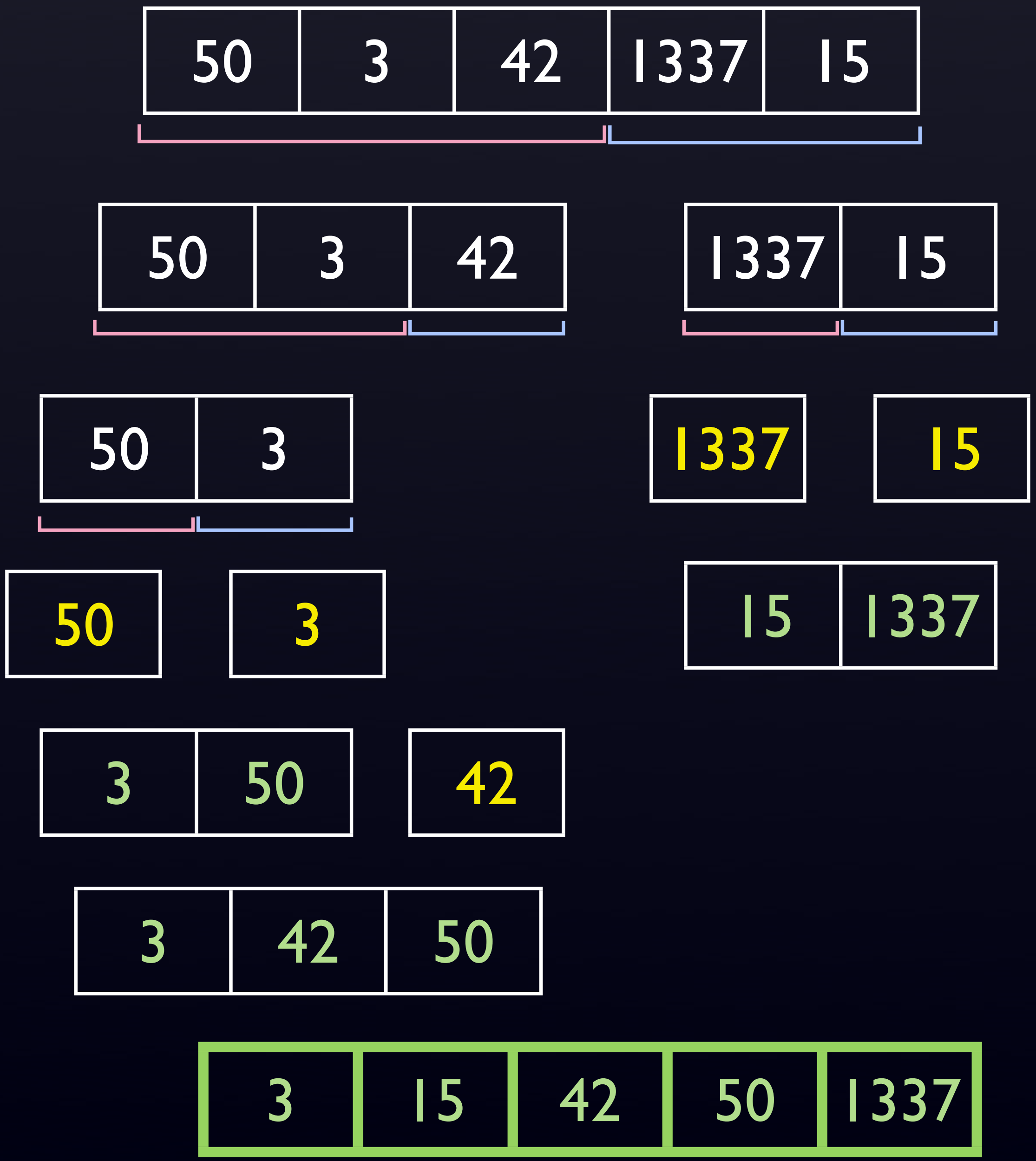
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  ⟶    merge sorted halves;
```



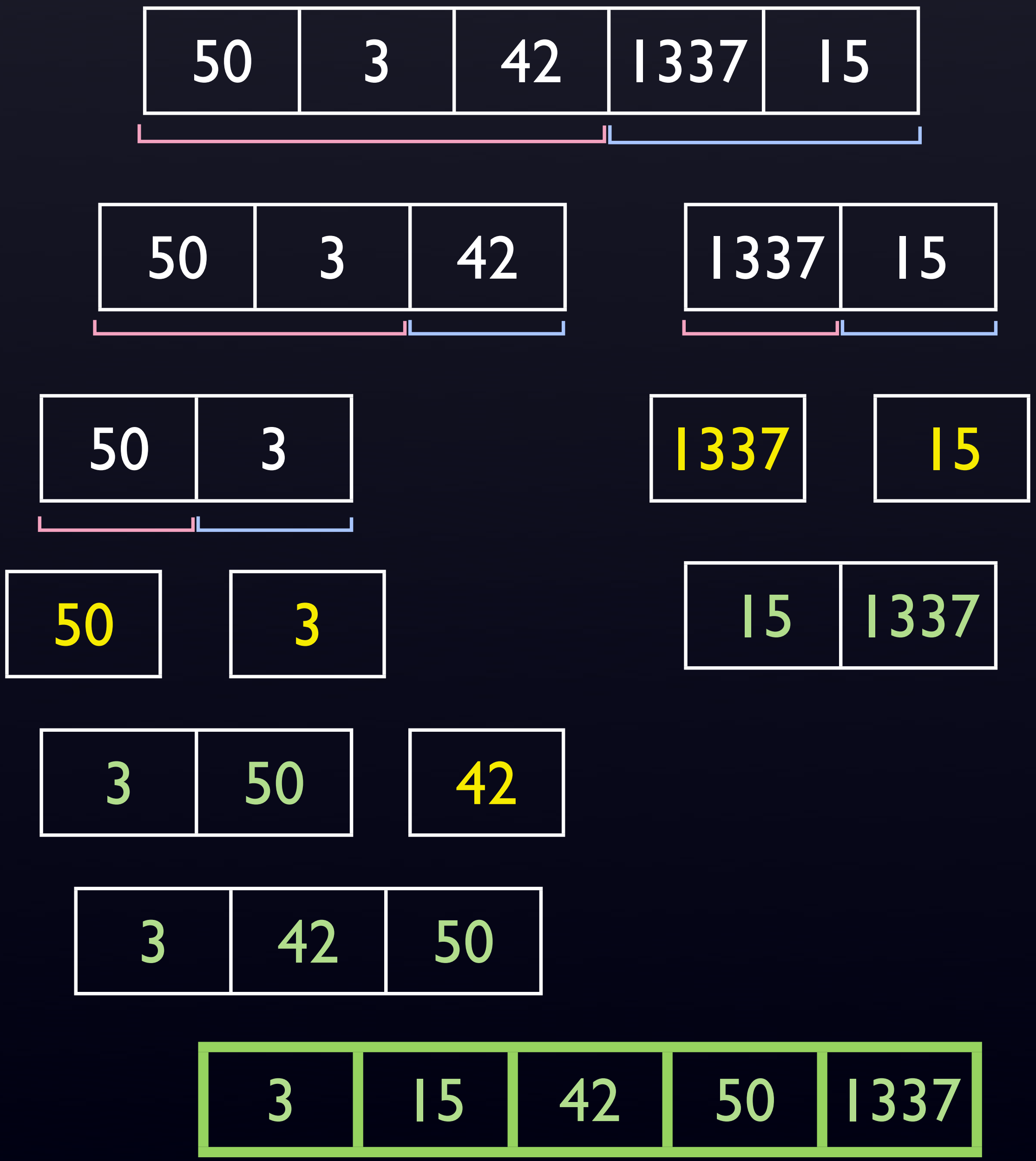
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



$O(n \log n)$

$\Omega(n \log n)$

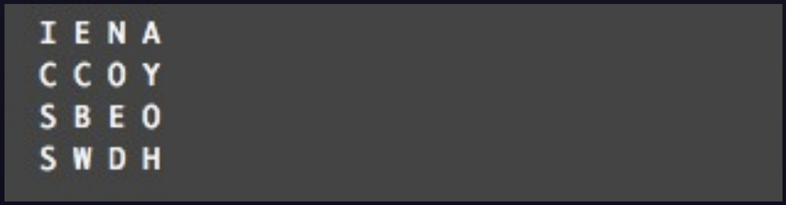
CS50: Quiz 0

Super Basic File I/O

fprintf

Prints to a file, instead of the terminal's standard output.

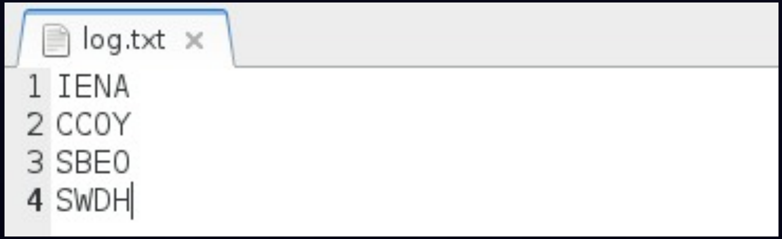
```
// print board to standard output
for (int row = 0; row < DIMENSION; row++)
{
    for (int col = 0; col < DIMENSION; col++)
        printf("%c", grid[row][col]);
    printf("\n");
}
```



```
I E N A
C C O Y
S B E O
S W D H
```

↓

```
// log board to a file
for (int row = 0; row < DIMENSION; row++)
{
    for (int col = 0; col < DIMENSION; col++)
        fprintf(log, "%c", grid[row][col]);
    fprintf(log, "\n");
}
```



```
IENA
CCOY
SBEO
SWDH
```

fopen

Hmm, but wait, where did that file come from?

```
// open log
FILE* log = fopen("log.txt", "a");

// check if successfully opened
if (log == NULL)
{
    printf("Could not open log.\n");
    return 1;
}
```

fopen's arguments

Overwrite a file, append to it, or open it read-only.

```
// open log
FILE* log = fopen("log.txt", "a");

// check if successfully opened
if (log == NULL)
{
    printf("Could not open log.\n");
    return 1;
}
```

w = overwrite existing file completely
a = append to the end of existing file
r = open the file, read-only

fclose

Once we're done with the file, remember to close it!

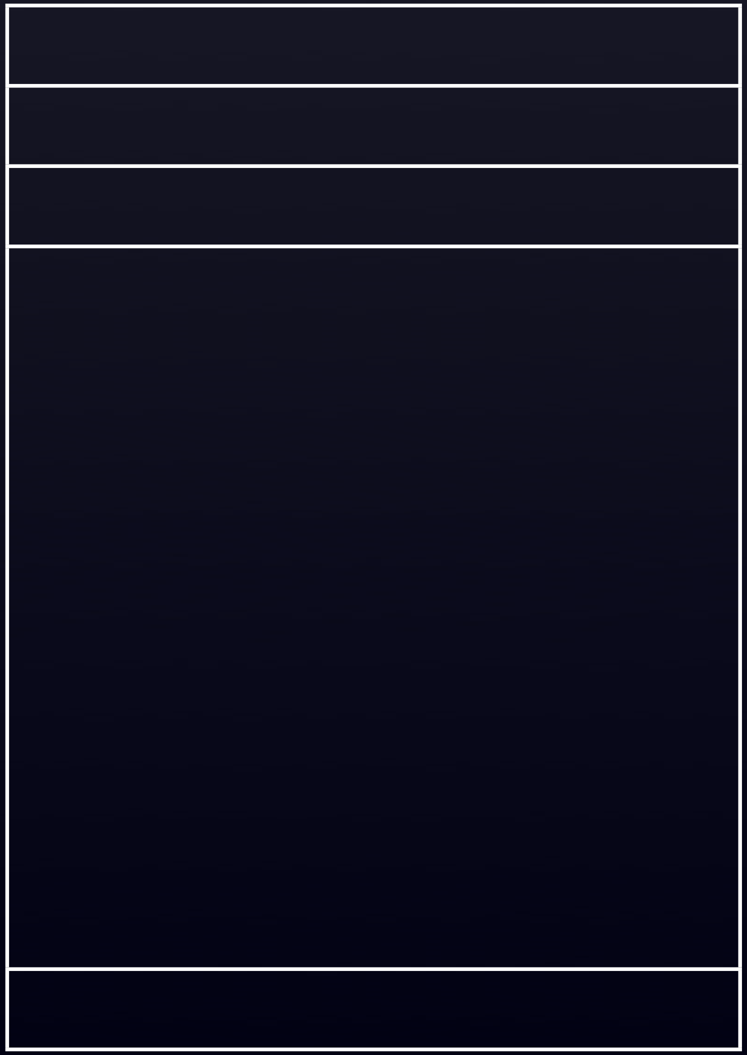
```
// close our log file
fclose(log);
```

CS50: Quiz 0

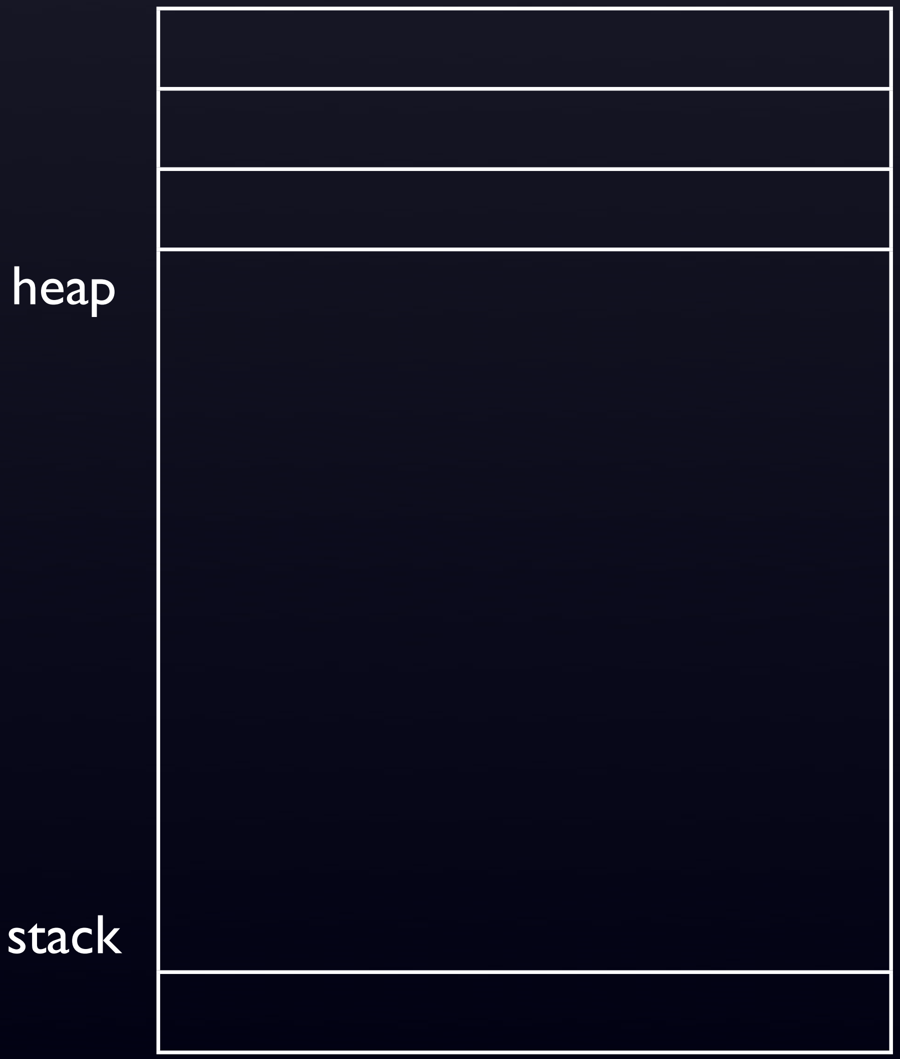
Memory, Stack, Heap

Memory Layout

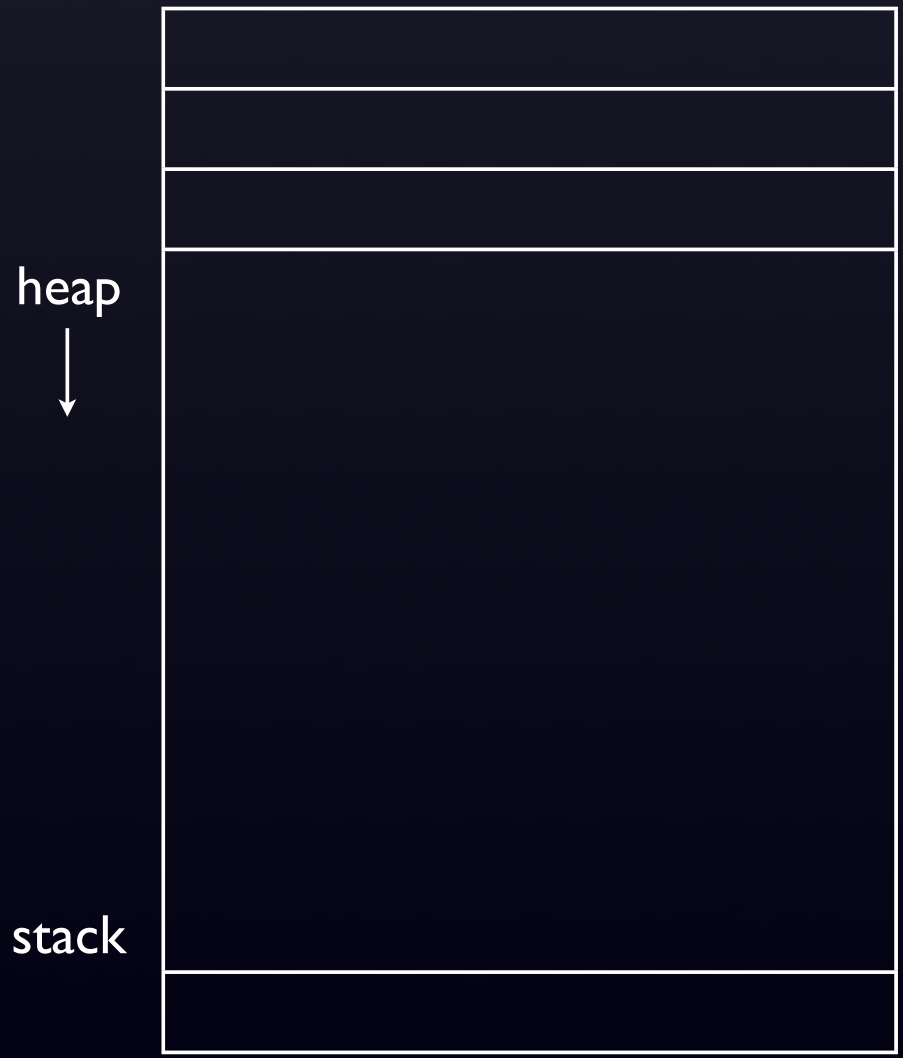
Memory Layout



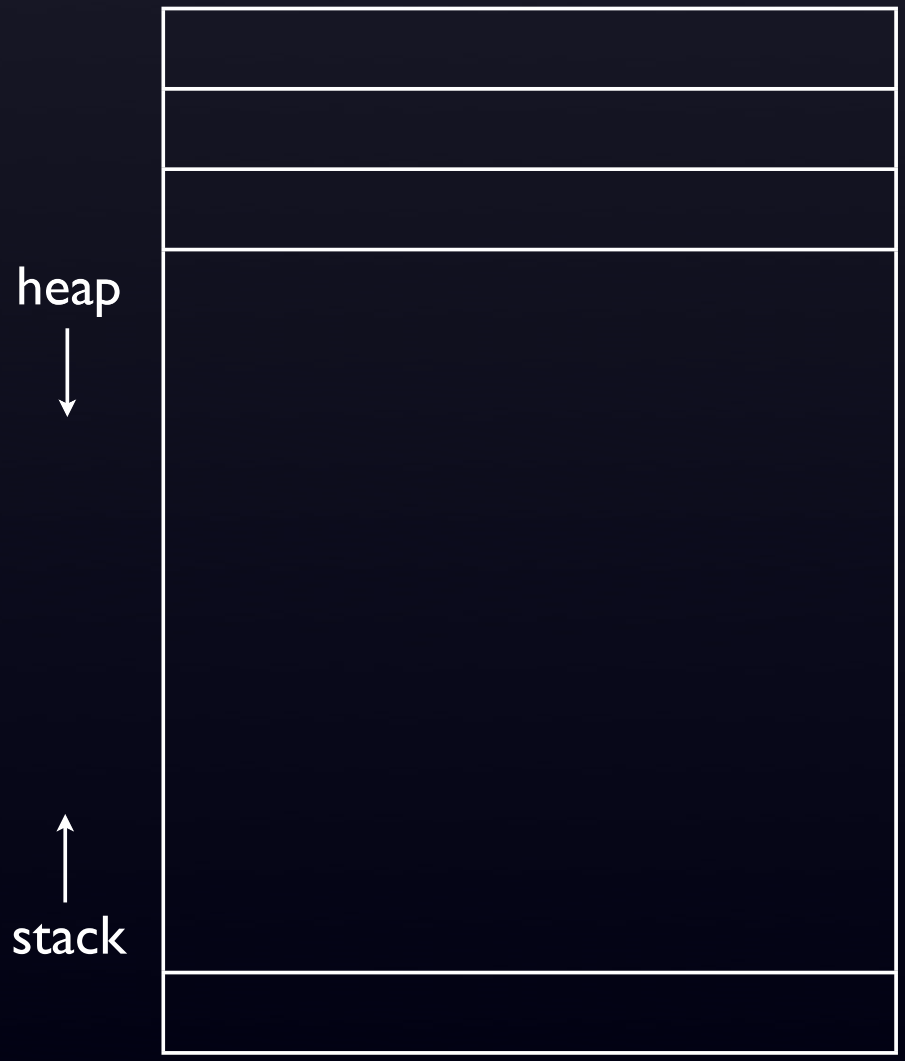
Memory Layout



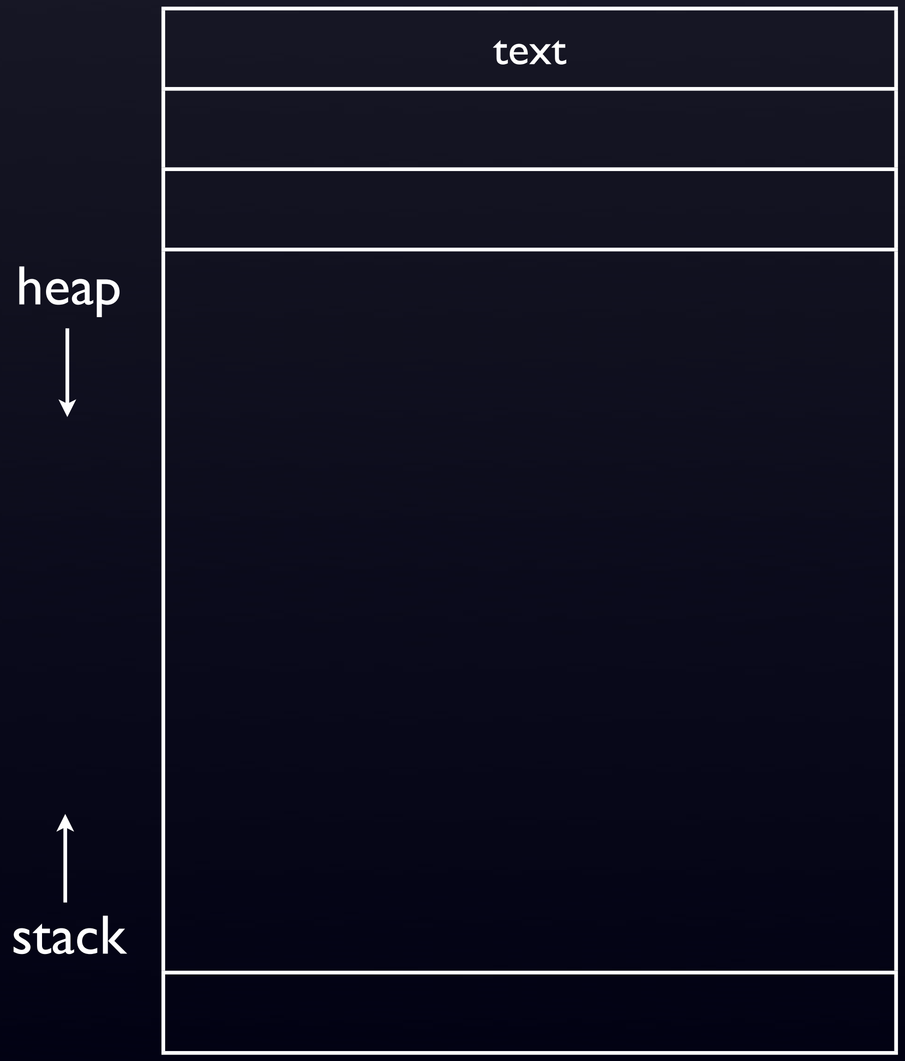
Memory Layout



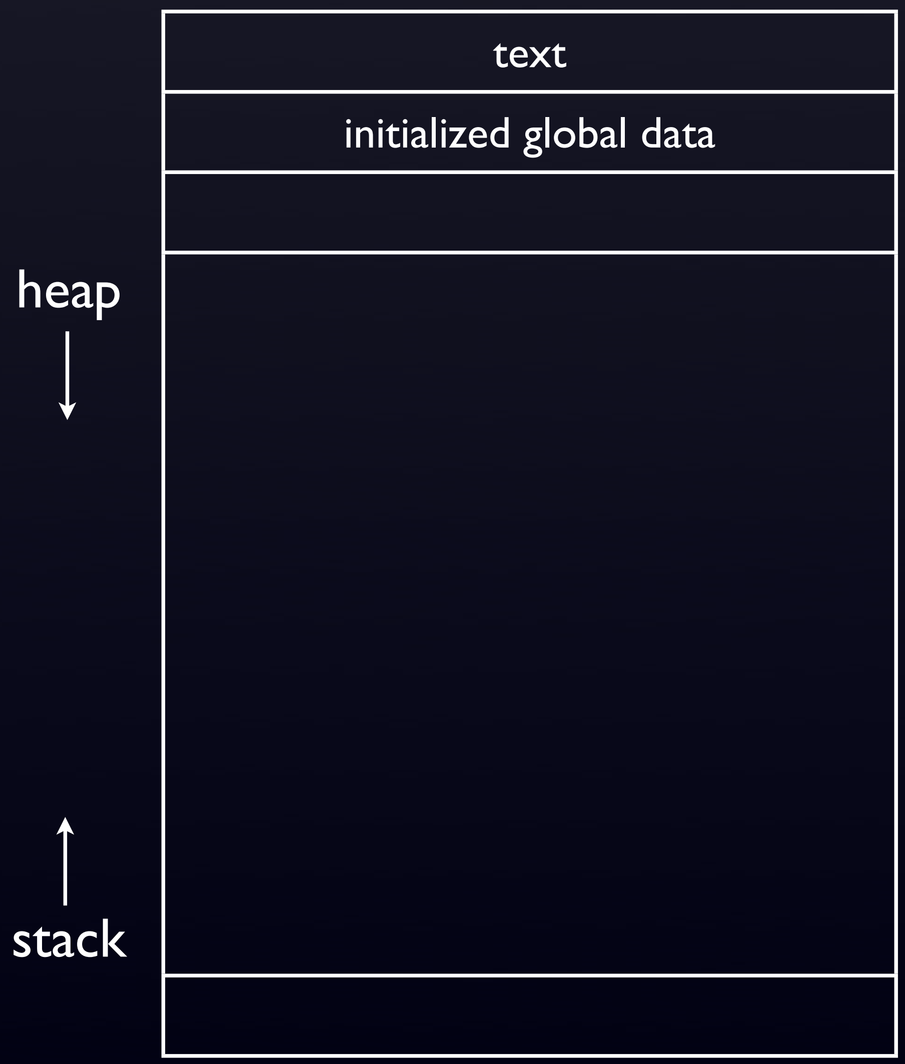
Memory Layout



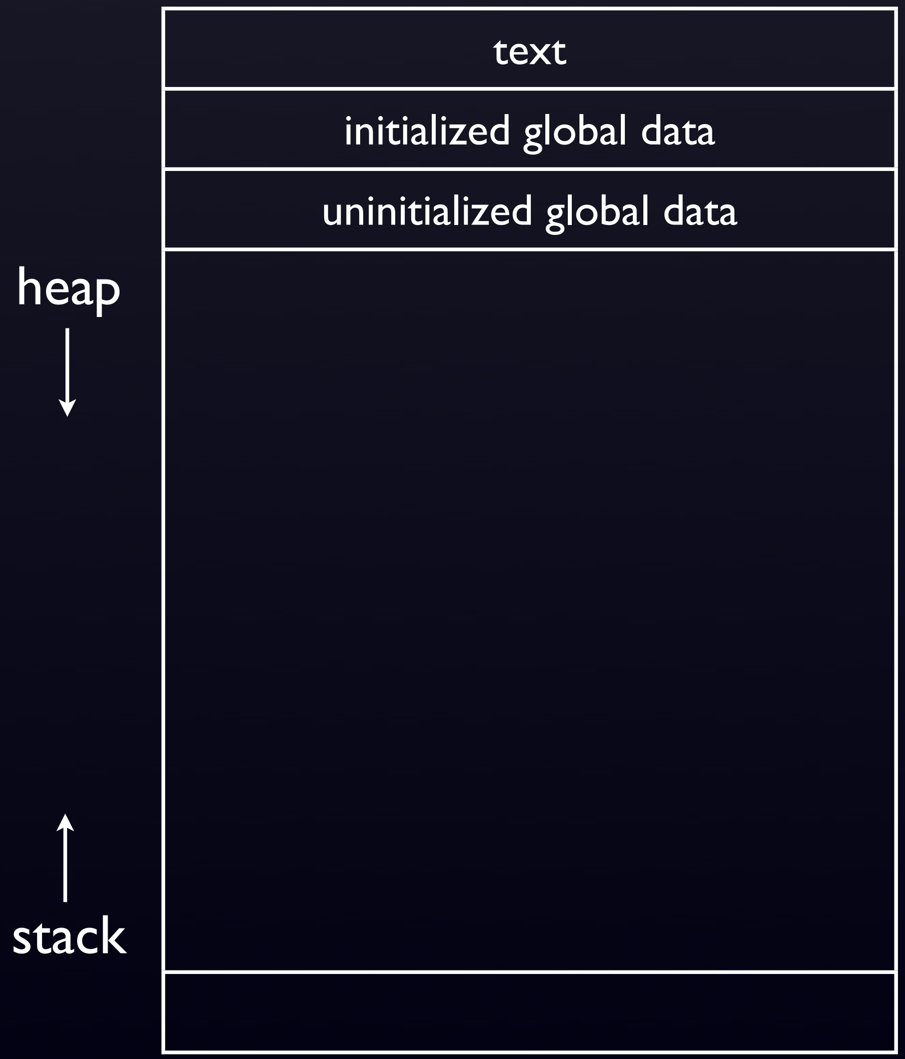
Memory Layout



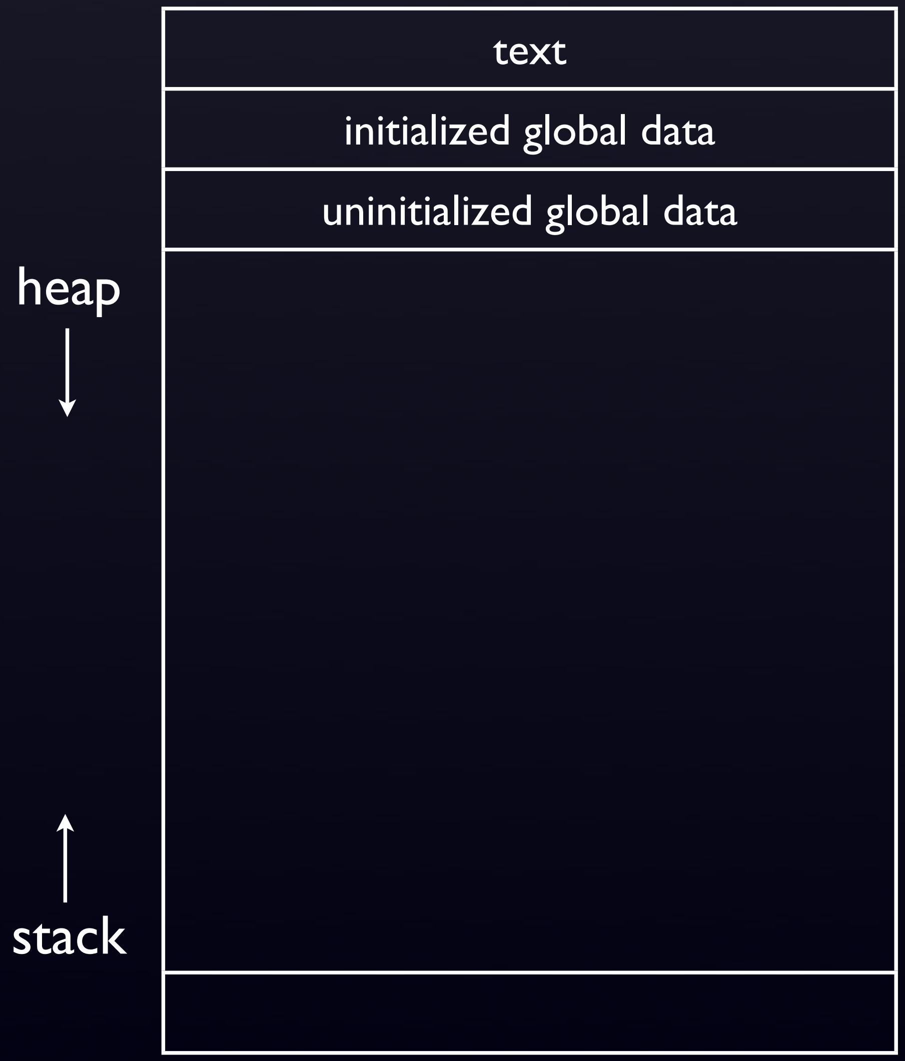
Memory Layout



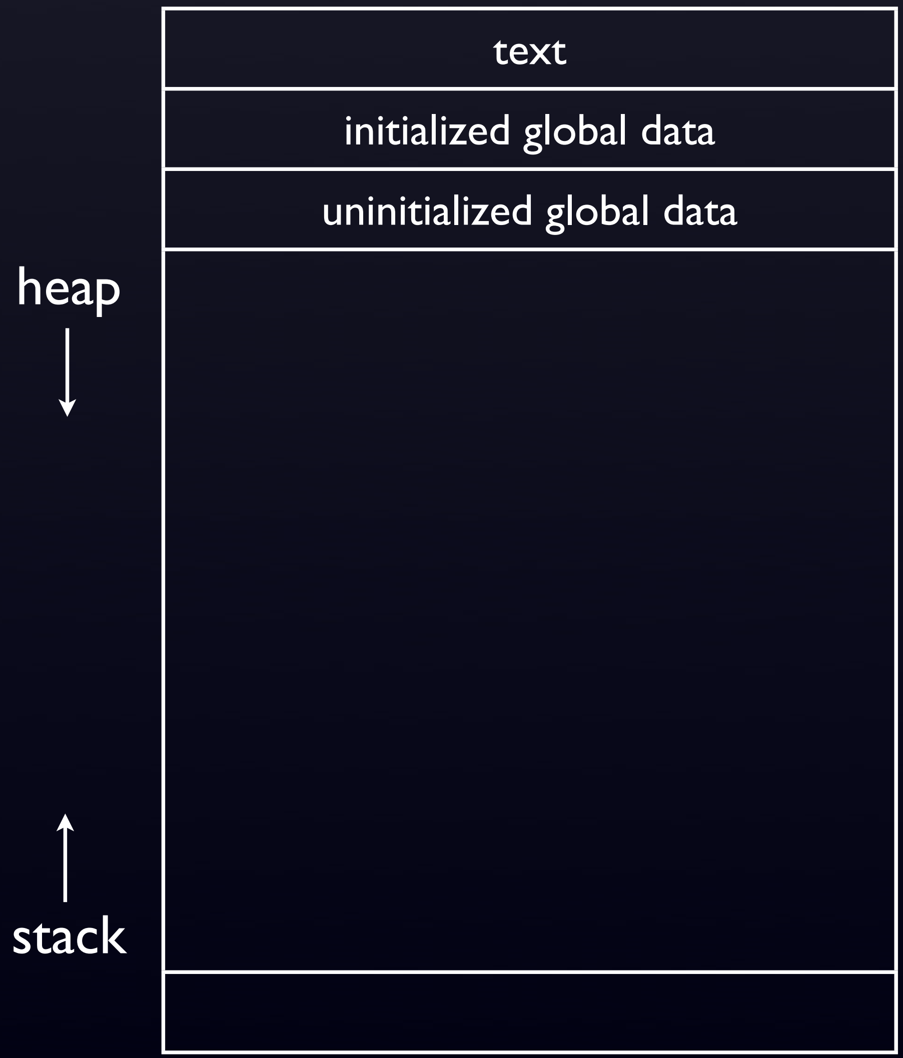
Memory Layout



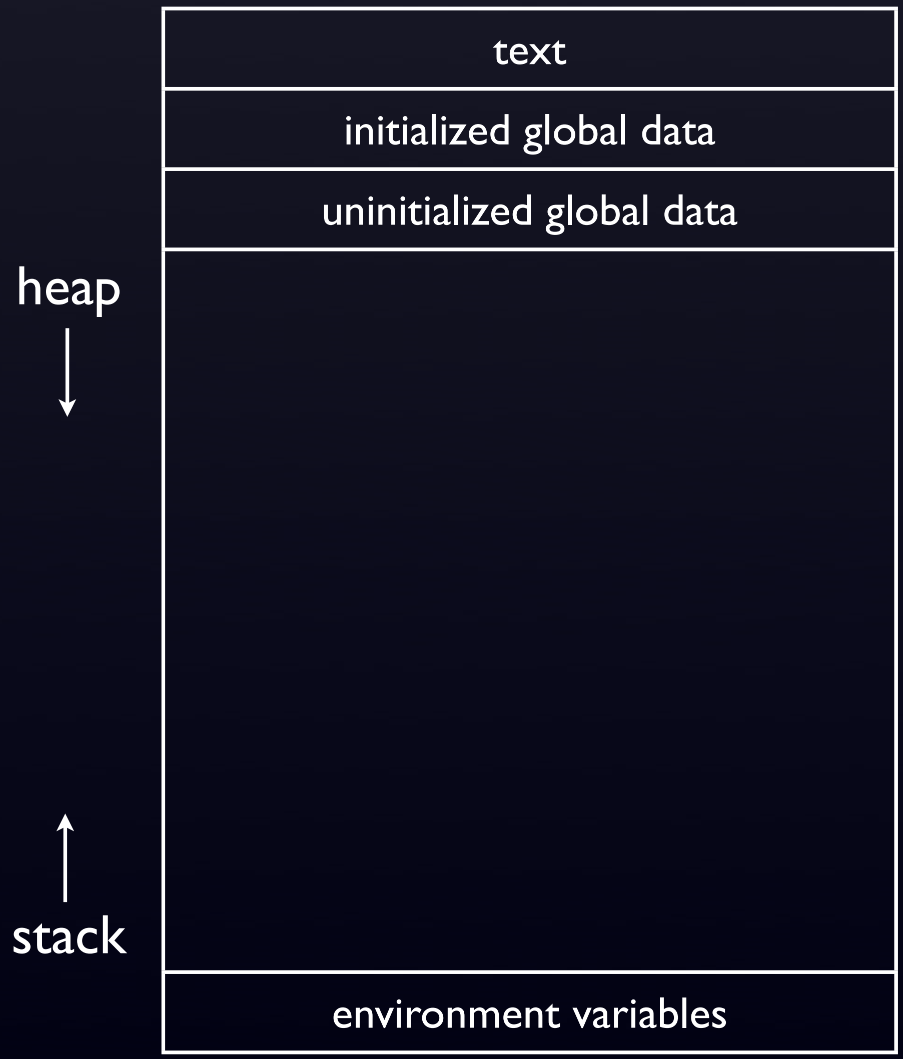
Memory Layout



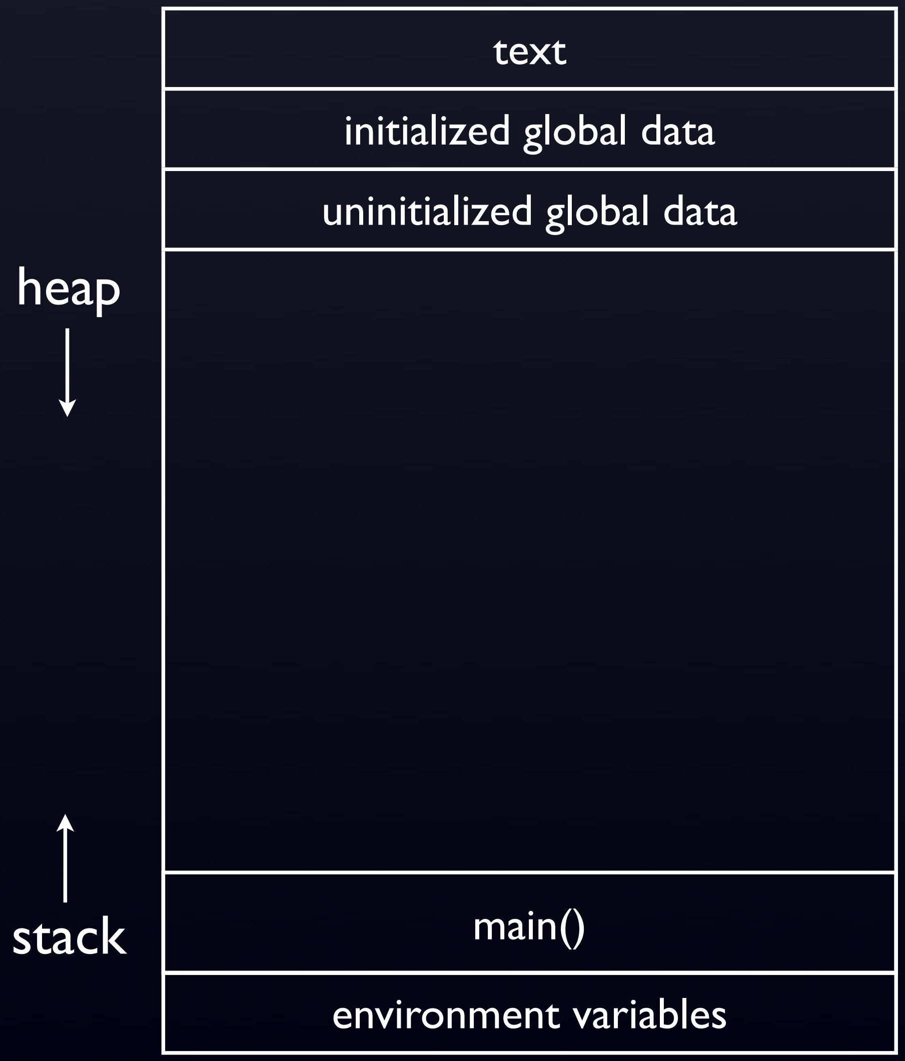
Memory Layout



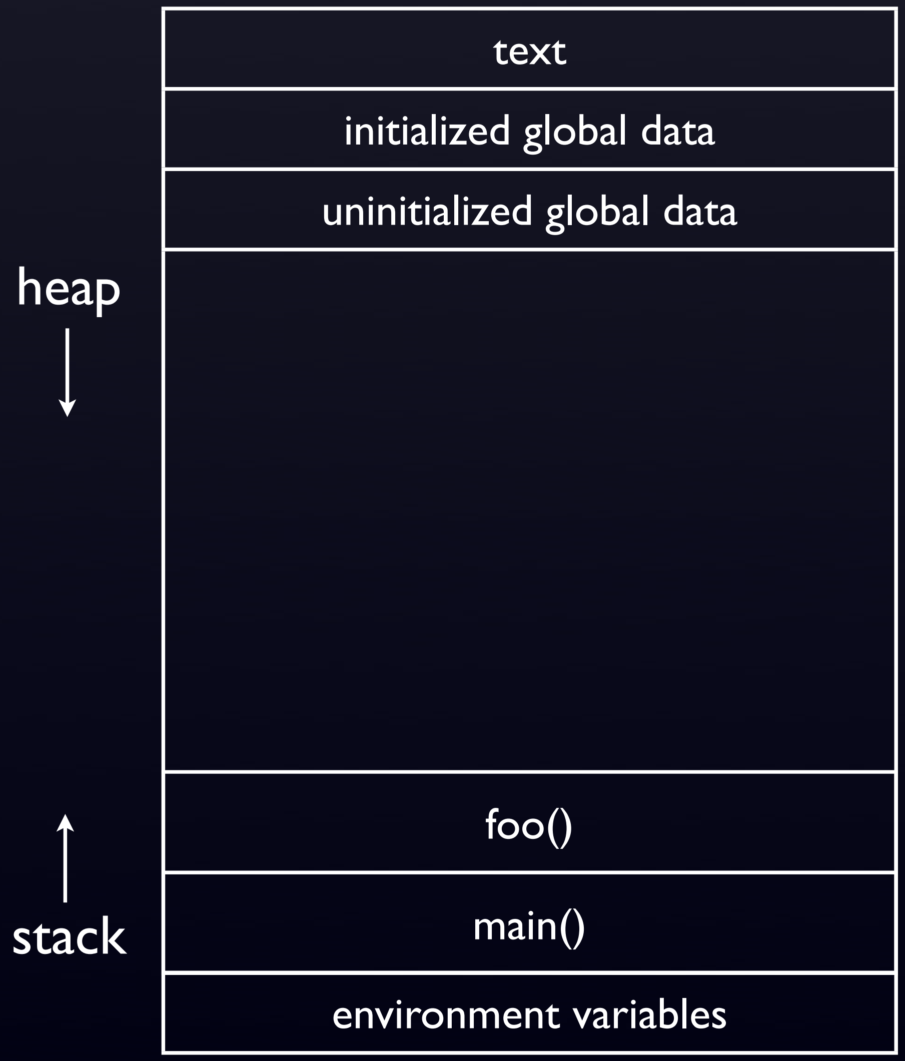
Memory Layout



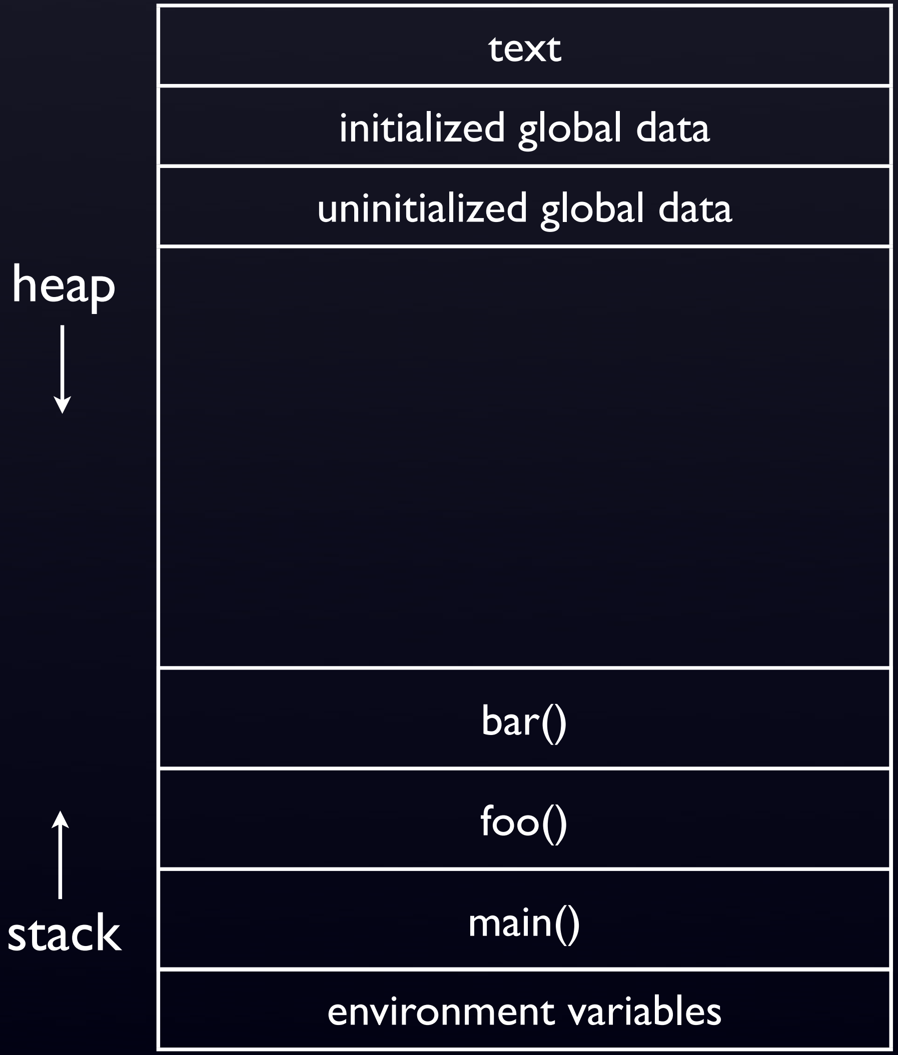
Memory Layout



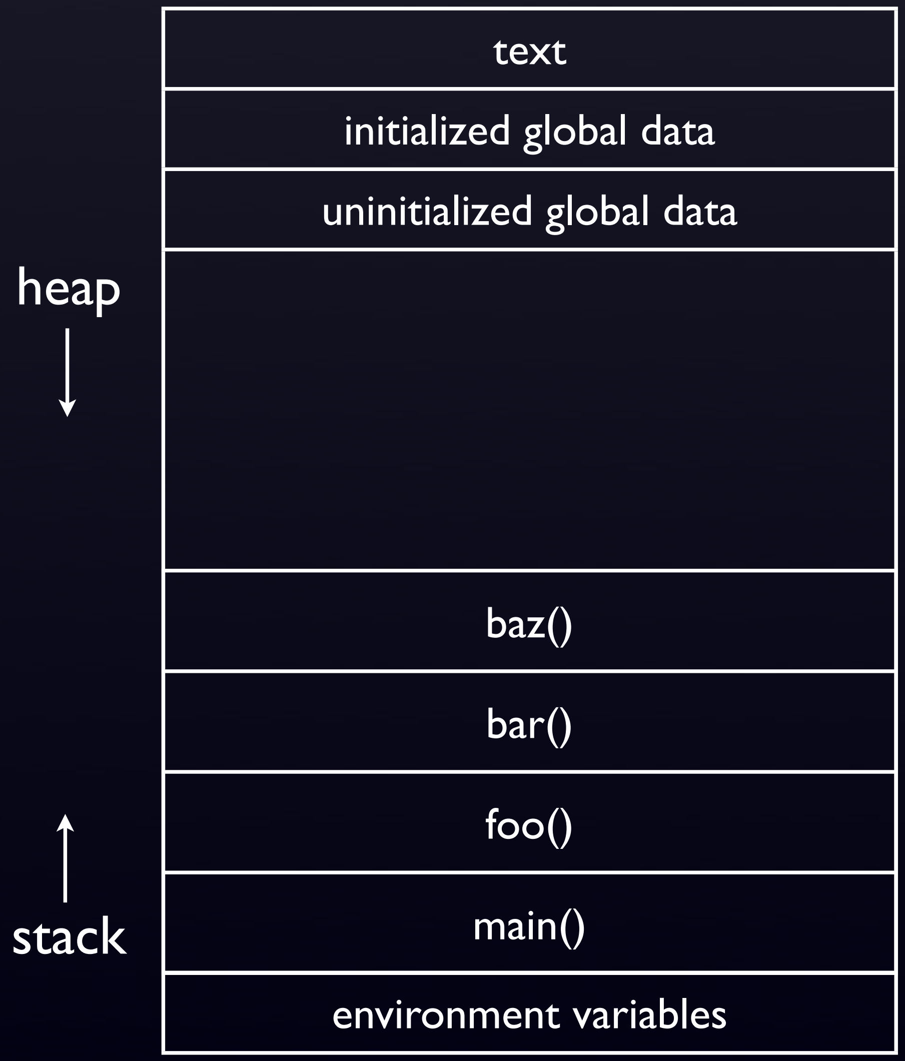
Memory Layout



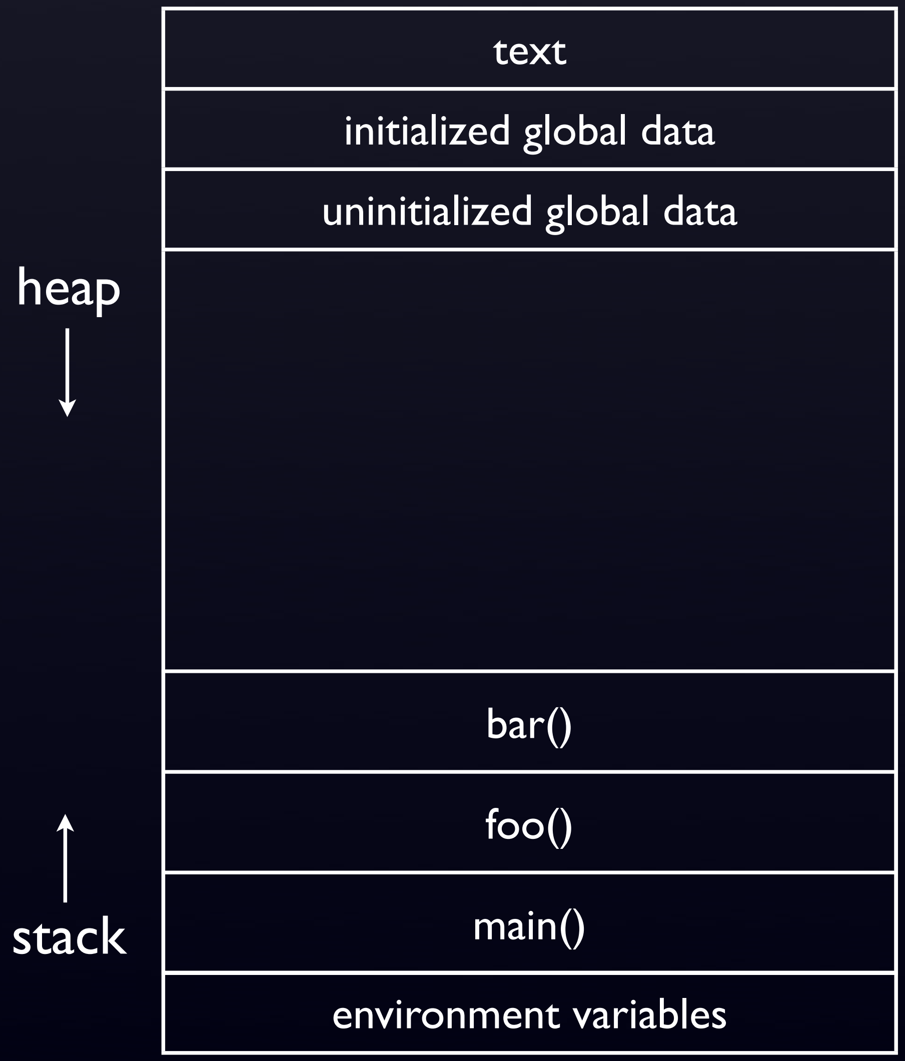
Memory Layout



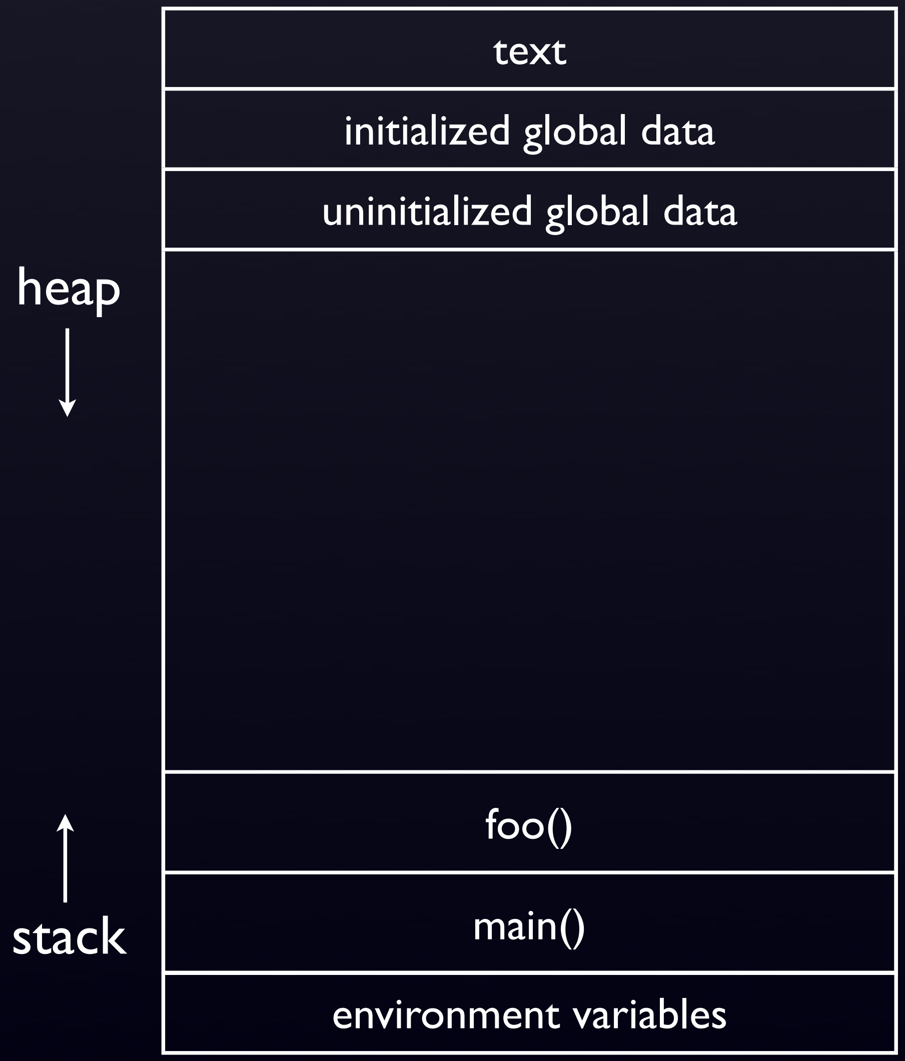
Memory Layout



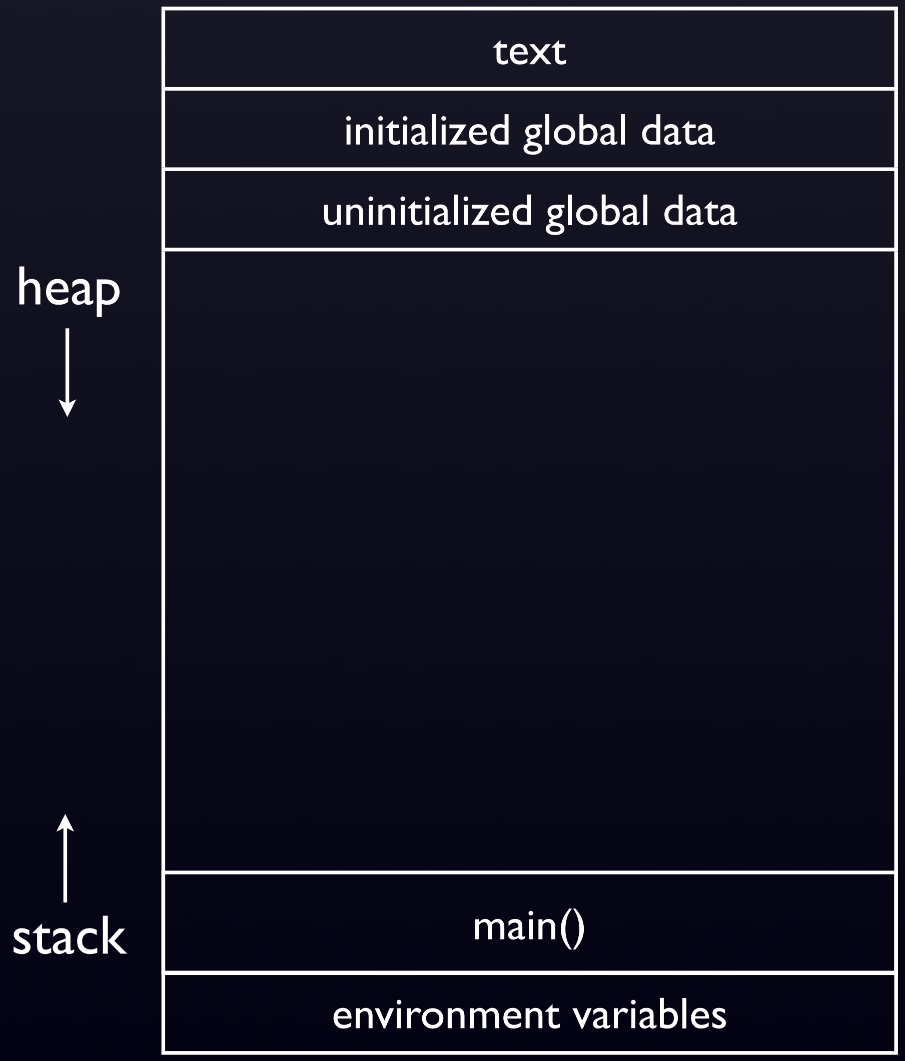
Memory Layout



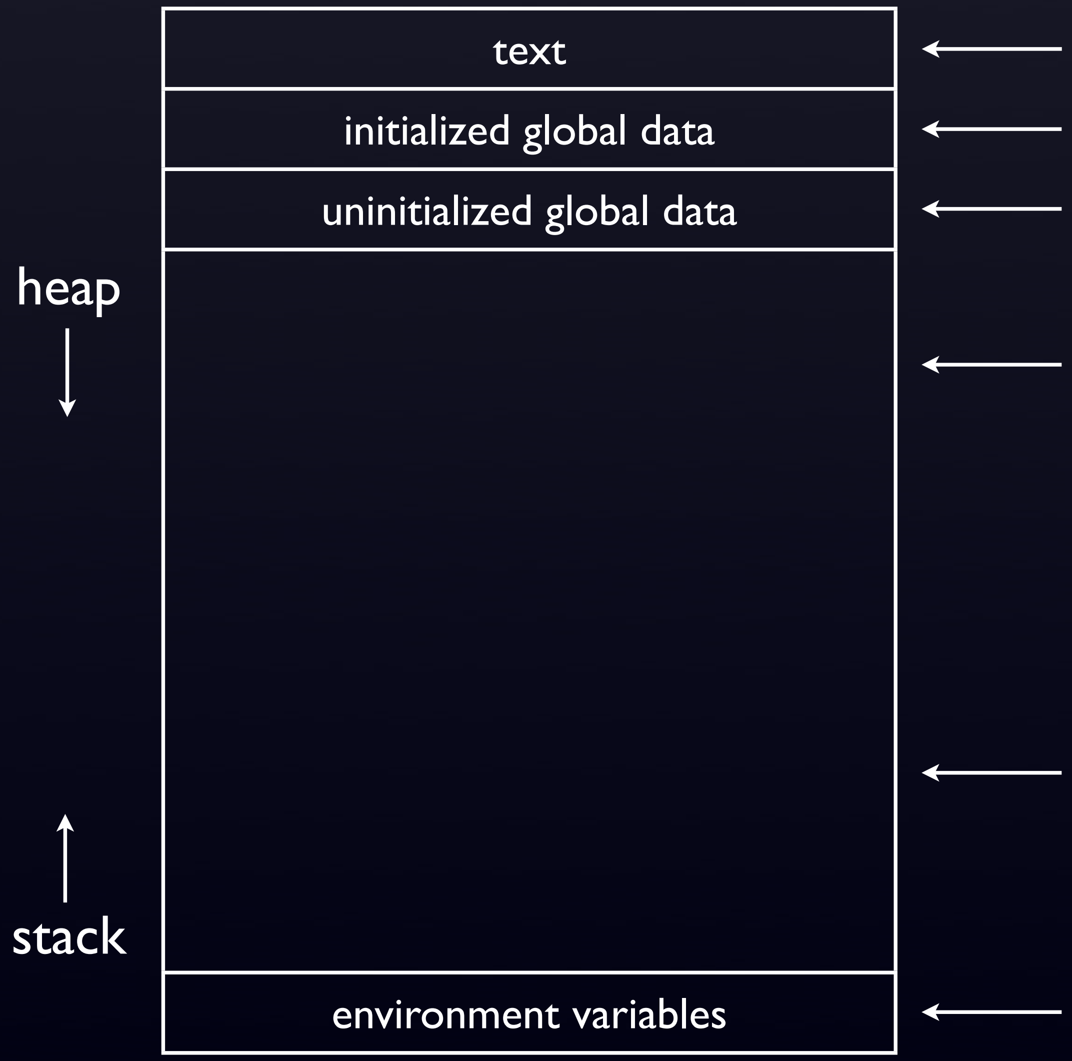
Memory Layout



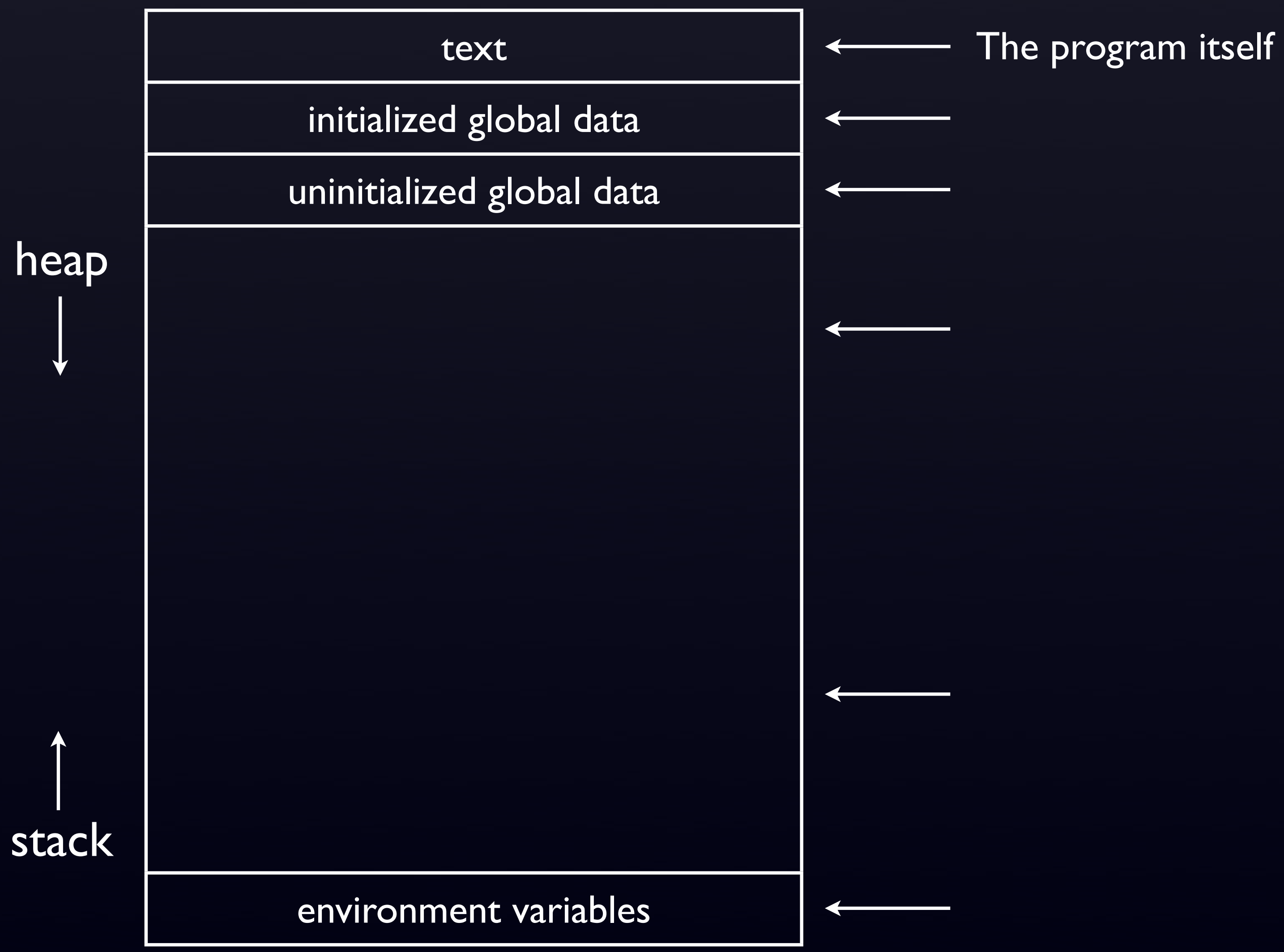
Memory Layout



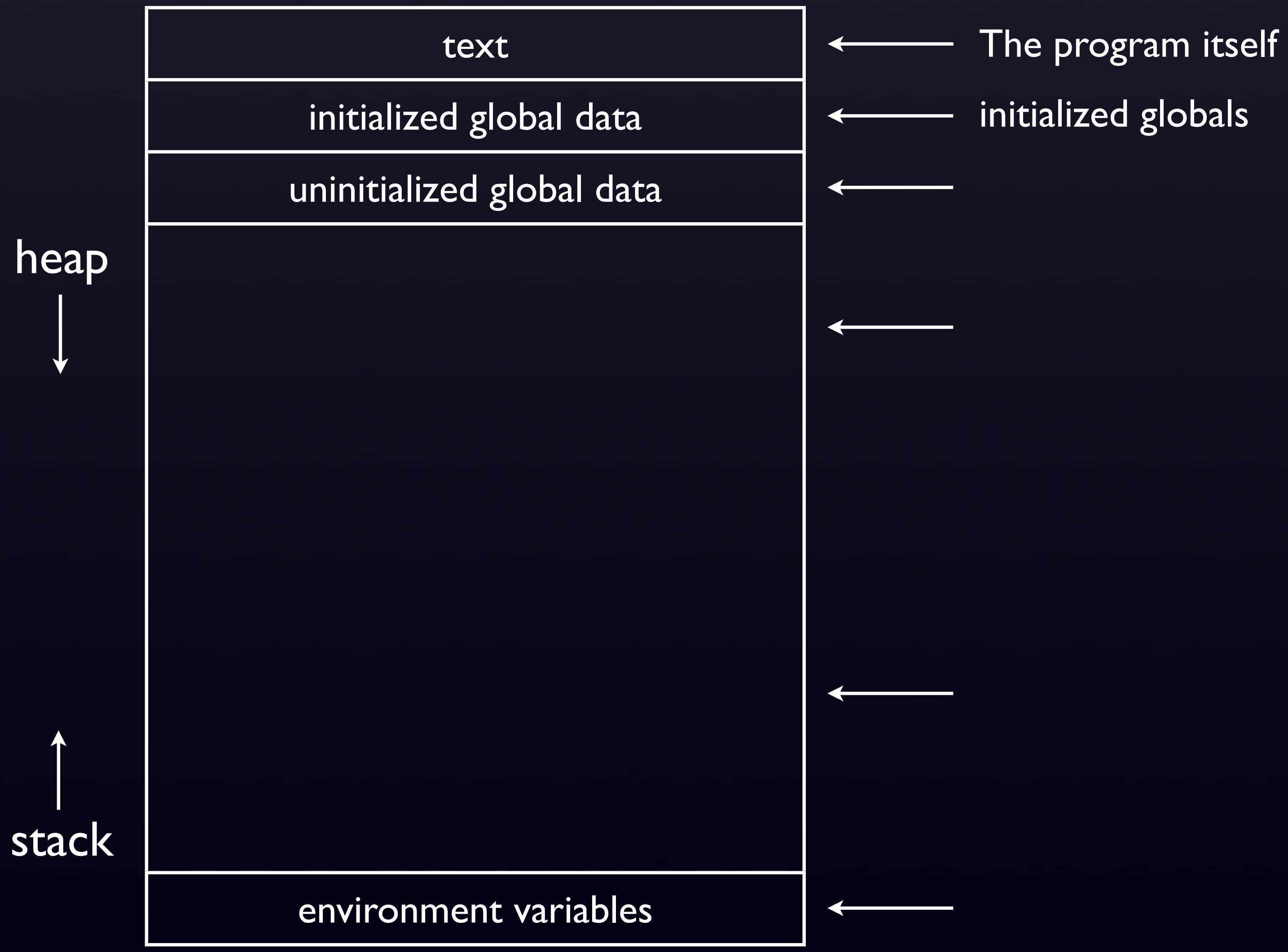
Memory Layout



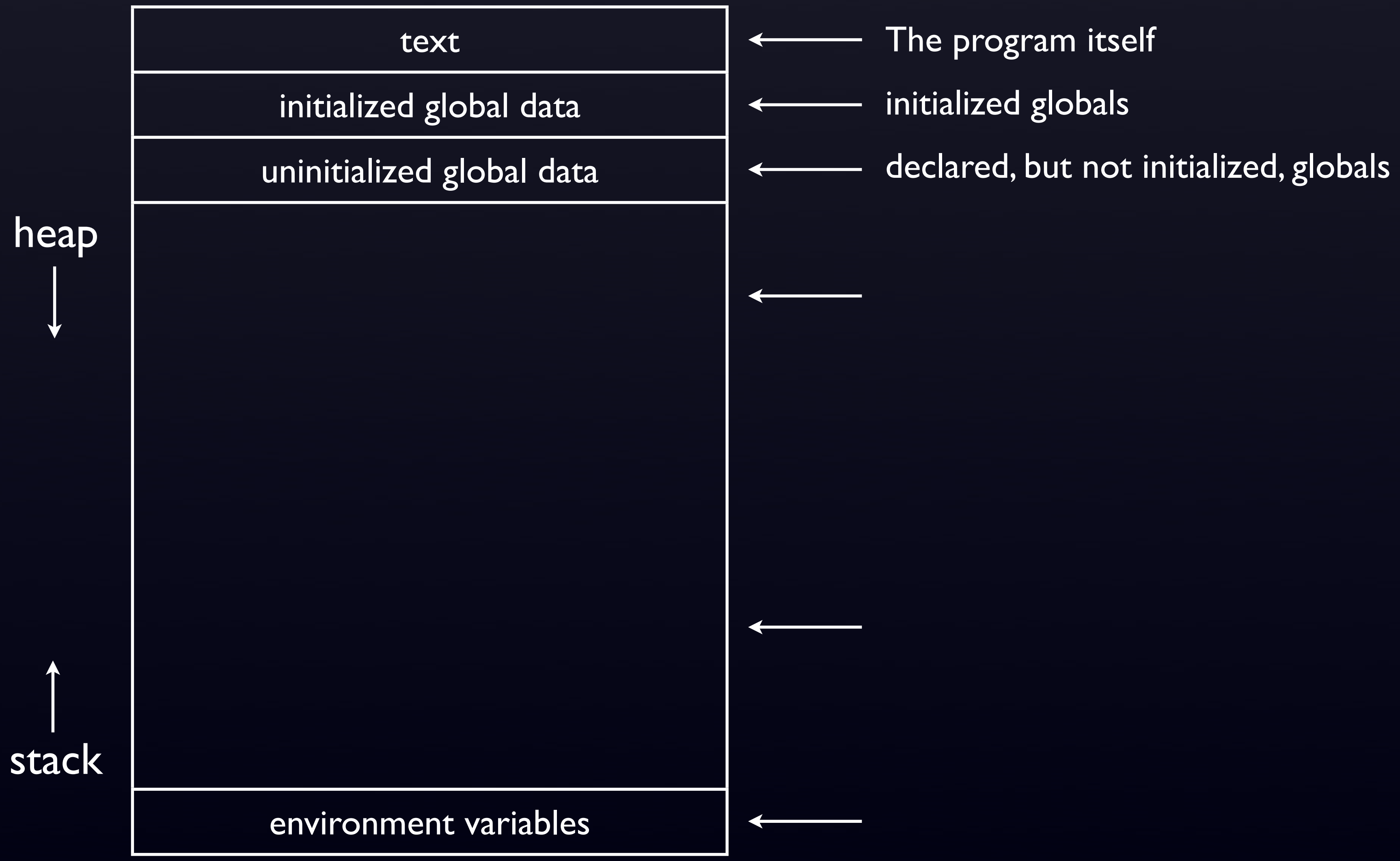
Memory Layout



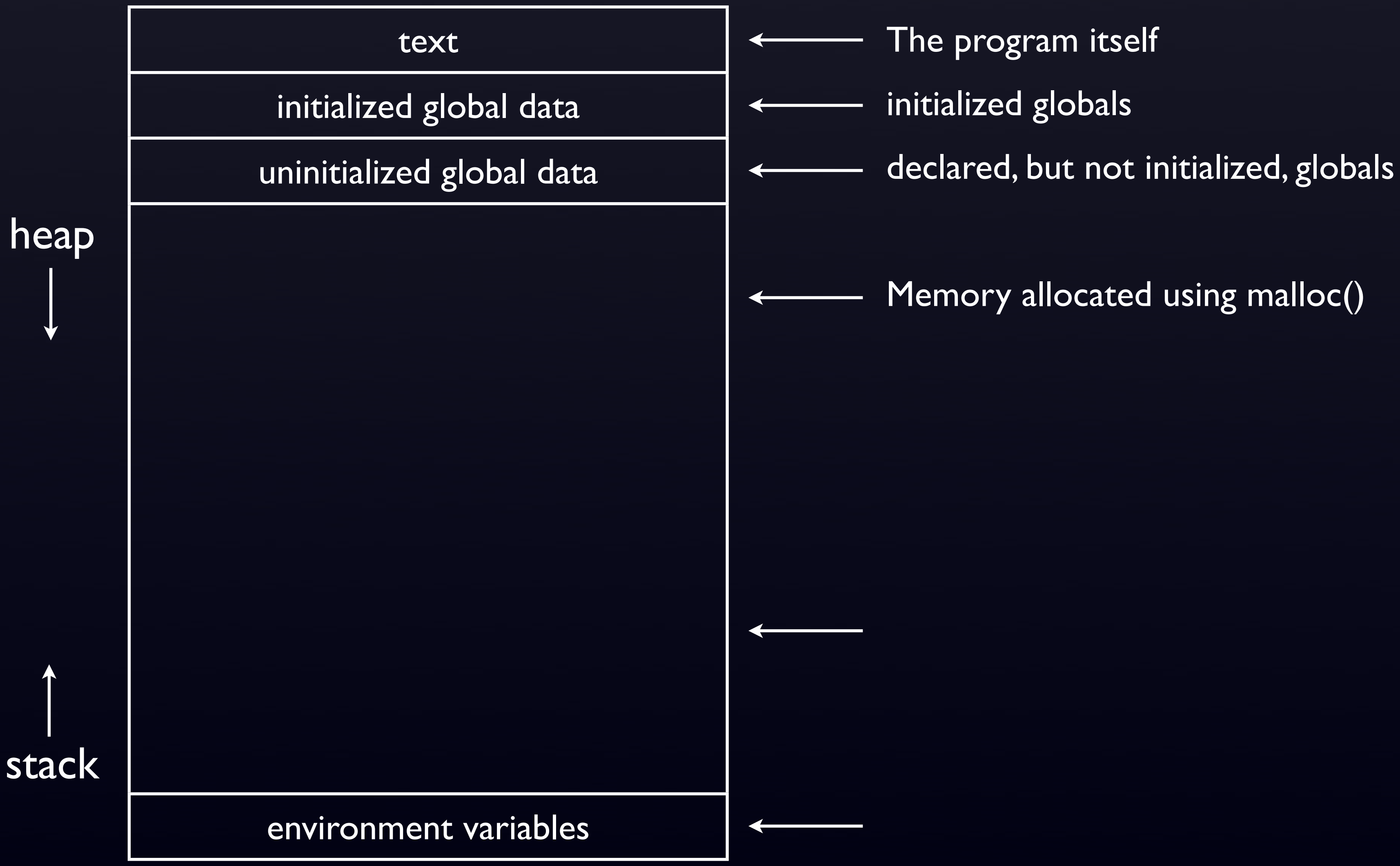
Memory Layout



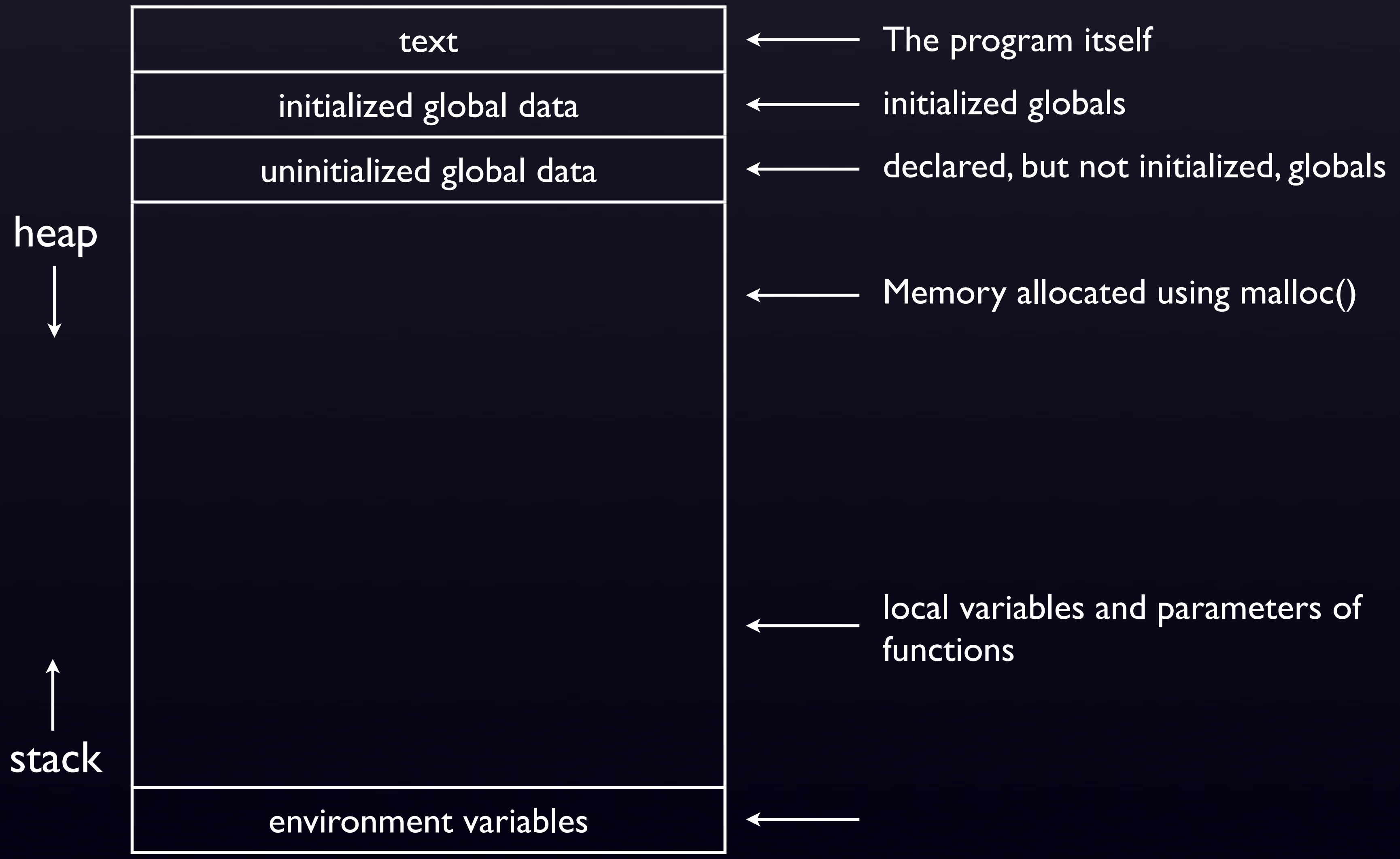
Memory Layout



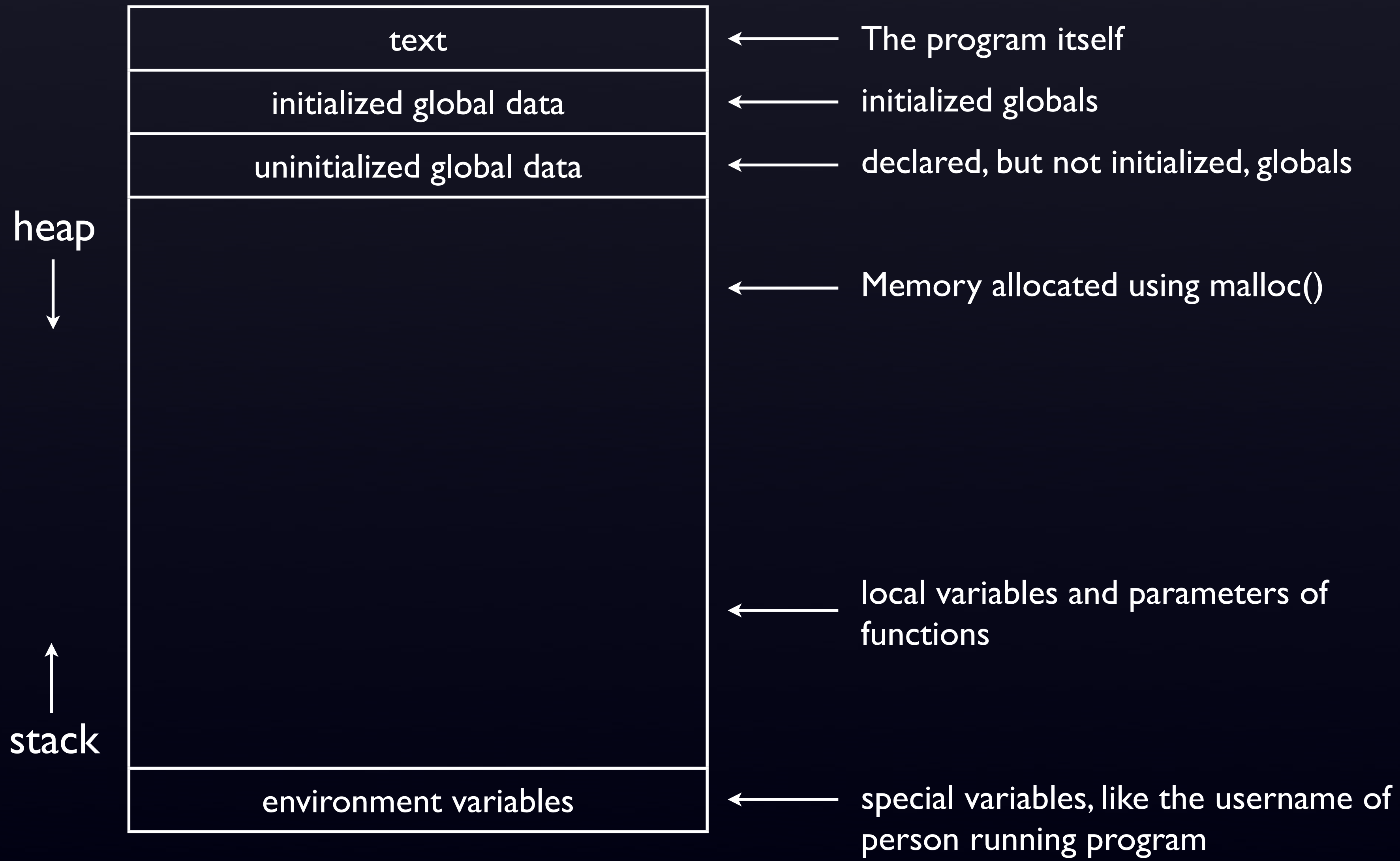
Memory Layout


text
initialized global data
uninitialized global data
heap
stack
environment variables
The program itself
initialized globals
declared, but not initialized, globals
Memory allocated using malloc()

Memory Layout


text
initialized global data
uninitialized global data
heap
stack
environment variables
The program itself
initialized globals
declared, but not initialized, globals
Memory allocated using malloc()
local variables and parameters of functions

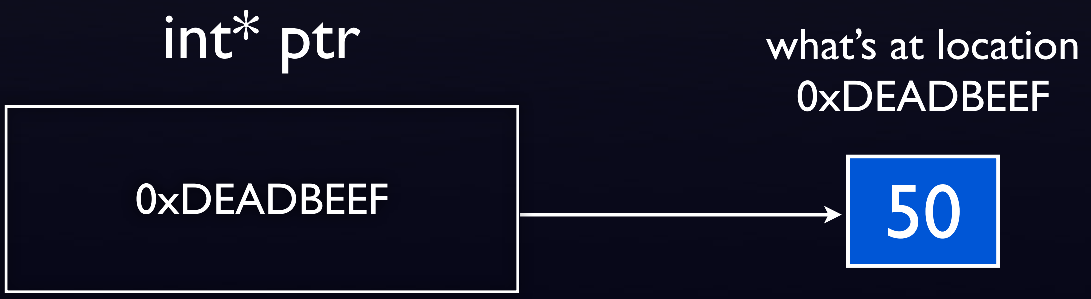
Memory Layout


text
initialized global data
uninitialized global data
heap
stack
environment variables
The program itself
initialized globals
declared, but not initialized, globals
Memory allocated using malloc()
local variables and parameters of functions
special variables, like the username of person running program

What are pointers?

They are data types that refer to another location in memory, where other data is stored.

In this case, ptr “references” 50.



Just fyi, on 32-bit systems, pointers take up 32 bits, or 4 bytes, of space, just like an int does.

Dynamic Memory Allocation

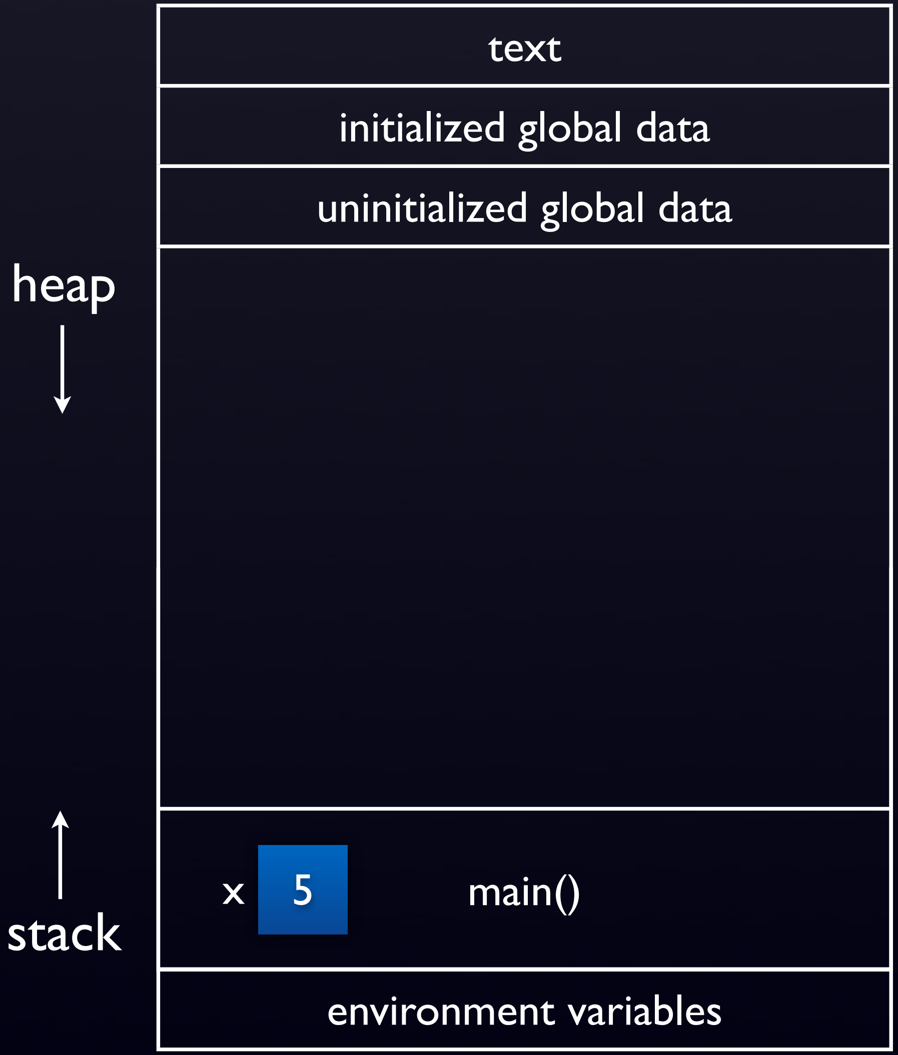
Recall, local variables are allocated on the stack, and we can't access them outside the scope of the functions or loops they belong to.

So, what dynamic memory allocation lets us do is hold on to data for the entire duration of the program.

This is done by:

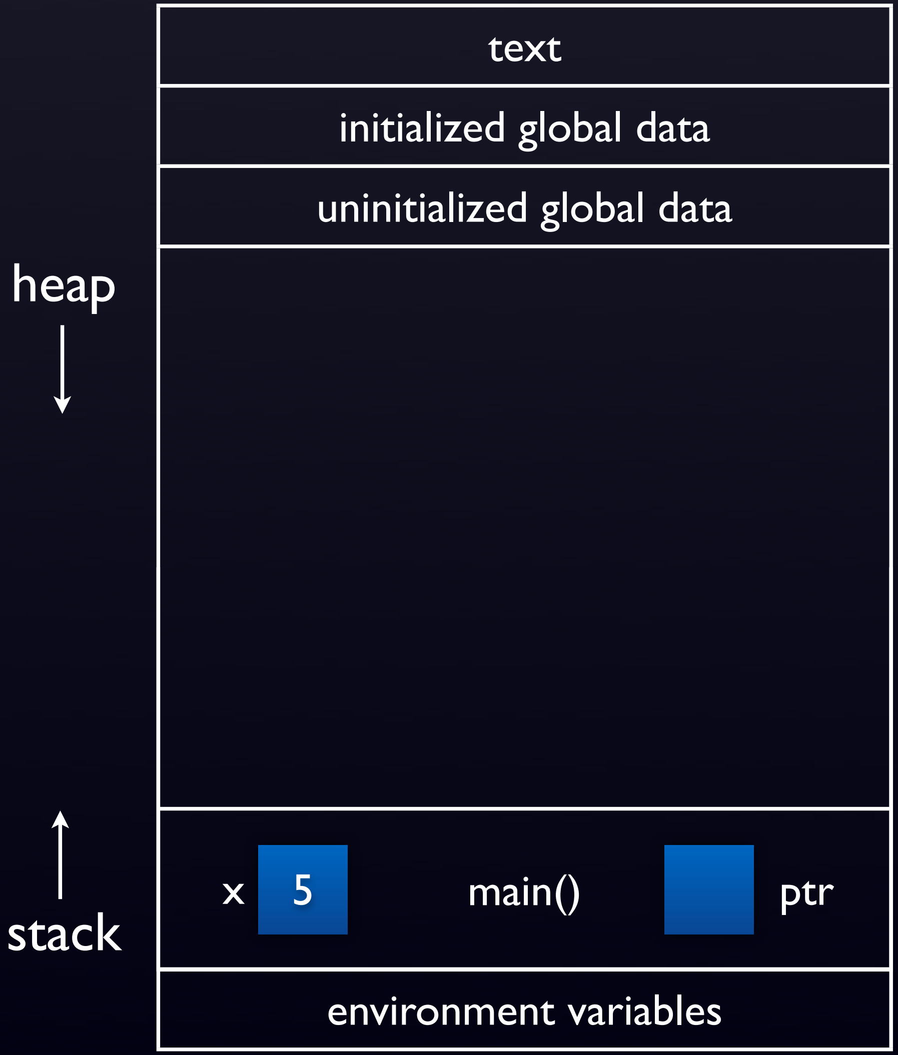
1) Allocating data in a permanent space on the heap.
2) Keeping track of a pointer to that location in memory.

malloc()



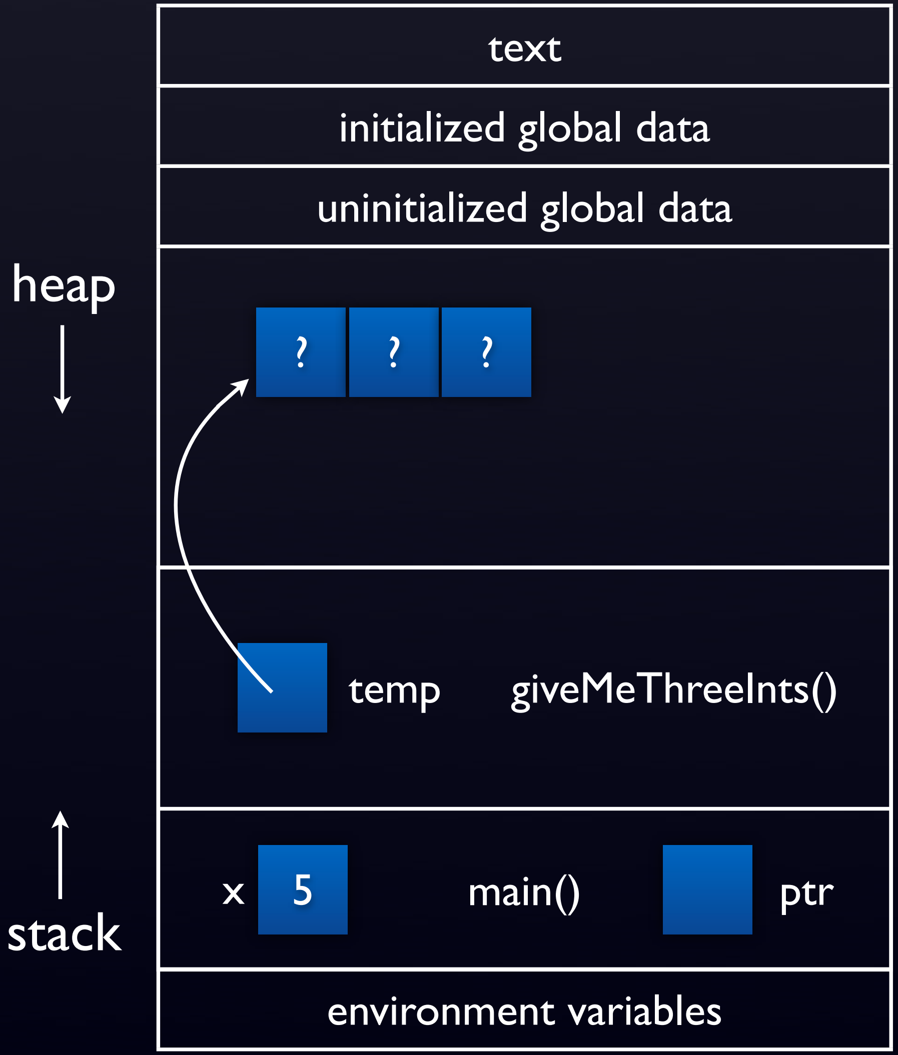
```
int main(void)
{
→       int x = 5;
        int* ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int* giveMeThreeInts(void)
{
        int* temp = malloc(sizeof(int) * 3);

        return temp;
}
```

malloc()



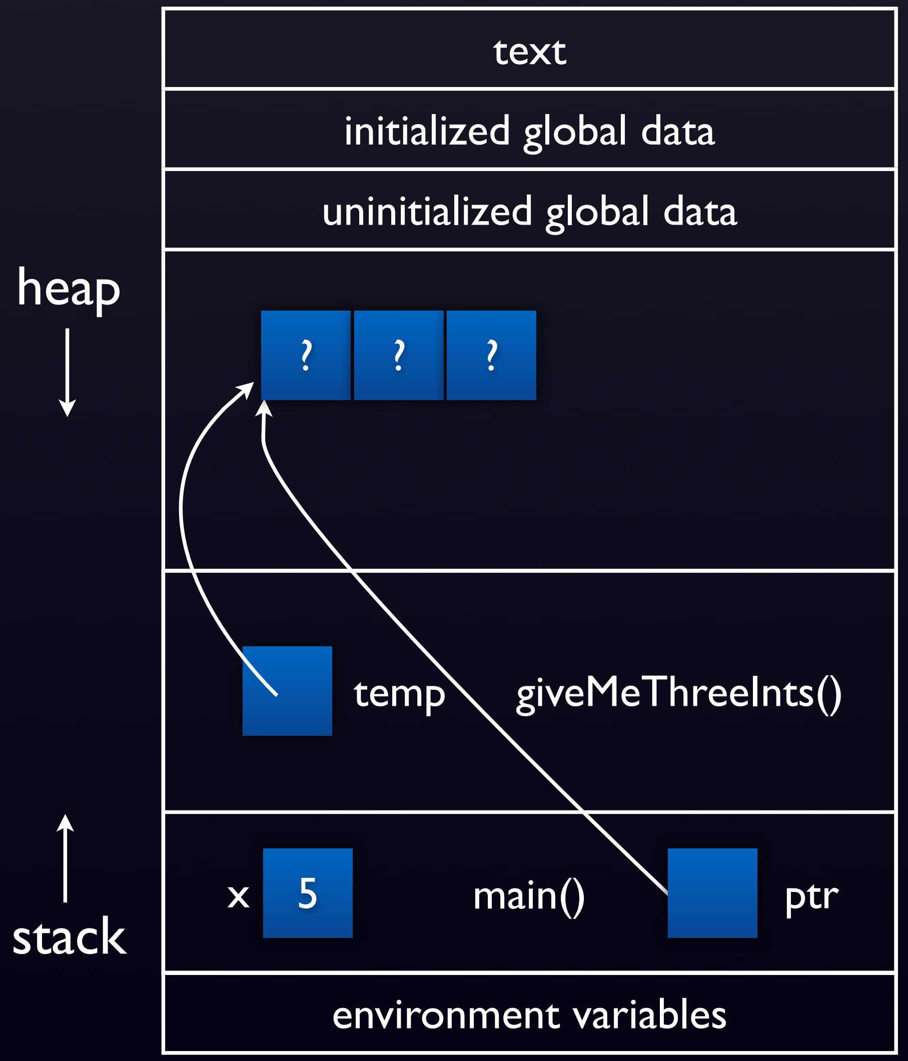
```
int main(void)
{
        int x = 5;
  ───>  int* ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int* giveMeThreeInts(void)
{
        int* temp = malloc(sizeof(int) * 3);

        return temp;
}
```

malloc()



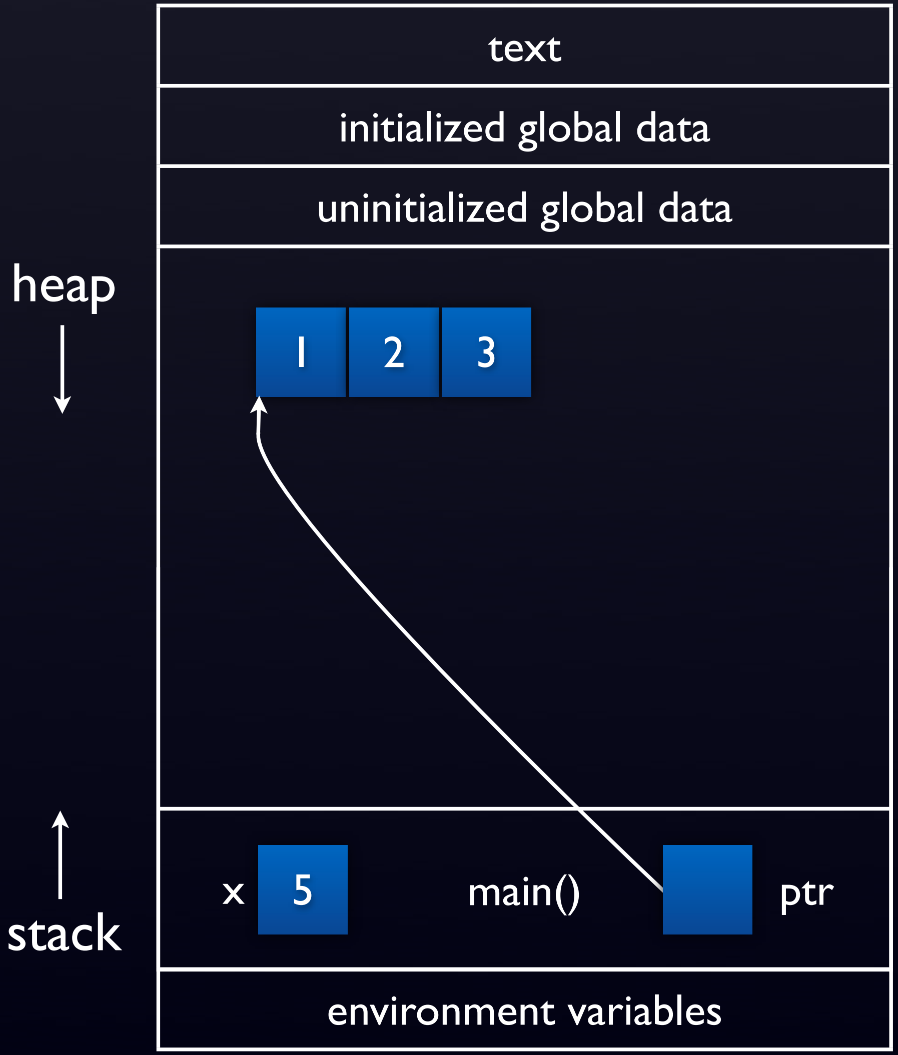
```
int main(void)
{
        int x = 5;
        int* ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int* giveMeThreeInts(void)
{
  ⟶    int* temp = malloc(sizeof(int) * 3);

        return temp;
}
```

malloc()



```
int main(void)
{
        int x = 5;
──→     int* ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int* giveMeThreeInts(void)
{
        int* temp = malloc(sizeof(int) * 3);

        return temp;
}
```

malloc()



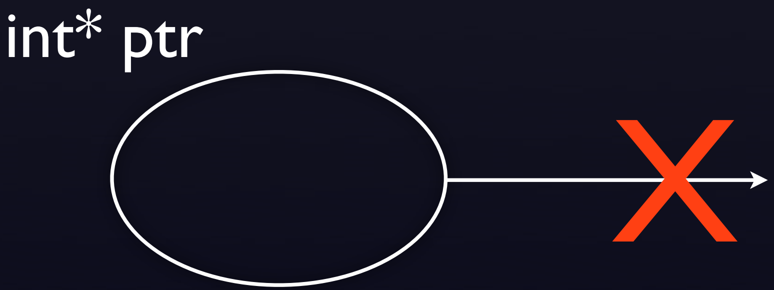
```
int main(void)
{
        int x = 5;
        int* ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int* giveMeThreeInts(void)
{
        int* temp = malloc(sizeof(int) * 3);

        return temp;
}
```

CS50: Quiz 0

Pointers

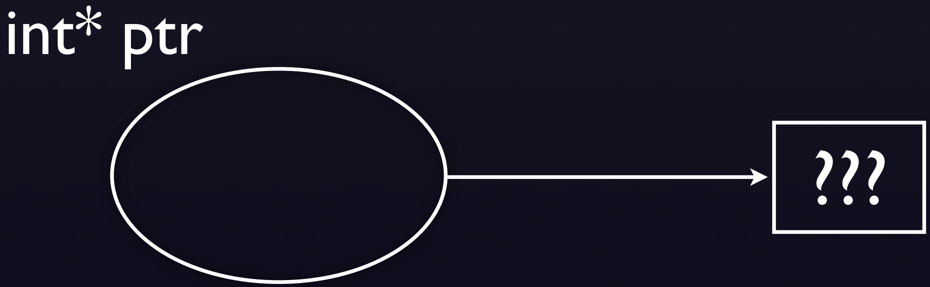
Recall Binky

```
int main(void)
{
      // usually done in same step
──→   int* ptr;
      ptr = malloc(sizeof(int));

      if (ptr == NULL)
            return 1;

      *ptr = 1;
      free(ptr);

      int x = 5;
      ptr = &x;

      return 0;
}
```



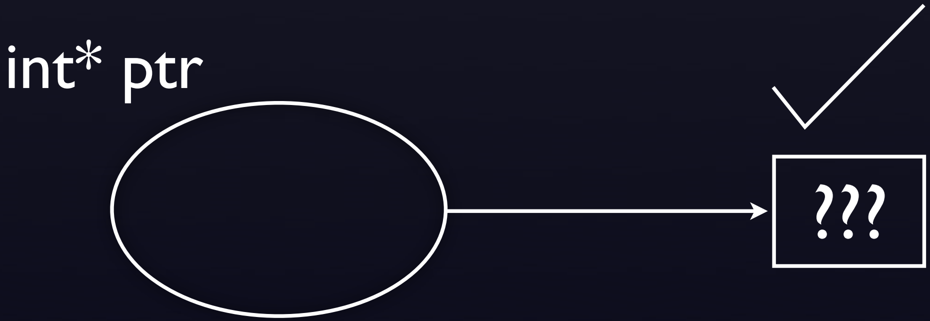
Recall Binky

```
int main(void)
{
      // usually done in same step
      int* ptr;
──→   ptr = malloc(sizeof(int));

      if (ptr == NULL)
            return 1;

      *ptr = 1;
      free(ptr);

      int x = 5;
      ptr = &x;

      return 0;
}
```



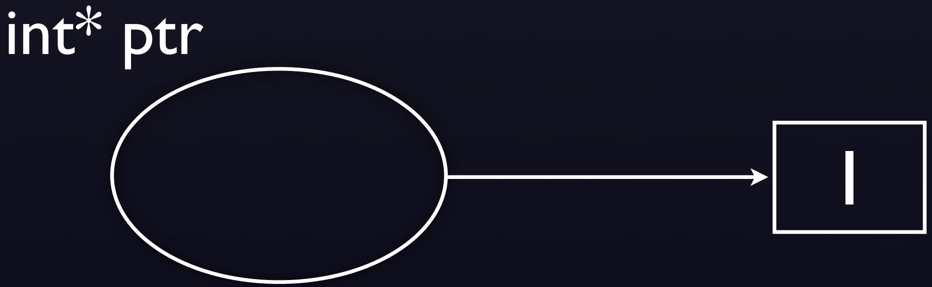
Recall Binky

```
int main(void)
{
      // usually done in same step
      int* ptr;
      ptr = malloc(sizeof(int));

────> if (ptr == NULL)
            return 1;

      *ptr = 1;
      free(ptr);

      int x = 5;
      ptr = &x;

      return 0;
}
```



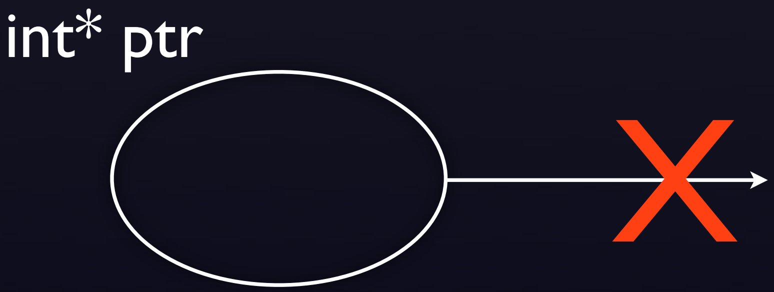
Recall Binky

```
  int main(void)
  {
       // usually done in same step
       int* ptr;
       ptr = malloc(sizeof(int));

       if (ptr == NULL)
            return 1;

──→    *ptr = 1;
       free(ptr);

       int x = 5;
       ptr = &x;

       return 0;
  }
```



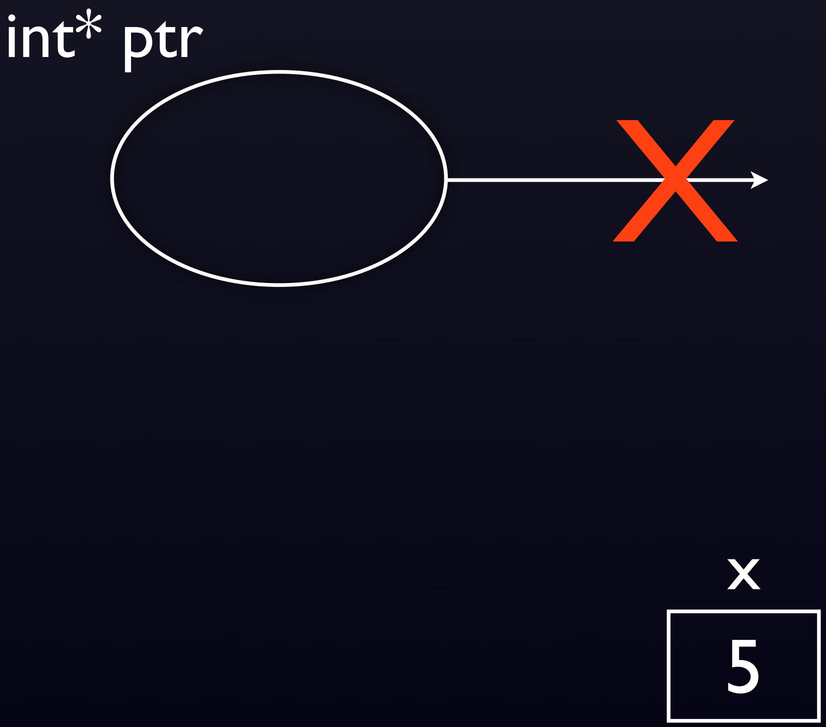
Recall Binky

```
int main(void)
{
    // usually done in same step
    int* ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
--> free(ptr);

    int x = 5;
    ptr = &x;

    return 0;
}
```



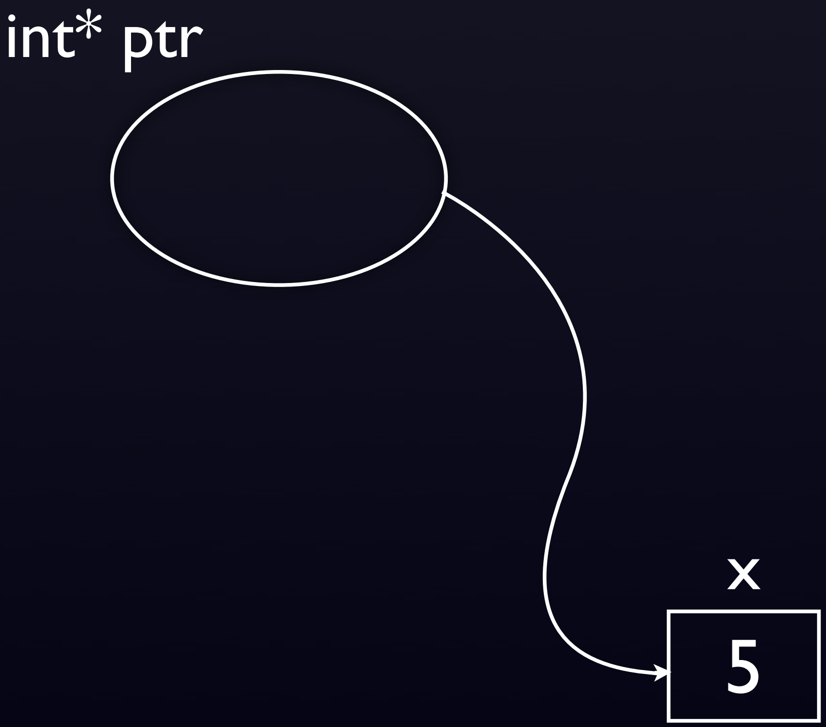
Recall Binky

```
int main(void)
{
    // usually done in same step
    int* ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

→   int x = 5;
    ptr = &x;

    return 0;
}
```



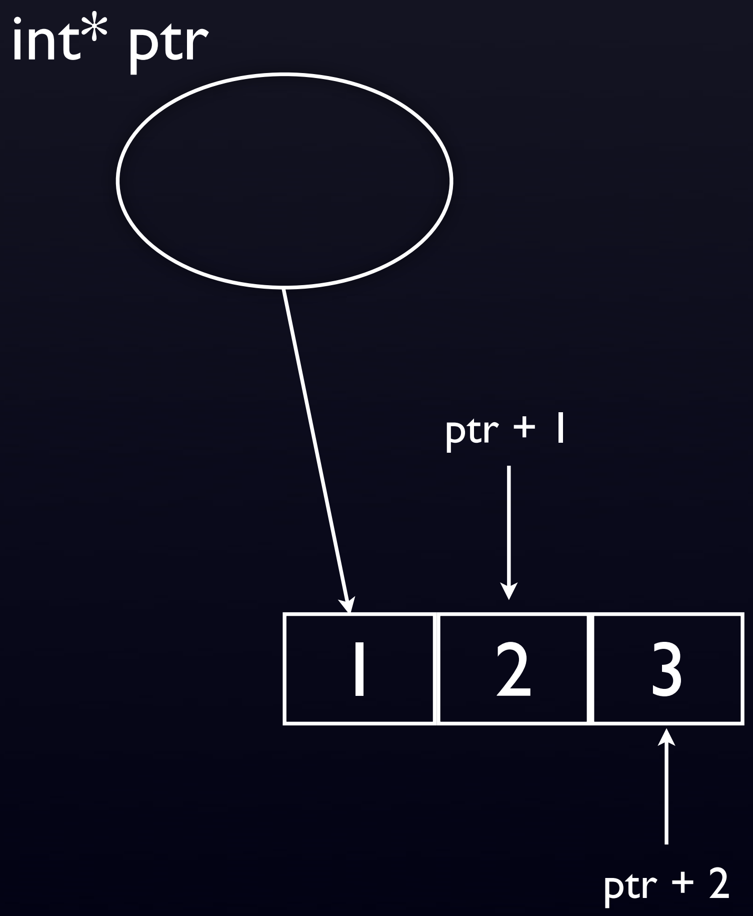
Recall Binky

```
int main(void)
{
    // usually done in same step
    int* ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    int x = 5;
→   ptr = &x;

    return 0;
}
```



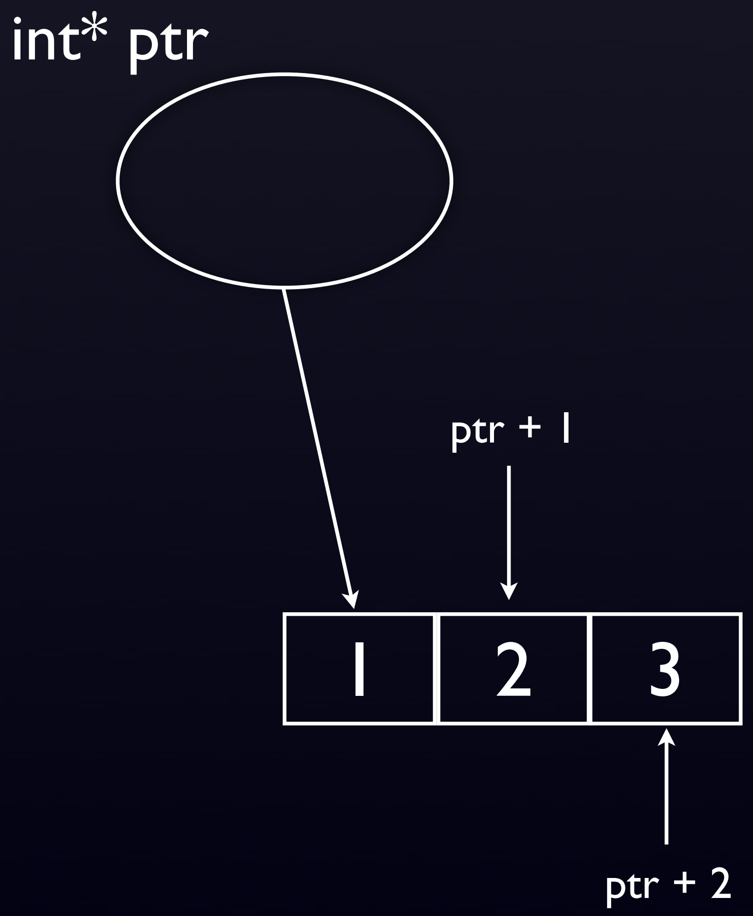
Pointer Arithmetic

```
int main(void)
{
    int* ptr = malloc(sizeof(int) * 3);

    *ptr = 1;
    *(ptr + 1) = 2;    // one int over from ptr
    *(ptr + 2) = 3;

    printf("%d", *(ptr + 1));

    ptr++;

    printf("%d", *(ptr + 1));

    ptr--;
    free(ptr);
}
```



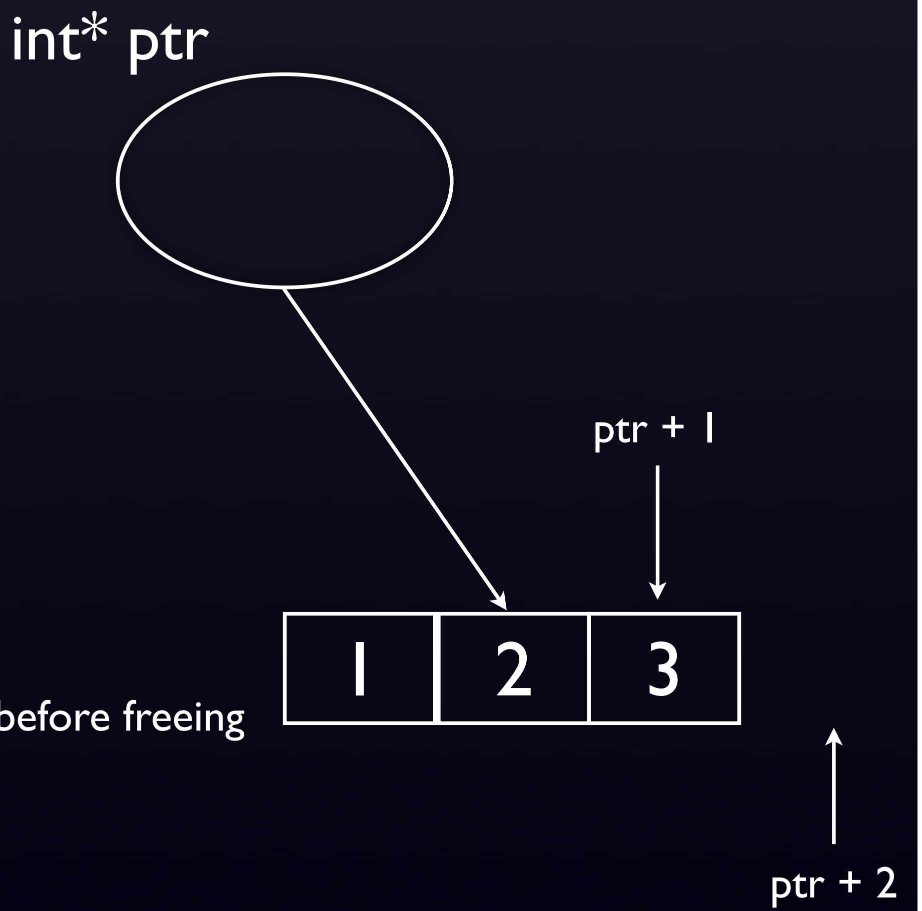
Pointer Arithmetic

```
int main(void)
{
	int* ptr = malloc(sizeof(int) * 3);

	*ptr = 1;
	*(ptr + 1) = 2;
	*(ptr + 2) = 3;

	printf("%d", *(ptr + 1));	// prints out 2

	ptr++;

	printf("%d", *(ptr + 1));

	ptr--;
	free(ptr);
}
```



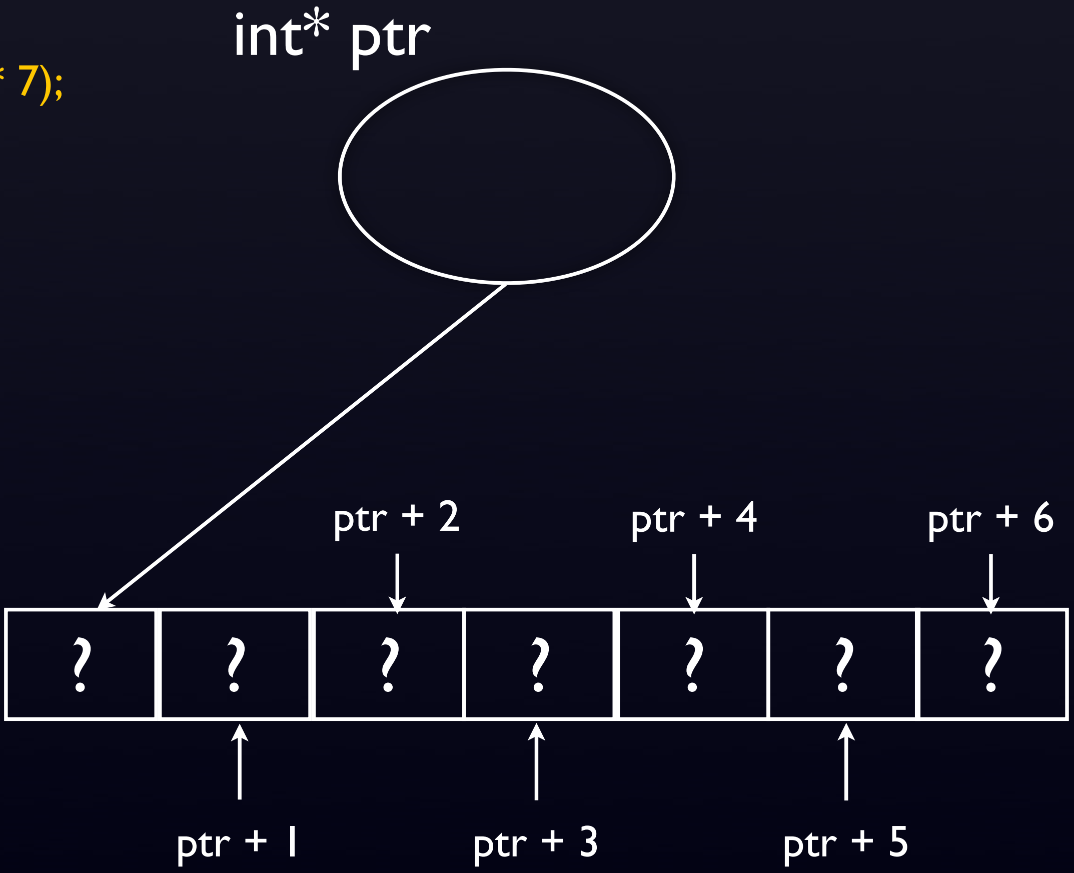
Pointer Arithmetic

```
int main(void)
{
	int* ptr = malloc(sizeof(int) * 3);

	*ptr = 1;
	*(ptr + 1) = 2;
	*(ptr + 2) = 3;

	printf("%d", *(ptr + 1));

	ptr++;                          // changes ptr

	printf("%d", *(ptr + 1));   // now prints out 3

	ptr--;    // move back to original ptr location before freeing
	free(ptr);
}
```



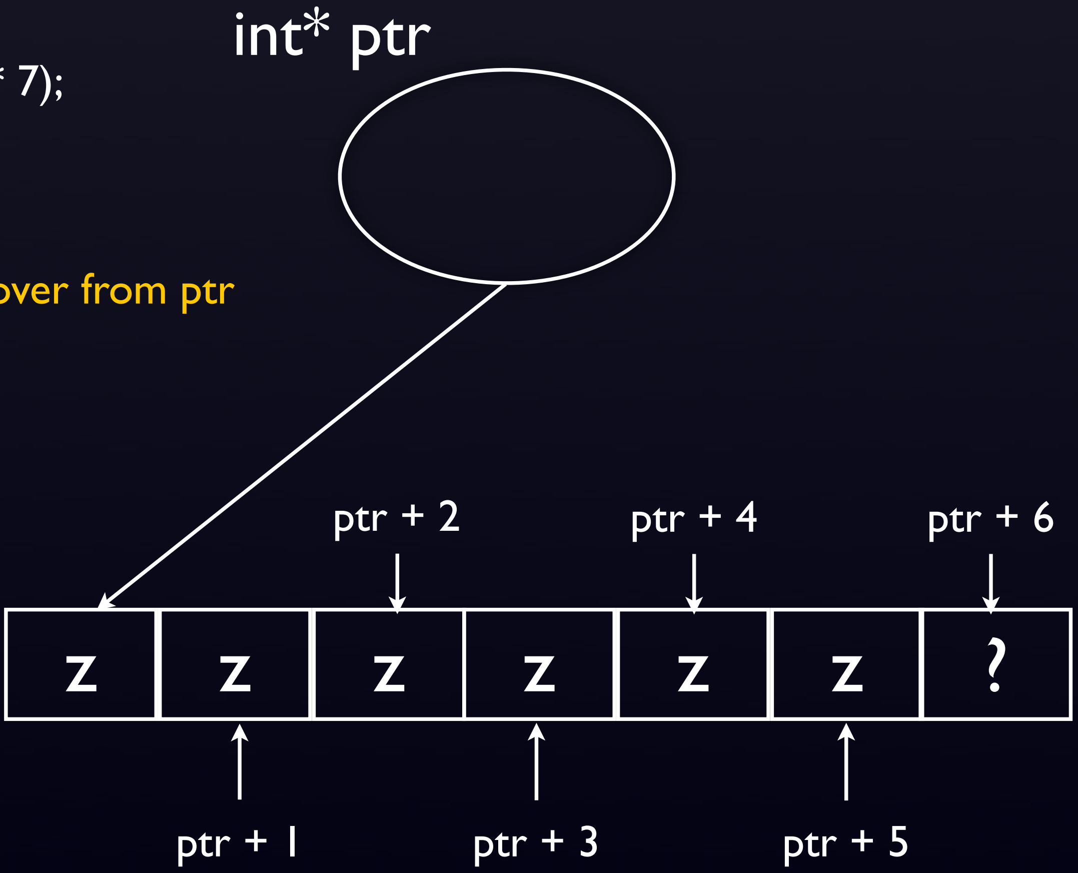
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



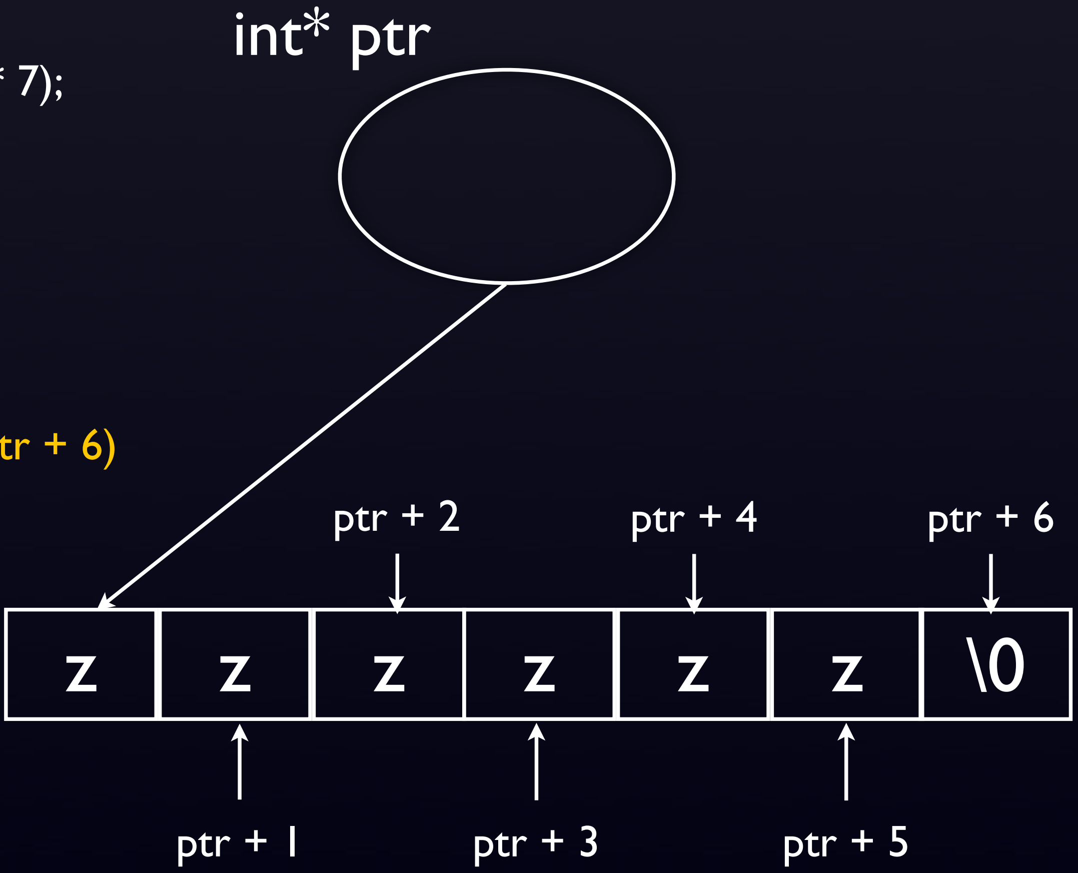
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;   //  i chars over from ptr
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



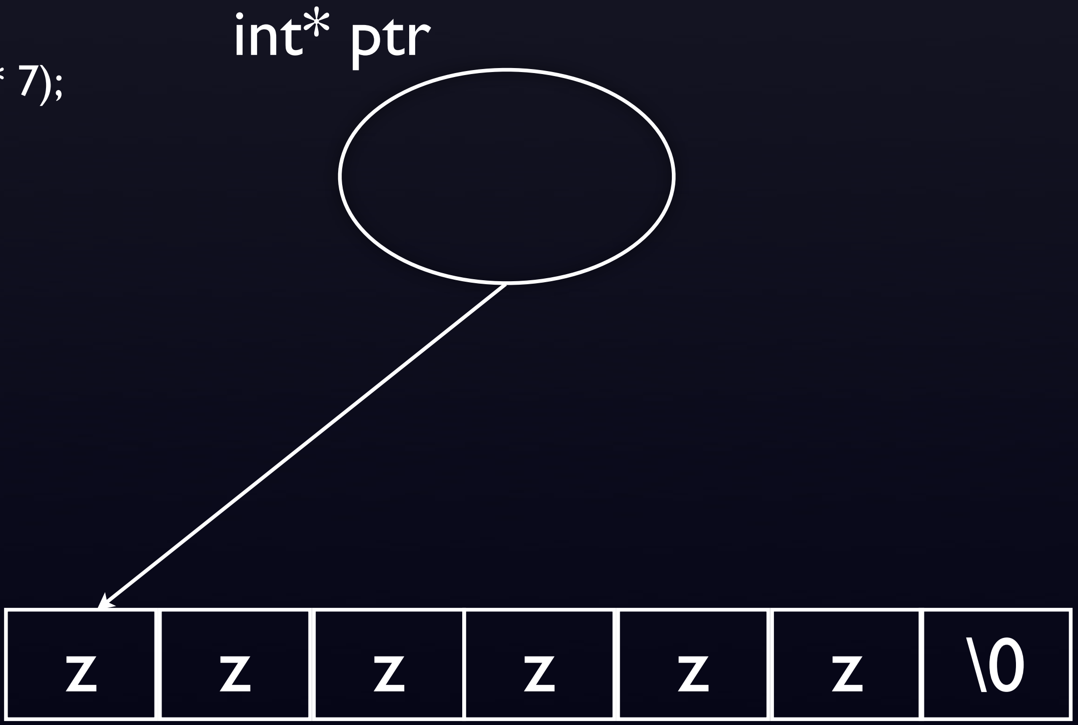
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;  //shorthand for *(ptr + 6)

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



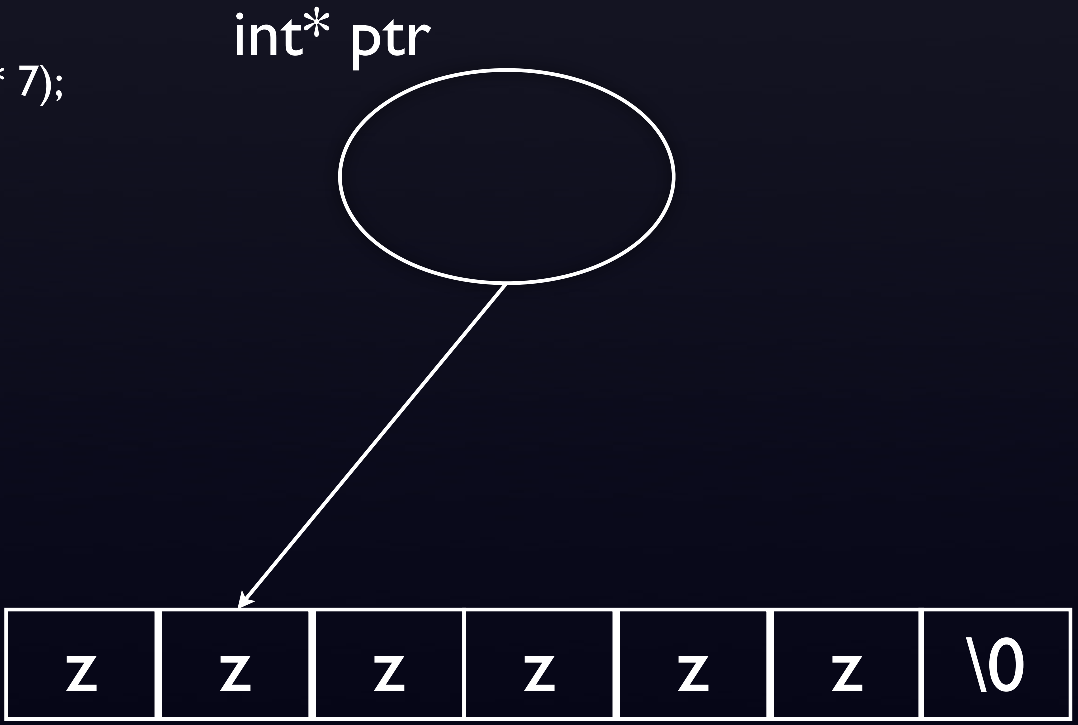
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



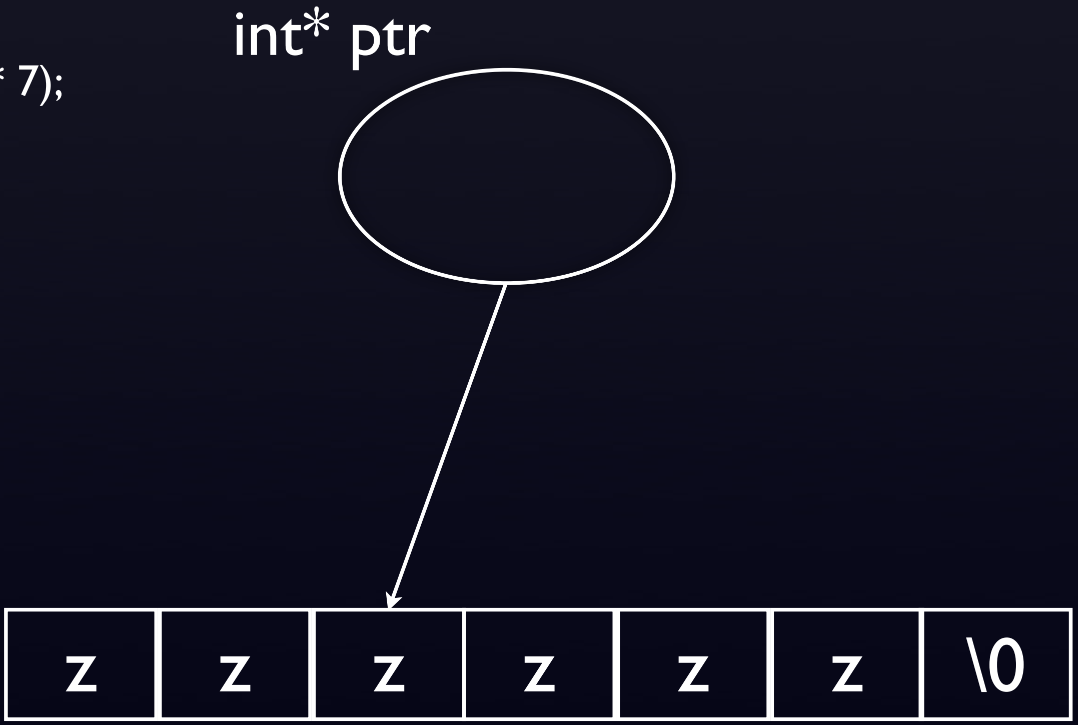
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



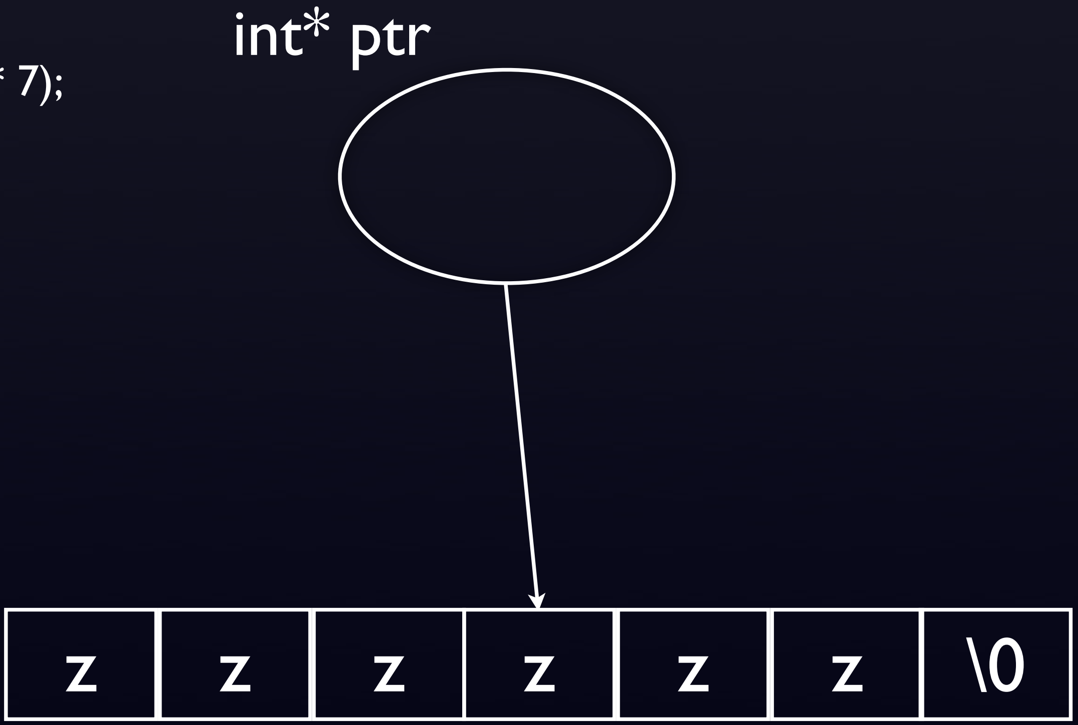
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



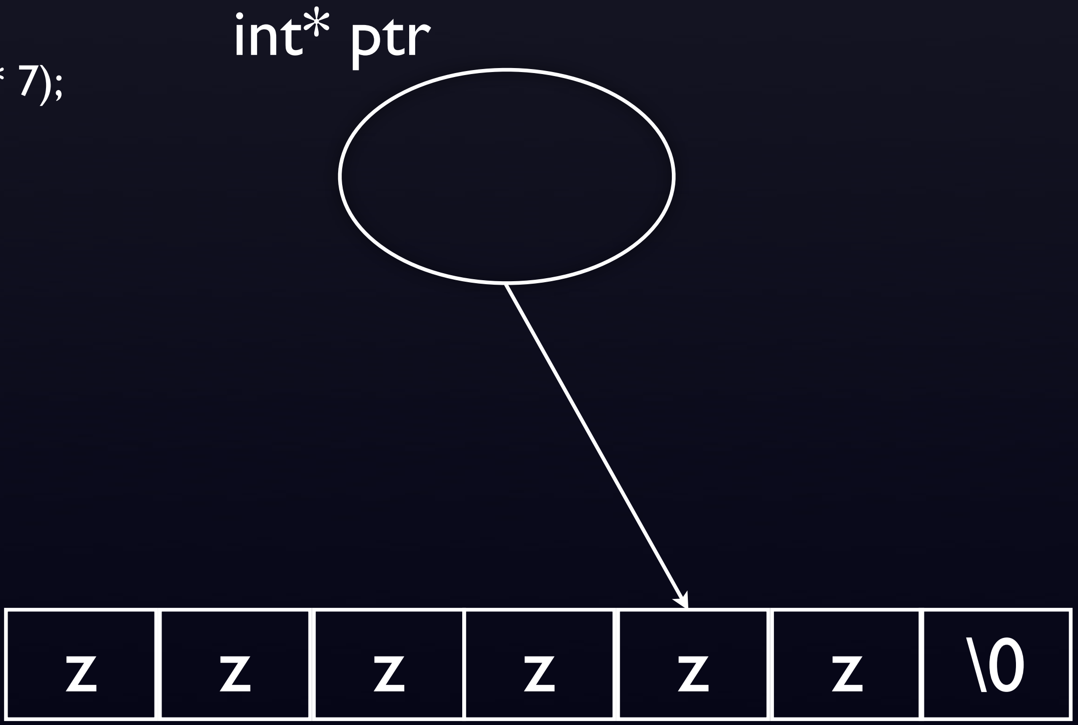
Pointer Arithmetic with Strings

```
int main(void)
{
      char* ptr = malloc(sizeof(char) * 7);

      for (int i = 0; i < 6 i++)
      {
            *(ptr + i) = ‘z’;
      }

      ptr[6] = ‘\0’;

      while (*ptr != ‘\0’)
      {
            printf("%c", *ptr);
            ptr++;
      }

      ptr -= 6;
      free(ptr);
}
```



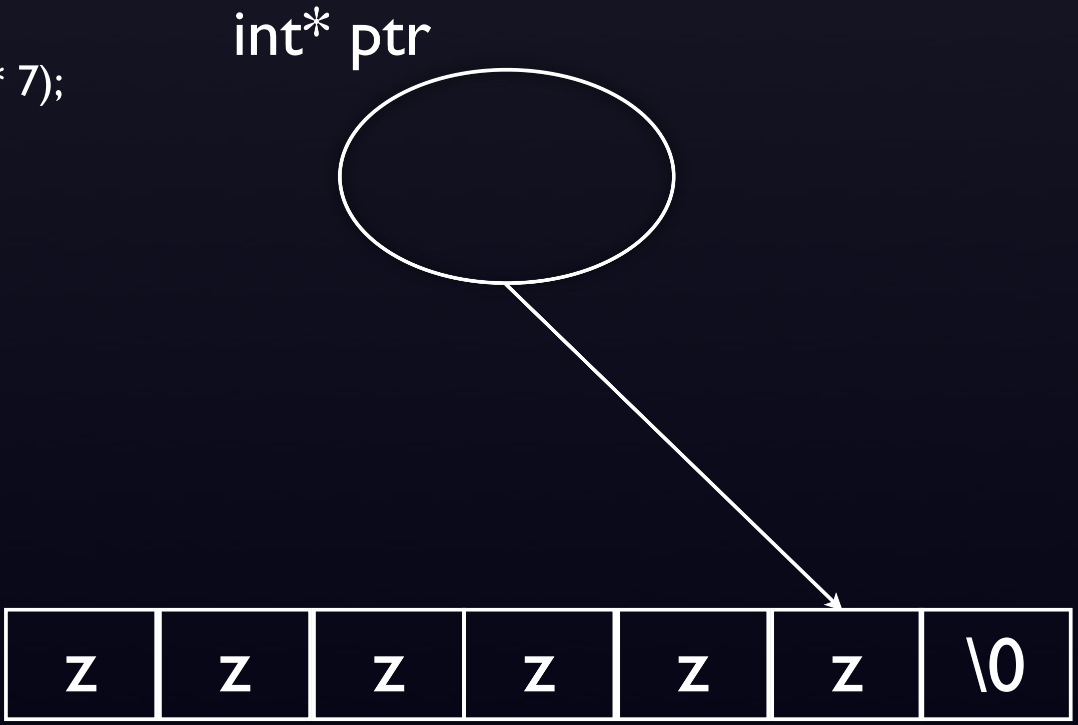
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



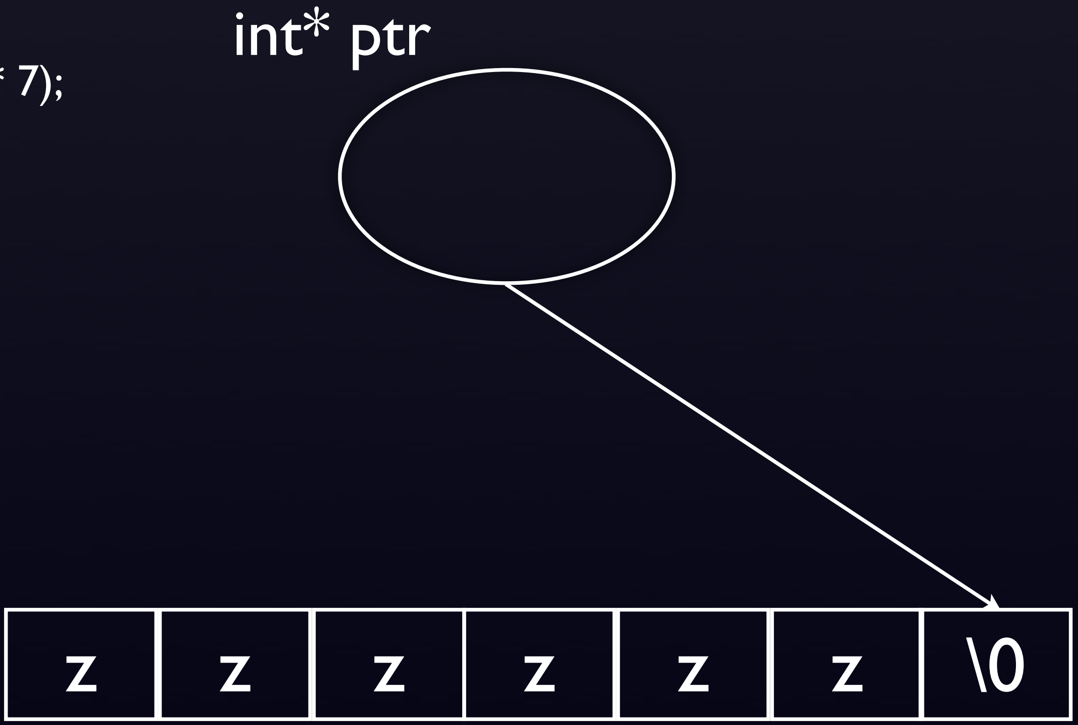
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



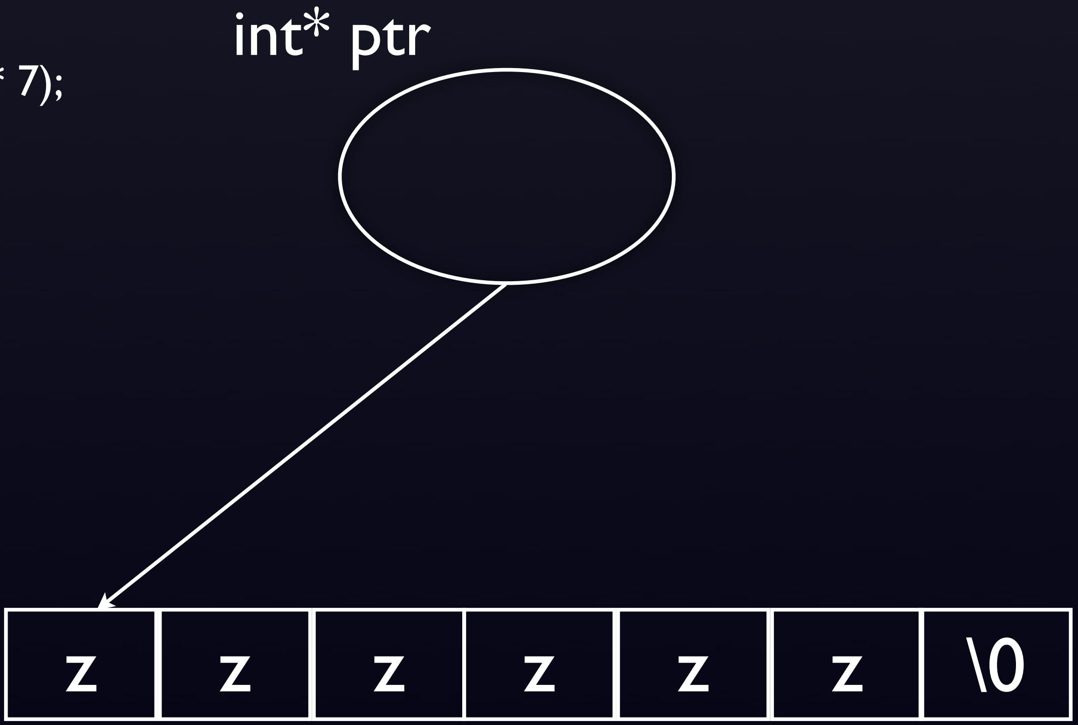
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')    // !!!
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



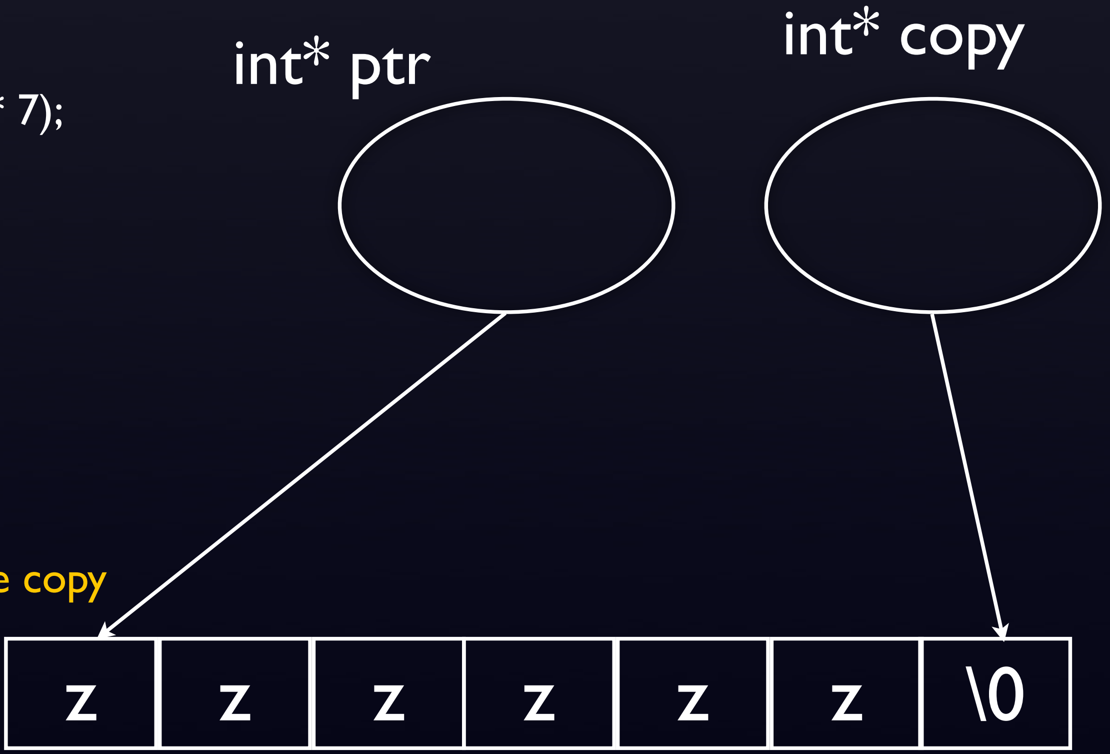
Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;   // move back to original memory location before freeing
    free(ptr);
}
```



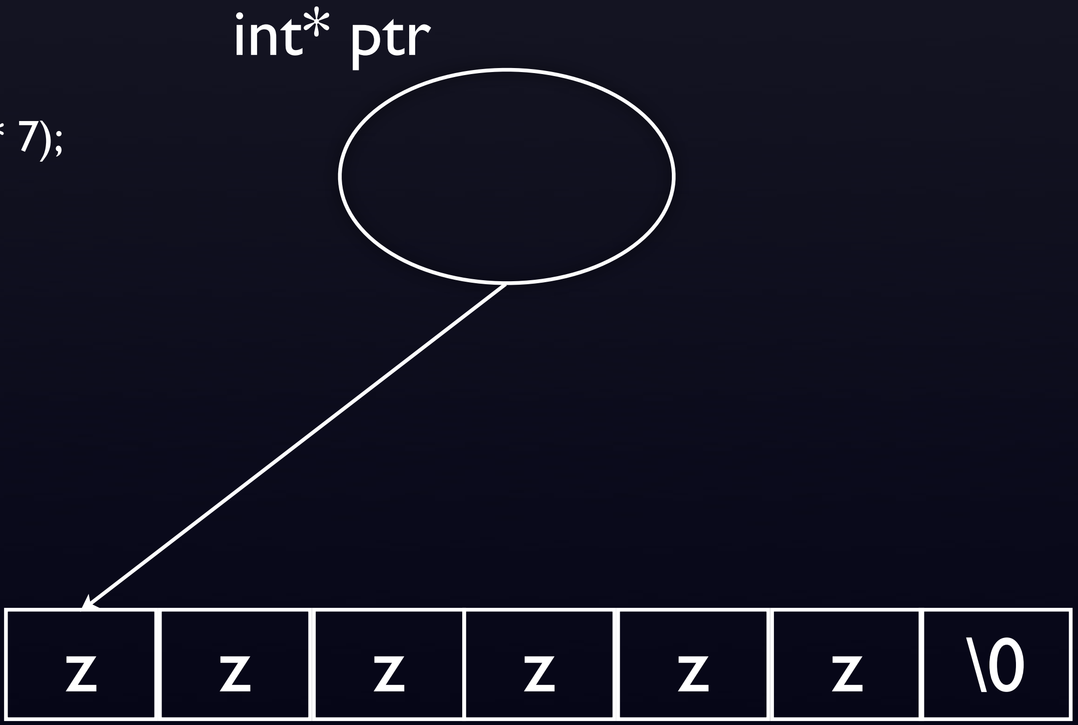
Pointer Arithmetic with Strings

```
int main(void)
{
	char* ptr = malloc(sizeof(char) * 7);

	for (int i = 0; i < 6 i++)
	{
		*(ptr + i) = 'z';
	}

	ptr[6] = '\0';

	char* copy = ptr;	// create/use copy
	while (*copy != '\0')
	{
		printf("%c", *copy);
		copy++;
	}

	free(ptr);  // since we never changed original pointer, no extra arithmetic
}
```



Pointer Arithmetic with Strings

```
int main(void)
{
    char* ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    printf("%s", ptr);    // no *

    free(ptr);
}
```



Arrays and Pointers

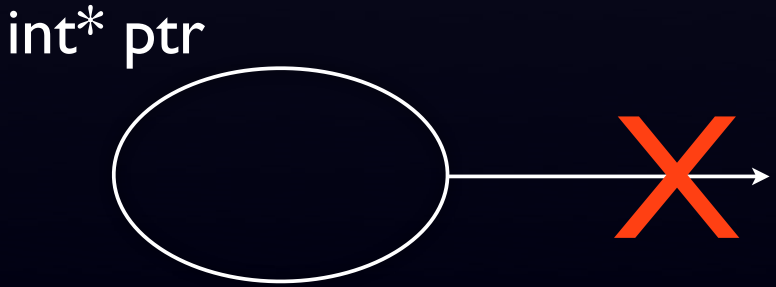
So, arrays and pointers are equivalent!

$$\begin{tikzcd} \text{x[y]} \arrow[d, leftrightarrow] \\ \textbf{*(x + y)} \end{tikzcd}$$

So... what can go wrong?

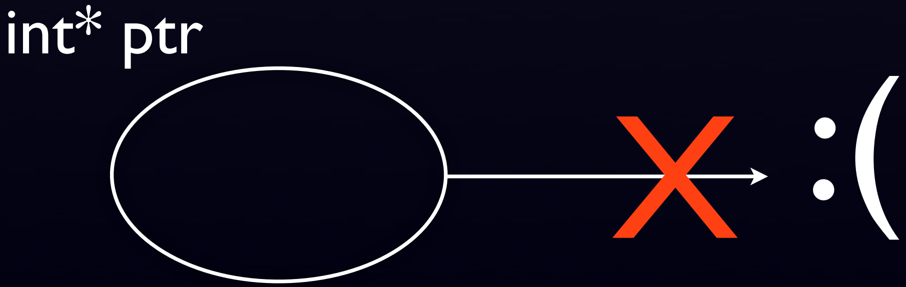
DANGER BAD THINGS D:

```
int main(void)
{
    // oops, pretty sure we don't have that much memory
    // malloc will fail, returning a NULL pointer
──> int* ptr = malloc(sizeof(int) * 2147483647);

    // oops, we forgot to check if it was NULL

    *ptr = 1;

    return 0;
}
```



Null Pointer Dereference

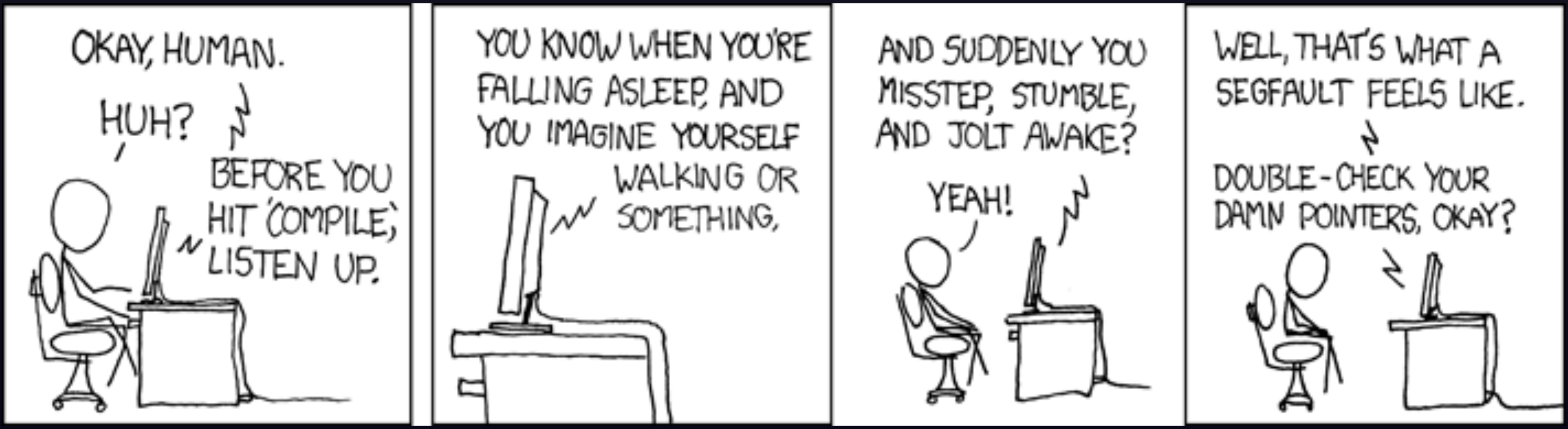
```
int main(void)
{
    // oops, pretty sure we don't have that much memory
    // malloc will fail, returning a NULL pointer
    int* ptr = malloc(sizeof(int) * 2147483647);

    // oops, we forgot to check if it was NULL

──> *ptr = 1;     // oops, we just died x.x,  aka  "dereferencing a null pointer"

    return 0;
}
```



Null Pointer Dereference

```
int main(void)
{
    // oops, pretty sure we don't have that much memory
    // malloc will fail, returning a NULL pointer
    int* ptr = malloc(sizeof(int) * 2147483647);

    // solution, check if null, and exit the program
    if (ptr == NULL)
        return 1;

    *ptr = 1;      // no longer dereferenced if ptr is NULL

    return 0;
}
```

:D


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Memory Leaks

```
int main(void)
{
    while (1)
    {
        int* ptr = malloc(sizeof(int));

        if (ptr == NULL)
            return 1;

        *ptr = 1;
        // oops, we forgot to free memory, we'll get a memory leak!
    }

    return 0;
}
```

Memory Leaks

```
int main(void)
{
    while (1)
    {
        int* ptr = malloc(sizeof(int));

        if (ptr == NULL)
            return 1;

        *ptr = 1;
        // oops, we forgot to free memory, we'll get a memory leak!
    }

    return 0;
}
```

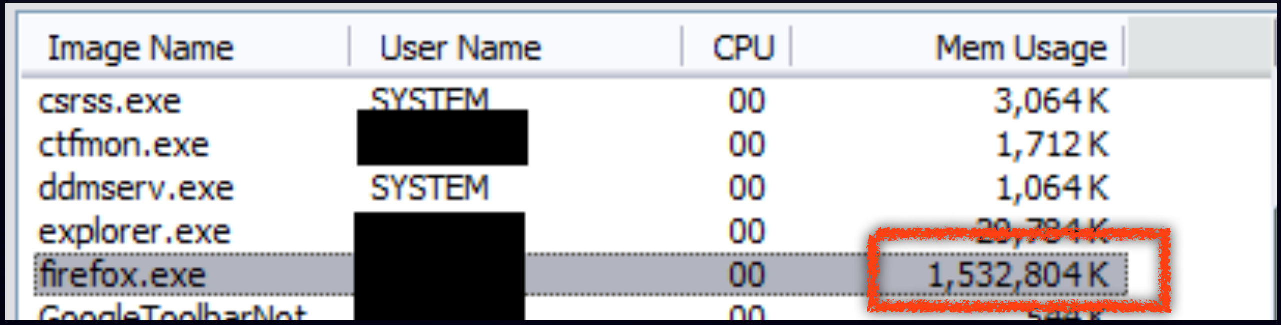
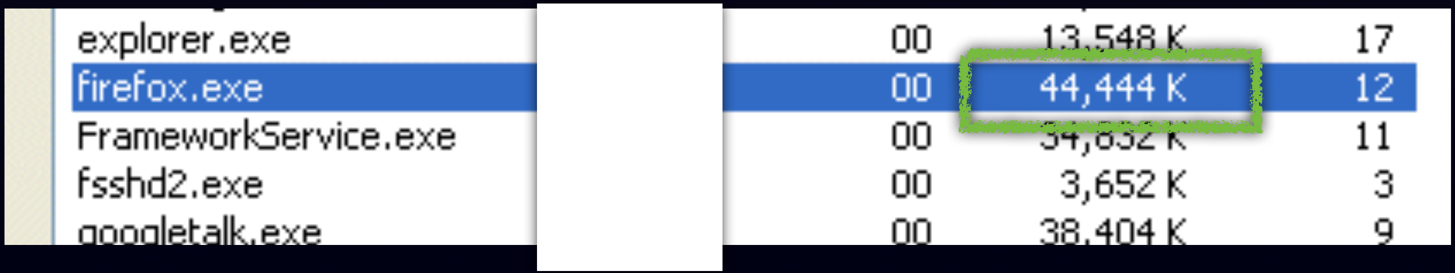


Image Name	User Name	CPU	Mem Usage
csrss.exe	SYSTEM	00	3,064 K
ctfmon.exe	[redacted]	00	1,712 K
ddmserv.exe	SYSTEM	00	1,064 K
explorer.exe	[redacted]	00	29,784 K
firefox.exe	[redacted]	00	1,532,804 K
GoogleToolbarNot	[redacted]	00	[illegible]

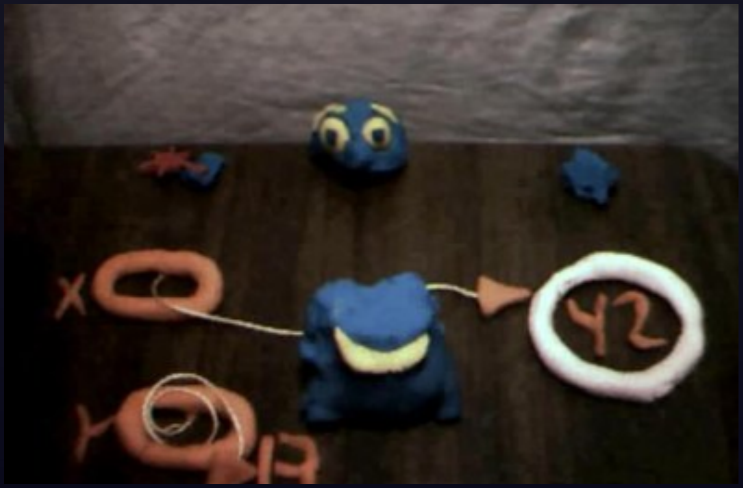
Memory Leaks

```
int main(void)
{
    while (1)
    {
        int* ptr = malloc(sizeof(int));

        if (ptr == NULL)
            return 1;

        *ptr = 1;
        free(ptr);    // fix't!
    }

    return 0;
}
```



```
explorer.exe                 00    13,548 K    17
firefox.exe                  00    44,444 K    12
FrameworkService.exe         00    34,632 K    11
fsshd2.exe                   00     3,652 K     3
googletalk.exe               00    38,404 K     9
```

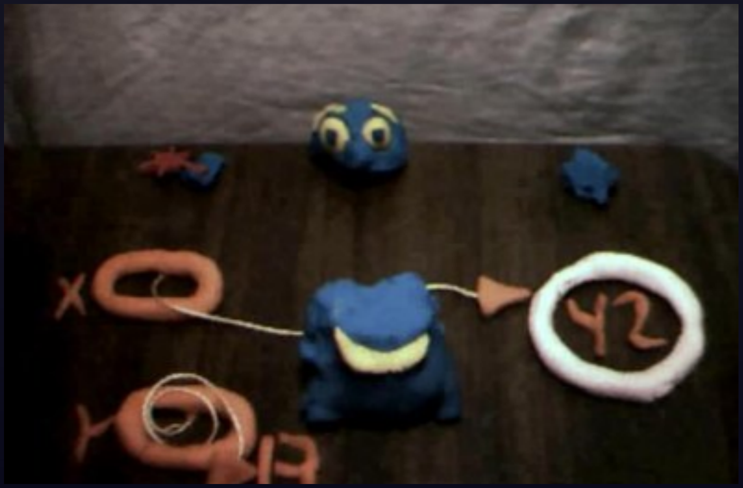
Freeing Twice (or n > 1 times)



```
int main(void)
{
    int* ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    free(ptr);      // oops, we freed something we already freed earlier.

    return 0;
}
```

Freeing Twice (or n > 1 times)

```
int main(void)
{
    int* ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    // fix't!

    return 0;
}
```



Failure to use sizeof()

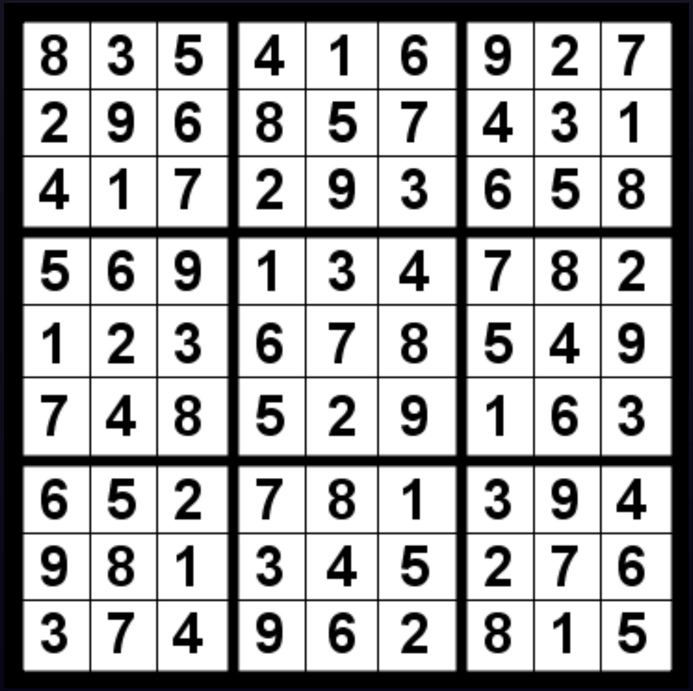
```
int main(void)
{
    // wants to malloc 2 ints. 8 bytes? Right?
    int* ptr = malloc(8);

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    return 0;
}
```

Failure to use sizeof()

```
int main(void)
{
    // actually, an int isn't necessarily 4 bytes on all systems.
    // this is safer and is more compatible with different architectures.
    int* ptr = malloc(sizeof(int) * 2);

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    return 0;
}
```

CS50: Quiz 0

Structs



8	3	5	4	1	6	9	2	7
2	9	6	8	5	7	4	3	1
4	1	7	2	9	3	6	5	8
5	6	9	1	3	4	7	8	2
1	2	3	6	7	8	5	4	9
7	4	8	5	2	9	1	6	3
6	5	2	7	8	1	3	9	4
9	8	1	3	4	5	2	7	6
3	7	4	9	6	2	8	1	5

Structs

A struct is a container that can hold and organize meaningfully related variables of different types.

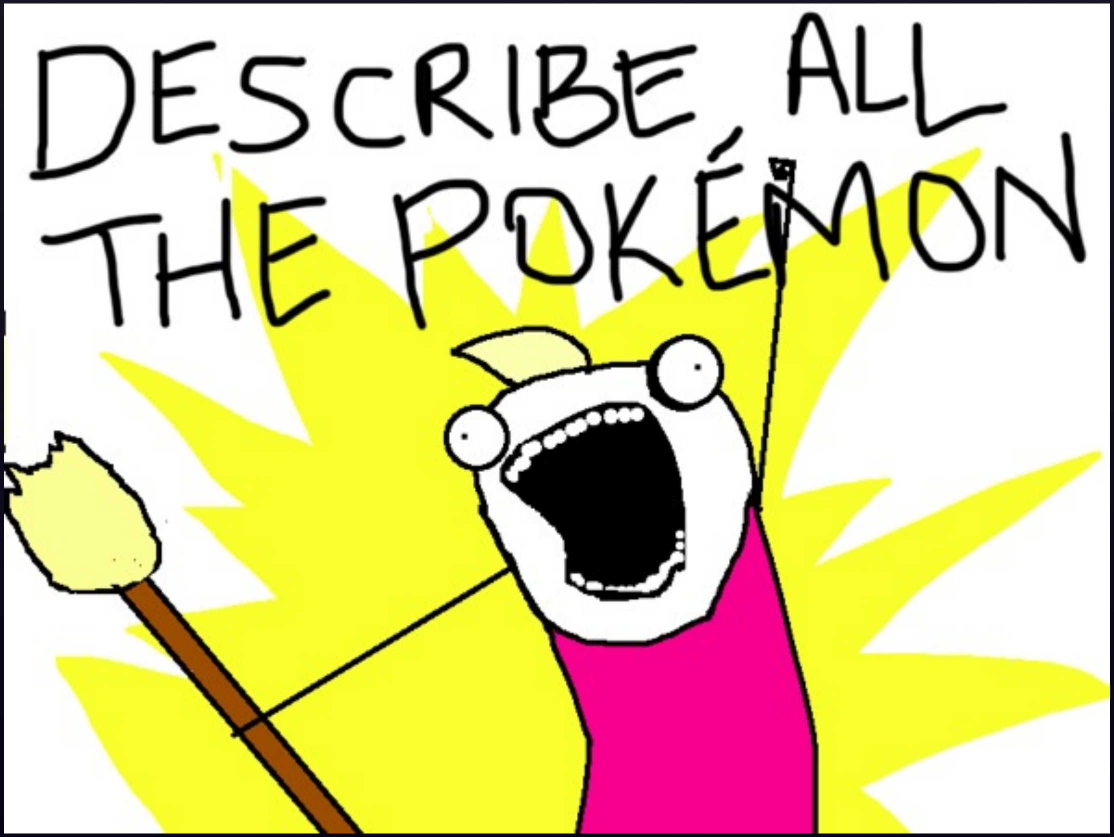
For example, let's say we want to make a collection of variables to represent a Sudoku board!

```
typedef struct
{
        int board[9][9];

        char* level;
        int x, y;
        int timeSpent;
        int totalMoves;
}
sudokuBoard;
```

```
int main(void)
{
        sudokuBoard board;

        board.board = {{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, ...};
        board.level = "n00b";
        board.x = 0;
        board.y = 0;
        board.timeSpent = 0;
        board.totalMoves = 0;

        // do stuff with board in rest of program
};
```

Structs

```
typedef struct
{
        int pokedexNo;
        int level;

        char* owner;
        char* pokemonType;
        char* nickName;

        int stats[6];
        char* moveset[4];

        ...

}
pokemon;
```



Questions?

CS50: Quiz 0

GDB

GDB

Let's you poke around the contents of memory of your program while it's executing.

How? Lots of things!

- Pausing program execution at "breakpoints"
- Printing out variables when program is paused
- Stepping through program execution, line by line
- Looking at the state of the stack (i.e. function calls)
- ...

Using GDB

```
clang hello_world.c
```

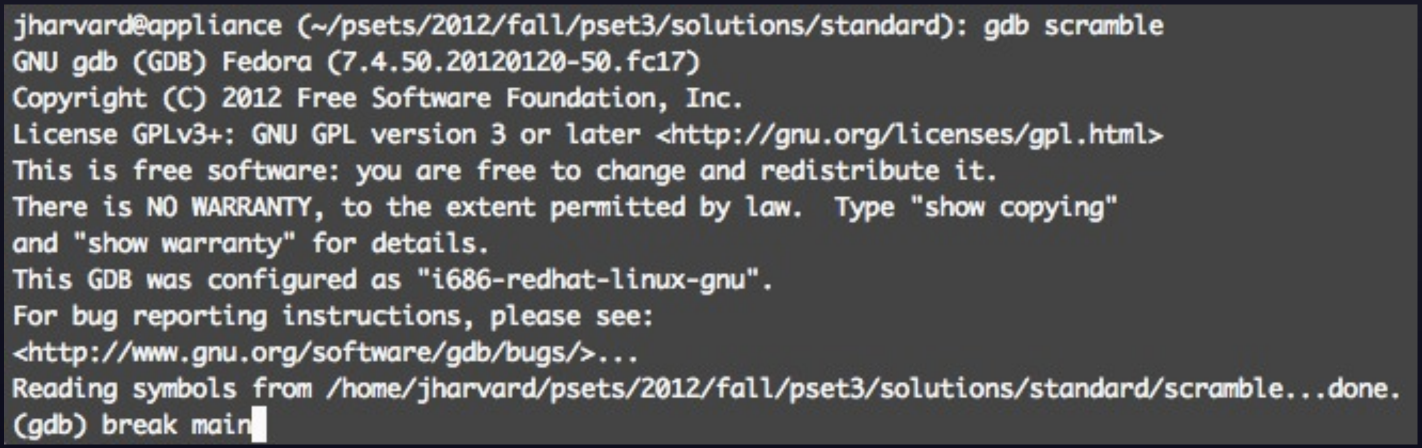
$\downarrow$

```
clang -ggdb  hello_world.c
```

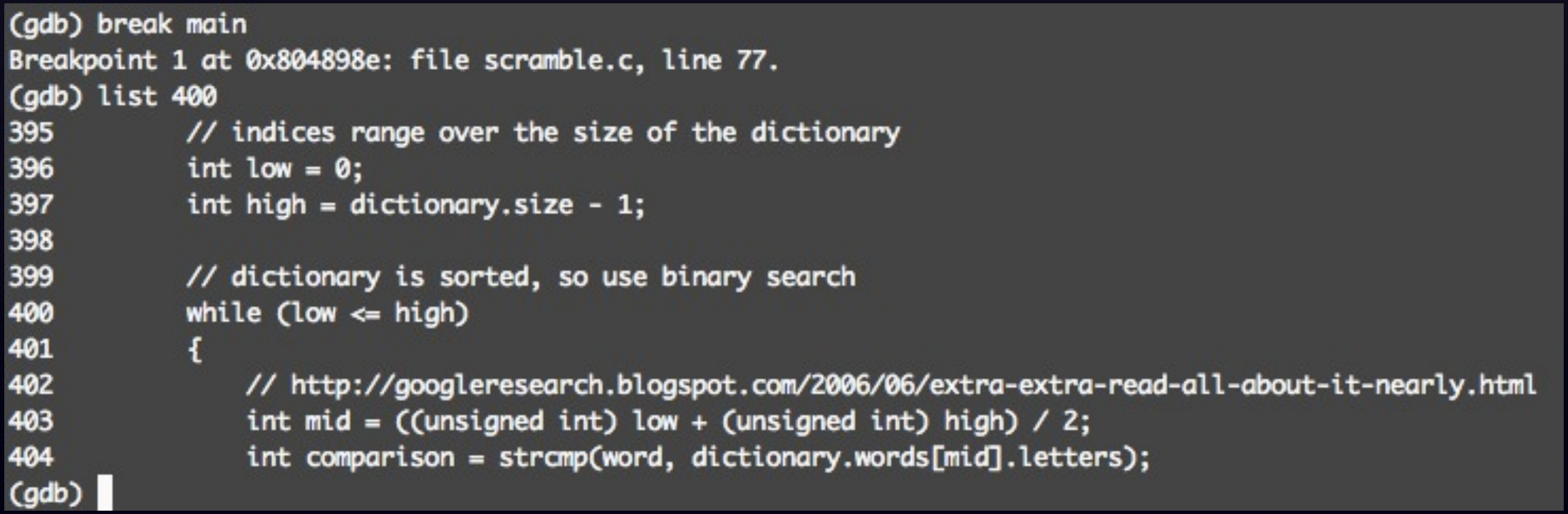
$\downarrow$

```
gdb  a.out
```

Using GDB



```
jharvard@appliance (~/psets/2012/fall/pset3/solutions/standard): gdb scramble
GNU gdb (GDB) Fedora (7.4.50.20120120-50.fc17)
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/jharvard/psets/2012/fall/pset3/solutions/standard/scramble...done.
(gdb) break main
```



```
(gdb) break main
Breakpoint 1 at 0x804898e: file scramble.c, line 77.
(gdb) list 400
395         // indices range over the size of the dictionary
396         int low = 0;
397         int high = dictionary.size - 1;
398
399         // dictionary is sorted, so use binary search
400         while (low <= high)
401         {
402             // http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
403             int mid = ((unsigned int) low + (unsigned int) high) / 2;
404             int comparison = strcmp(word, dictionary.words[mid].letters);
(gdb) 
```

GDB Commands

These are some you’ll need to know!

GDB Commands

These are some you'll need to know!

run arg1, arg2, ...	runs program with command line arguments

GDB Commands

These are some you'll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame

GDB Commands

These are some you'll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame
break function_name	sets a breakpoint at a function called function_name

GDB Commands

These are some you'll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame
break function_name	sets a breakpoint at a function called function_name
break line_number	sets a breakpoint at a line of your code

GDB Commands

These are some you'll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame
break function_name	sets a breakpoint at a function called function_name
break line_number	sets a breakpoint at a line of your code
frame n	gives you information about the nth stack frame

GDB Commands

These are some you'll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame
break function_name	sets a breakpoint at a function called function_name
break line_number	sets a breakpoint at a line of your code
frame n	gives you information about the nth stack frame
backtrace	tells you stack frames leading to current point in program

GDB Commands

These are some you'll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame
break function_name	sets a breakpoint at a function called function_name
break line_number	sets a breakpoint at a line of your code
frame n	gives you information about the nth stack frame
backtrace	tells you stack frames leading to current point in program
next	moves forward one line in the current execution of code

GDB Commands

These are some you'll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame
break function_name	sets a breakpoint at a function called function_name
break line_number	sets a breakpoint at a line of your code
frame n	gives you information about the nth stack frame
backtrace	tells you stack frames leading to current point in program
next	moves forward one line in the current execution of code
step	moves forward one line, stepping into a function where applicable

GDB Commands

These are some you'll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame
break function_name	sets a breakpoint at a function called function_name
break line_number	sets a breakpoint at a line of your code
frame n	gives you information about the nth stack frame
backtrace	tells you stack frames leading to current point in program
next	moves forward one line in the current execution of code
step	moves forward one line, stepping into a function where applicable
continue	moves forward in the program until the next breakpoint

GDB Commands

These are some you’ll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame
break function_name	sets a breakpoint at a function called function_name
break line_number	sets a breakpoint at a line of your code
frame n	gives you information about the nth stack frame
backtrace	tells you stack frames leading to current point in program
next	moves forward one line in the current execution of code
step	moves forward one line, stepping into a function where applicable
continue	moves forward in the program until the next breakpoint
list n	shows the lines of code around the "nth" line of code