Quiz 0 Review Session Part 0

October 14, 2013

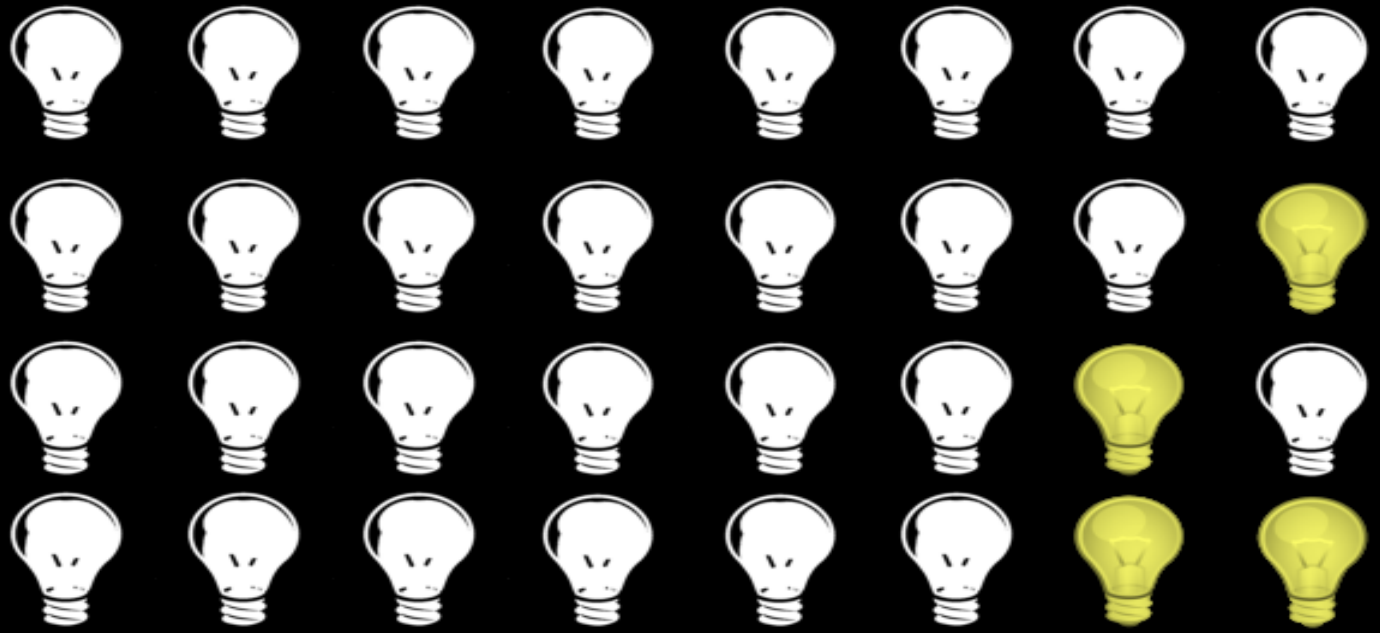
Karen Xiao

Hey! You have a quiz on Wednesday!

cs50.net/quizzes

Let’s get started!

Binary



Binary - Basics

$$\underset{2^7}{1}\ \underset{2^6}{0}\ \underset{2^5}{1}\ \underset{2^4}{0}\ \underset{2^3}{0}\ \underset{2^2}{0}\ \underset{2^1}{1}\ \underset{2^0}{1}$$

$$1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 163$$

Binary – Binary to Decimal

$1 = 1*2^0 = 1$

$10 = 1*2^1 + 0*2^0 = 2$

$11 = 1*2^1 + 1*2^0 = 3$

$100 = 1*2^2 + 0*2^1 + 0*2^0 = 4$

$101 = 1*2^2 + 0*2^1 + 1*2^0 = 5$

Binary – Arithmetic

```
      1 1
  101011
+ 010001
--------
  111100
```

```
  1     1
  14205
+ 19418
-------
  33623
```

ASCII

- Mapping between characters and numbers
- For expressing alphabetic, numeric, and other characters in binary, the “language” that is understood by a computer

ASCII - Math

- Because characters are fundamentally just numbers, we can do math with chars!

```
int A = 65;
int B = 'A' + 1;
char C = 'D' - 1;
char D = 68;

printf("%c %c %c %c", A, B, C, D);
```

What will this print out?

ASCII - Math

- Because characters are fundamentally just numbers, we can do math with chars!

```
int A = 65;
int B = 'A' + 1;
char C = 'D' - 1;
char D = 68;

printf("%c %c %c %c", A, B, C, D);
```

What will this print out? A B C D

ASCII

- Note: ‘5’ does not equal 5
- How might we convert them?

ASCII

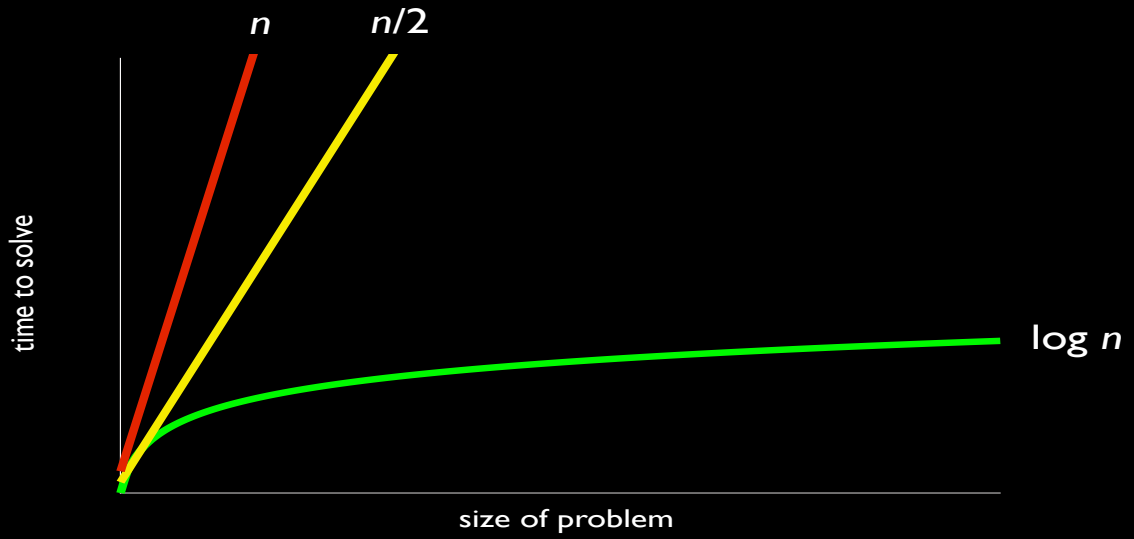
- Note: ‘5’ does not equal 5
- How might we convert them?

‘5’ – ‘0’ = 5

‘0’ + 5 = ‘5’

Algorithms

- A step-by-step set of instructions for how to perform a certain task (like a recipe?)



Pseudocode

- English-like syntax meant to represent a programming language
- Example: ask a user to guess my favorite number

```
get user's guess
if guess is correct
        tell them they are correct
else
        tell them they are not correct
```

Source Code

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

int main(void)
{
    printf("What is Karen's favorite number: ");
    int n = GetInt();

    if (n == 8)
    {
        printf("That is correct!\n");
    }
    else
    {
        printf("That is incorrect!\n");
    }
}
```

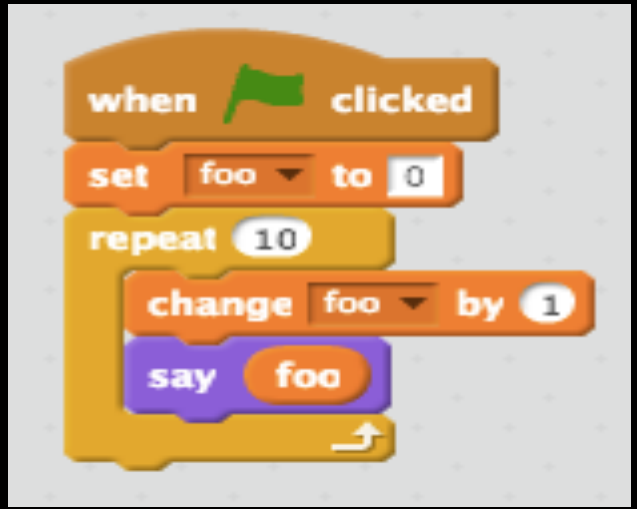
So how does your computer understand that?

Compiler

- `make` runs a compiler named `clang` for you with some command-line arguments.
- `clang` will then compile your source code to object code (0's and 1's that your computer understands)
- Source code -> Compiler -> Object code
- But more on that later…

Scratch

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



Let’s look at some of these building blocks that make up a program.

Boolean Expressions

Boolean expressions are those that have only two possible values: true or false, yes or no, on or off, 1 or 0.

```
bool happy = true;
if (happy)
{
    printf("smile");
}
```

Boolean Operators

&&	and
||	or
!	not
==	equal to
<=	less than or equal to
>=	greater than or equal to
<	less than
>	greater than

Conditions

Conditions are forks in the logic of a program that execute depending on whether or not certain criteria are met.

```
int x = GetInt();
if (x < 8)
{
    printf("%i is less than 8", x);
}
else if (x > 8)
{
    printf("%i is greater than 8", x);
}
else
{
    printf("%i is equal to 8", x);
}
```

Loops

```
int x;
do
{
    printf("Give me an int\n");
    x = GetInt();
}
while (x != 8);
```

Loops

- for
- while
- do while

- How do we know which one to use?

Loops

- for
  - We know how many times we want to iterate
- while
  - We need some condition to be true to keep running
- do while
  - Like while, but we want our code to run *at least once*

Loops - for

```
for (initialization; condition; update)
{
    execute this code
}
```

Loops - while

```
initialization
while (condition)
{
    execute this code
    update
}
```

Loops – do while

```
initialization
do
{
    execute this code
    update
}
while (condition);
```

Functions

- Some functions we've seen already
  - main
    - `int main(int argc, string argv[])`
  - `printf, GetInt, toupper`
    - These have been implemented for us already
- But now you can write your own!

Functions

return type

function name

parameter list

```
int cube(int input)
{
    int output = input * input * input;
    return output;
}
```

Functions – Why?

- Organization. Functions help to break up a complicated problem into more manageable subparts and help to make sure concepts flow logically into one another.

- Simplification. Smaller components are easier to design, easier to implement, and far easier to debug. Good use of functions makes code easier to read and problems easier to isolate.

- Reusability. Functions only need to be written once, and then can be used as many times as necessary, so you can avoid duplication of code.

Threads

- Threads are the concept of multiple sequences of code executing at the same time

- In Scratch, for example, multiple sprites execute scripts simultaneously
- Original example in class where we counted the number of people in the room

Events

- Events are the concept of different parts of your code “communicating” with each other

- In Scratch, this is the Broadcast/When I Receive blocks
- In Problem Set 4, `Gevent` (`waitForClick`)

Linux

- `ls`
  - stands for "list," shows the contents of the current directory
- `mkdir`
  - stands for "make directory," creates a new folder
- `cd`
  - stands for "change directory," the equivalent of double clicking on a folder
- `rm`
  - stands for "remove," deletes a file
- `rmdir`
  - stands for "remove directory," deletes a directory

Libraries

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

- what's in the stdio library?

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

- what's in the cs50 library?

Libraries

`#include <stdio.h>`

- what's in the stdio library?
  - `printf`

`#include <cs50.h>`

- what's in the cs50 library?
  - `GetInt()`, `GetString()`, etc.
  - `string`

Types

int	4 bytes
char	1 byte
float	4 bytes
double	8 bytes
long	4 bytes
long long	8 bytes
char*,int*, etc.	4 bytes

Standard Output

The printf function can take many different format codes:

- %c for char
- %i for int
- %f for float
- %lld for long long
- %s for string

A few escape sequences:

- \n for newline
- \r for carriage return (think typewriter)
- \' for single quote
- \" for double quote
- \\ for backslash
- \0 for NUL terminator

Casting

A way to treat a value as another type

char to int
float to int
long long to double

Strange Behavior

```
float f = 1.31;
int n = (int) (f * 10000);
printf("%i\n", n);
```

What does this output?

13099

Why?

Imprecision

Floats aren’t perfect.

Can only represent numbers to a certain number of significant figures

```
float f = 1.31;
printf("%.8f\n", f);
```

What does this output?

1.30999994

Switches

```
printf("Give me a number between 1 and 4\n");
int n = GetInt();
switch (n)
{
     case 1:
          printf("Low\n");
          break;
     case 2:
     case 3:
          printf("Middle\n");
          break;
     case 4:
          printf("High\n");
          break;
     default:
          printf("Wrong\n");
          break;
}
```

Scope

The range that a declared variable extends

```
for (int i = 0; i < 10; i++)
{
   // STUFF
}

printf("%d\n", i);
```

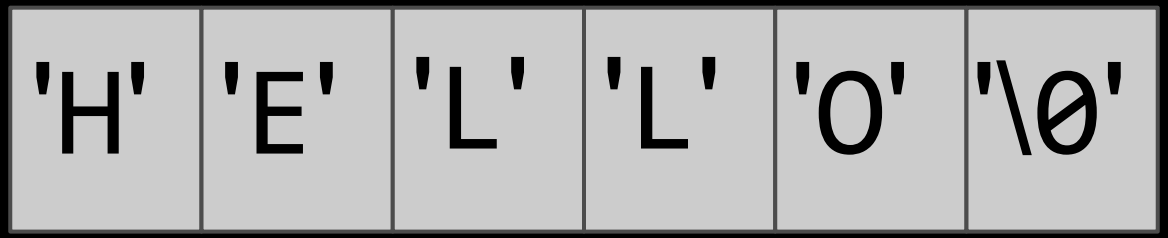
Strings

string is char*
ends with '\0'

NULL != '\0'

Arrays

Continuous blocks of memory
Instant access — `name[index]`
Zero-Indexed
Declared `type name[size]`



Command-Line Arguments

Gets input from the user as arguments to main

```
int main(int argc, string argv[])
```

argc is the number of arguments

argv is the array of arguments (last is `NULL`)

Security

To be truly secure, you rely on no one, and you allow no one access to any of your information

Which is why everyone builds their own computers, operating systems, and programs from scratch, and don’t connect to any other machine

Cryptography

We have secrets

Sometimes we have to move our secrets through insecure channels

We want them to stay secret

So, we encrypt them

Debugging

GDB is the best

Commands include:
break
print
next
step

Searching

Linear search:

Look through the search space one element at a time

Binary search (needs sorted elements):

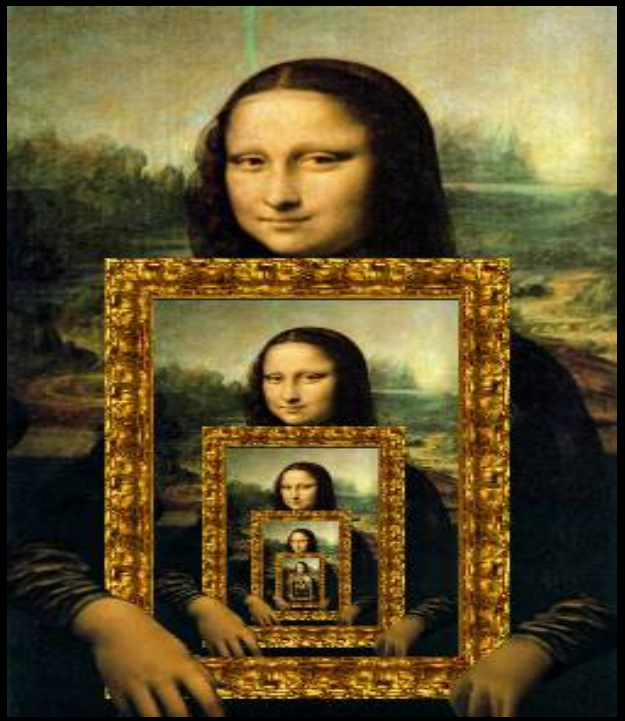
Go to the middle of the elements

See if the element you're looking for is larger or smaller

Reduce the search space accordingly

Repeat

Recursion



Adding all numbers from 1 to n

$$sum(n) = n + \mathbf{(n - 1) + (n - 2) + \ldots + 1}$$

Adding all numbers from 1 to n

$$sum(n) = n + \mathbf{(n - 1) + (n - 2) + \ldots + 1}$$

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

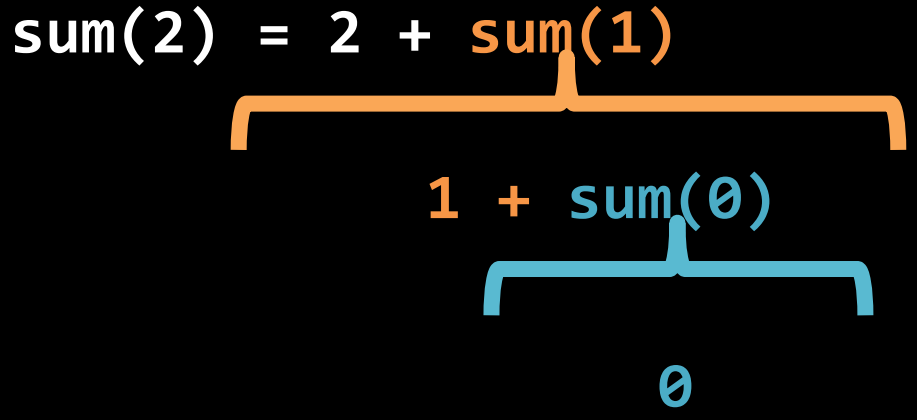
Adding all numbers from 1 to n

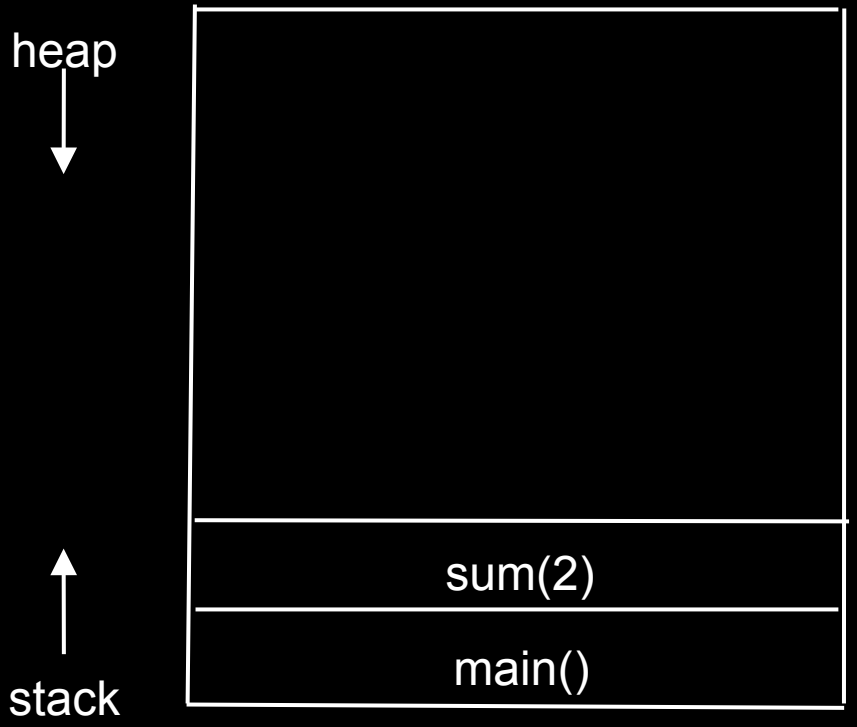
sum(n) = n + **(n - 1) + (n - 2) + ... + 1**

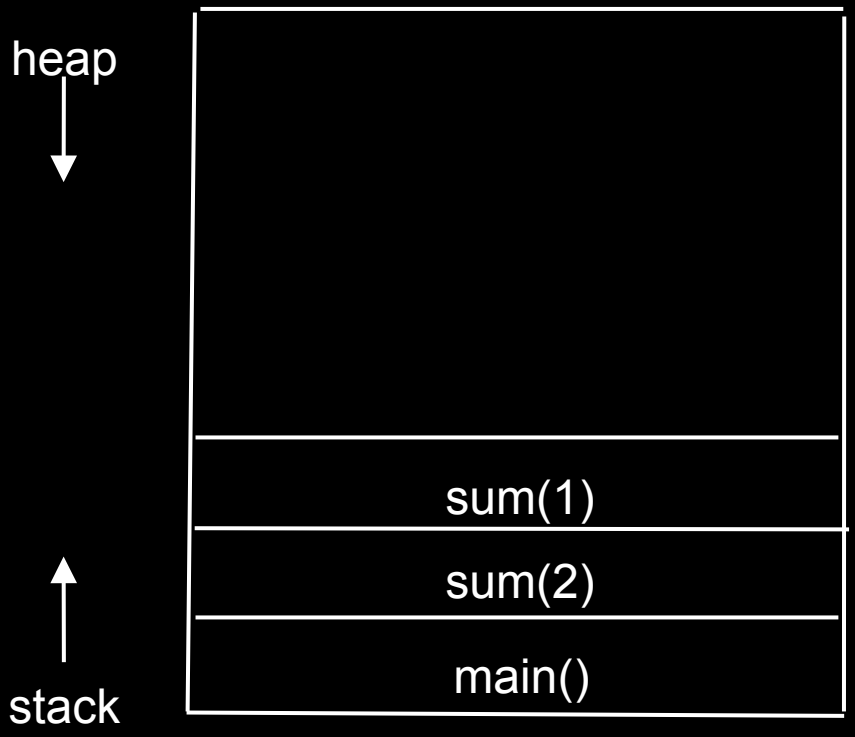
sum(n) = n + **sum(n - 1)**

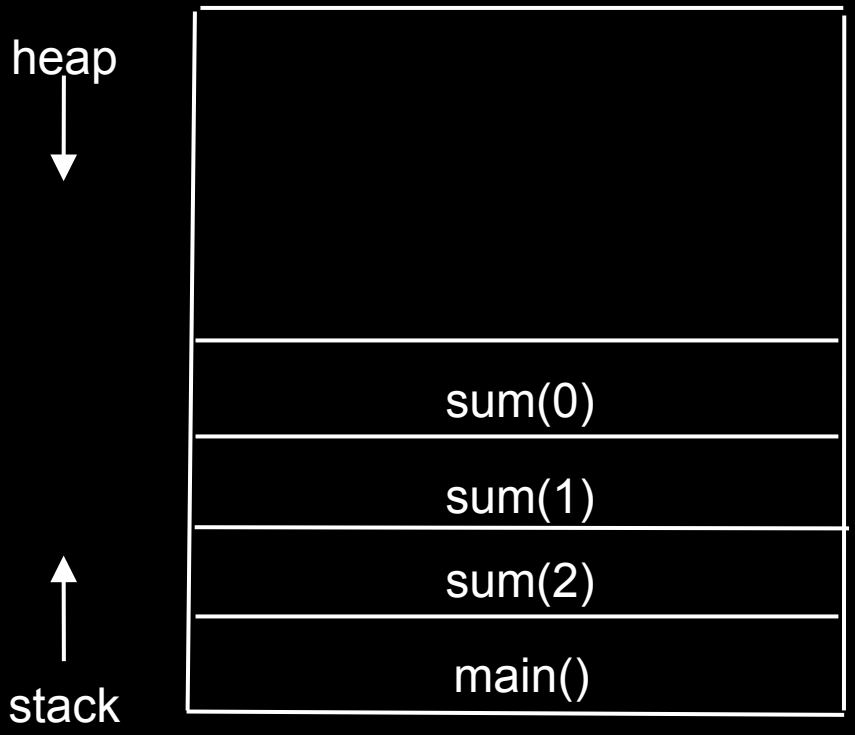
sum(0) = 0 (base case)

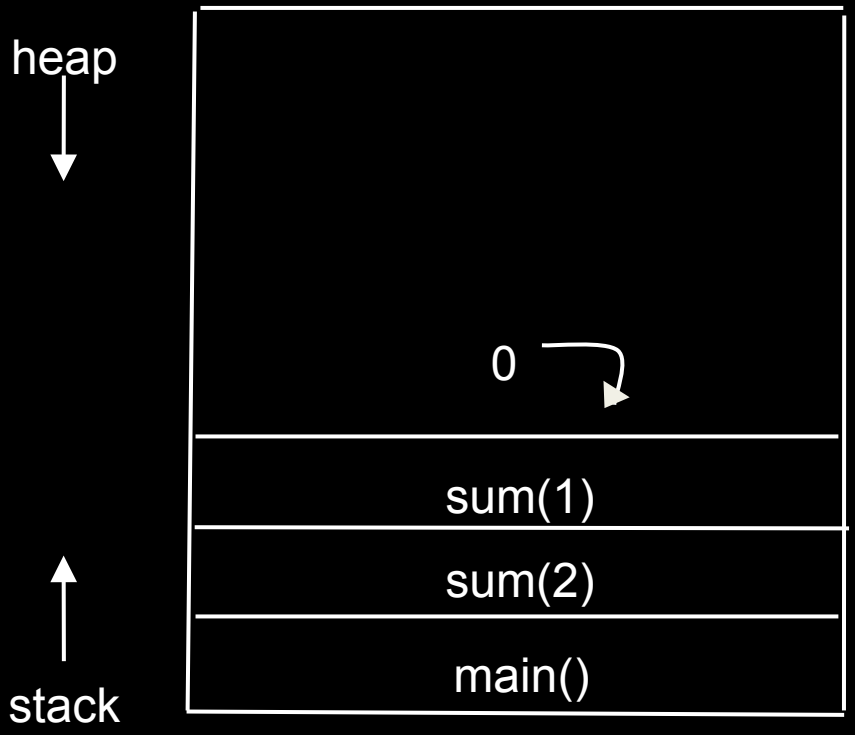
```
int sum(int n)
{
    if (n <= 0)
    {
        return 0;
    }
    else
    {
        return n + sum(n - 1);
    }
}
```

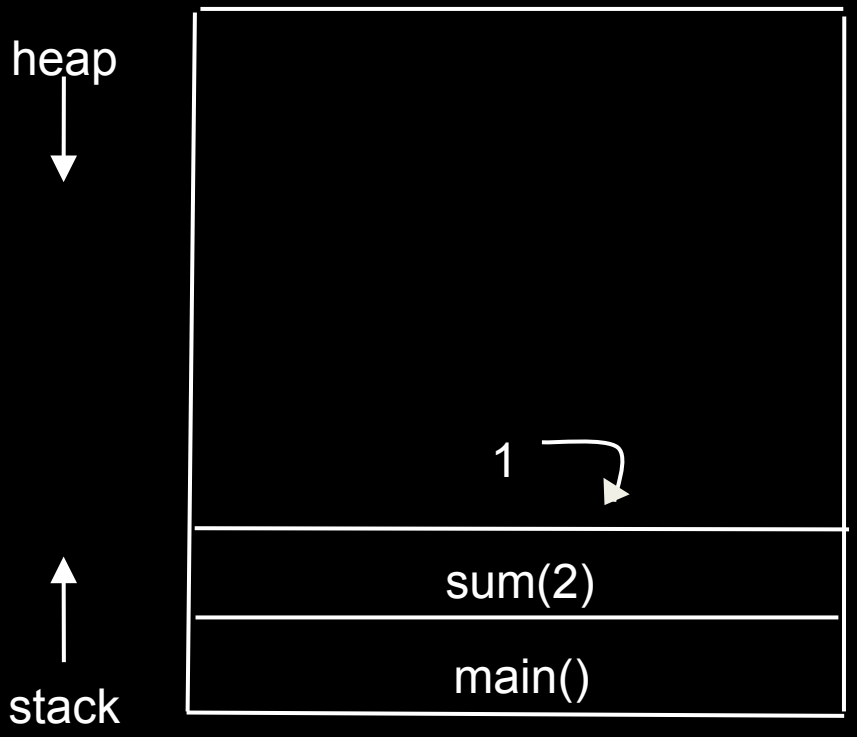

sum(2) = 2 + sum(1)
1 + sum(0)
0

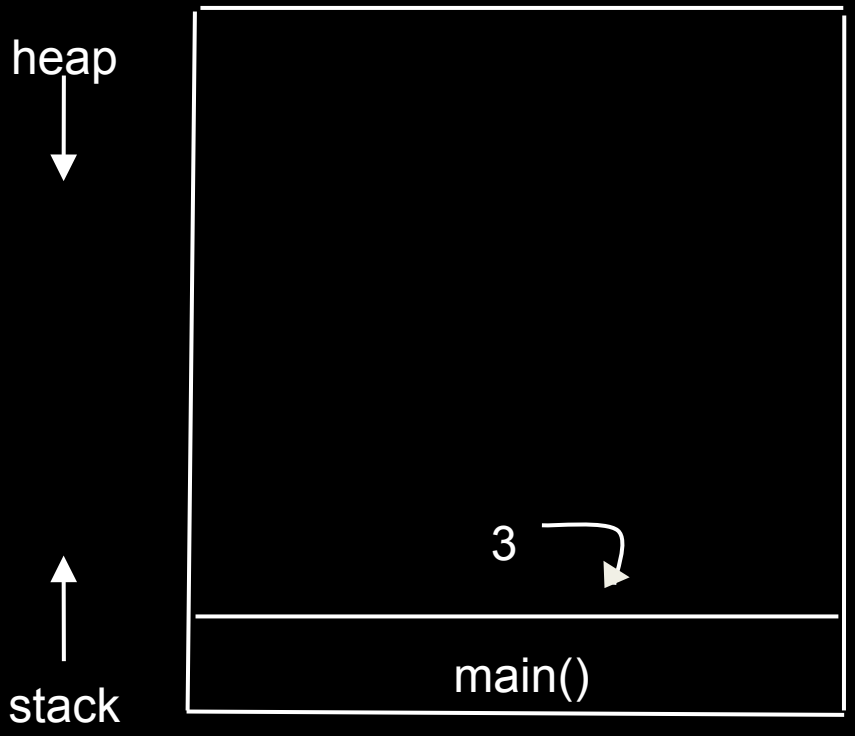

heap
sum(2)
main()
stack


heap
sum(1)
sum(2)
main()
stack


heap
sum(0)
sum(1)
sum(2)
main()
stack


heap
0
sum(1)
sum(2)
main()
stack


heap
1
sum(2)
main()
stack


heap
3
main()
stack

Sorting

- Dictionary
- Facebook
- Organizing data
- Pokémon!

Selection Sort

1. **Select** the smallest unsorted value

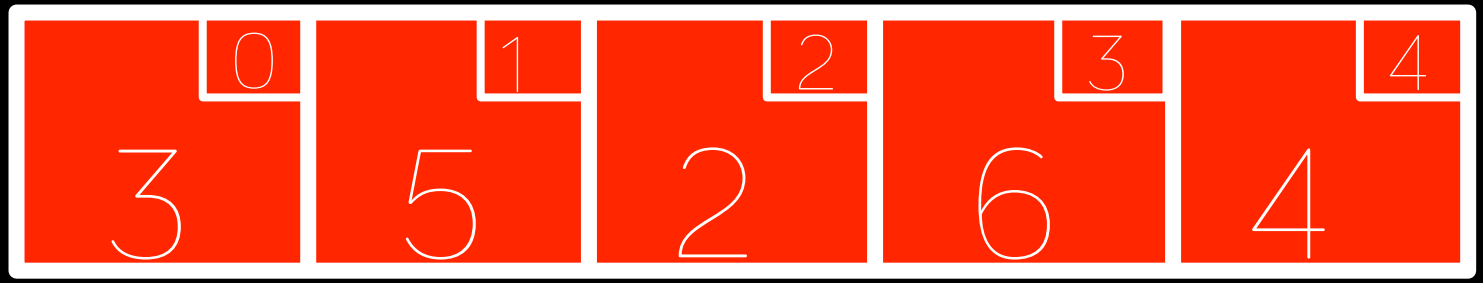
2. Move that value to the end of the “sorted” part of the list

3. Repeat from step 1 if there are still unsorted items

All values start as Unsorted

Sorted

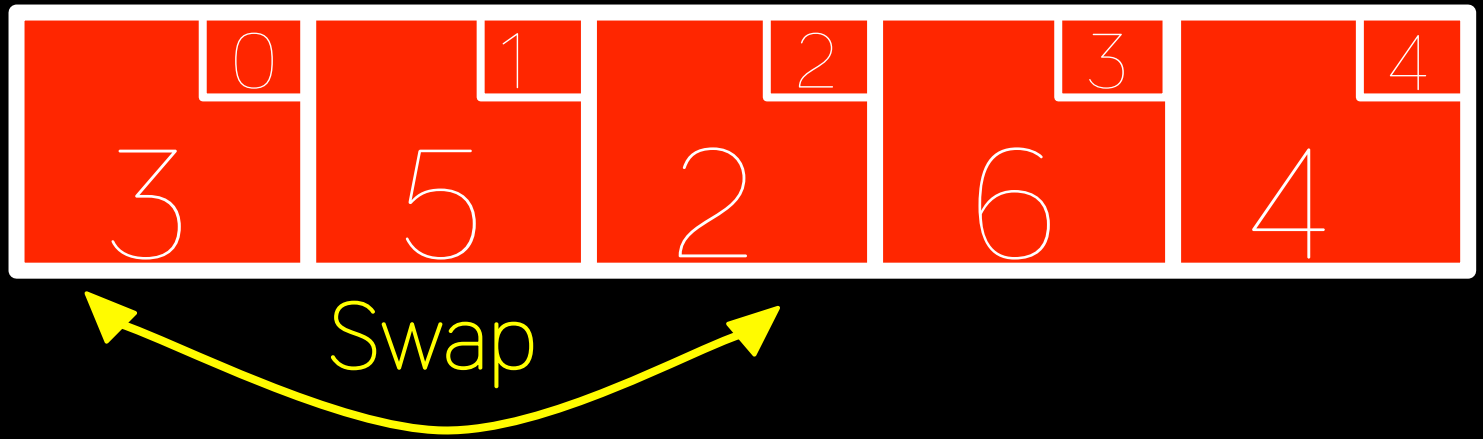
Unsorted



First pass:
2 is smallest, swap with 3

Sorted

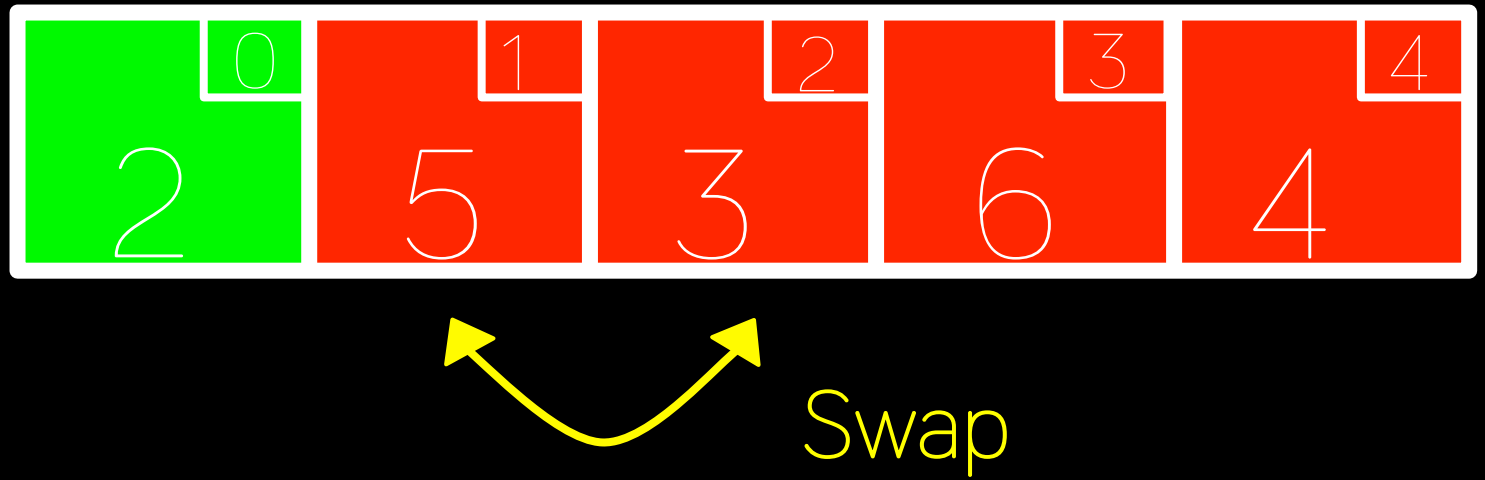
Unsorted



Second pass:
3 is smallest, swap with 5

Sorted

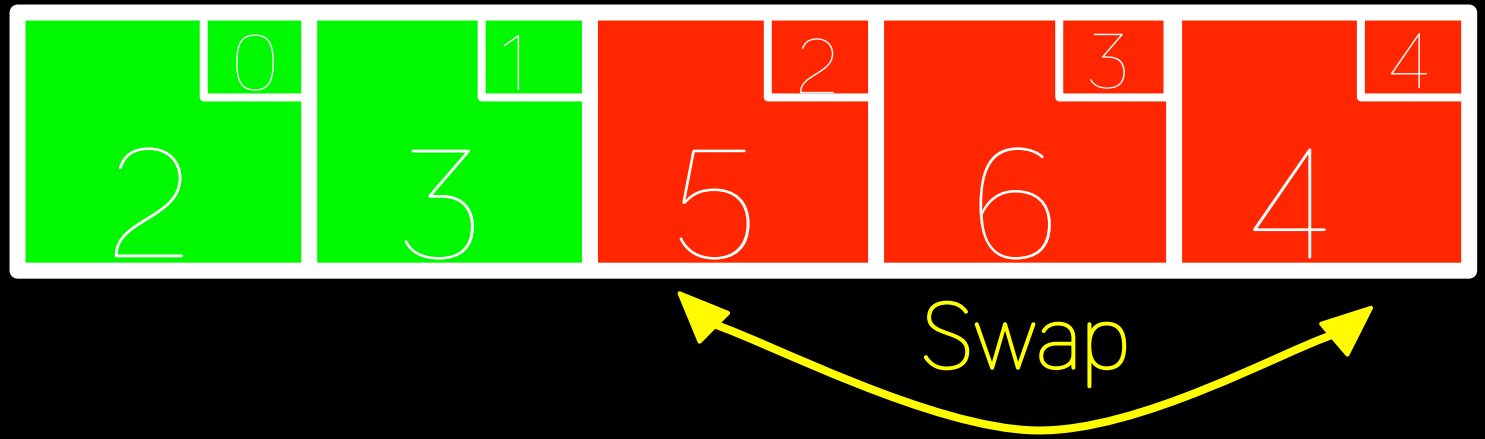
Unsorted



Third pass:
4 is smallest, swap with 5

Sorted

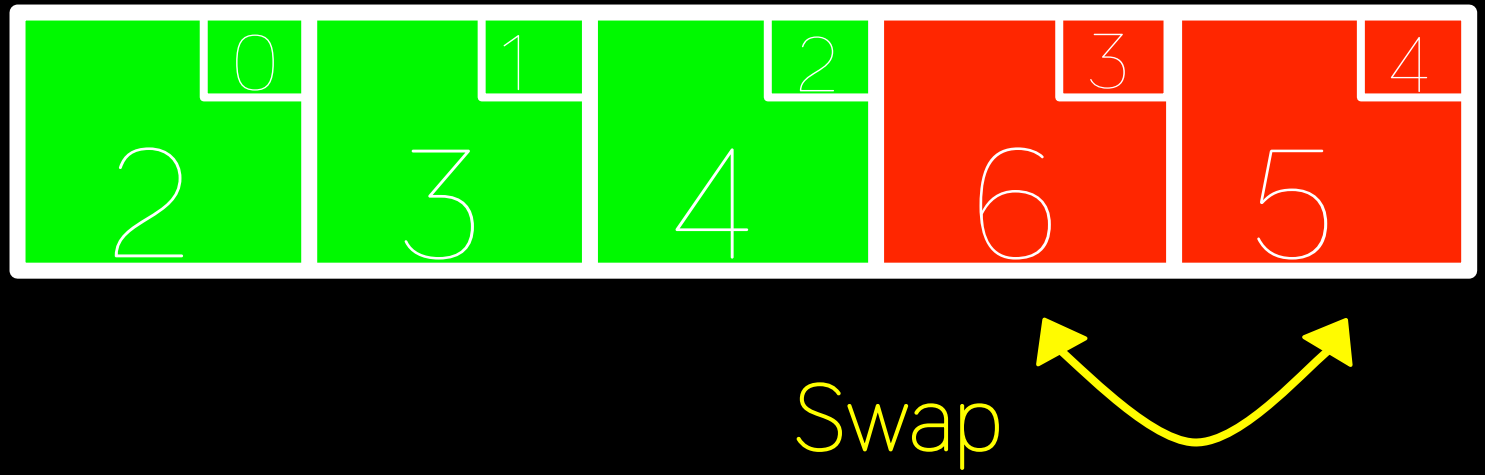
Unsorted



Fourth pass:
5 is smallest, swap with 6

Sorted

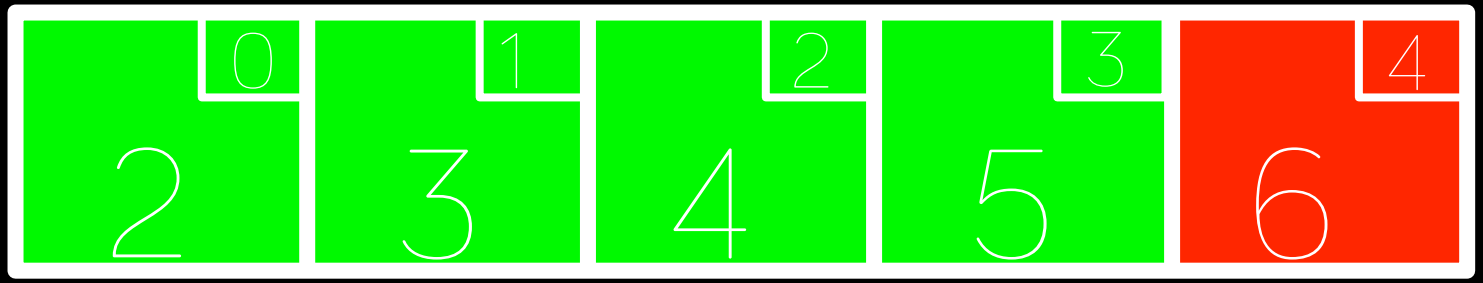
Unsorted



Fifth pass:
6 is the only value left, done!

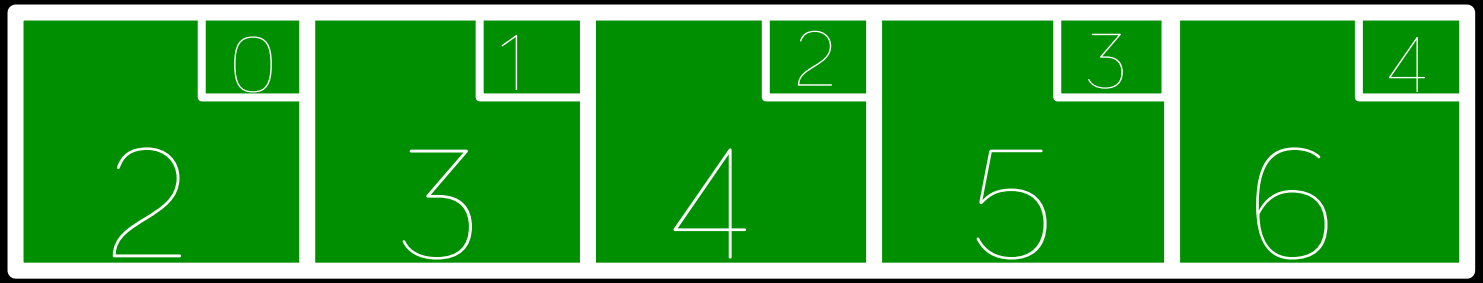
Sorted

Unsorted



Fifth pass:
6 is the only value left, done!

All Sorted



Insertion Sort

For each unsorted element n:

1. Determine where to insert n on the sorted portion of the list

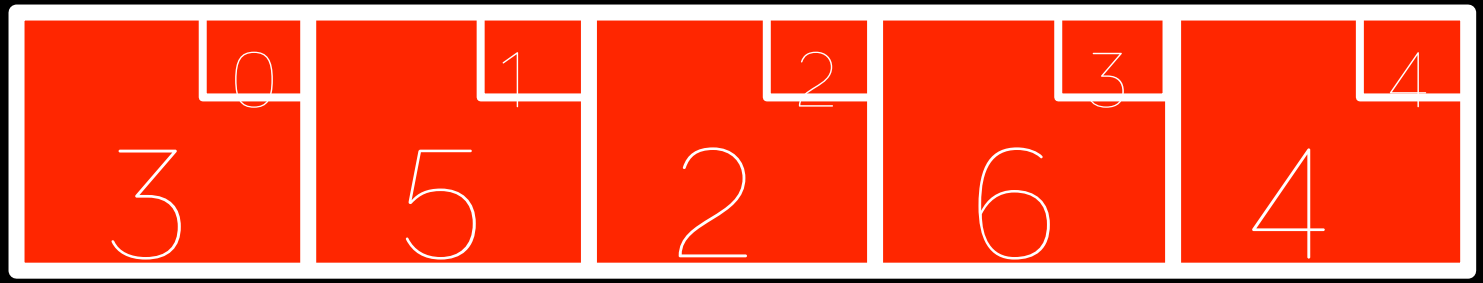
2. Shift sorted elements rightwards as necessary to make room for n

3. **Insert** n into sorted portion of the list

All values start as Unsorted

Sorted

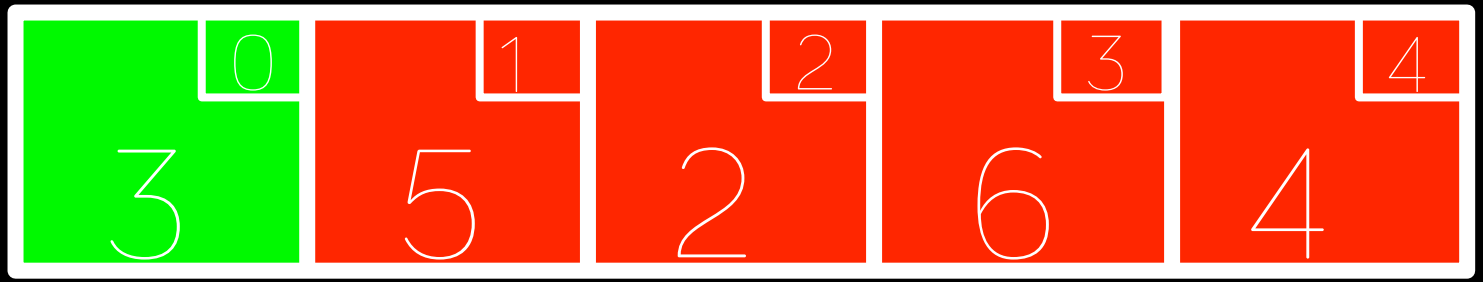
Unsorted



Add first value to Sorted

Sorted

Unsorted

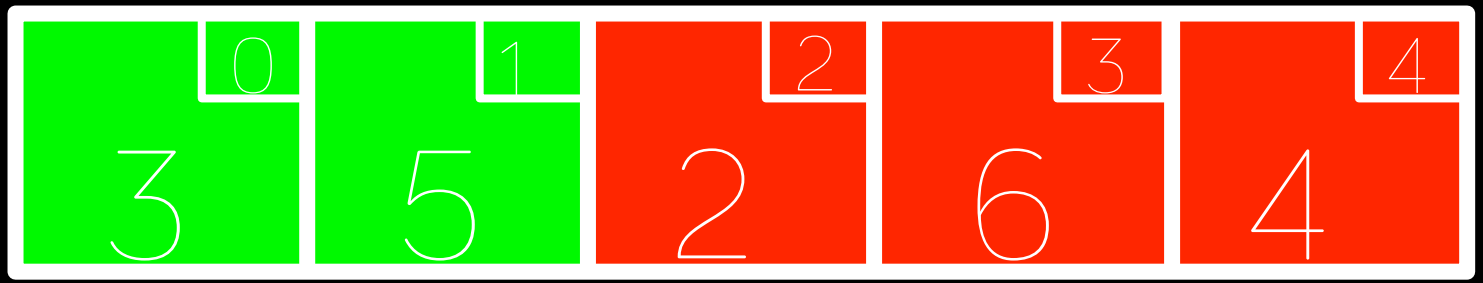


5 > 3

insert 5 to right of 3

Sorted

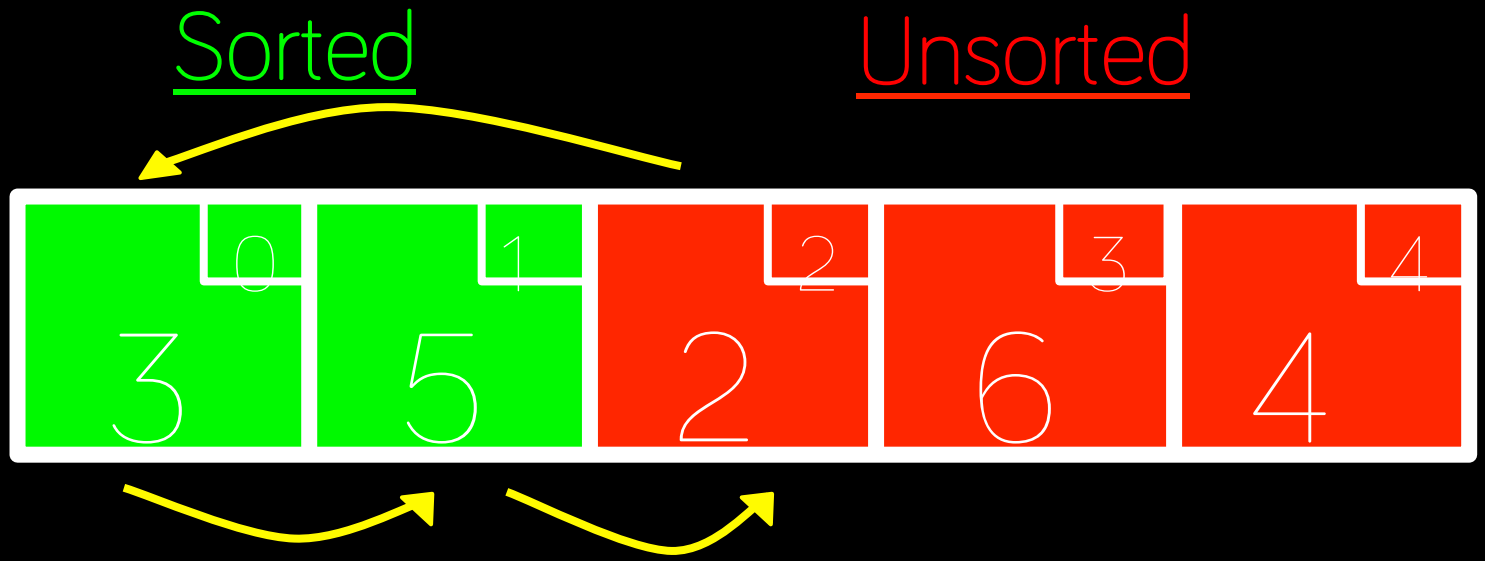
Unsorted



$2 < 5$ and $2 < 3$

shift 3 and 5

insert 2 to left of 3

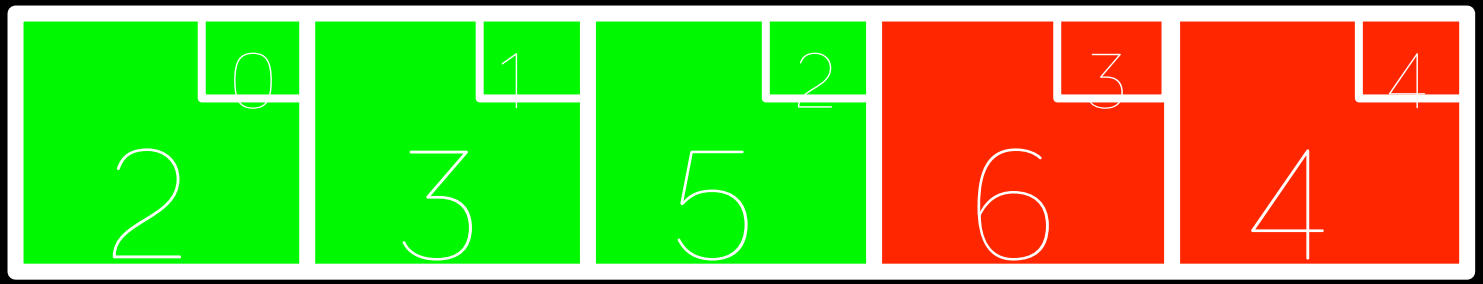


6 > 5

insert 6 to right of 5

Sorted

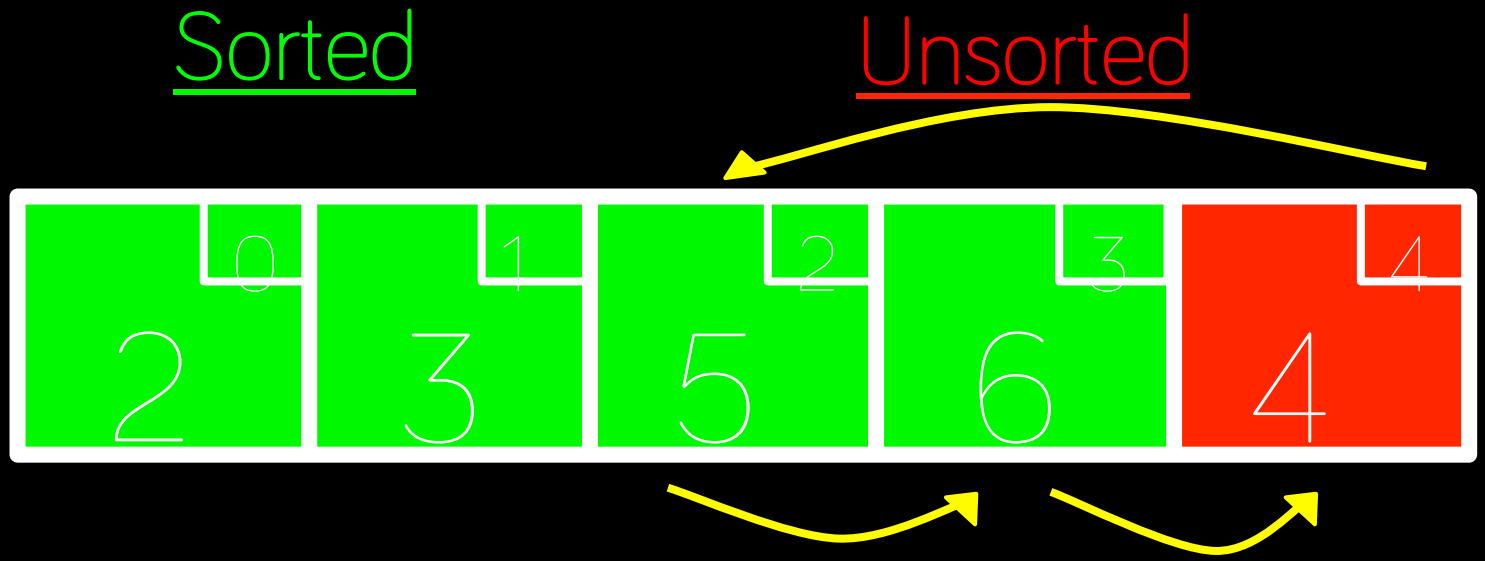
Unsorted



4 < 6, 4 < 5, and 4 > 3

shift 5 and 6

insert 4 to right of 3

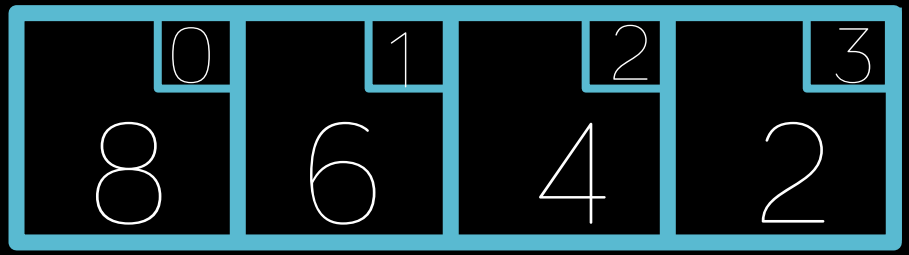


Bubble Sort

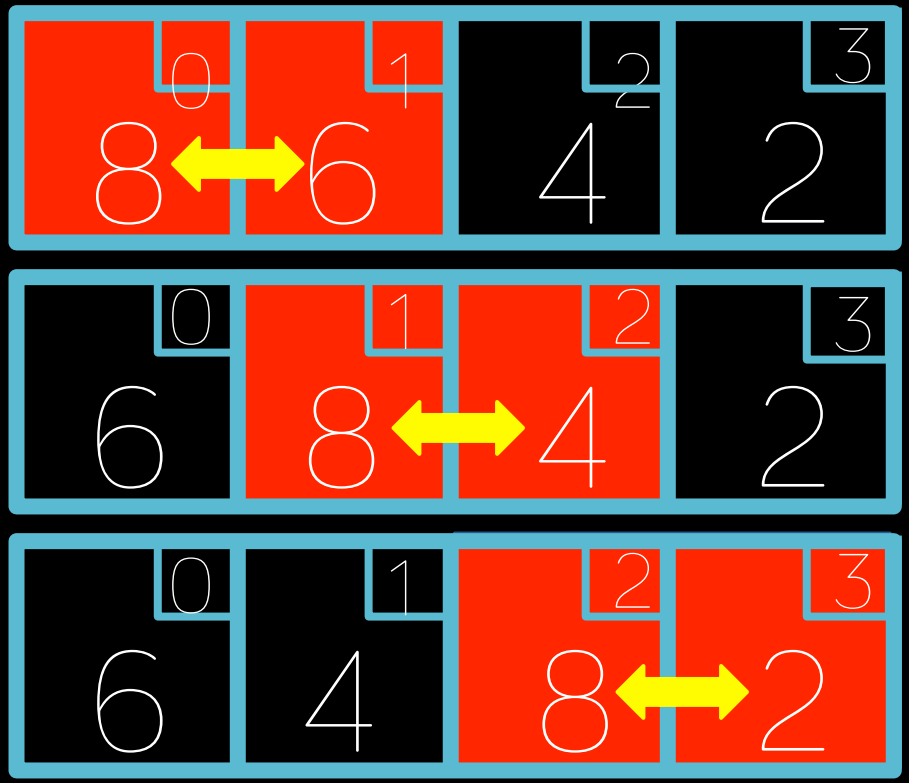
1. Step through entire list, swapping adjacent values if not in order

2. Repeat from step 1 if any swaps have been made

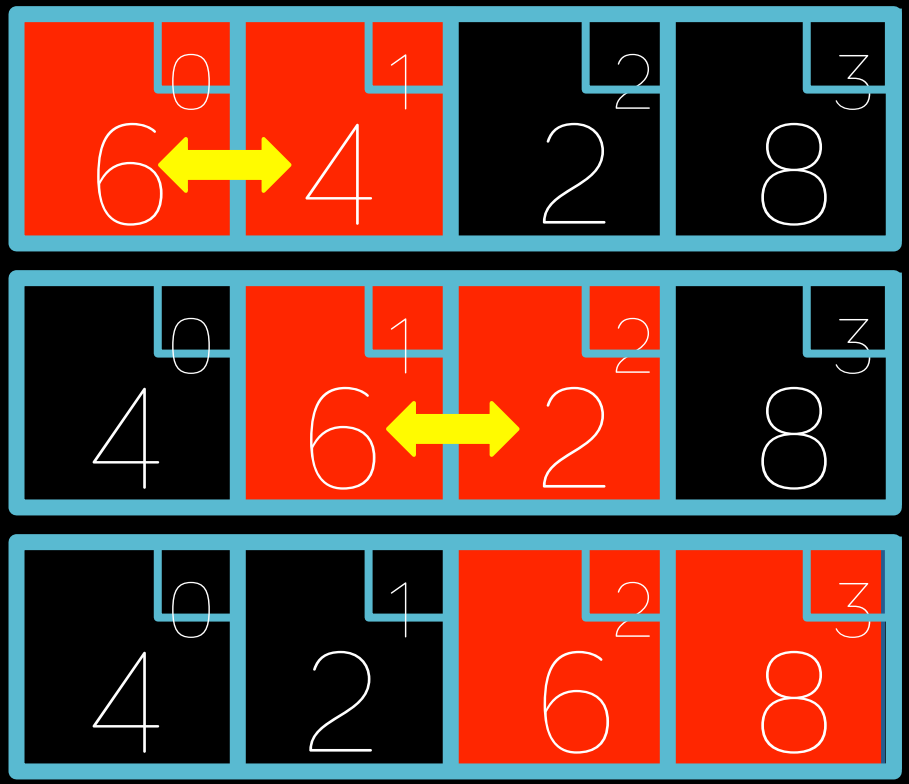
At each step, the largest value **bubbles** to the end of the list


0
1
2
3
8
6
4
2

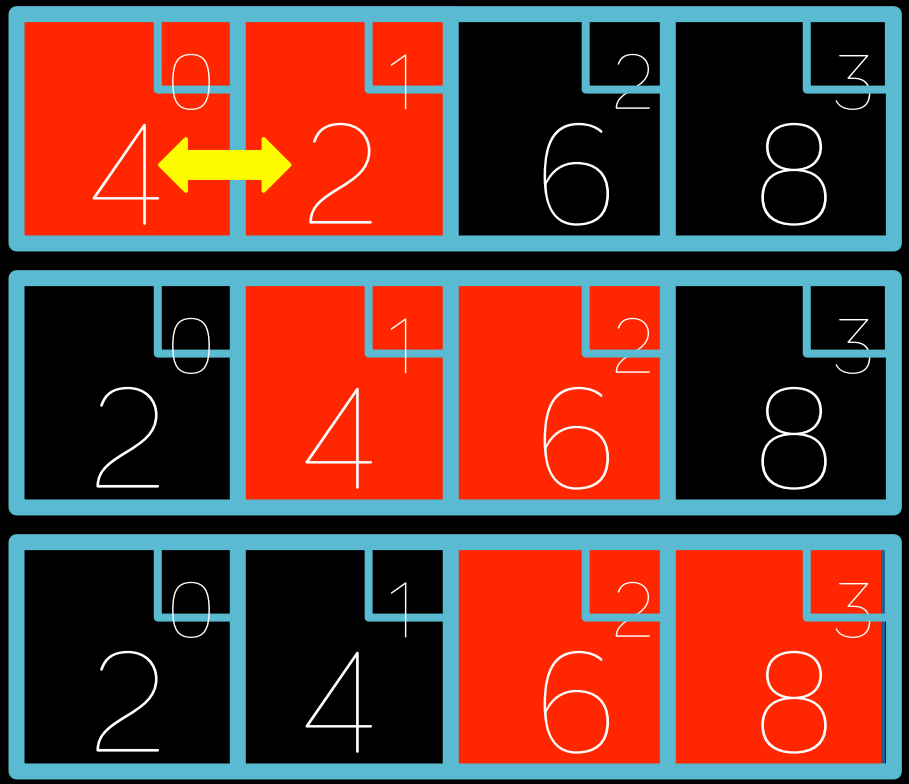
First pass: 3 swaps



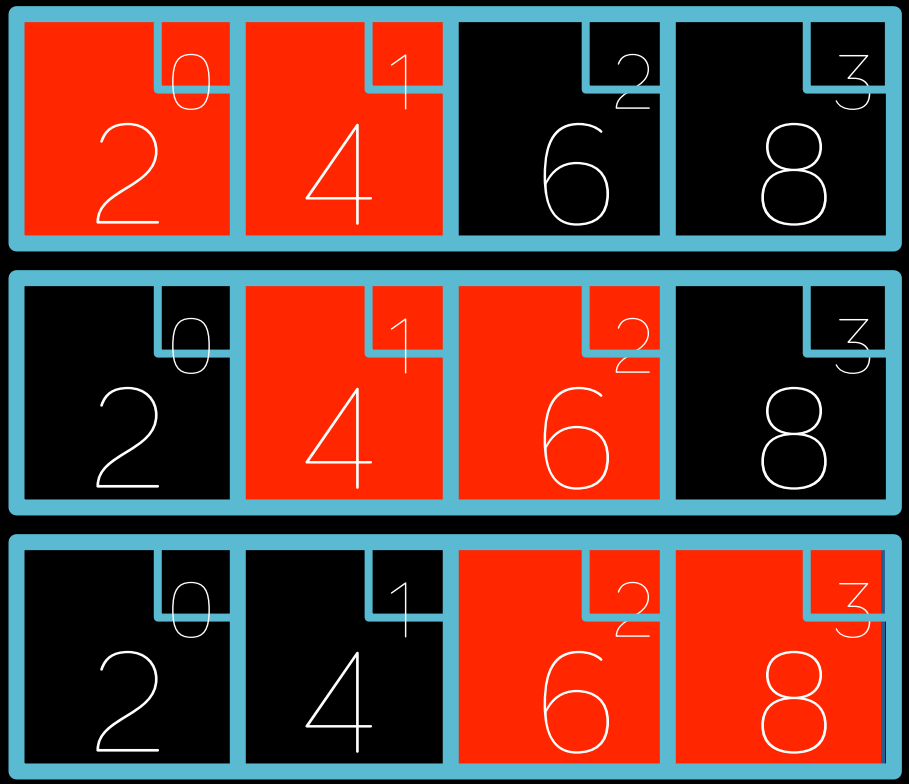
Second pass: 2 swaps



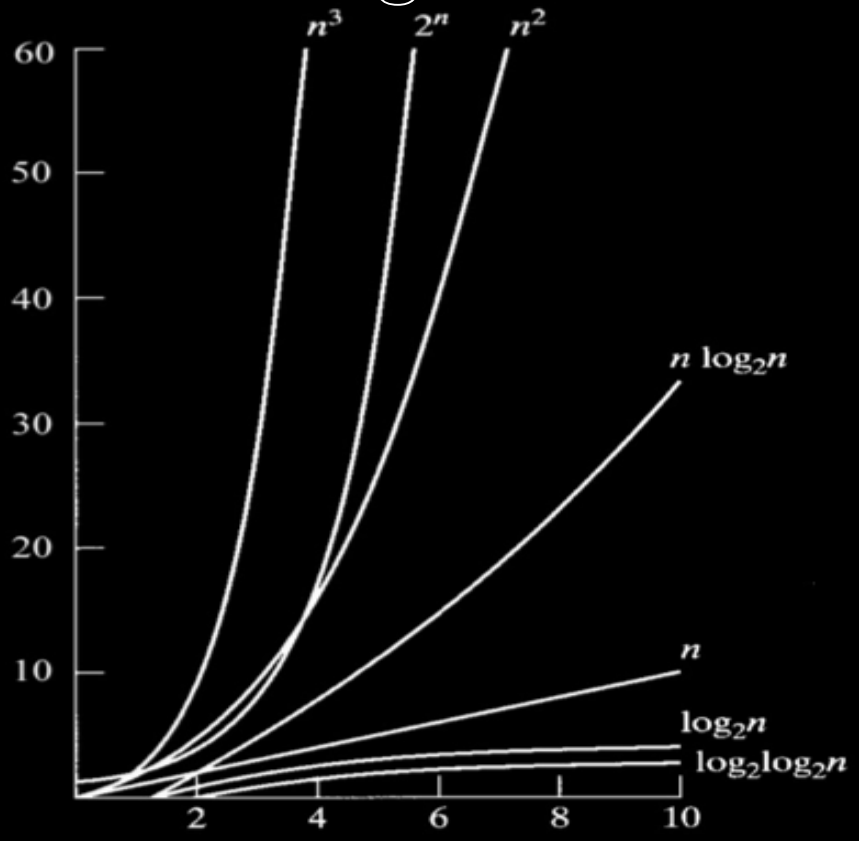
Third pass: 1 swap



Fourth pass: 0 swaps



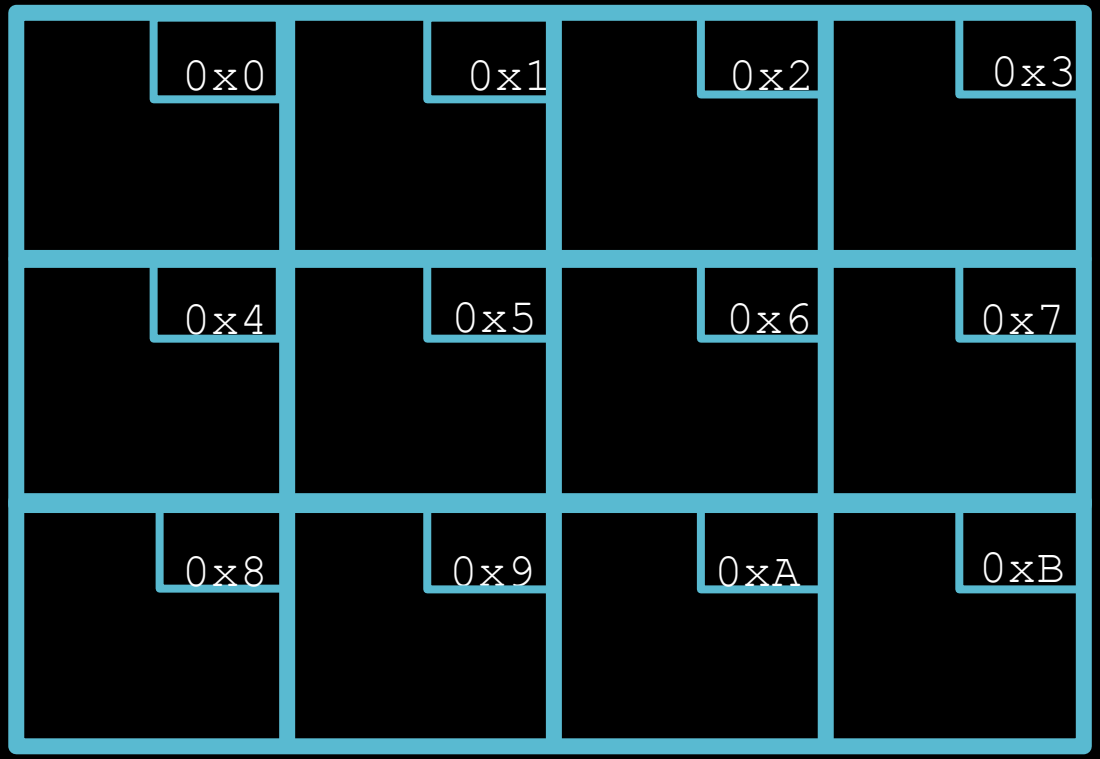
Running Time



	O	Ω	Θ
Bubble sort	n^2	n	
Selection sort	n^2	n^2	n^2
Insertion sort	n^2	n	
Merge sort	$n\log n$	$n\log n$	$n\log n$

Pointers

Memory



Creating Pointers

Declaring pointers:

<type>* <variable name>

Examples:

```
int* x;
char* y;
float* z;
```

Size: 4 bytes for 32-bit machine

Referencing and Dereferencing

Referencing:
&<variable name>
& is the same as saying “address of”

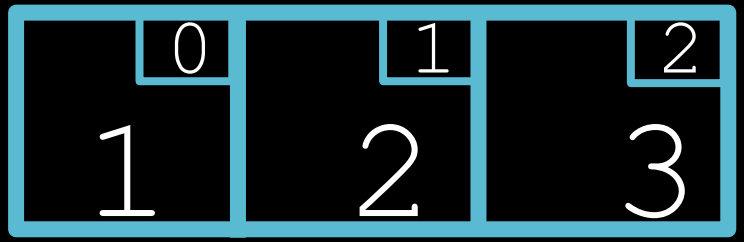
Dereferencing:
*<pointer name>
***** is the same as saying “content of”

Let's see this in memory

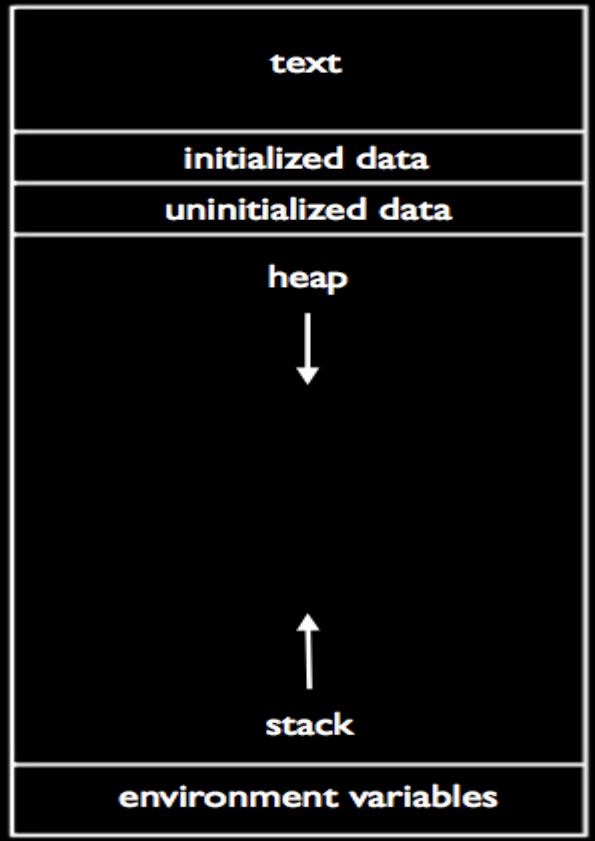
```
int a = 3;
int b = 4;
int c = 5;
int* pa = &a;
int* pb = &b;
int* pc = &c;
```

Pointers and Arrays

```
int array[3];

*array = 1;
*(array + 1) = 2;
*(array + 2) = 3;
```



Dynamic Memory Allocation



m(emory)alloc(ation)

- Allocates memory on the heap
- Returns pointer to allocated memory
- Returns **`NULL`** if can't allocate memory

malloc

```
void* malloc(size in bytes);
```

example:

```
int* ptr;
ptr = malloc(sizeof(int) * 10);
```

Check for NULL!

```
int* ptr = malloc(sizeof(int) * 10);

if (ptr == NULL)
{
      printf("Error -- out of memory.\n");
      return 1;
}
```

Don’t forget to free!

```
free(ptr);
```

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

int main(void)
{
    int* ptr = malloc(sizeof(int));
    if (ptr == NULL)
    {
        printf("Error -- out of memory.\n");
        return 1;
    }

    *ptr = GetInt();
    printf("You entered %i.\n", *ptr);

    free(ptr);

    return 0;
}
```

Week 5 Review

Rob Bowden

Stack

- One stack frame per “active” function call
- Stores local variables and passed-in arguments

Stack

```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```

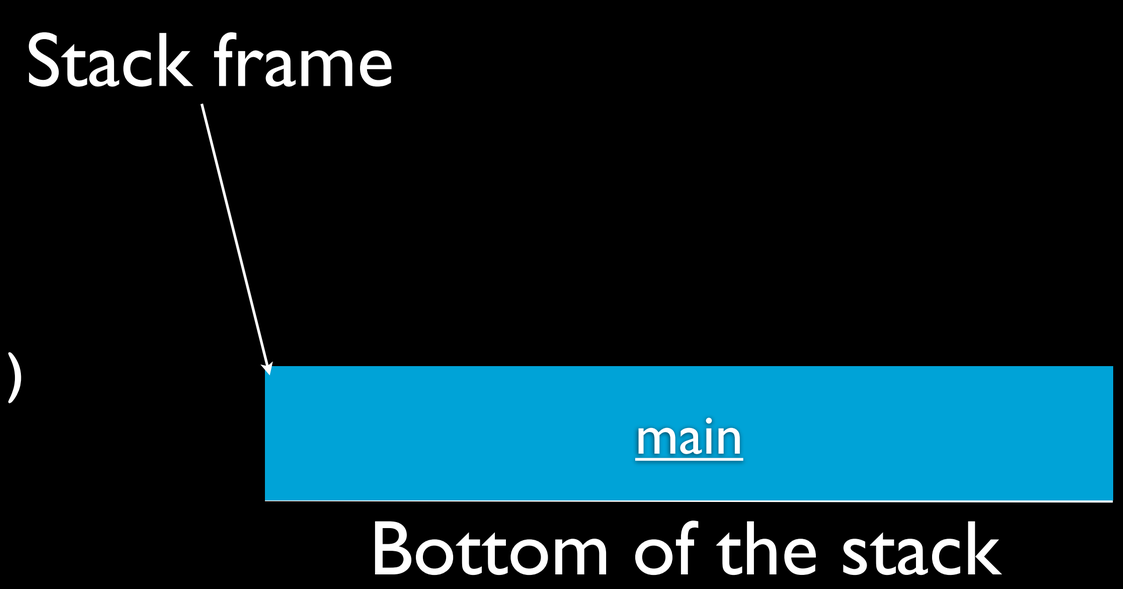
Stack

```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```

Bottom of the stack

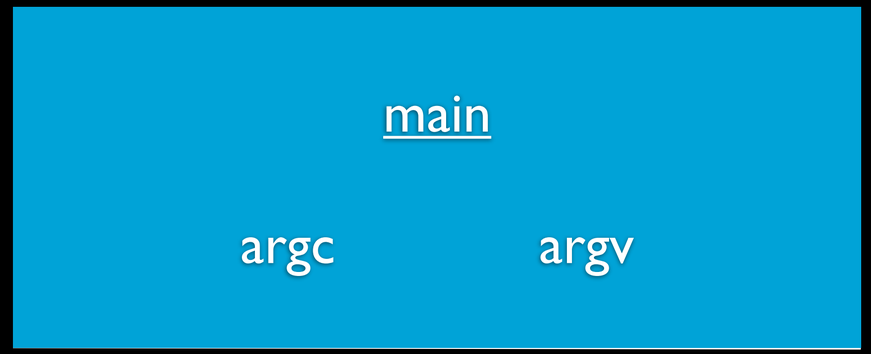
Stack

```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Stack

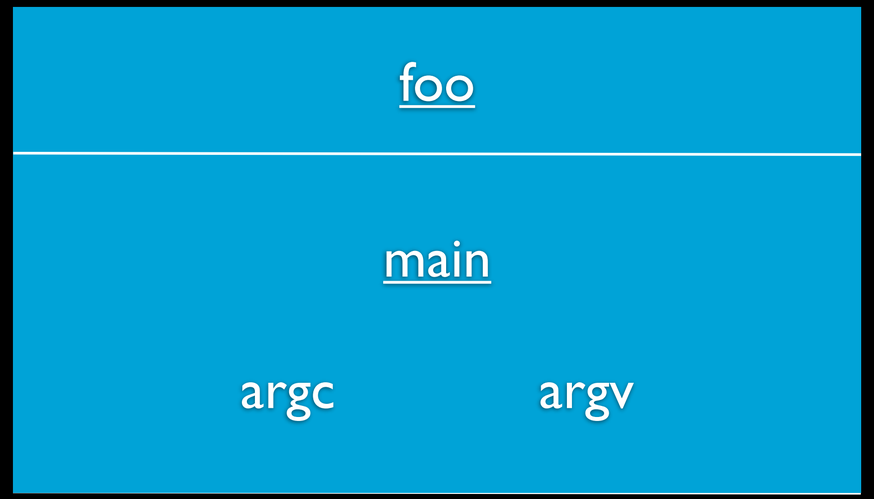
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

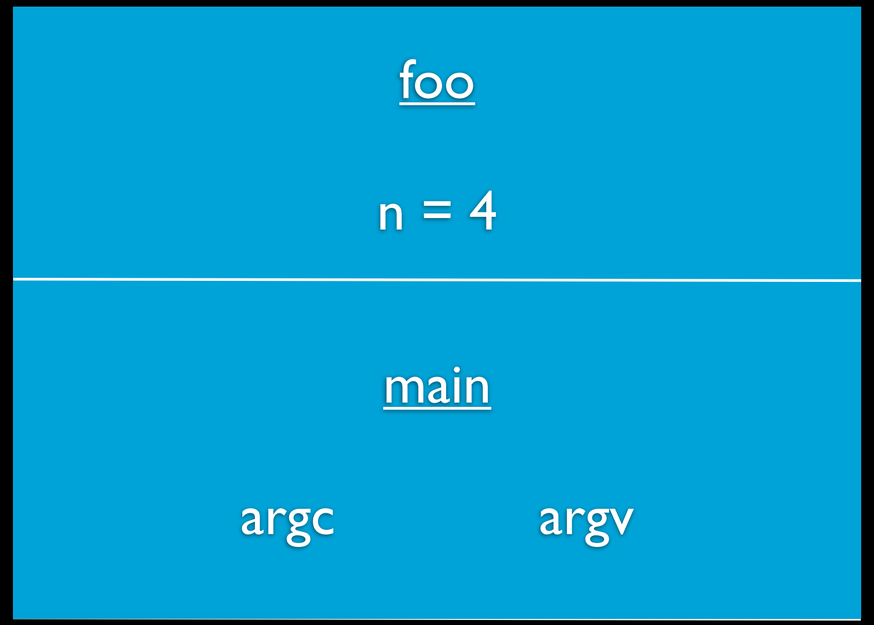
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

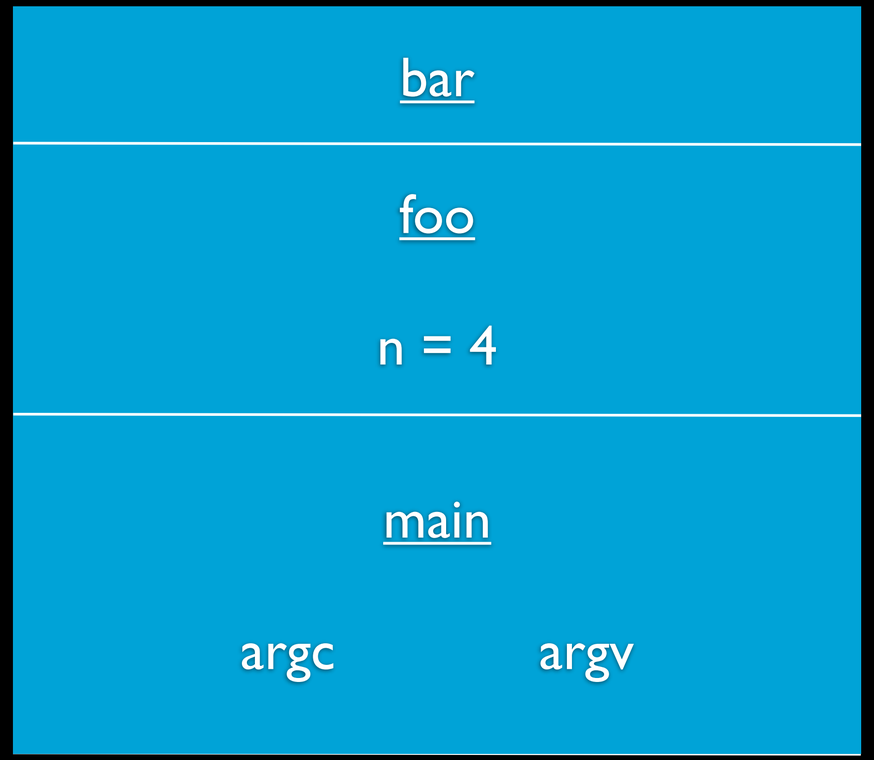
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

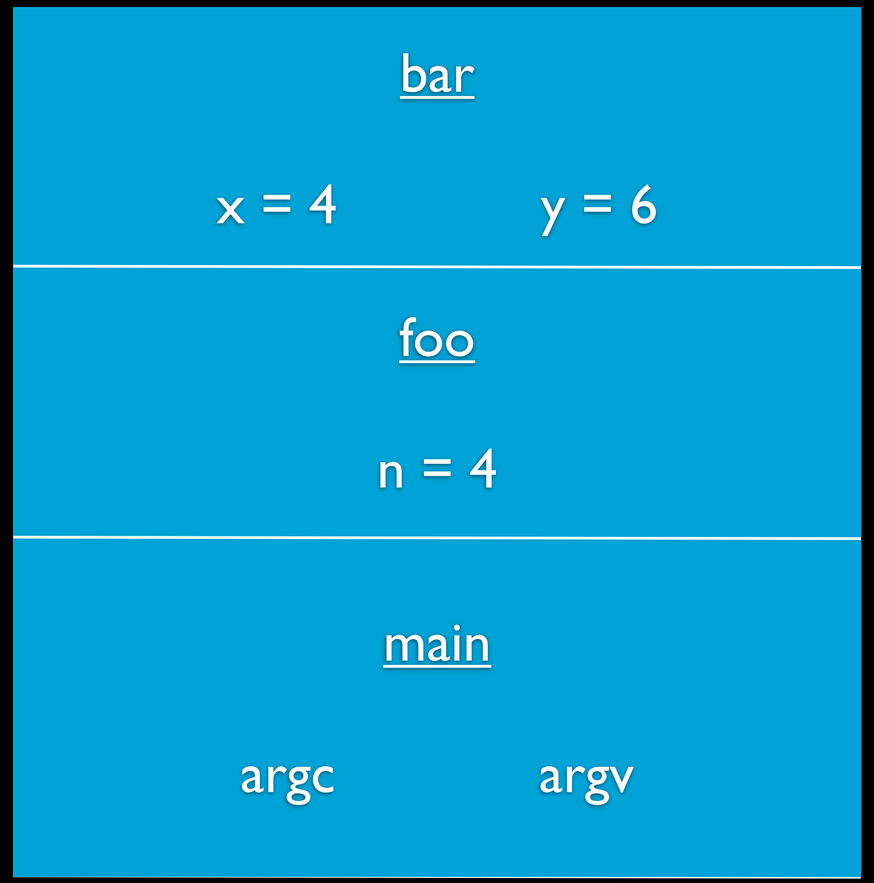
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

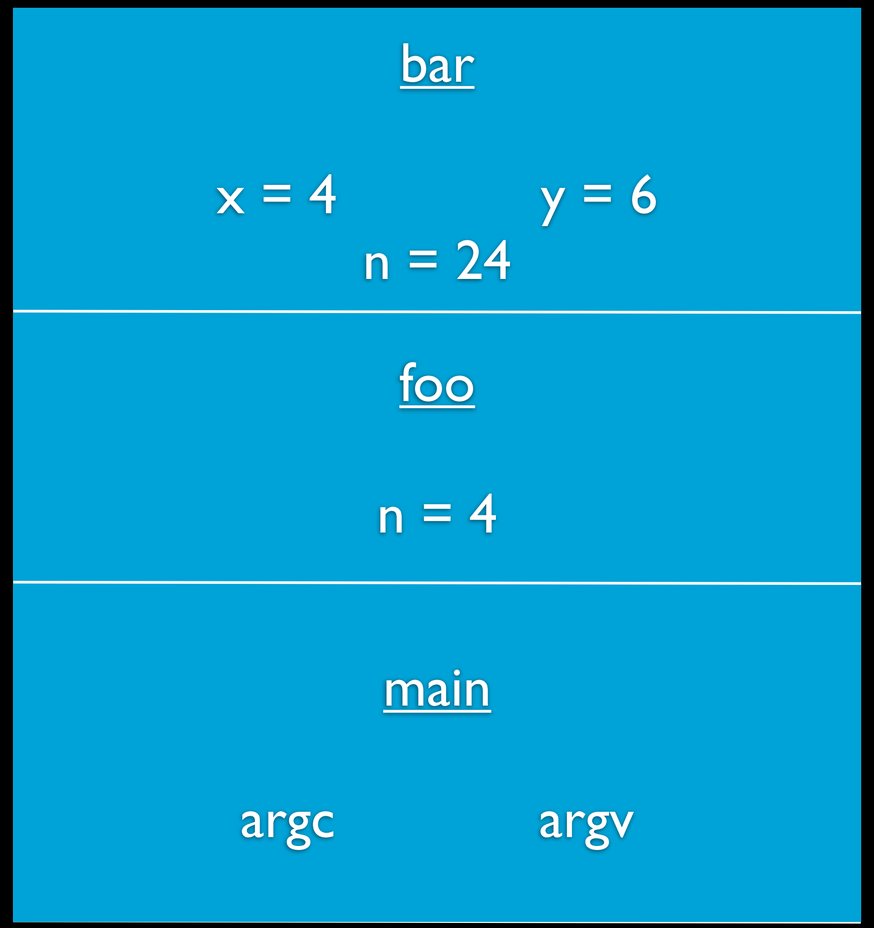
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

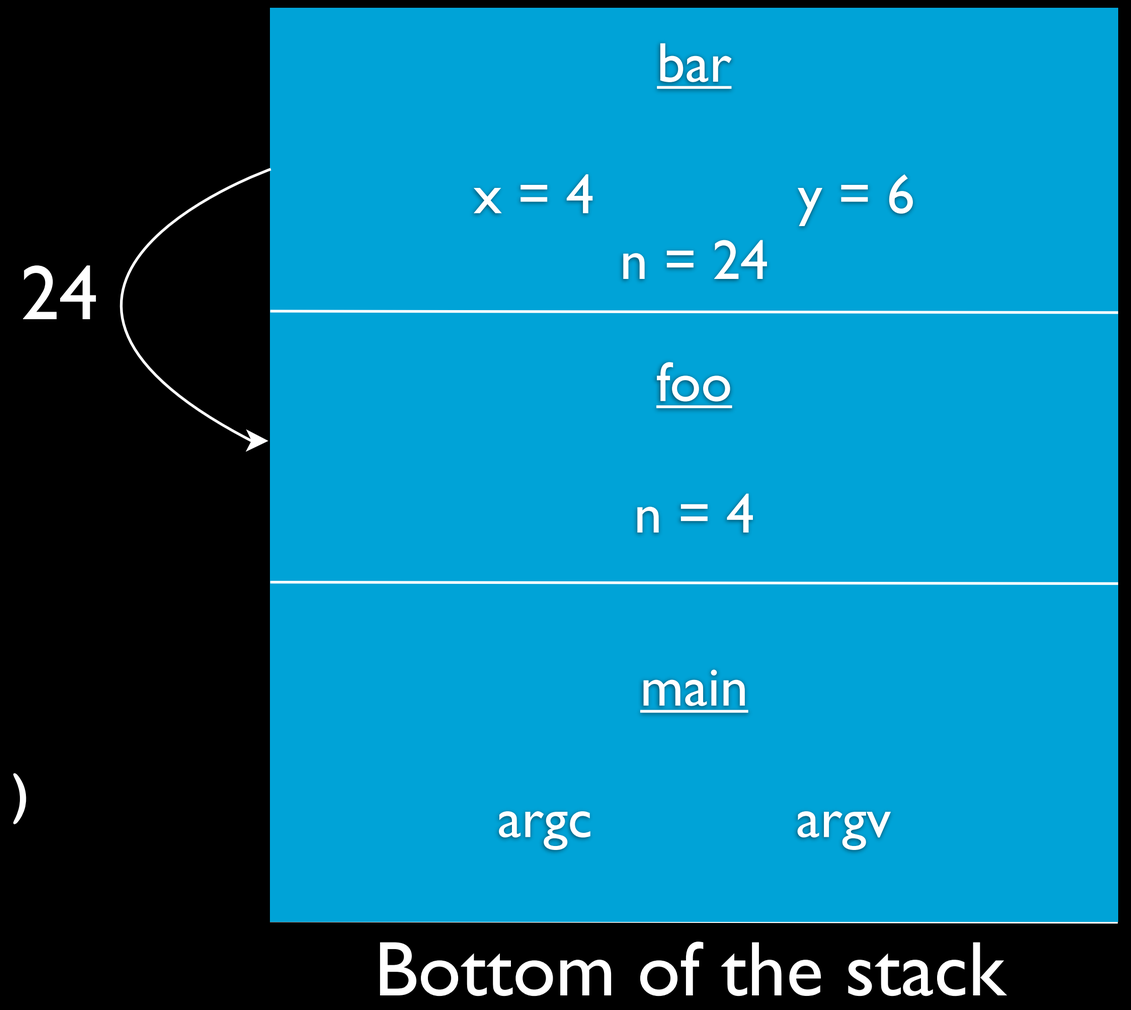
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

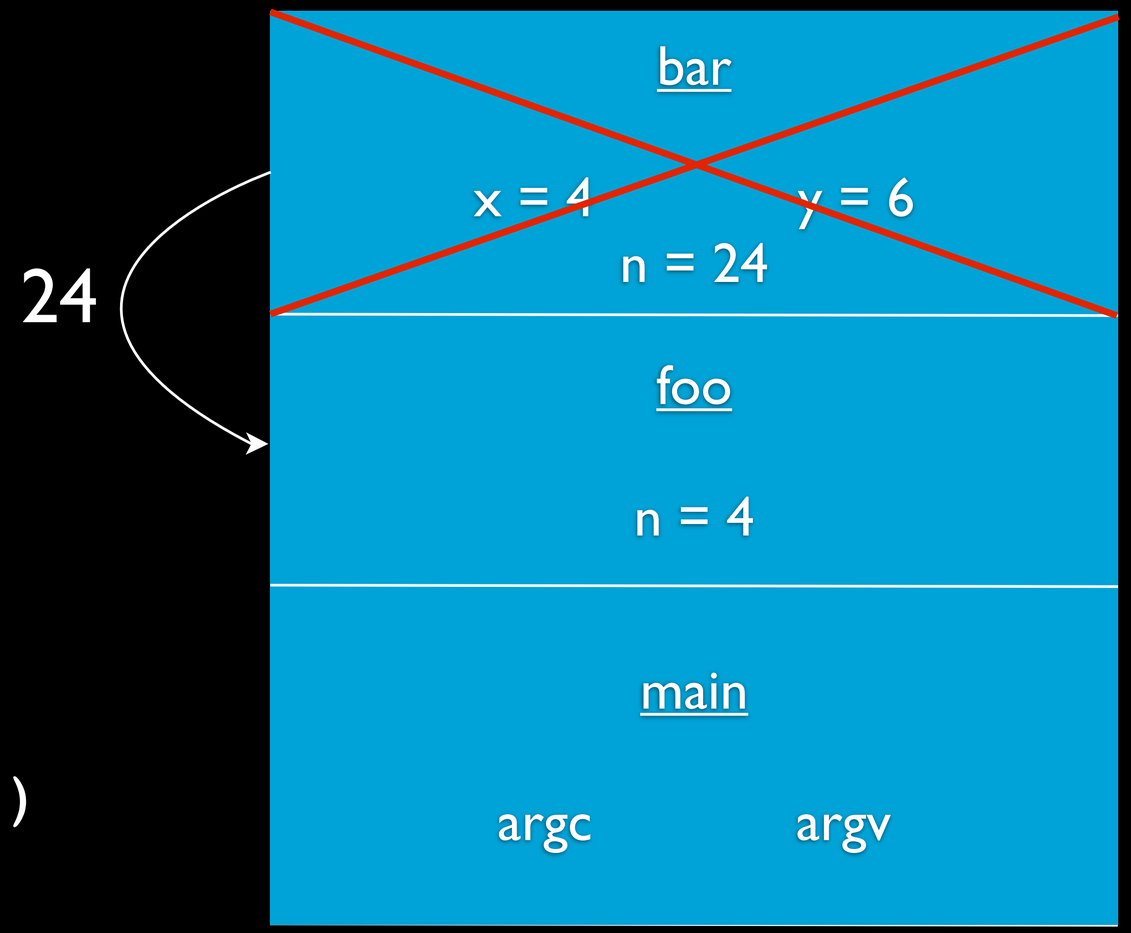
Stack

```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Stack

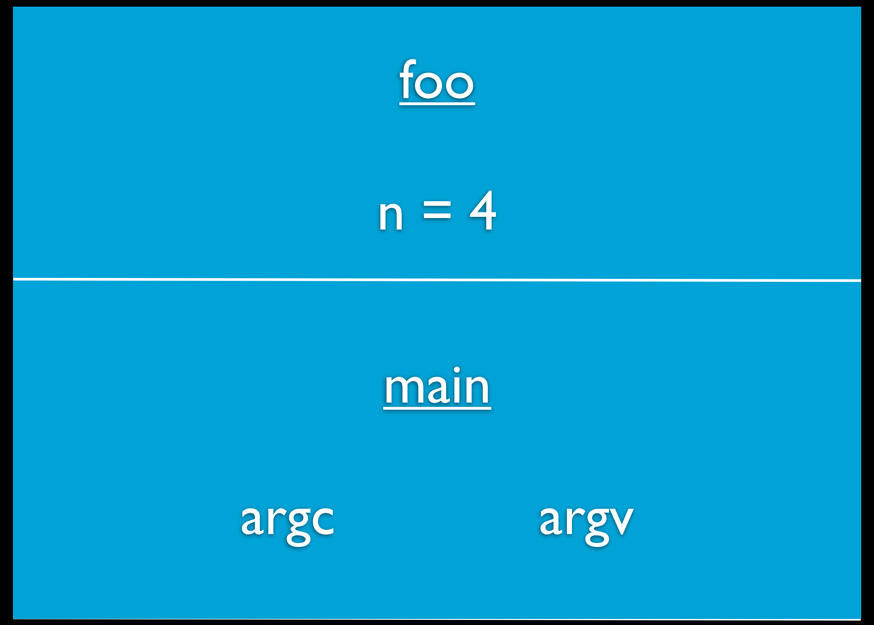
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

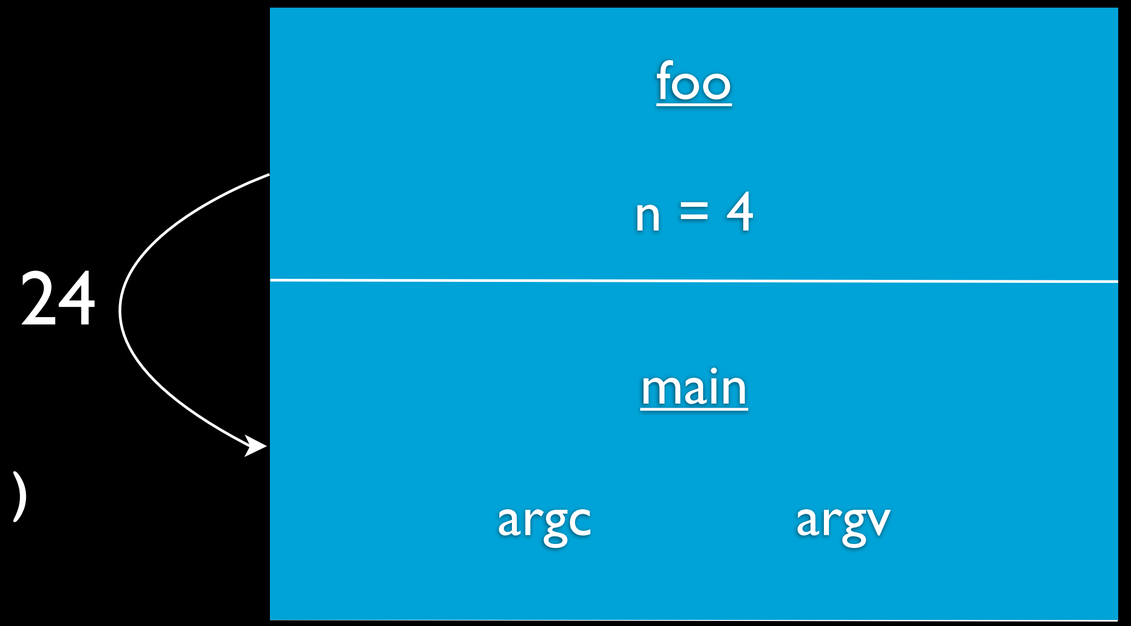
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

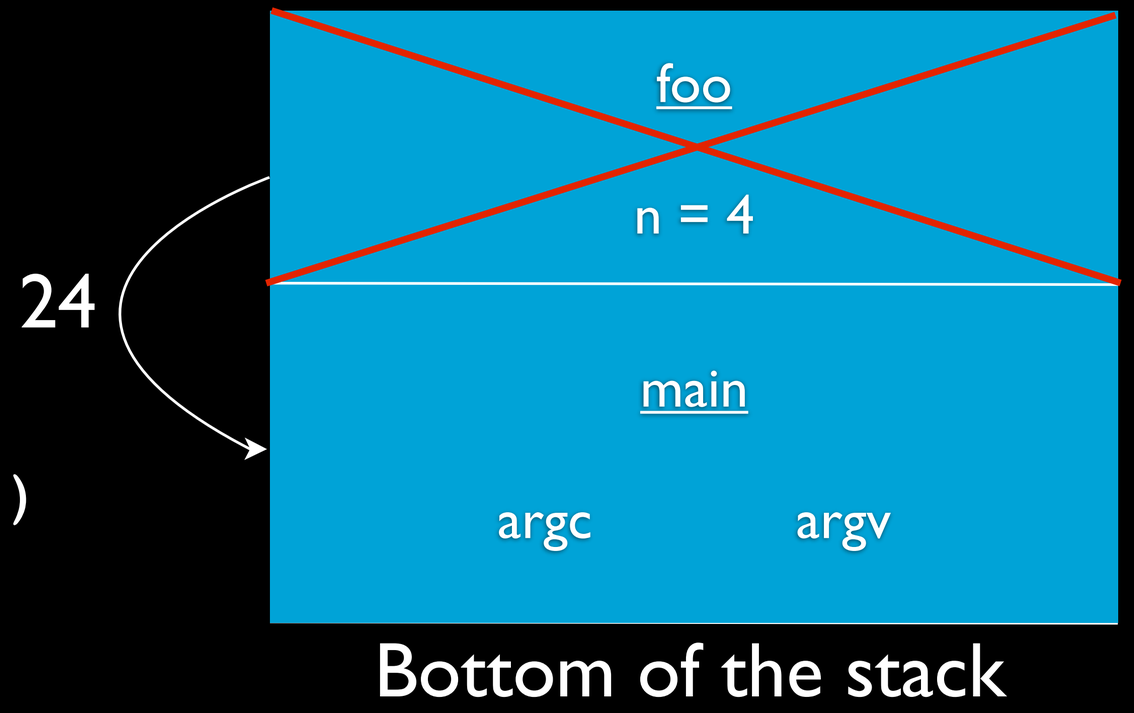
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

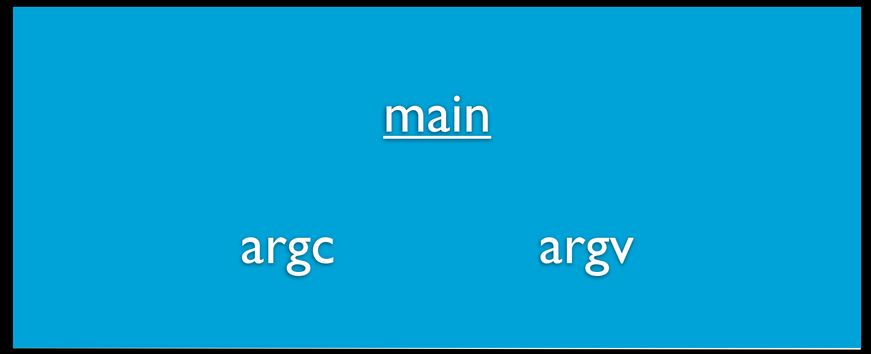
Stack

```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Stack

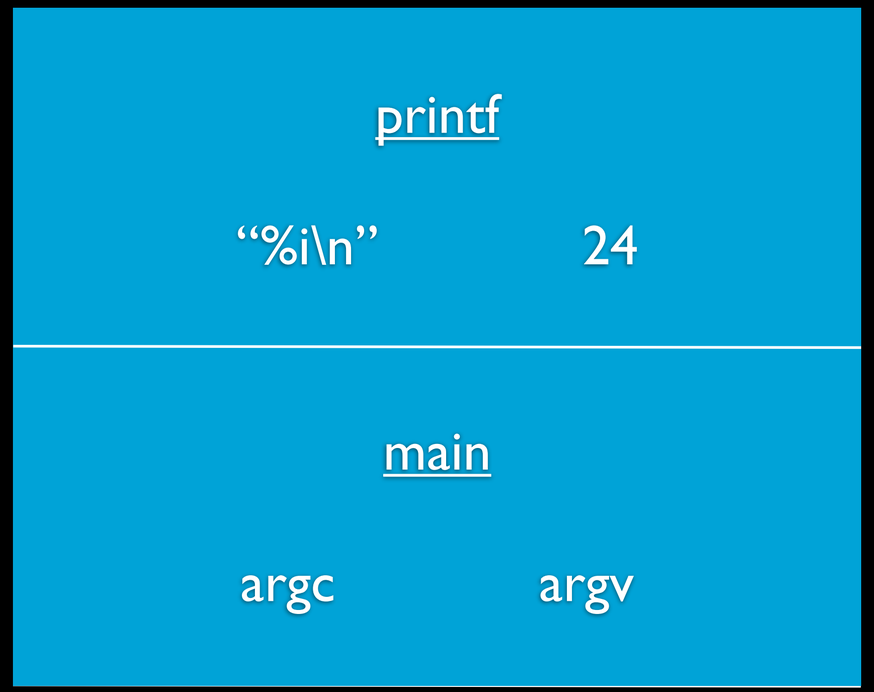
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

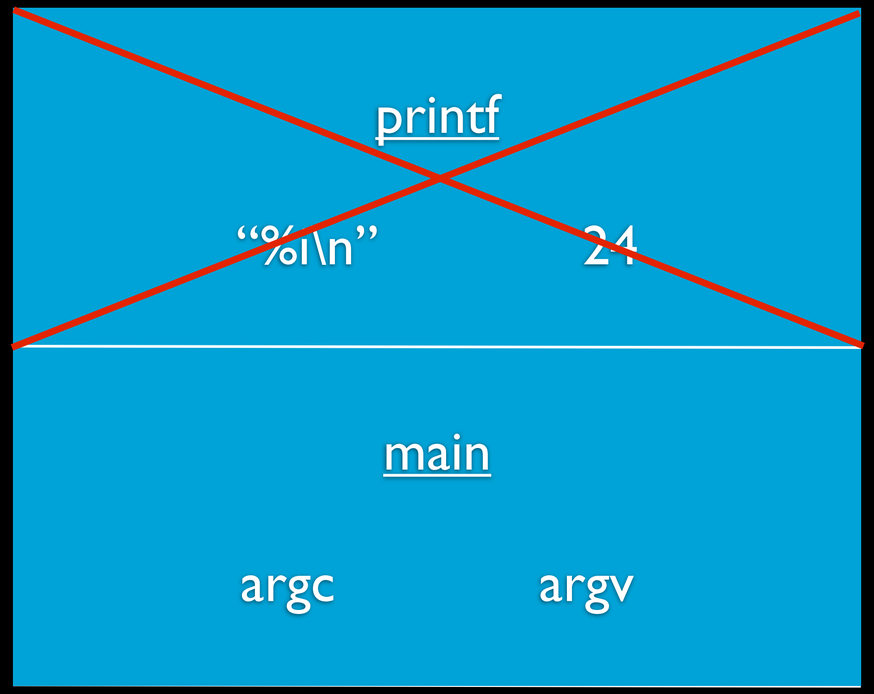
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

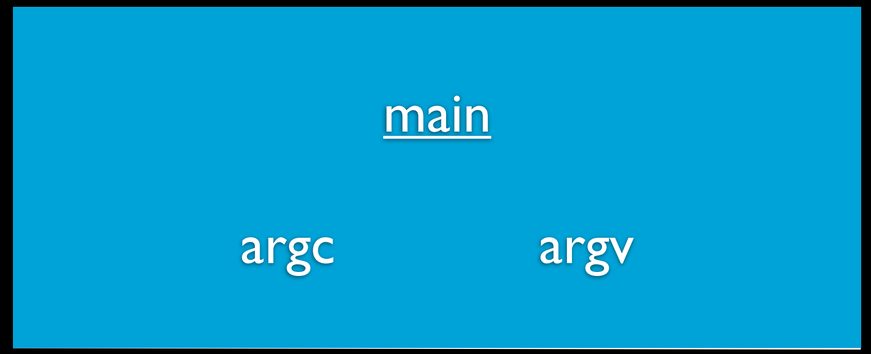
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

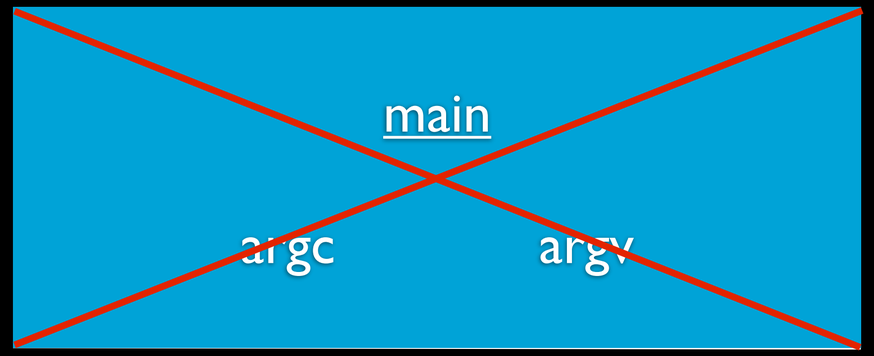
```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```



Bottom of the stack

Stack

```
int bar(int x, int y)
{
    int n = x * y;
    return n;
}

int foo(int n)
{
    return bar(n, n + 2);
}

int main(int argc, string argv[])
{
    printf("%i\n", foo(4));
}
```

Bottom of the stack

Heap

- Dynamic memory management
- malloc and free

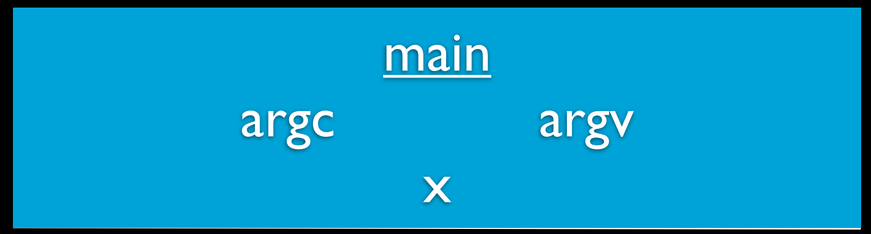
Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Top of the heap

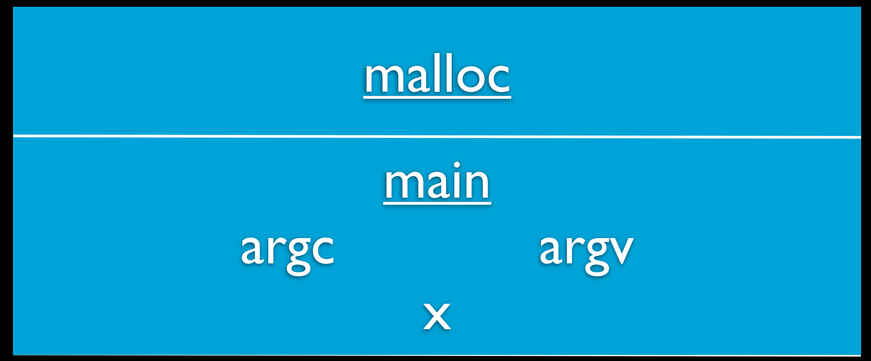


Bottom of the stack

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Top of the heap



Bottom of the stack

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Top of the heap

Bottom of the stack

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

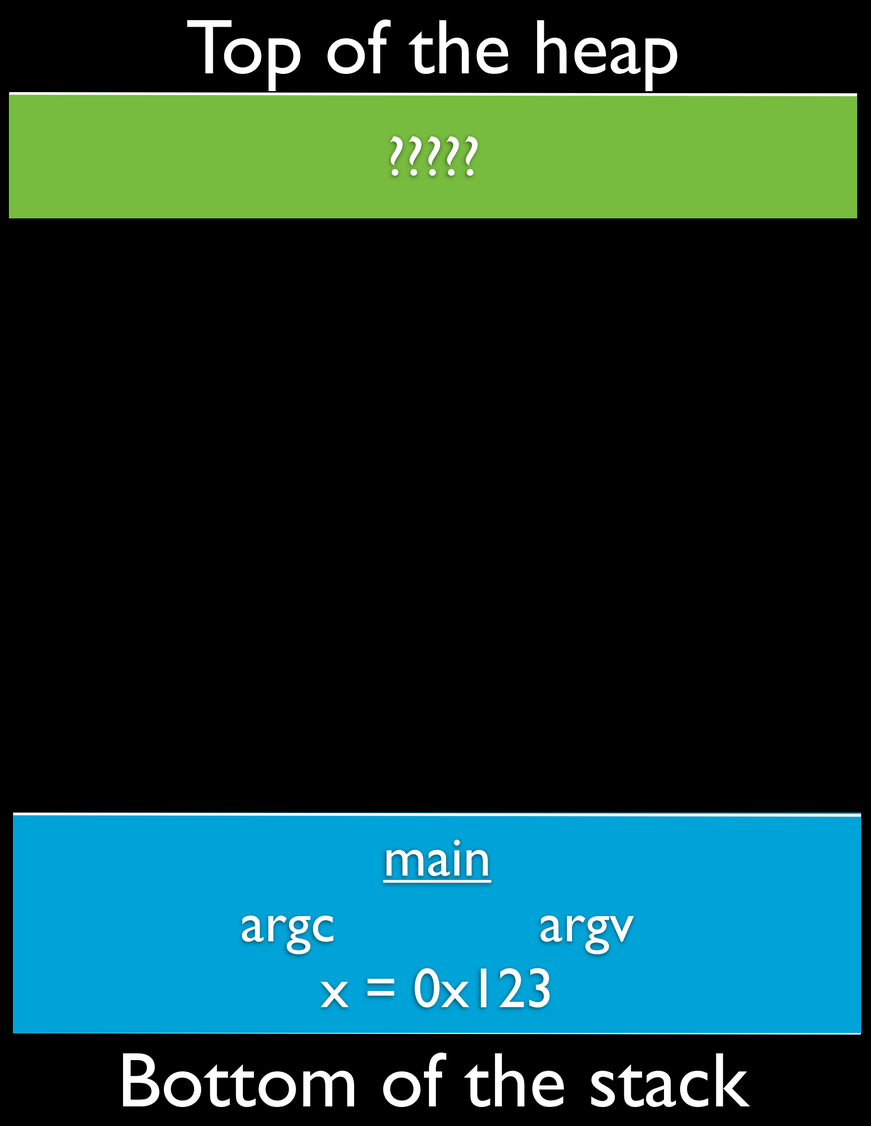
0x123

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

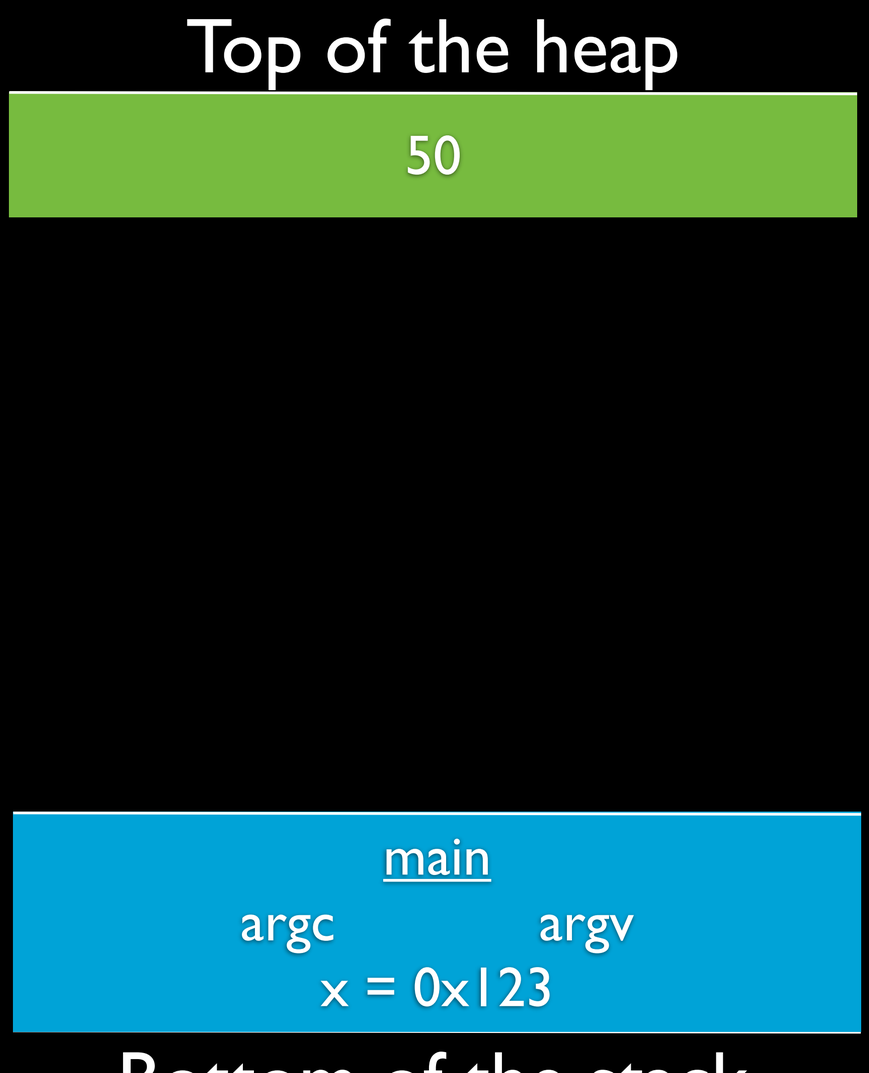


Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

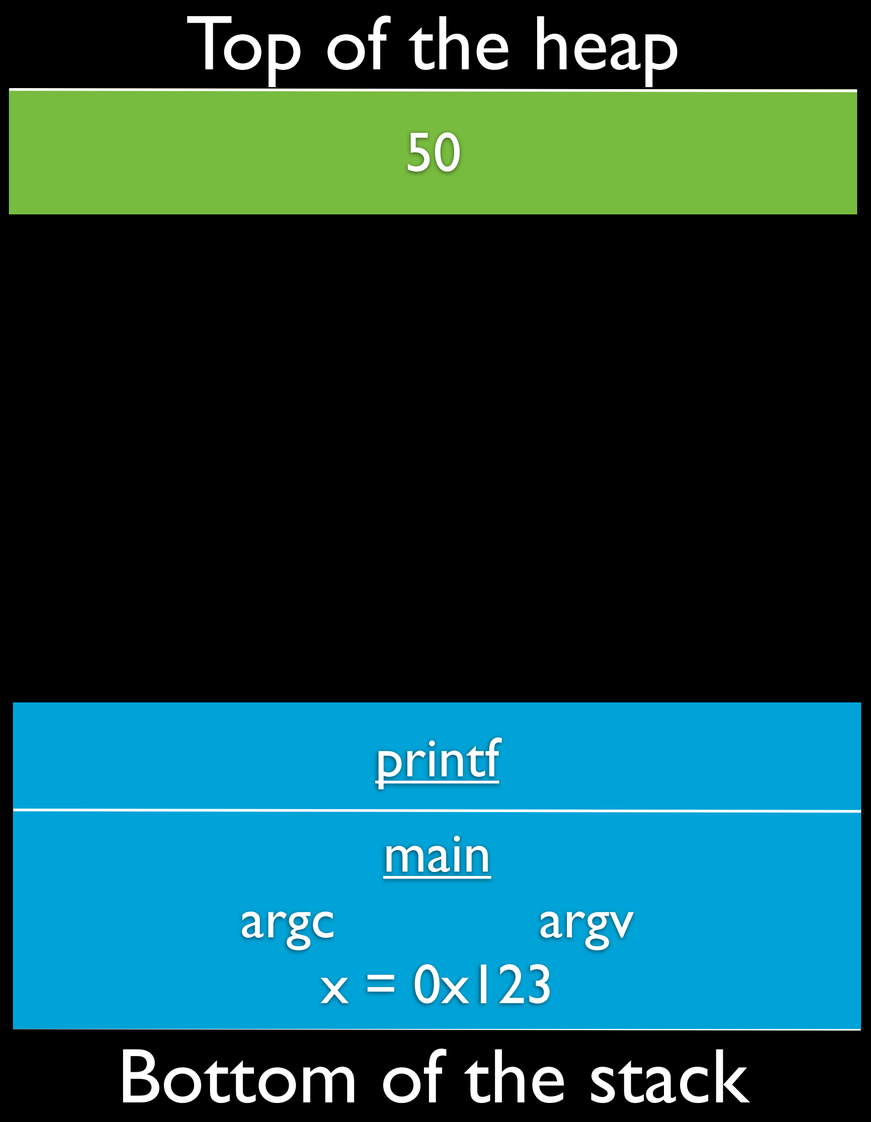
Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```



Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

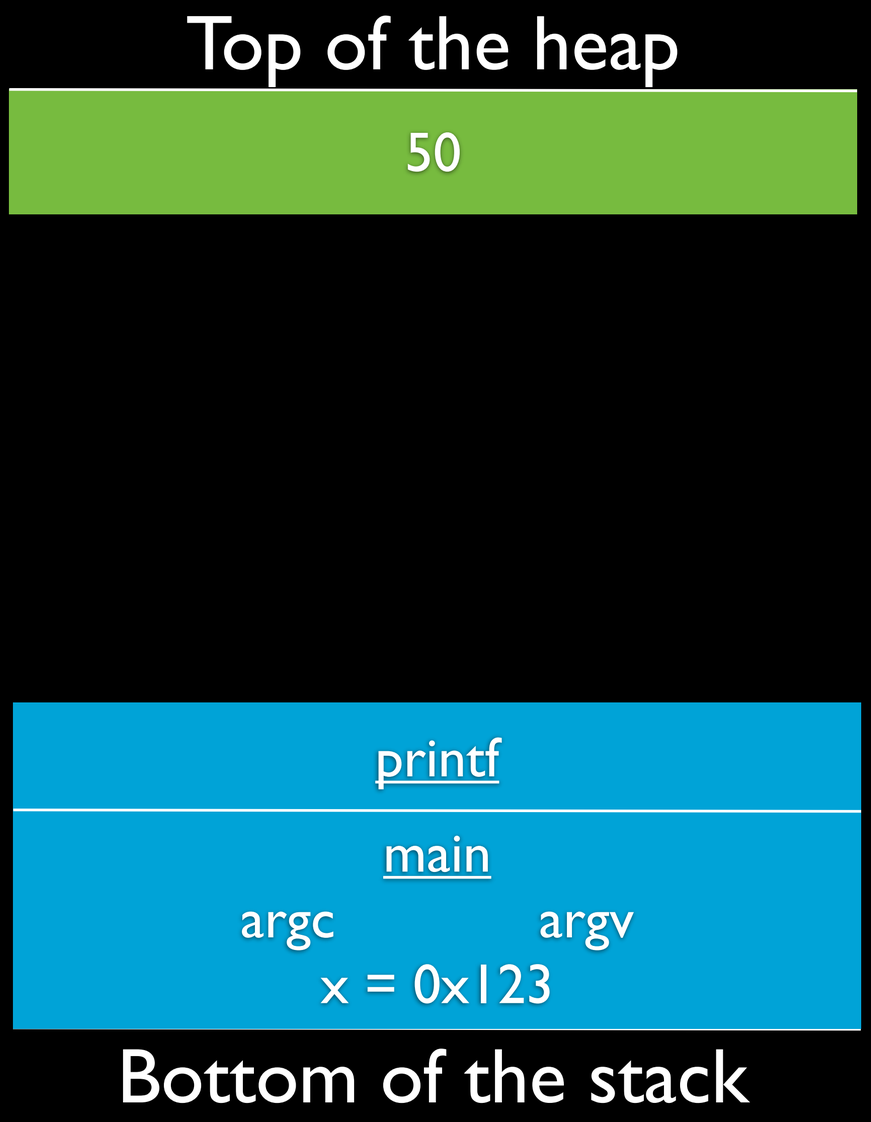


Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

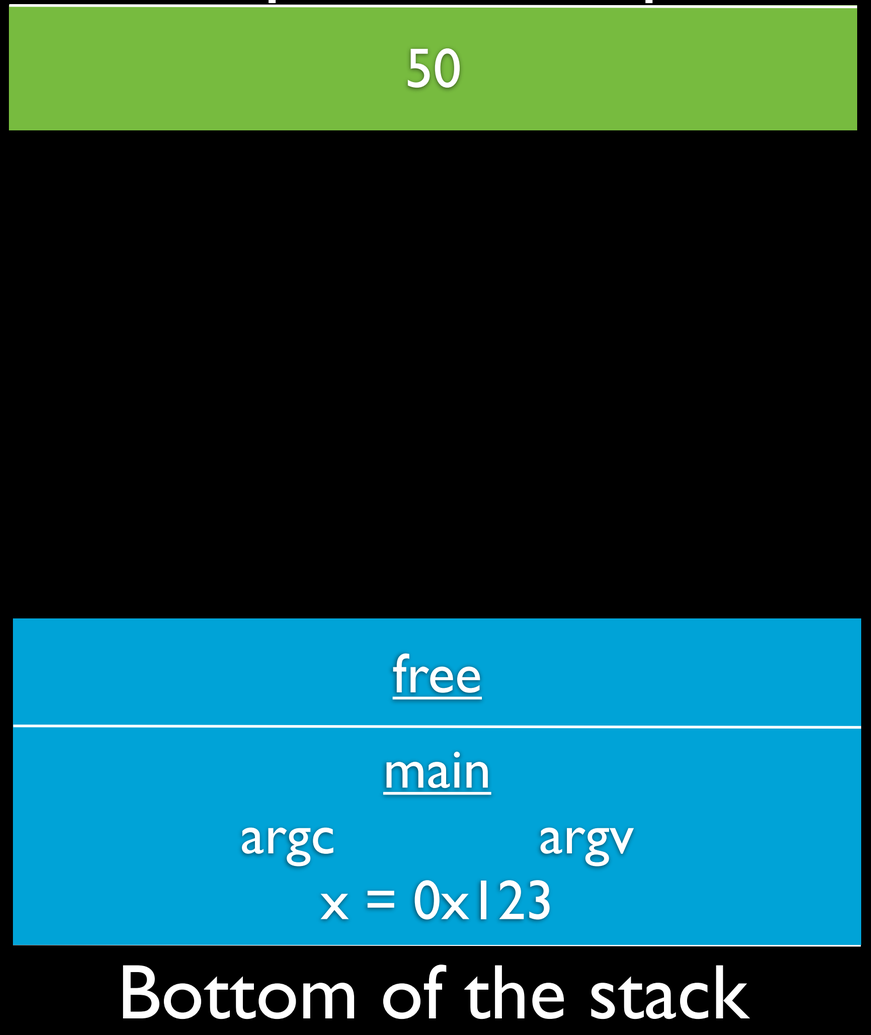


Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```



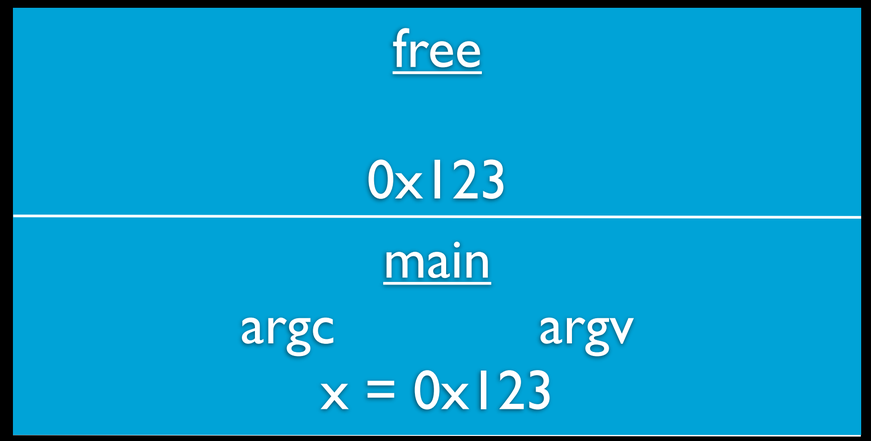
Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Top of the heap

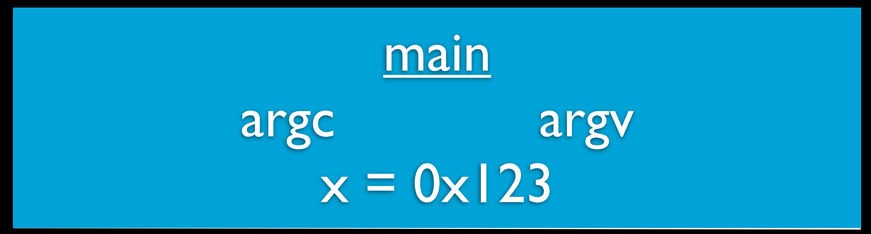


Bottom of the stack

Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Top of the heap



Bottom of the stack

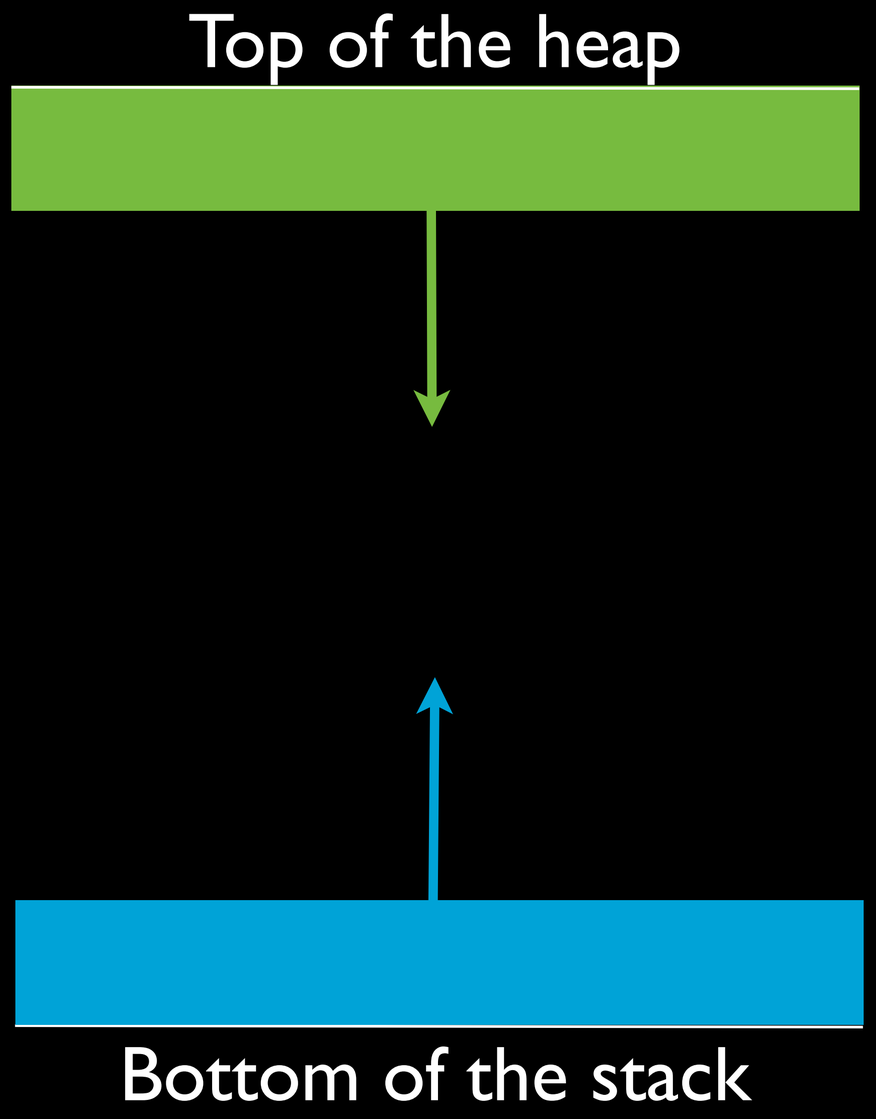
Heap

```
int main(int argc, string argv[])
{
    int* x = malloc(sizeof(int));
    if (x == NULL)
    {
        return 1;
    }
    *x = 50;
    printf("%i\n", *x);
    printf("%p\n", x);
    free(x);
    return 0;
}
```

Top of the heap

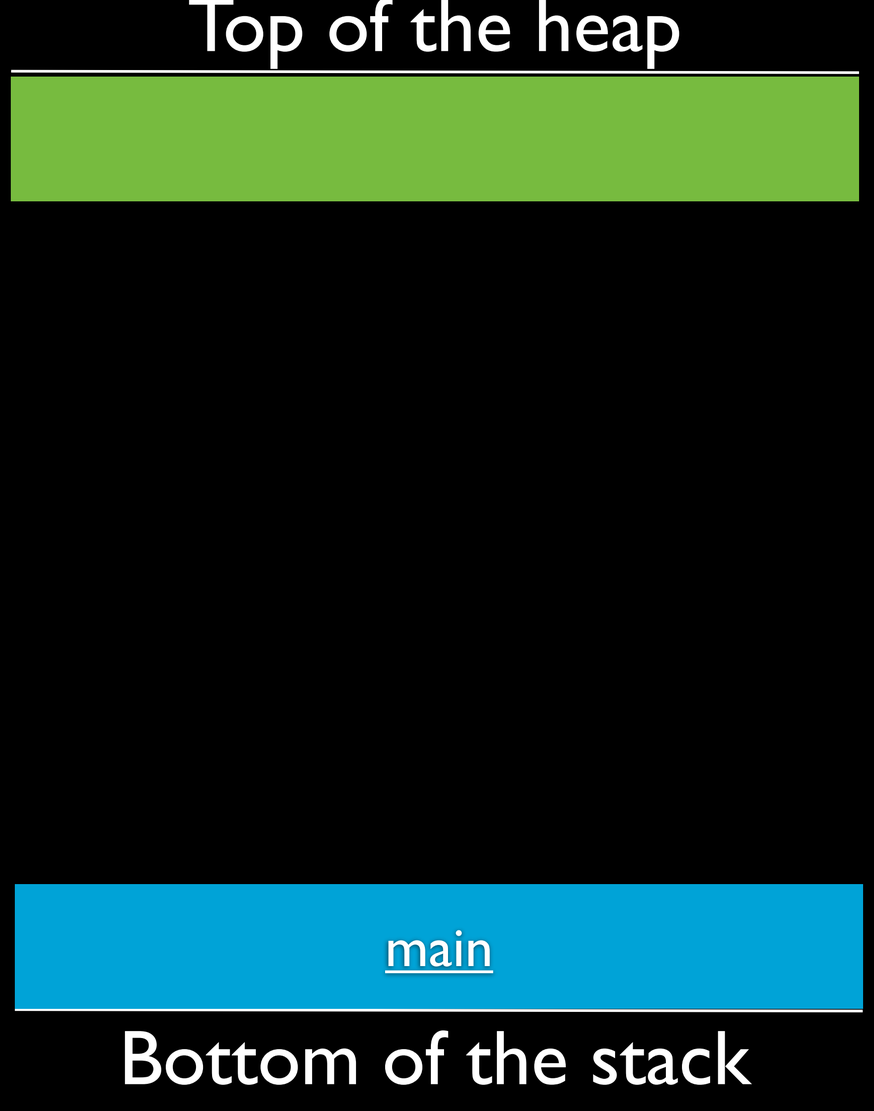
Bottom of the stack

Stack overflow



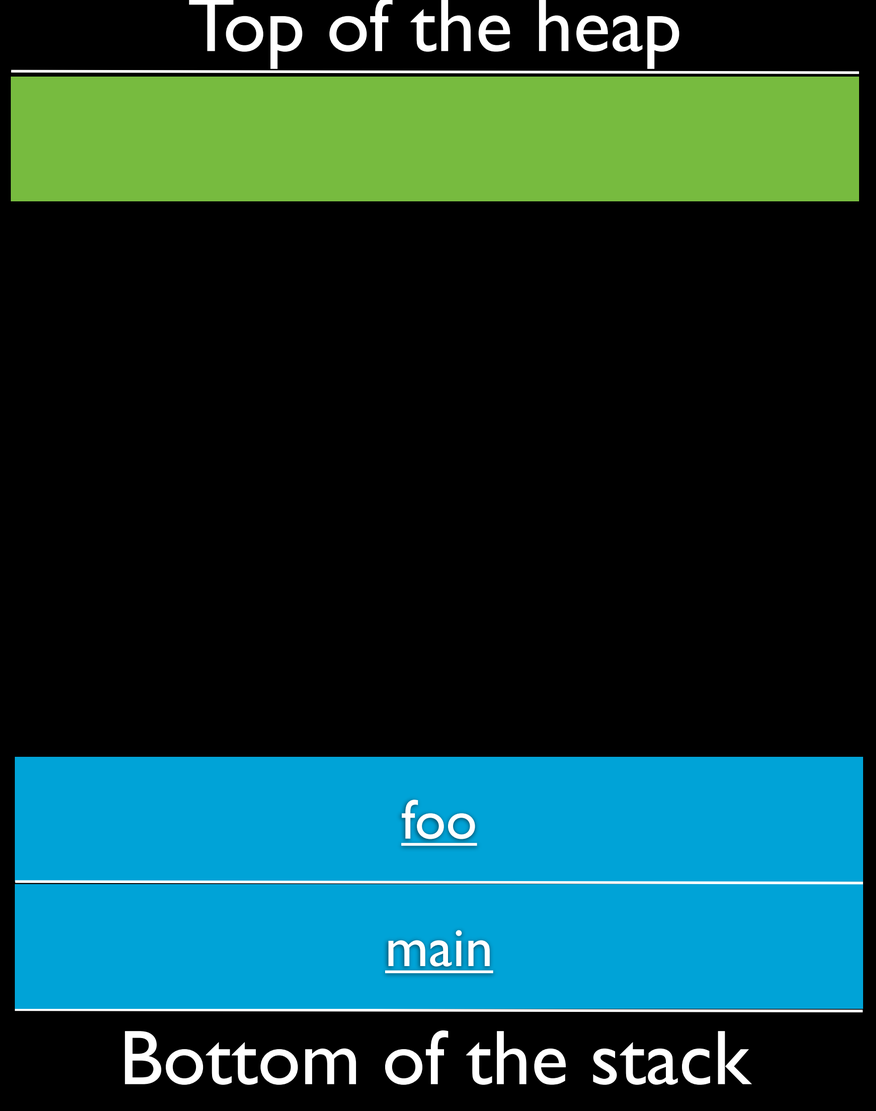
Stack overflow

```
void foo(void)
{
    foo();
}

int main(int argc, string argv[])
{
    foo();
}
```



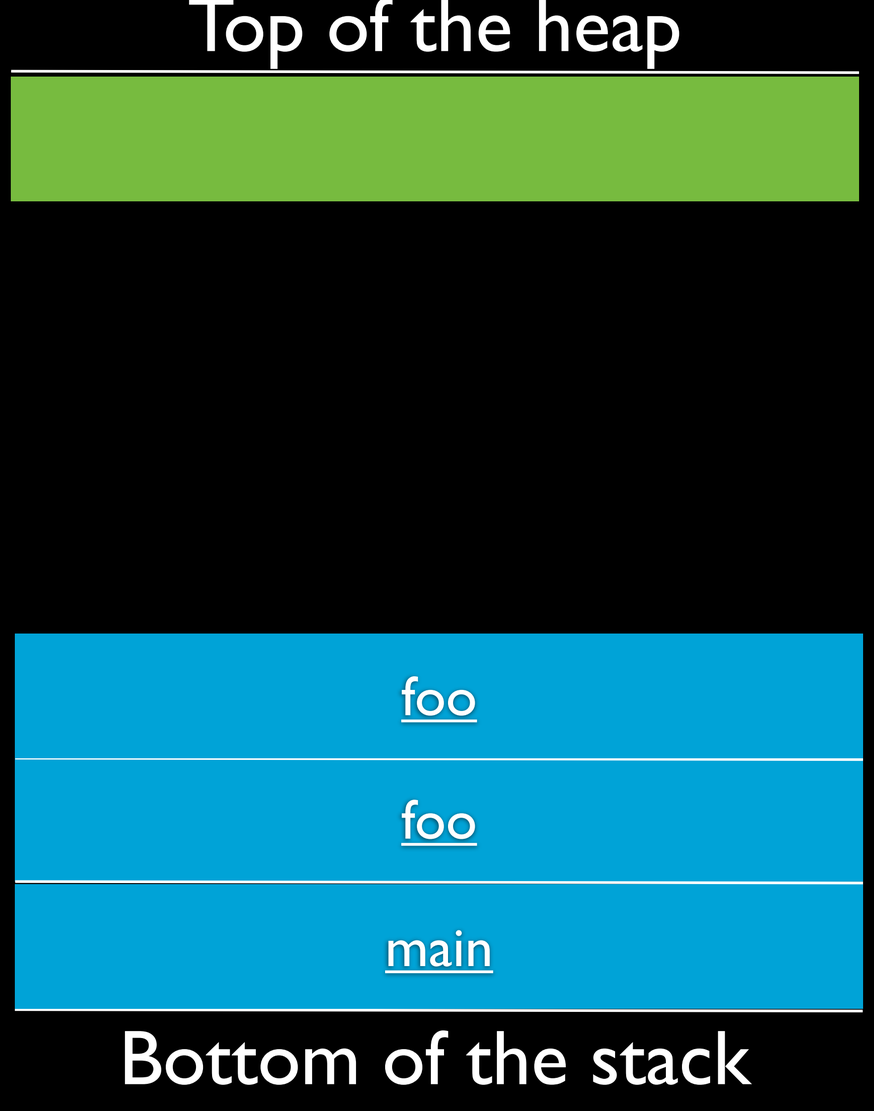
Stack overflow

```
void foo(void)
{
    foo();
}

int main(int argc, string argv[])
{
    foo();
}
```



Stack overflow

```
void foo(void)
{
    foo();
}

int main(int argc, string argv[])
{
    foo();
}
```



Stack overflow

```
void foo(void)
{
    foo();
}

int main(int argc, string argv[])
{
    foo();
}
```

Stack overflow

```
void foo(void)
{
    foo();
}

int main(int argc, string argv[])
{
    foo();
}
```

Stack overflow

```
void foo(void)
{
    foo();
}

int main(int argc, string argv[])
{
    foo();
}
```

Stack overflow

```
void foo(void)
{
    foo();
}

int main(int argc, string argv[])
{
    foo();
}
```

Compilation

- Pre-processing (#)
- Compiling (C => Assembly)
- Assembling (Assembly => Binary)
- Linking (Binary => Executable <= Binary)