C Memory Model: Structs & Heap

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

TAs:

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

pollev.com/tqm

- How are you? Any Questions from last lecture?

Administrivia (Pt. 1)

- Pre-semester survey: **Due Tomorrow Night @ Midnight**
  - Free points
  - Anonymous on canvas
  - Give us feedback on how to shape the course better

- Check-ins
  - All current ones are due tomorrow at midnight (To account for people joining late)
  - Next one comes out at end of week
  - Due before lectures on Tuesday, No extensions

Administrivia (Pt. 2)

- HW01 out & autograder
  - HW01 is out and the autograder is posted on gradescope
  - Due Friday night @ midnight

- 1-on-1's
  - We have a new form to sign-up for 1-on-1's with TA's
  - Linked on the course website
  - Should not replace office hours entirely

- Recitation
  - First one starts tomorrow! Should be useful for HW01
  - Will be recorded if all goes well
  - 4:00 pm in DRLB room 3C6

Lecture Outline

- **C memory Review:**
  - **An array of bytes**
  - **Pointers & arrays**
  - **The Stack**
- Structs, Pass-by-value, Output Parameters
- The Heap
  - malloc() & free()

Memory is an array of bytes

- Memory can be thought of as an array of bytes
  - Each byte has an index called an "Address"
  - All Variables* and Data is stored inside of Memory.
  - Exact address of variables will change execution to execution
    - Hardcoding exact addresses, is not a thing (for us)
    - Addresses matter more as "references", but this is how they work

```
char c = 'a';

char x = 'b';
```

0xF0	0xF1	0xF2	0xF3	0xF4	0xF5	0xF6	0xF7	0xF8	0xF9
'a'	????	'b'	???	????	????	???	???	????	???

Size Does Matter When Talking About Range

Type	Size (Bytes)	Minimum	Maximum
char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	8	-9223372036854775808	9223372036854775807
unsigned long	8	0	18446744073709551615

Memory is an array of bytes

- Memory can be thought of as an array of bytes
  - Each byte has an index called an "Address"
  - Some data can span multiple bytes / addresses.
    - Address of (&) a multi-byte variable is the lowest address of it
  - Example: an int is four bytes

```
char c = 'a';
int i = 4;
char x = 'b';
```

0xF0	0xF1	0xF2	0xF3	0xF4	0xF5	0xF6	0xF7	0xF8	0xF9
'a'	????	'b'	???	0x 00 00 00 04				????	???

Pointers & Arrays in Memory

- Pointers themselves can be stored in memory as variables
  - Pointer are 8-bytes
  - There are 2^{64} addresses in memory (thus memory is 2^{64} bytes)
- Arrays are also stored in memory
  - Note how arrays only have memory to store their members, there is no address stored or length stored

```
char c[] = "Hi";
char *str = c;
```

0xF0	0xF1	0xF2	0xF3	0xF4	0xF5	0xF6	0xF7	0xF8	0xF9	0xFA
'H'	'i'	'\0'	0x 00 00 00 00 00 00 00 F0							

It Is All Just Bytes

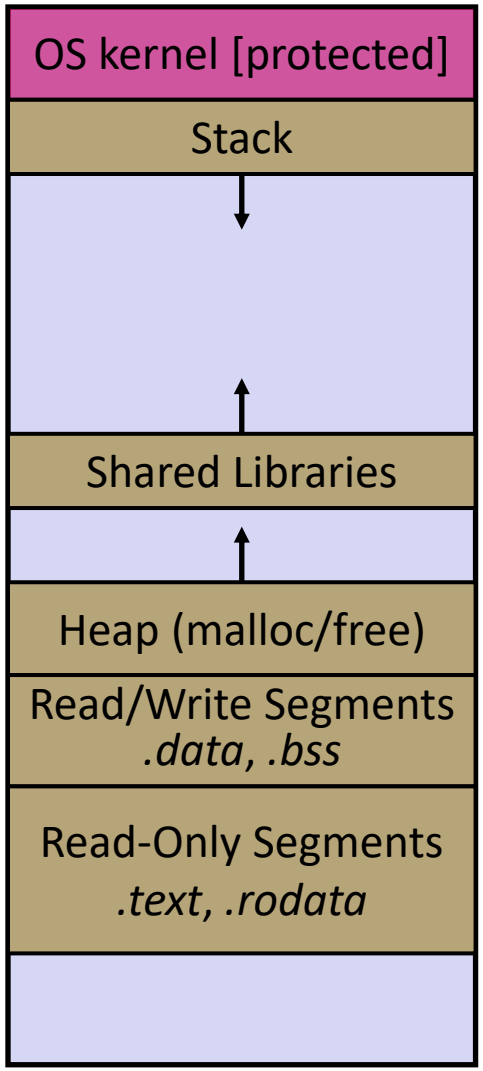
- Remember the check-setup files?

```
int array[] = { 1735289167, 1866604655,
                1869049449, 1920287264,
                2053208673, 8565};
printf("%s\n", (char*) array);
```

- This prints “Oingo Boingo Burazazu!”
- I took the ASCII encoding of each character and combined them into integers.
- I told C to read the integer array as a sequence of characters (a string) and it will interpret the same bytes as ASCII instead of integers.

Whole View of C Memory Model

- Where all data, code, etc are stored for a program
- Broken up into several segments:
  - The stack (Last Lecture)
  - The heap (This & next lecture)
  - The kernel (Take CIS 4480)
  - Etc.
- Each "unit" of memory has an address



The Stack

- The stack is where we hold all local variables for the functions we call in C

- Each time we call a function, we “push” a “stack frame” onto the stack.
  - Each stack frame stores the variables for that function call
  - New Stack Frame for each time we call the function

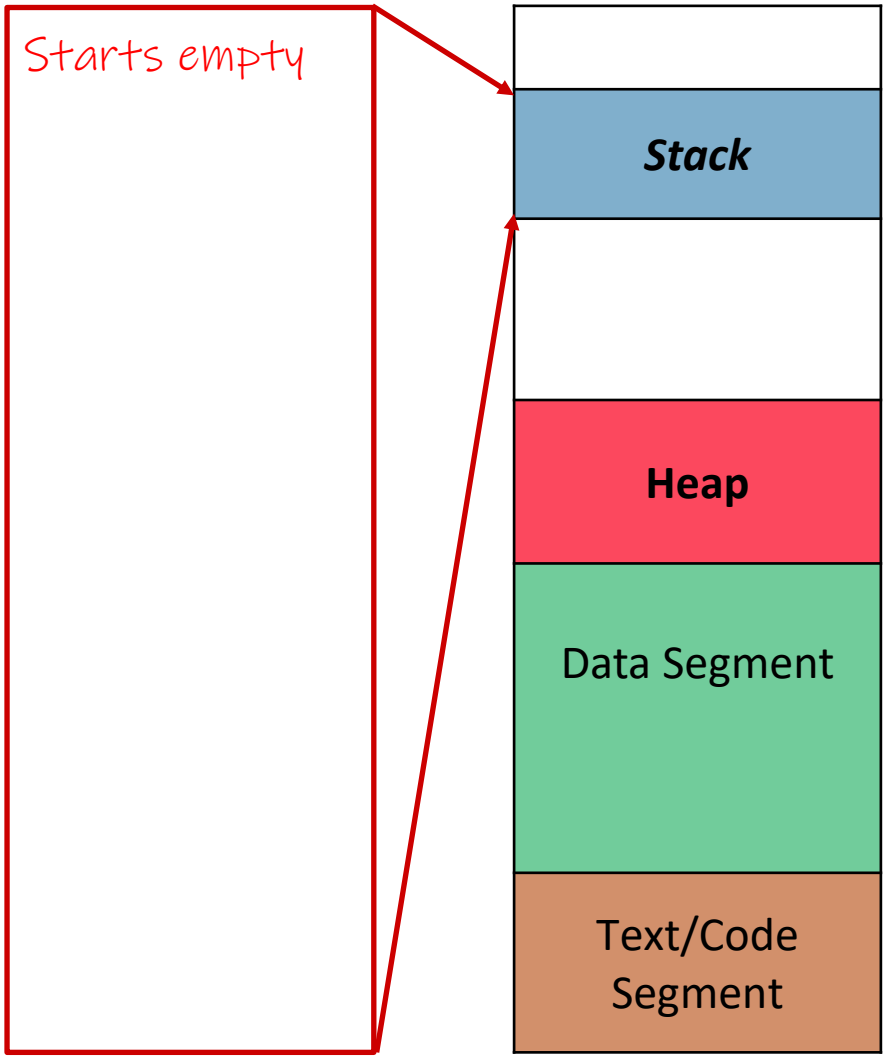
- That stack frame gets "popped" when we return from a function

Stack Example 1:

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

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

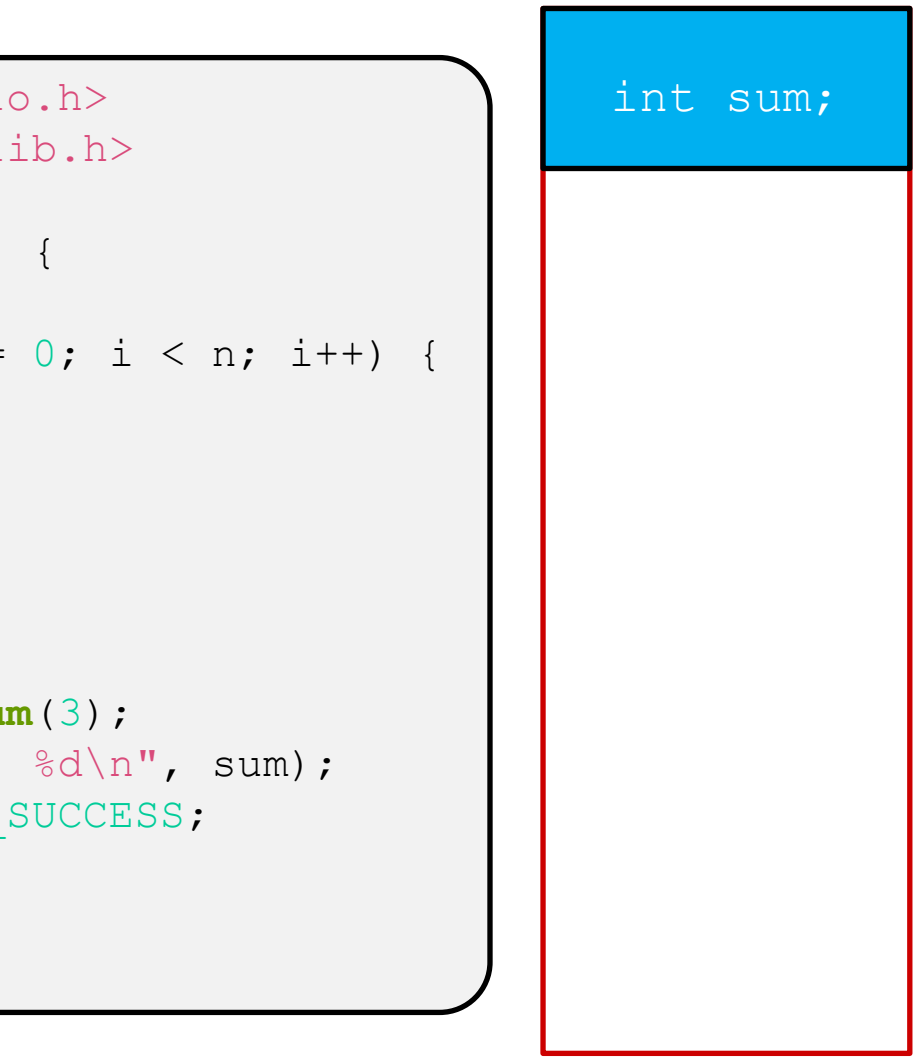
Zooming in on the bottom of the stack

Starts empty



Stack Example 1:

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

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

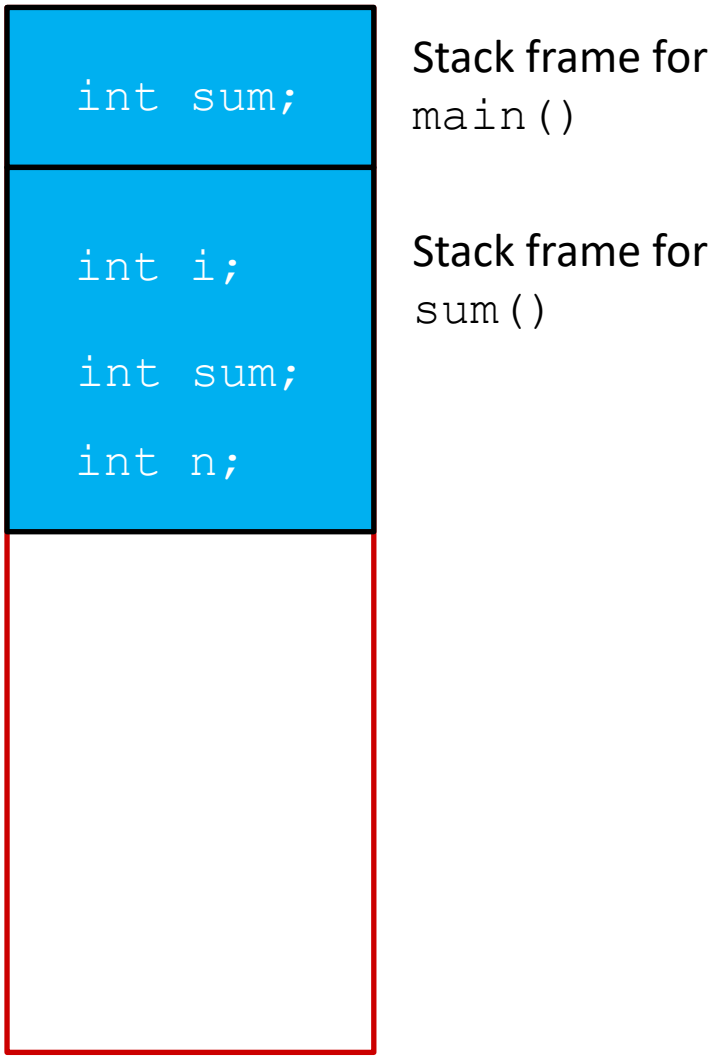


Stack frame for `main()`

Stack frame for main is created when CPU starts executing it

Stack Example 1:

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

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

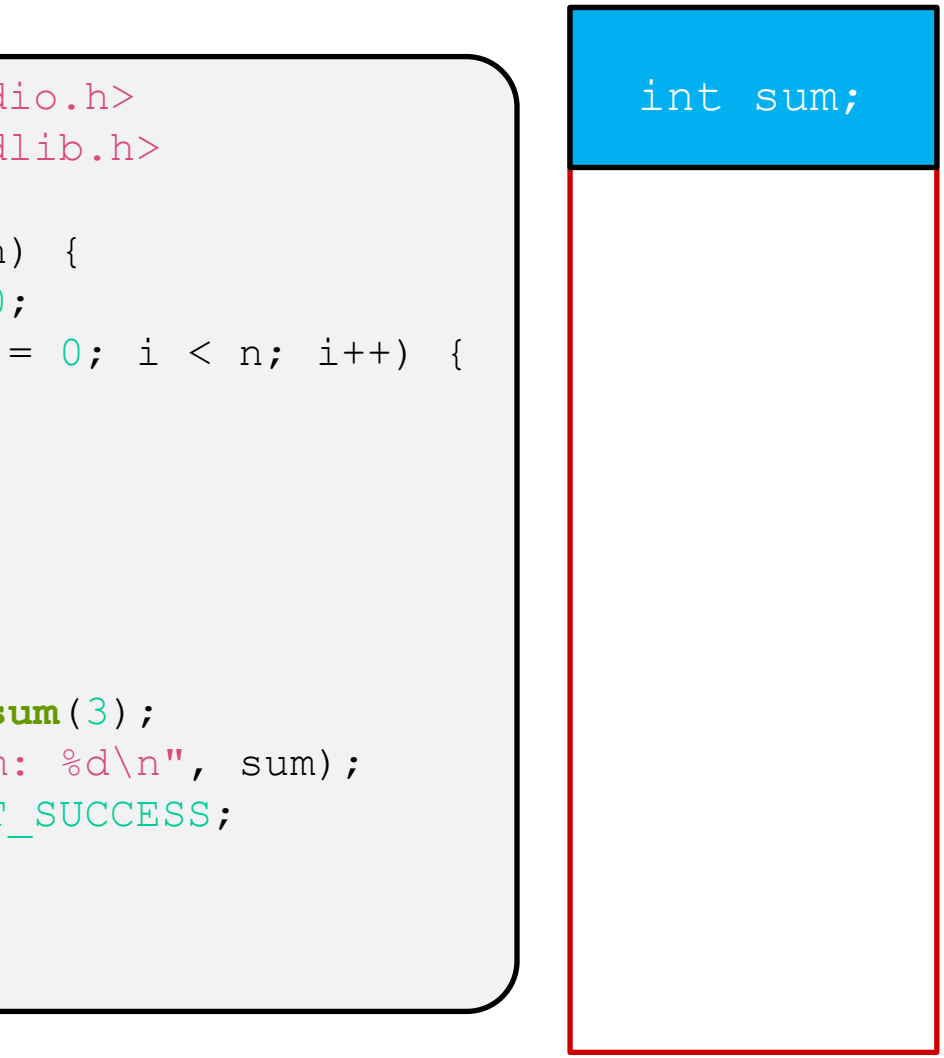


Stack frame for `main()`

Stack frame for `sum()`

Stack Example 1:

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

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



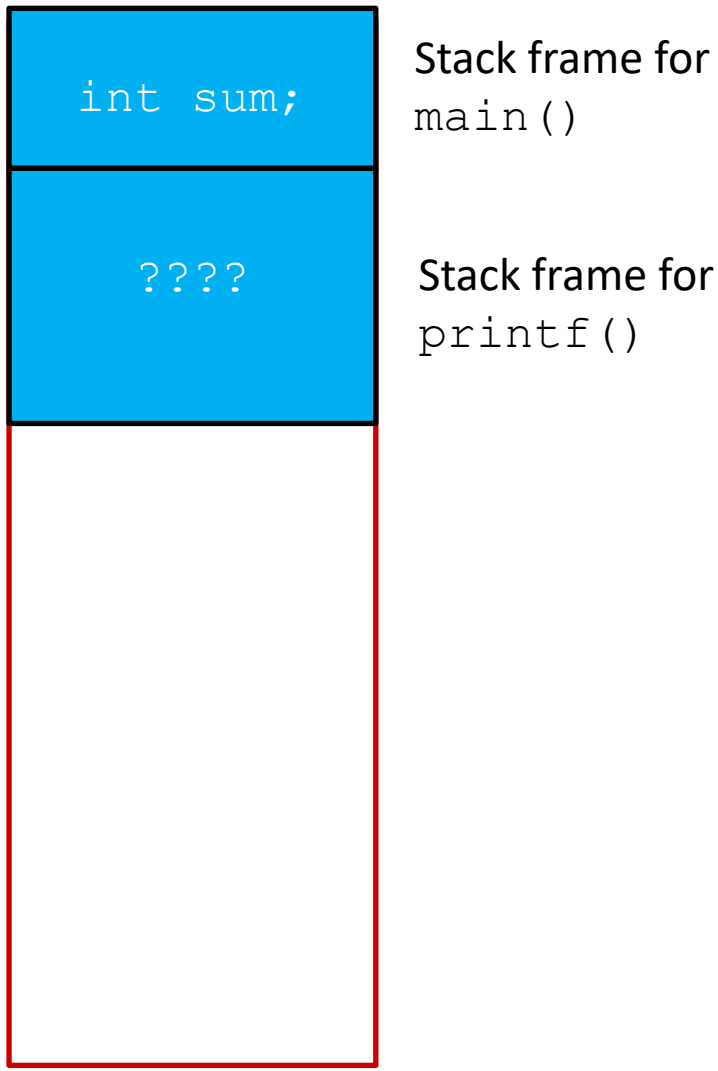
Stack frame for `main()`

`main()`'s stack frame is now top of the stack and we keep executing **`main()`**

`sum()`'s stack frame goes away after **`sum()`** returns.

Stack Example 1:

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

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

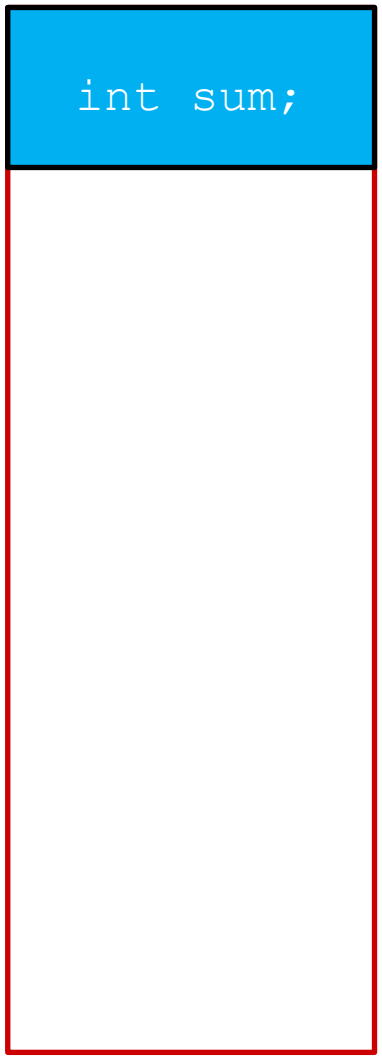
Stack frame for `printf()`

Blank Slide

- Make clear there is a gap between the Prev example and the next one.

Stack Example 2:

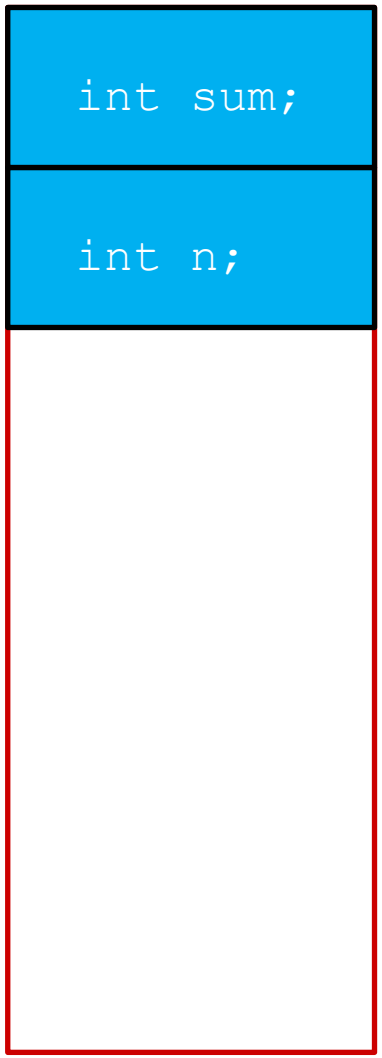
```c
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

Stack Example 2:

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

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

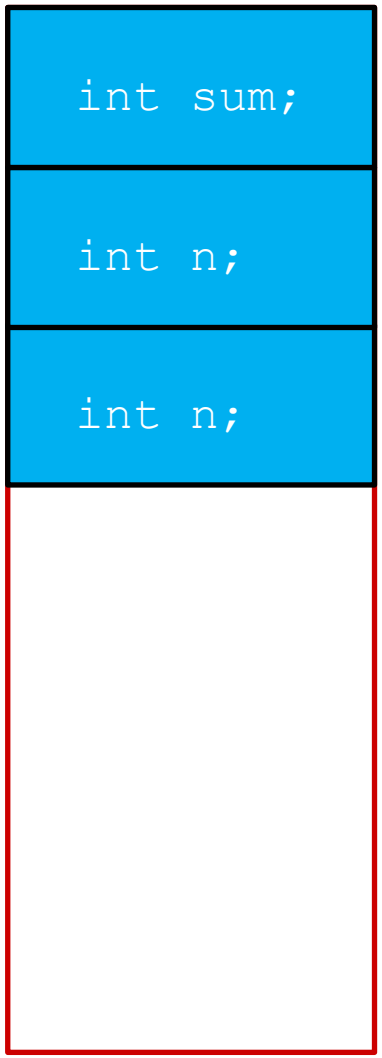


Stack frame for `main()`

Stack frame for `sum_recursive(3)`

Stack Example 2:

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

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



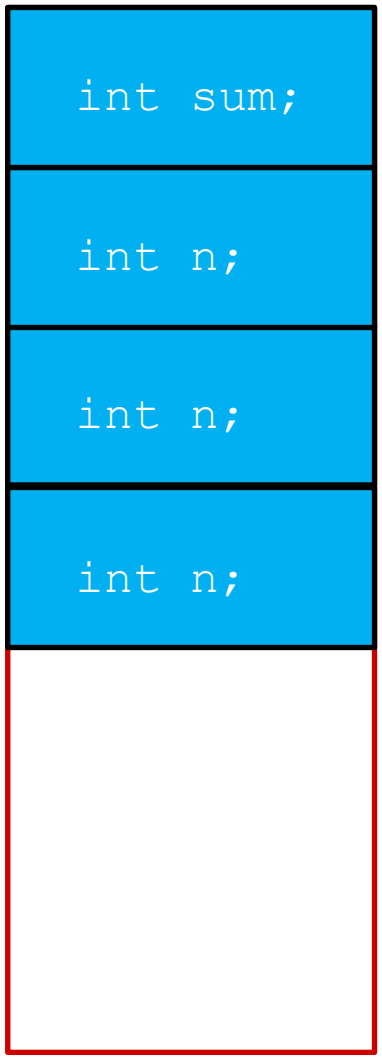
Stack frame for `main()`

Stack frame for `sum_recursive(3)`

Stack frame for `sum_recursive(2)`

Stack Example 2:

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

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

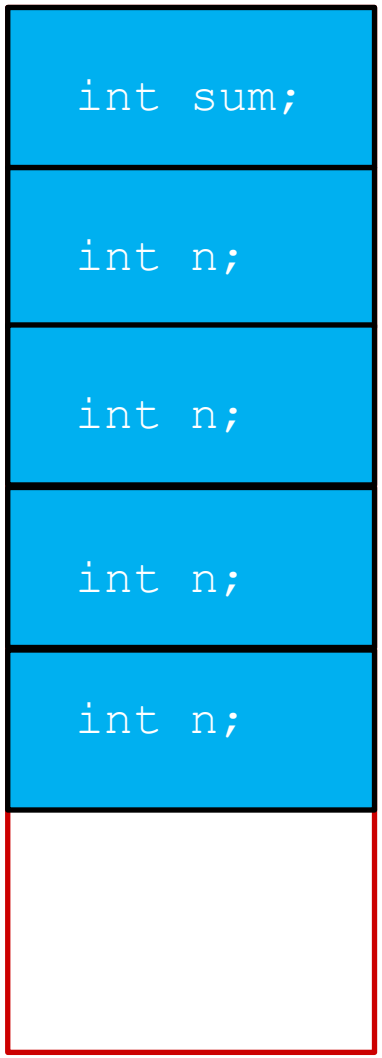
Stack frame for `sum_recursive(3)`

Stack frame for `sum_recursive(2)`

Stack frame for `sum_recursive(1)`

Stack Example 2:

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

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

Stack frame for `sum_recursive(3)`

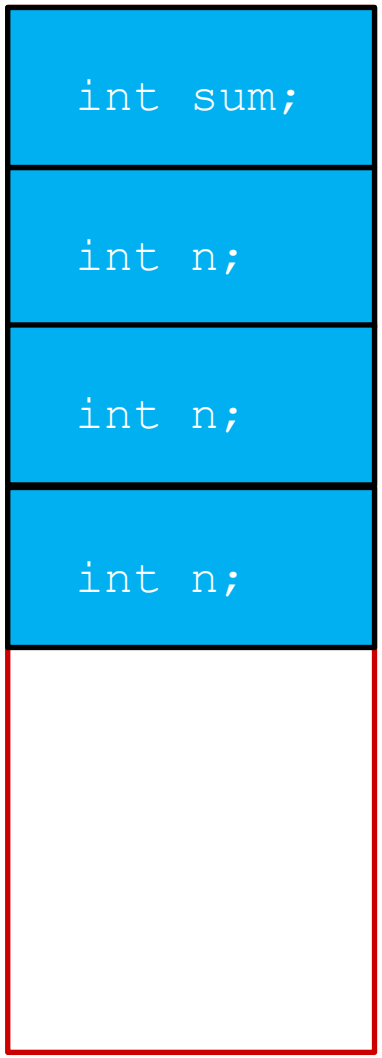
Stack frame for `sum_recursive(2)`

Stack frame for `sum_recursive(1)`

Stack frame for `sum_recursive(0)`

Stack Example 2:

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

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

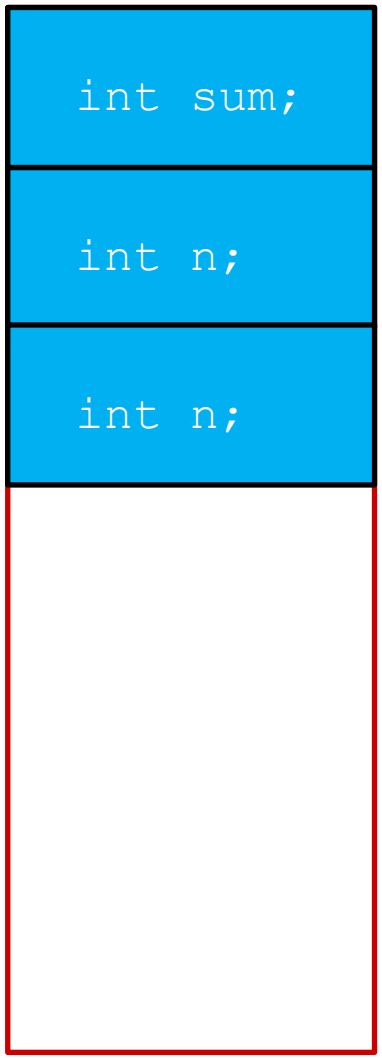
Stack frame for `sum_recursive(3)`

Stack frame for `sum_recursive(2)`

Stack frame for `sum_recursive(1)`

Stack Example 2:

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

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



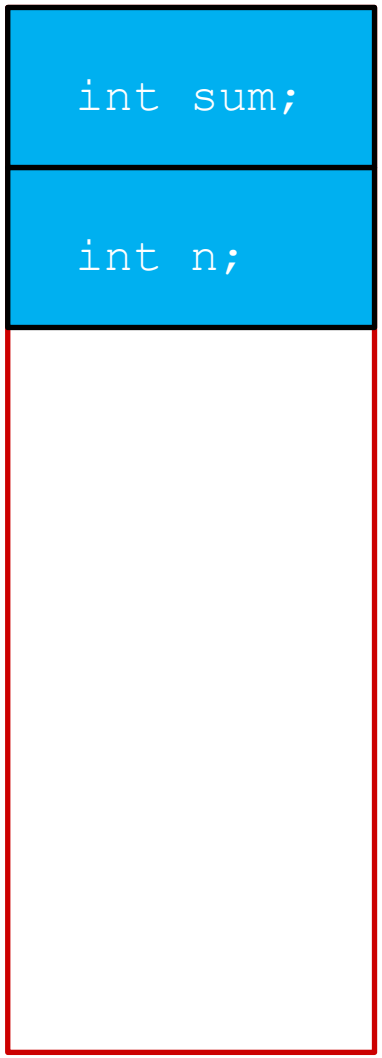
Stack frame for `main()`

Stack frame for `sum_recursive(3)`

Stack frame for `sum_recursive(2)`

Stack Example 2:

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

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

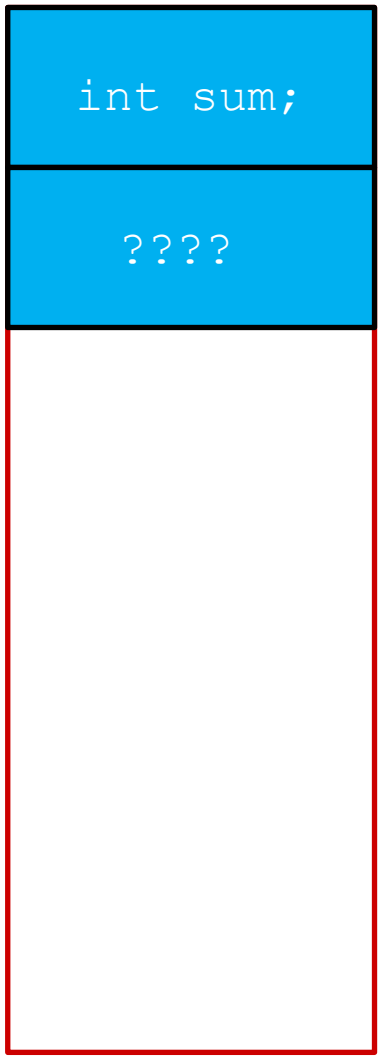


Stack frame for `main()`

Stack frame for `sum_recursive(3)`

Stack Example 2:

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

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

Stack frame for `printf()`

Global Variables in C

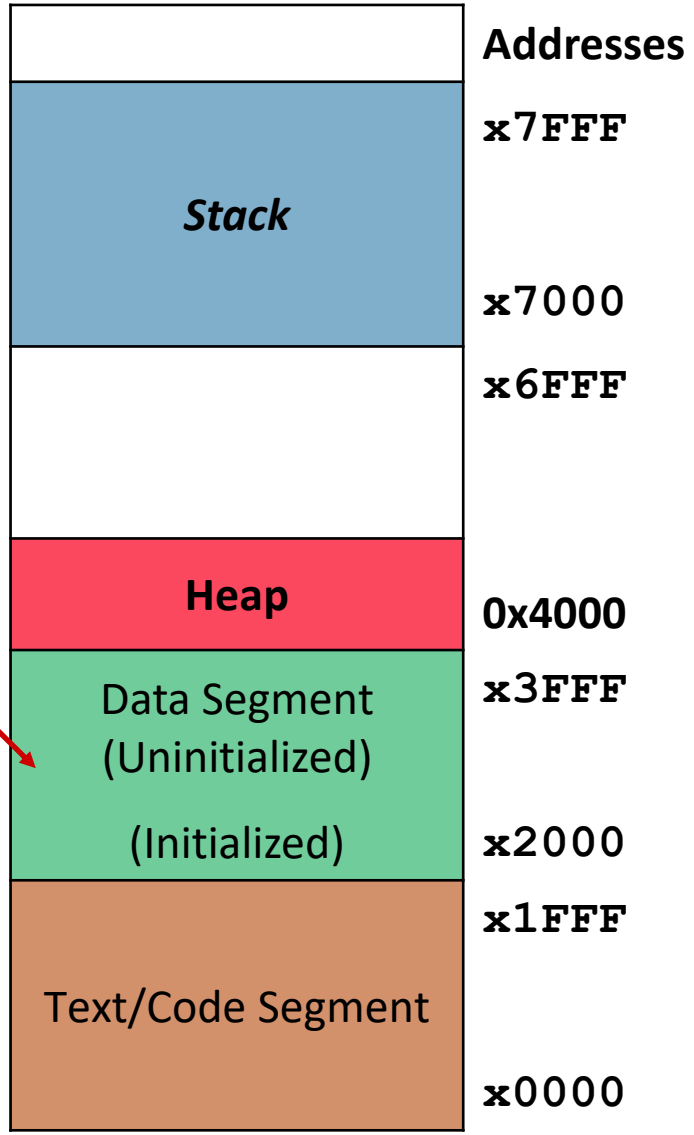
```c
#include <stdio.h>
#include <stdlib.h>

int x = 0;

void incr_globals() {
  x++;
}

int main() {
  printf("x: %d\n", x);   // prints 0
  incr_globals();
  printf("x: %d\n", x);   // prints 1
  return EXIT_SUCCESS;
}
```

Declaring a variable outside of a function makes it "global"

- Global variables exist outside of any function, can be accessed from any function
- Exist throughout the entire lifespan of a program

Global Variables in Memory

- Global variables can be stored at a static (un-changing) address.
- Reading/writing to that variable just involves going to that static memory location.
- These variable are “allocated as soon as the program is loaded. Program exiting will “de-allocate” the variable.



What questions do you have? Thoughts? Feelings?

Lecture Outline

- C memory Review:
  - An array of bytes
  - Pointers & arrays
  - The Stack
- **Structs, Pass-by-value, Output Parameters**
- The Heap
  - malloc() & free()

Typedef

- **`typedef`** is used for "defining" a "new" type

```c
typedef int my_int;

my_int x = 3;
// same as "int x = 3;"

typedef unsigned int my_uint;

// unsigned means value cannot be < 0
my_uint ux = 0U; // unsigned 0

typedef char* str;

str mf = "DOOM";
```

Structured Data

- A `struct` is a C datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors
  - Useful for defining new structured types of data
  - Acts similarly to primitive variables
- Generic declaration:

There are other ways to define structs aside from this

```
typedef struct {
  float x;
  float y;
} Point;

Point pt;
Point origin = {0.0f, 0.0f};
pt = origin; // pt now contains 0.0f, 0.0f
```

Default values are still garbage!

<- Initializer List

Can be assigned into,
used as parameters, etc.

Accessing struct Fields

- Use "." to refer to a field in a struct
- Use "->" to refer to a field from a struct pointer
  - Dereferences pointer first, then accesses field

```
typedef struct {
  float x, y;
} Point;

int main(int argc, char** argv) {
  Point p1 = {0.0, 0.0};
  Point* p1_ptr = &p1;

  p1.x = 1.0;
  p1_ptr->y = 2.0;  // equivalent to (*p1_ptr).y = 2.0;
  return 0;
}
```

Initializing a struct

- Can initialize specific members of a struct
  - Use " . " to refer to a field in a struct

```
typedef struct {
  float x, y;
} Point;

int main(int argc, char** argv) {
  Point p1 = {3.0, 2.0};
  Point p2 = (Point) {
    .x = 0.0,
    .y = 0.0
  };
  return 0;
}
```

Poll Everywhere

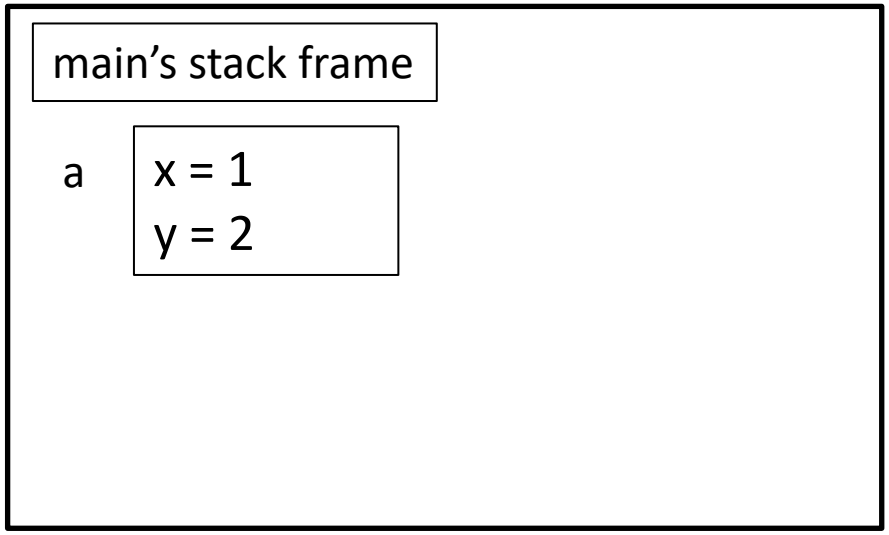
pollev.com/tqm

- What does this code print?

```
typedef struct {
  int x, y;
} point;

int main() {
  point a = {1, 2};
  point b = a;
  a.x = 5;
  printf("%d %d\n", a.x, b.x);
}
```

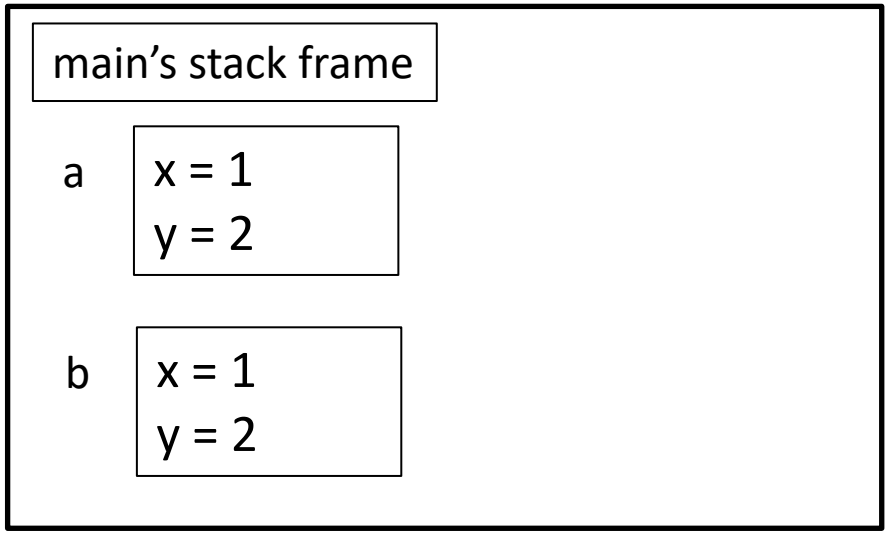
Poll Soln

```c
typedef struct {
  int x, y;
} point;

int main() {
  point a = {1, 2};
  point b = a;
  a.x = 5;
  printf("%d %d\n", a.x, b.x);
}
```



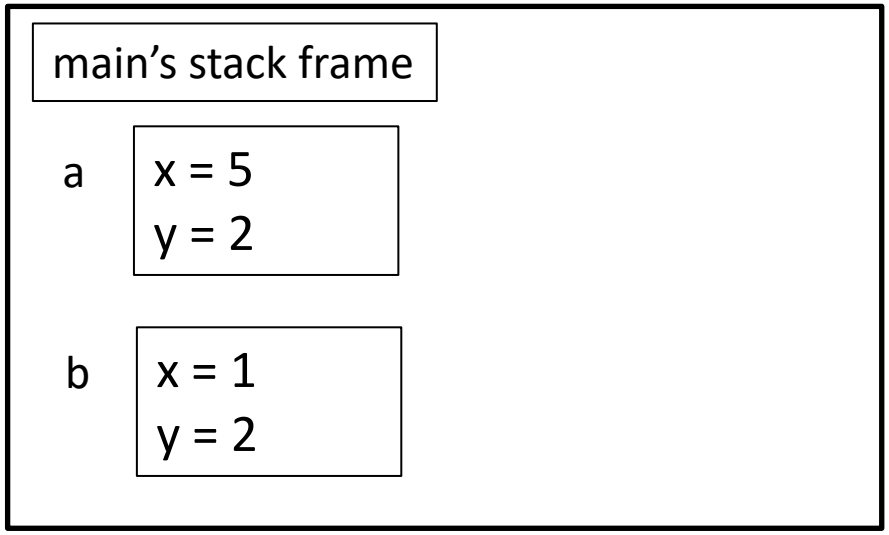
Poll Soln

```c
typedef struct {
  int x, y;
} point;

int main() {
  point a = {1, 2};
  point b = a;
  a.x = 5;
  printf("%d %d\n", a.x, b.x);
}
```



Poll Soln

```c
typedef struct {
  int x, y;
} point;

int main() {
  point a = {1, 2};
  point b = a;
  a.x = 5;
  printf("%d %d\n", a.x, b.x);
}
```



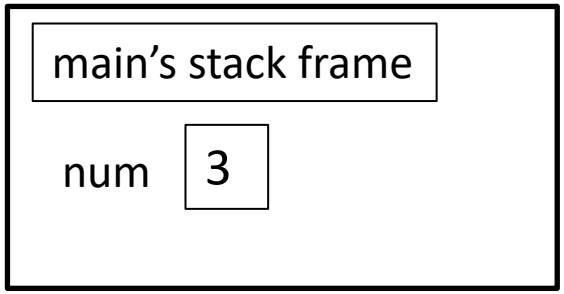
Poll Everywhere

pollev.com/tqm

- What does this code print?

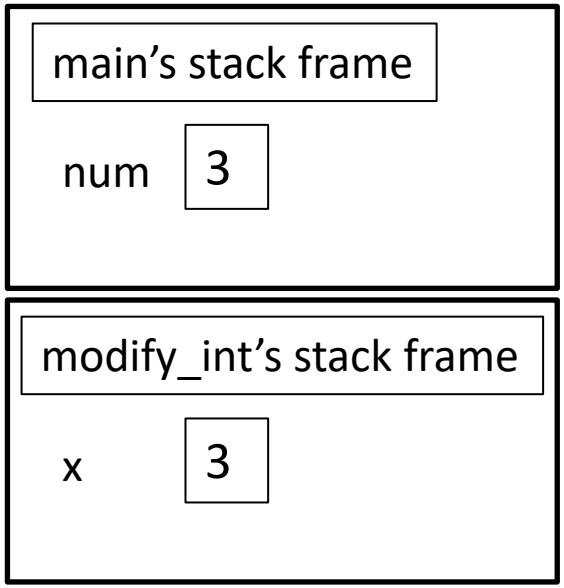
```c
#include <stdio.h>
#include <stdlib.h>

void modify_int(int x) {
  x = 5;
}

int main() {
  int num = 3;
  modify_int(num);
  printf("%d\n", num);
  return EXIT_SUCCESS;
}
```

Poll Soln



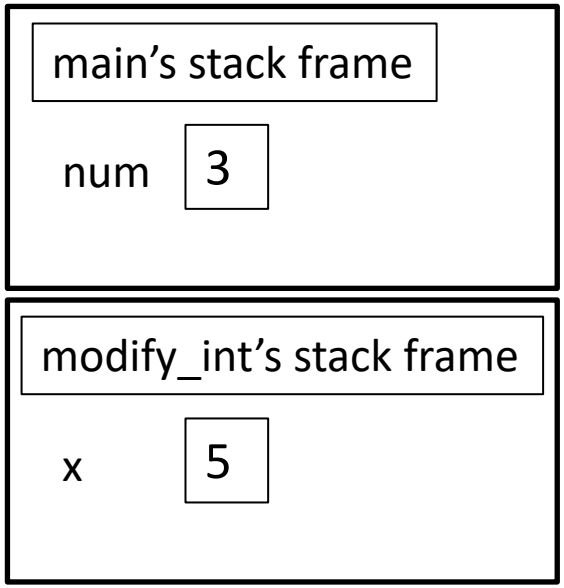

```c
#include <stdio.h>
#include <stdlib.h>

void modify_int(int x) {
  x = 5;
}

int main() {
  int num = 3;
  modify_int(num);
  printf("%d\n", num);
  return EXIT_SUCCESS;
}
```

Poll Soln



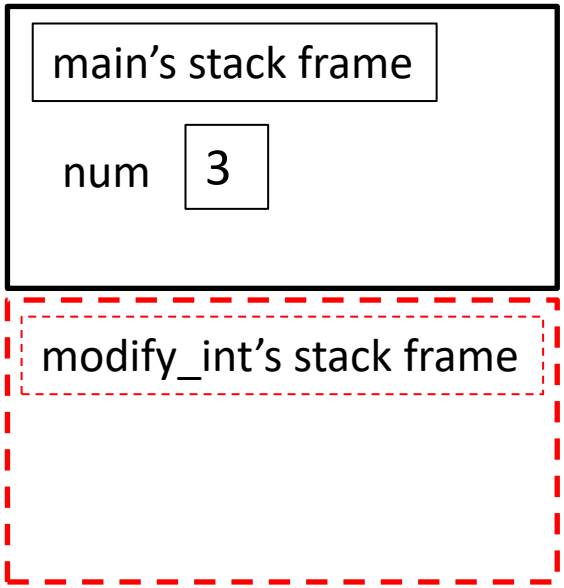

```c
#include <stdio.h>
#include <stdlib.h>

void modify_int(int x) {
  x = 5;
}

int main() {
  int num = 3;
  modify_int(num);
  printf("%d\n", num);
  return EXIT_SUCCESS;
}
```

Poll Soln




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

void modify_int(int x) {
  x = 5;
}

int main() {
  int num = 3;
  modify_int(num);
  printf("%d\n", num);
  return EXIT_SUCCESS;
}
```

Poll Soln




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

void modify_int(int x) {
  x = 5;
}

int main() {
  int num = 3;
  modify_int(num);
  printf("%d\n", num);
  return EXIT_SUCCESS;
}
```

Pass-by Value

- Everything in C is pass-by-value.
  - This means when we call a function, we pass in copies of the argument
  - We can pass pointers to simulate pass-by-reference if needed
    - We often call these “output” parameters since we are able to “return” multiple things.
    - Can also be though of as allowing us to modify things outside the function.

Output Parameters

- Pointers can be used to "return" more than one value from a function

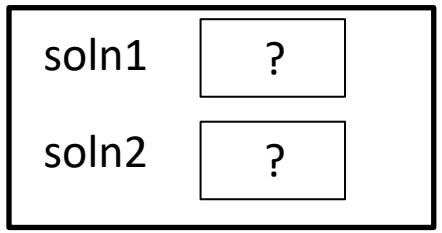
```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```

Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```

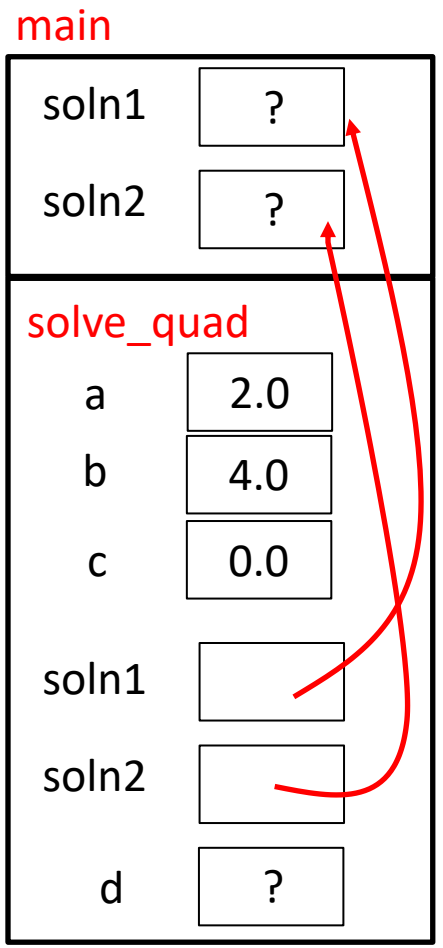


Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```

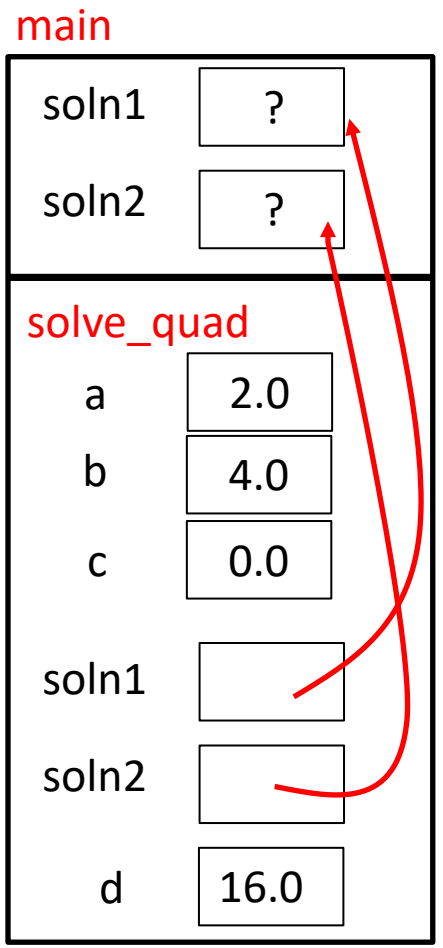


Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```

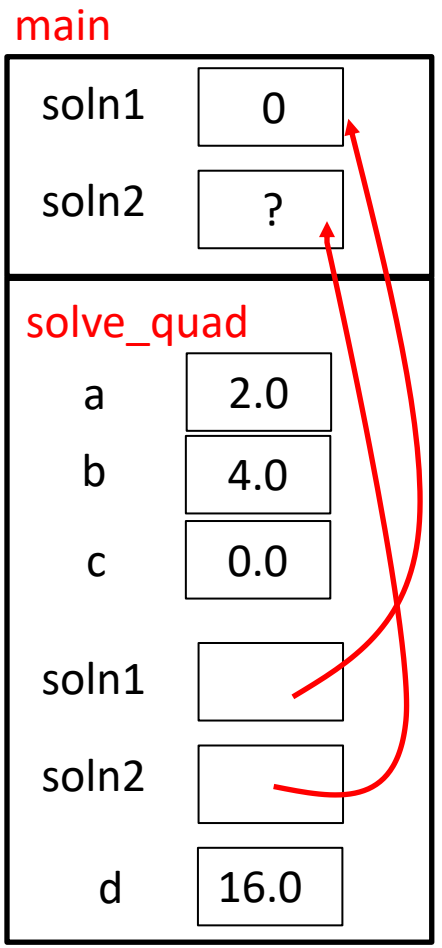


Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```

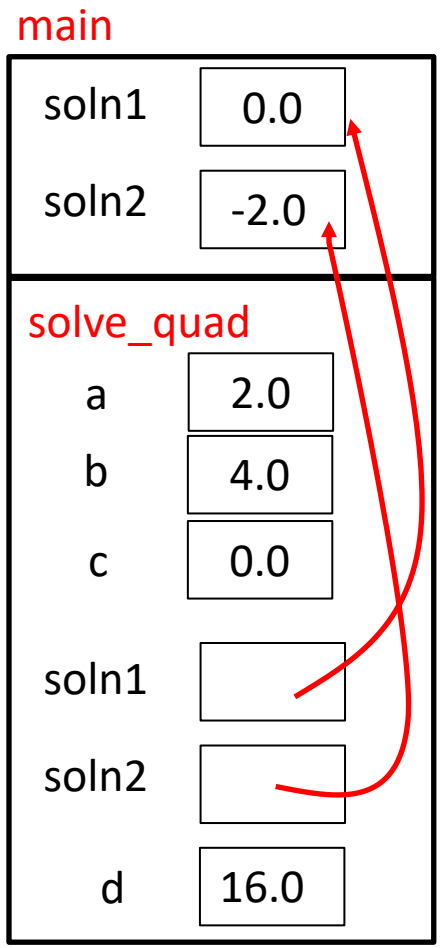


Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```



Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```

main

soln1	0.0
soln2	-2.0

Poll Everywhere

pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

A. 5, 22, 42

B. 42, 42, 37

C. 42, 22, 37

D. 5, 42, 37

E. I'm not sure

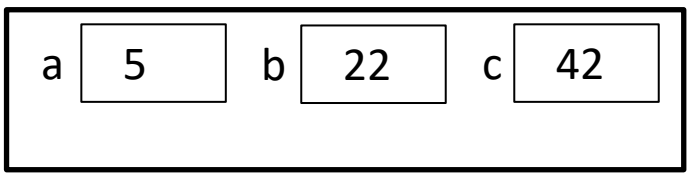
Poll Everywhere

pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



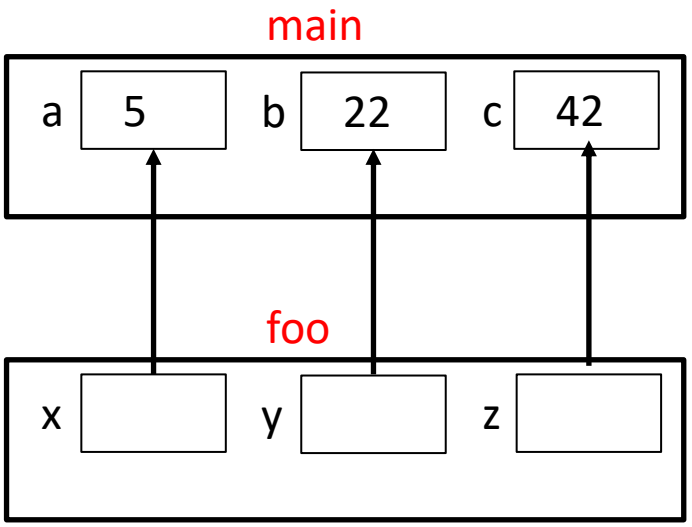
Poll Everywhere

pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



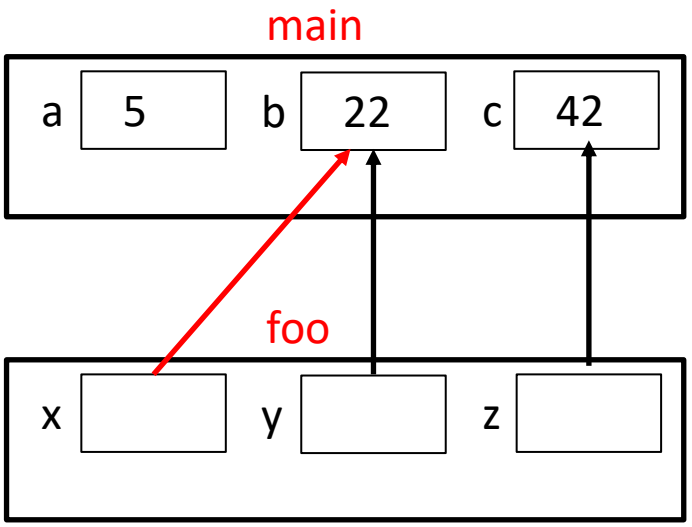
Poll Everywhere

pollev.com/tqm

- What is printed in this program?

Red arrow indicates the NEXT line to execute

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```



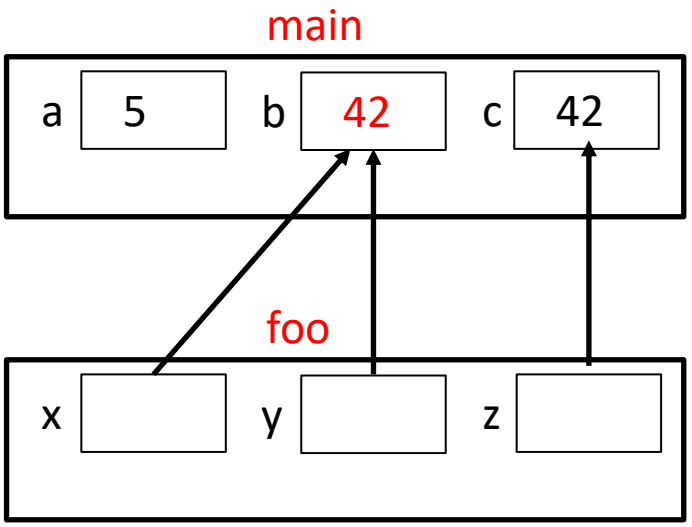
Poll Everywhere

pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



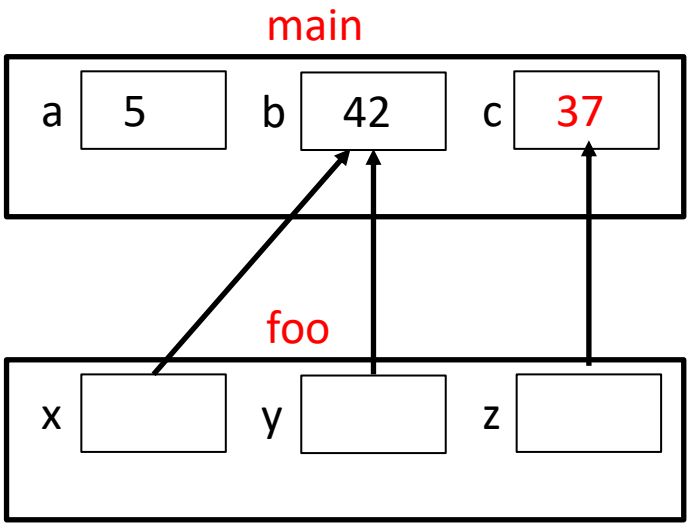
Poll Everywhere

pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



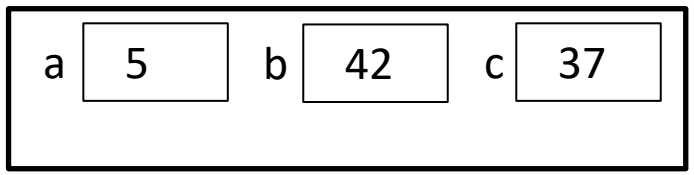
Poll Everywhere

pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



D. 5, 42, 37

pollev.com/tqm

- What does this code print?

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

typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point p) {
  p.x = 3800;
  p.y = 4710;
}

int main() {
  Point p = {1100, 2400};
  modify_point(p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Discuss

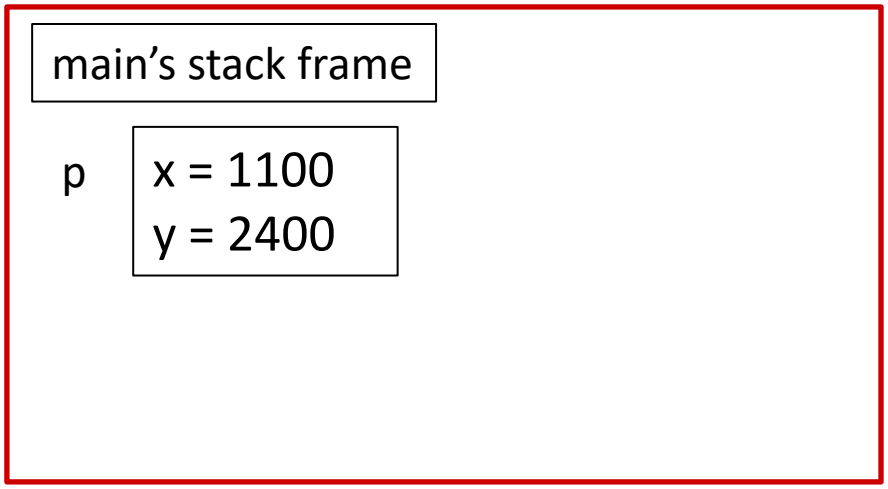
- How can we fix this?

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

typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point p) {
  p.x = 3800;
  p.y = 4710;
}

int main() {
  Point p = {1100, 2400};
  modify_point(p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Demo: pass_by.c

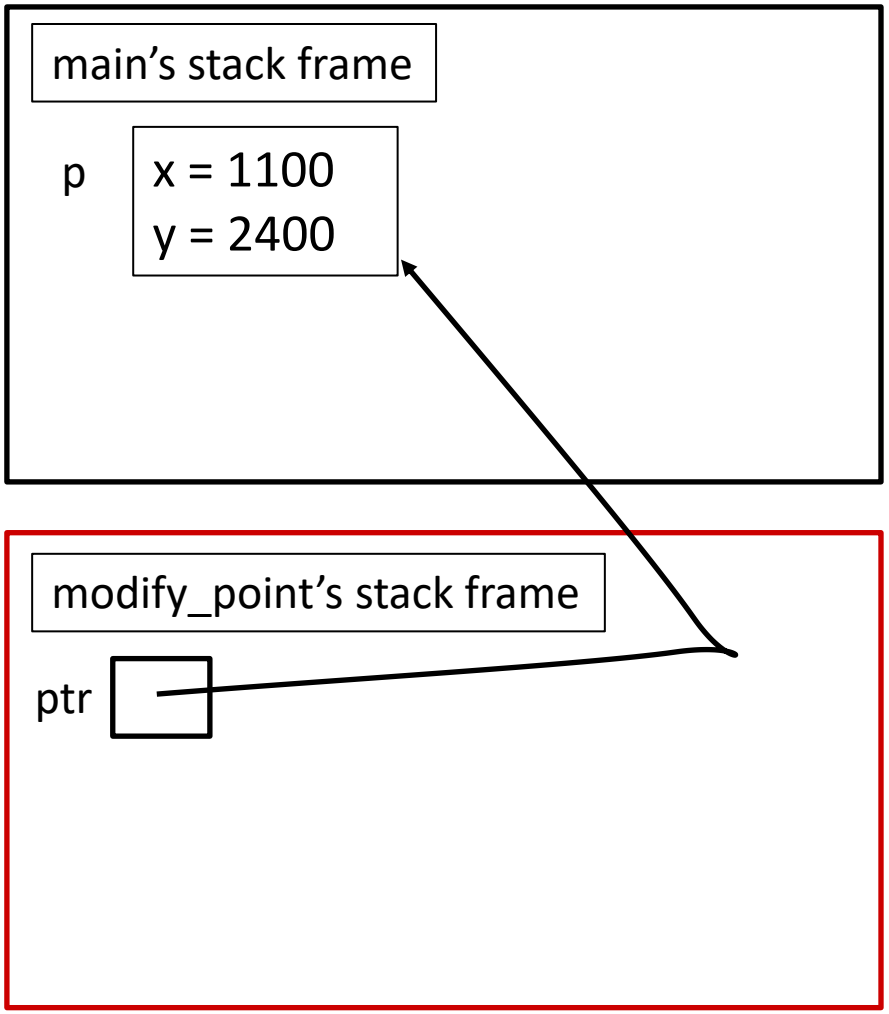
- Everything in C is pass-by value (e.g. a copy is passed to the function)
- HOWEVER, we can pass a copy of a pointer (e.g. a reference to something) to mimic pass-by-reference.
- Demo pass_by.c
  - Note: most lecture code will be available on the course website

Visualization: faulty pass by reference



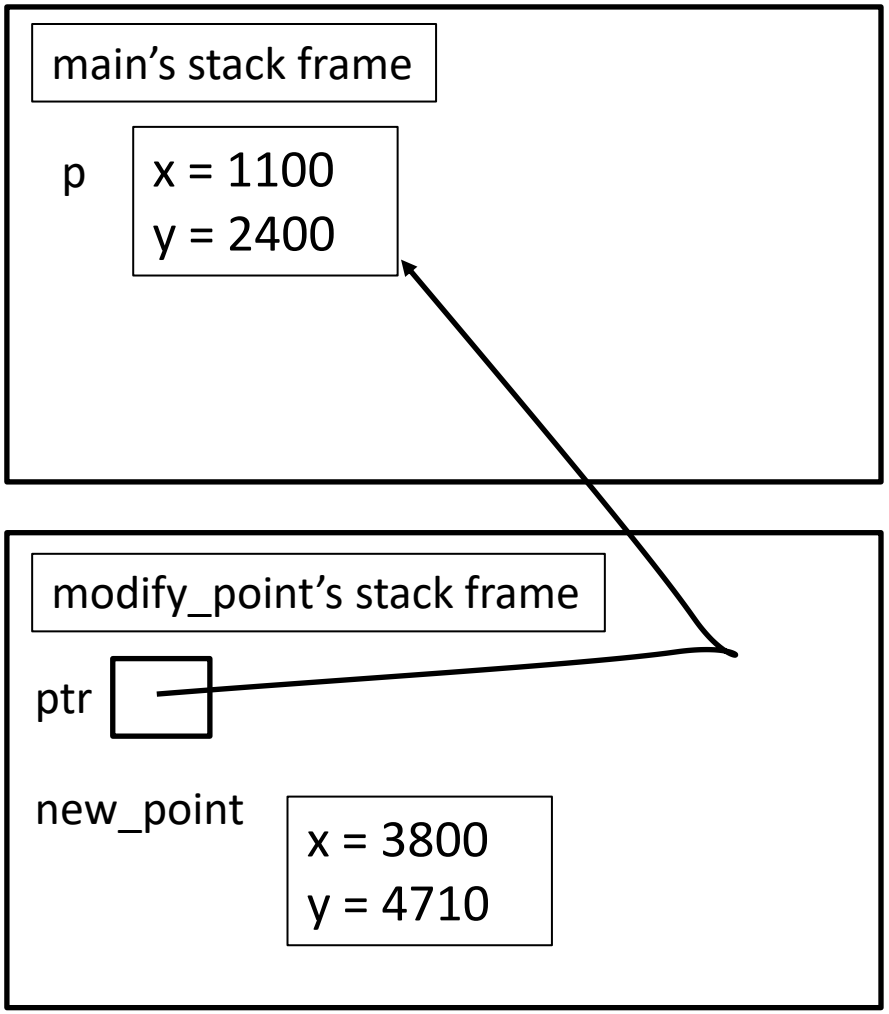

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: faulty pass by reference



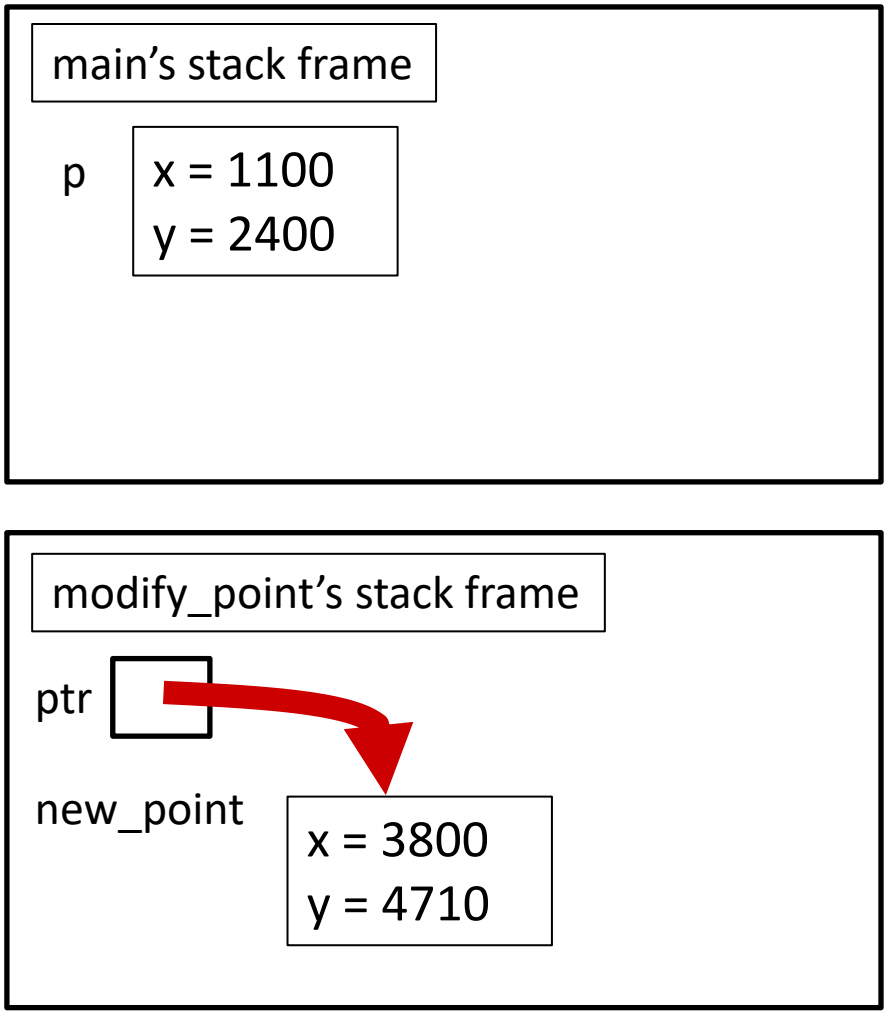

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: faulty pass by reference



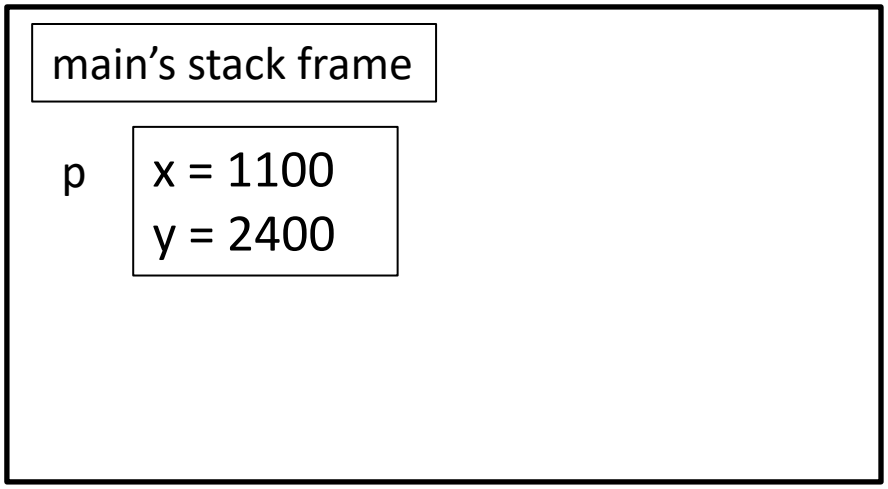

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: faulty pass by reference




```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: faulty pass by reference




```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Gap slide

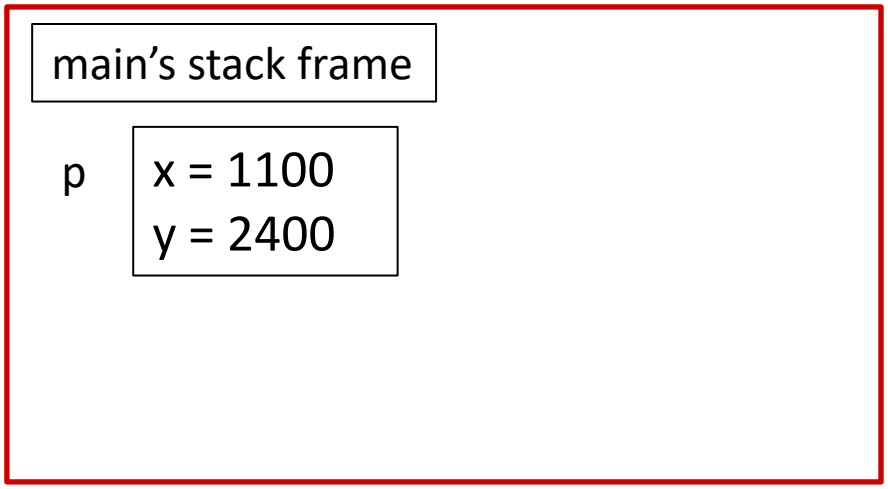
- Slide to make clear that we are moving onto a new example (that looks very similar)

Visualization: fixed pass by reference

Buggy version said:
`ptr = &new_point`

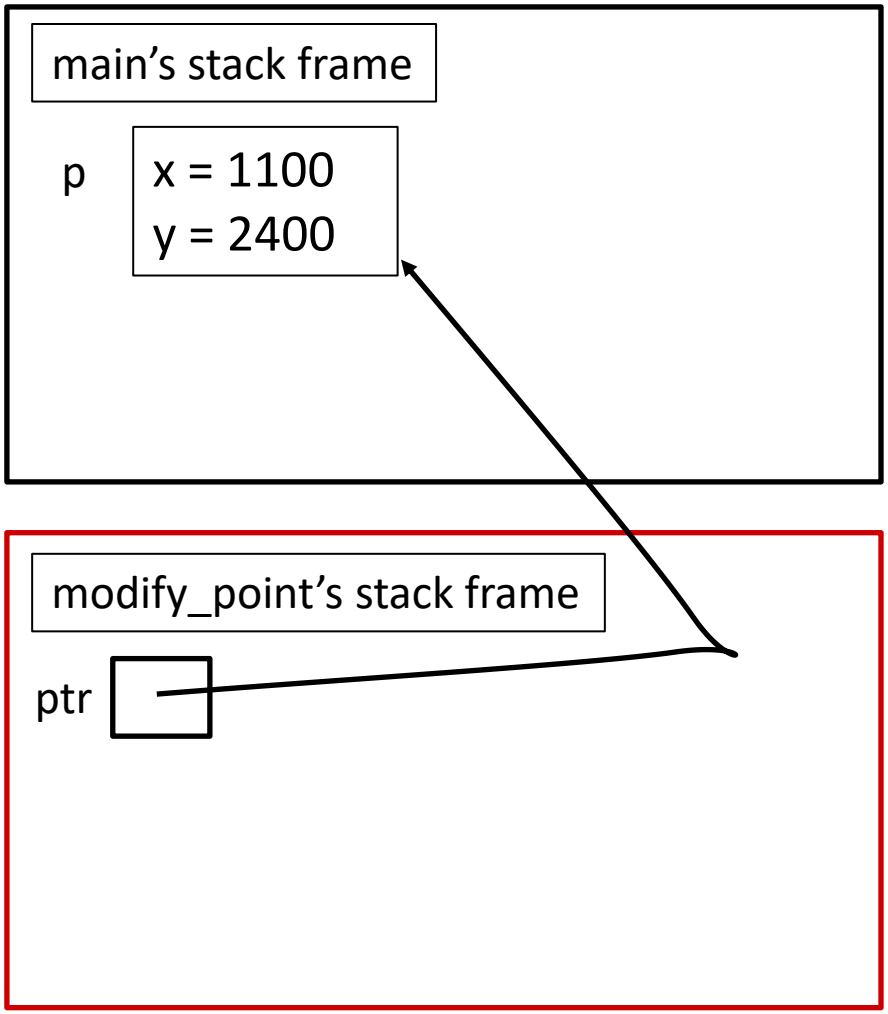
```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference



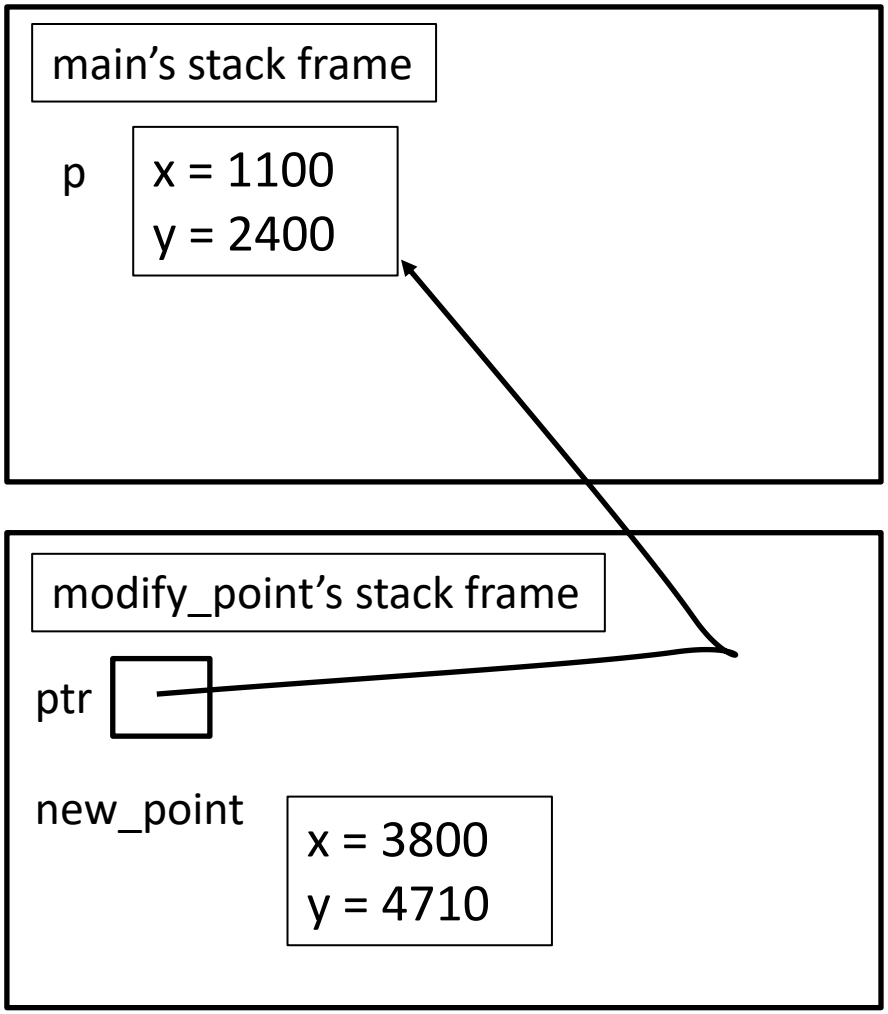

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference



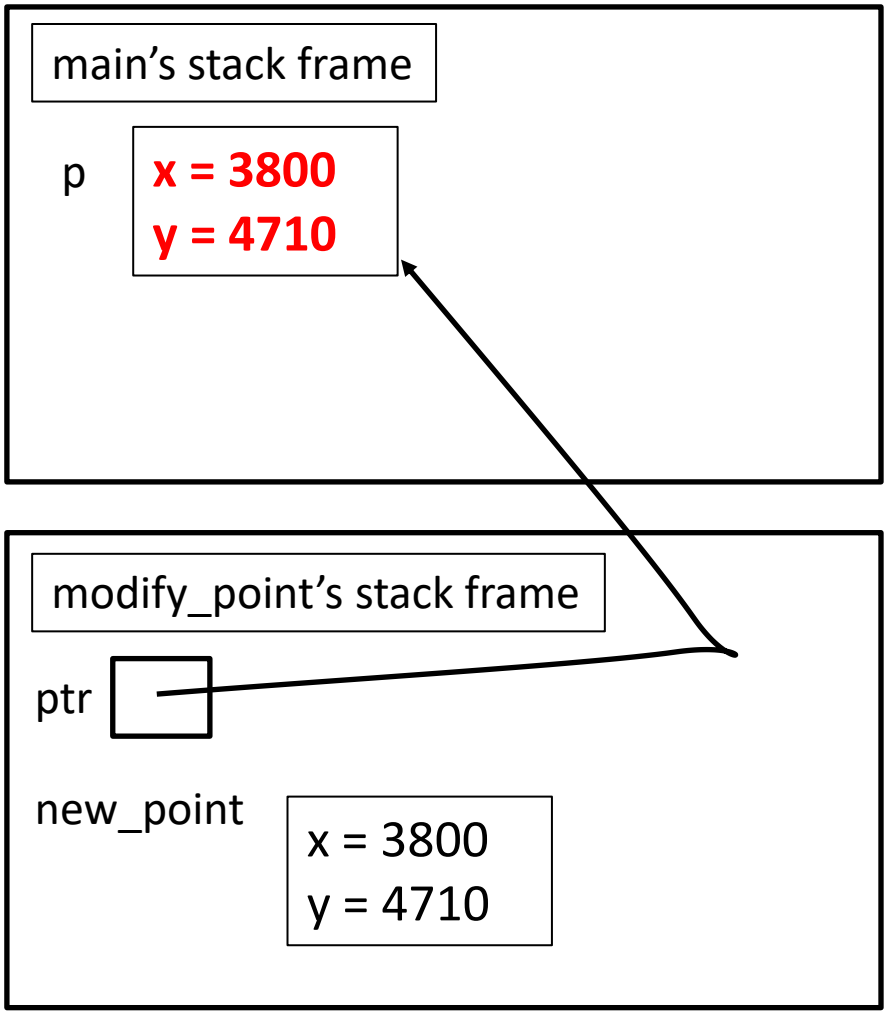

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference



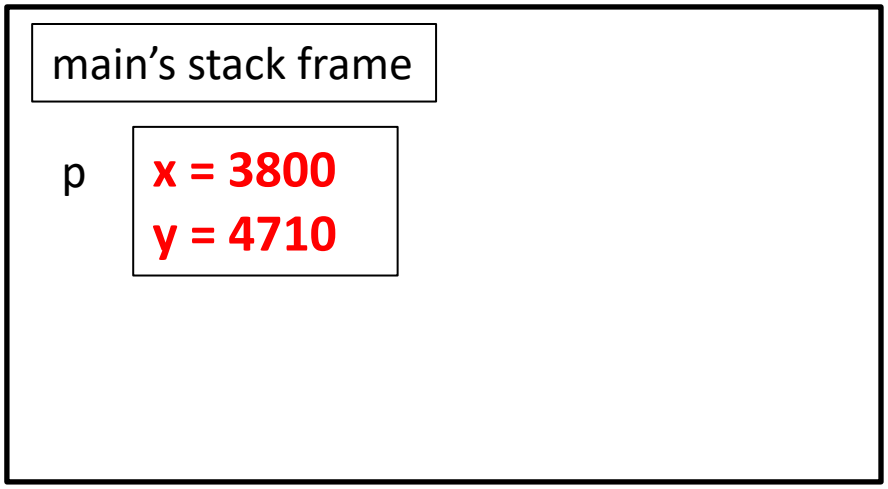

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference




```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference




```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Lecture Outline

- C memory Review:
  - An array of bytes
  - Pointers & arrays
  - The Stack
- Structs, Pass-by-value, Output Parameters
- **The Heap**
  - **malloc() & free()**

Aside: `sizeof`

- **`sizeof`** operator can be applied to a variable or a type and it evaluates to the size of that type in bytes
- Examples:
  - **`sizeof(int)`** – returns the size of an integer
  - **`sizeof(double)`** – returns the size of a double precision number
  - **`struct my_struct s;`**
    - **`sizeof(s)`** – returns the size of the struct s
  - **`my_type *ptr`**
    - **`sizeof (*ptr)`** – returns the size of the type pointed to by ptr
- Very useful for Dynamic Memory

What is Dynamic Memory Allocation?

- We want Dynamic Memory Allocation
  - Dynamic means “at run-time”
  - The compiler and the programmer don’t have enough information to make a final decision on how much to allocate
  - Your program explicitly requests more memory at run time
  - The language allocates it at runtime, maybe with help of the OS

- Dynamically allocated memory persists until either:
  - A garbage collector collects it (automatic memory management)
  - Your code explicitly deallocates it (manual memory management)

- C requires you to manually manage memory
  - More control, and more headaches

Heap API

- Dynamic memory is managed in a location in memory called the "Heap"
  - The heap is managed by user-level runetime library (libc)
  - Interface functions found in `<stdlib.h>`
- Most used functions:
  - **`void *malloc(size_t size);`**
    - Allocates memory of specified size
  - **`void free(void *ptr);`**
    - Deallocates memory
- Note: **`void*`** is “generic pointer”. It holds an address, but doesn’t specify what it is pointing at.
- Note 2: **`size_t`** is the integer type of **`sizeof()`**

malloc()

- `void *malloc(size_t size);`
- **malloc** allocates a block of memory of the requested size
  - Returns a pointer to the first byte of that memory
    - And returns `NULL` if the memory allocation failed!
  - You should assume that the memory initially contains garbage
  - You'll typically use `sizeof` to calculate the size you need

```
// allocate a 10-float array
float* arr = malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

ALWAYS CHECK FOR NULL

free()

- Usage: `free(pointer);`
- Deallocates the memory pointed-to by the pointer
  - Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by **malloc**)
  - Freed memory becomes eligible for future allocation
  - `free(NULL);` does nothing.
  - The bits in the pointer are *not changed* by calling free
    - Defensive programming: can set pointer to NULL after freeing it

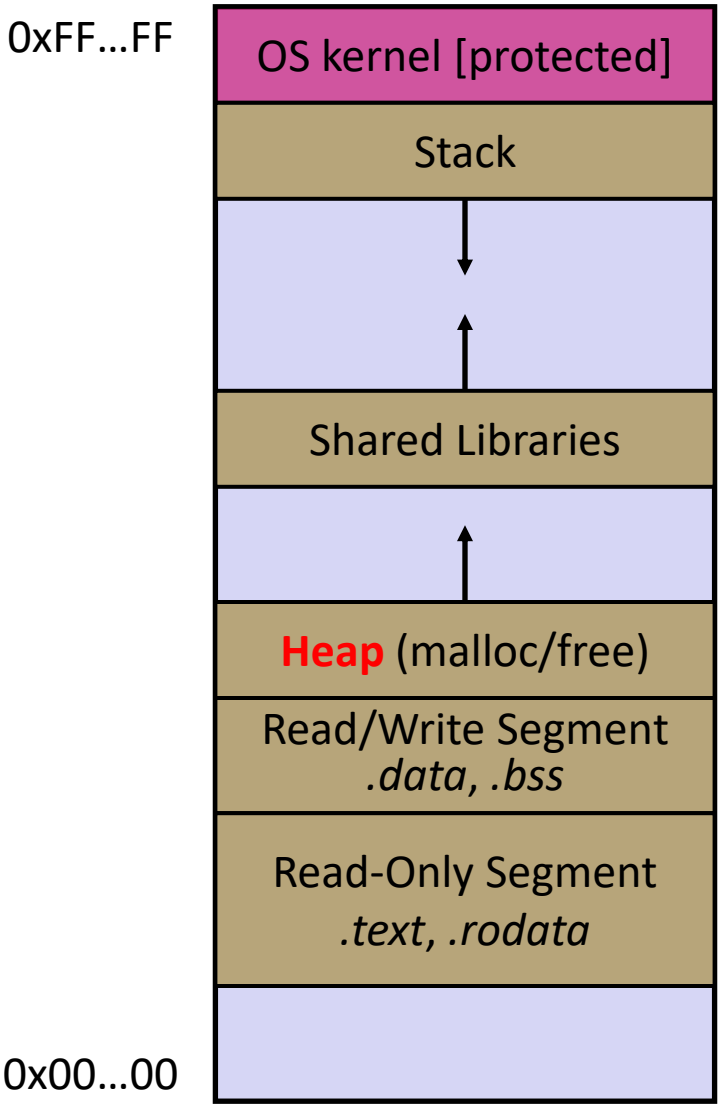
```
float* arr = malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...             // do stuff with arr
free(arr);
arr = NULL;     // OPTIONAL
```

The Heap

- The Heap is a large pool of available memory to use for Dynamic allocation
- This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- **`malloc`**:
  - searches for a large enough unused block of memory
  - marks the memory as allocated.
  - Returns a pointer to the beginning of that memory
- **`free`**:
  - Takes in a pointer to a previously allocated address
  - Marks the memory as free to use.

The Heap

- The Heap is a large pool of available memory used to hold dynamically-allocated data
  - `malloc` allocates chunks of data in the Heap; `free` deallocates those chunks
  - `malloc` maintains bookkeeping data in the Heap to track allocated blocks

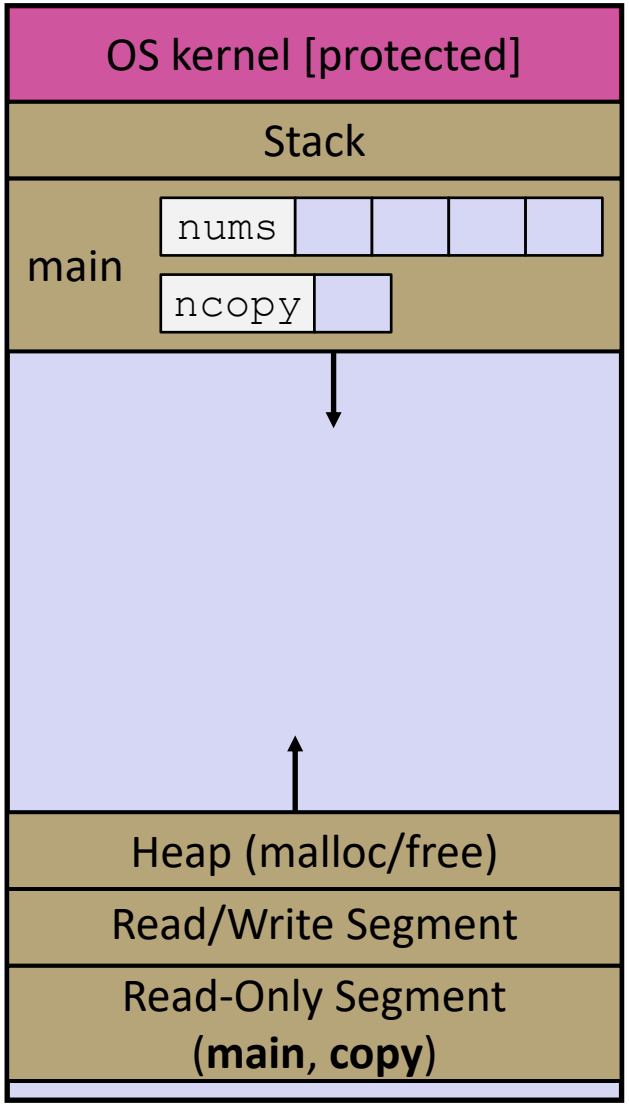


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

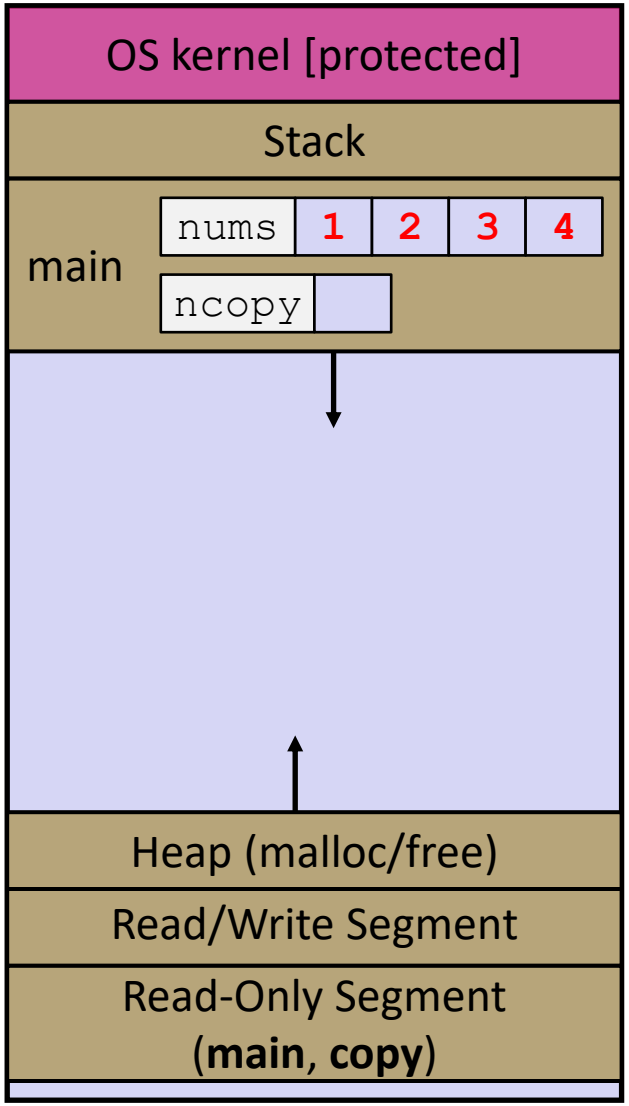


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

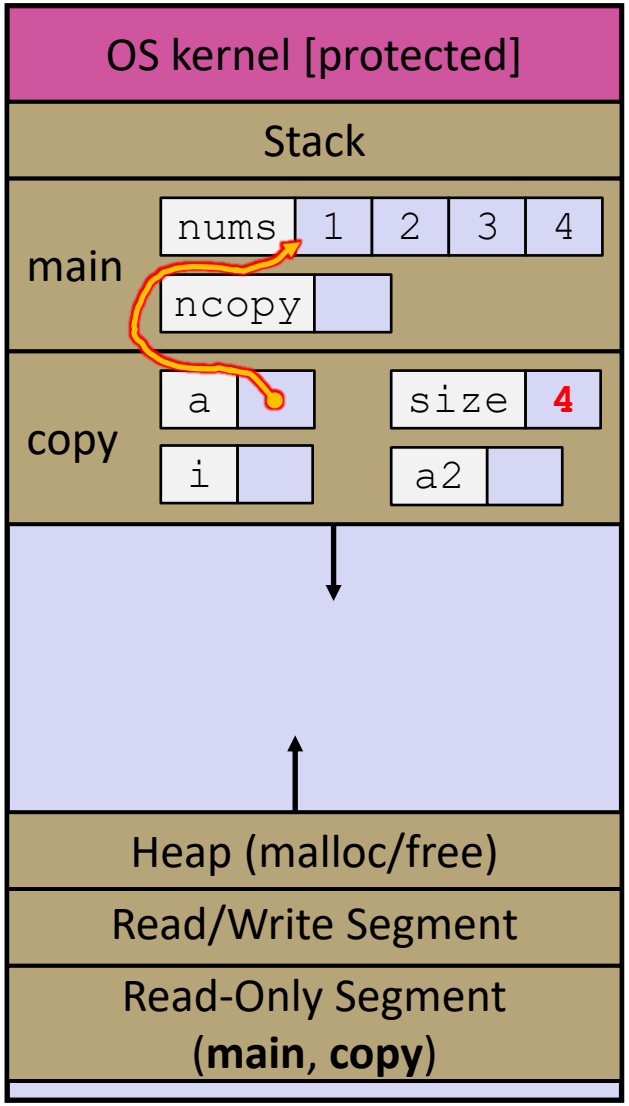


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

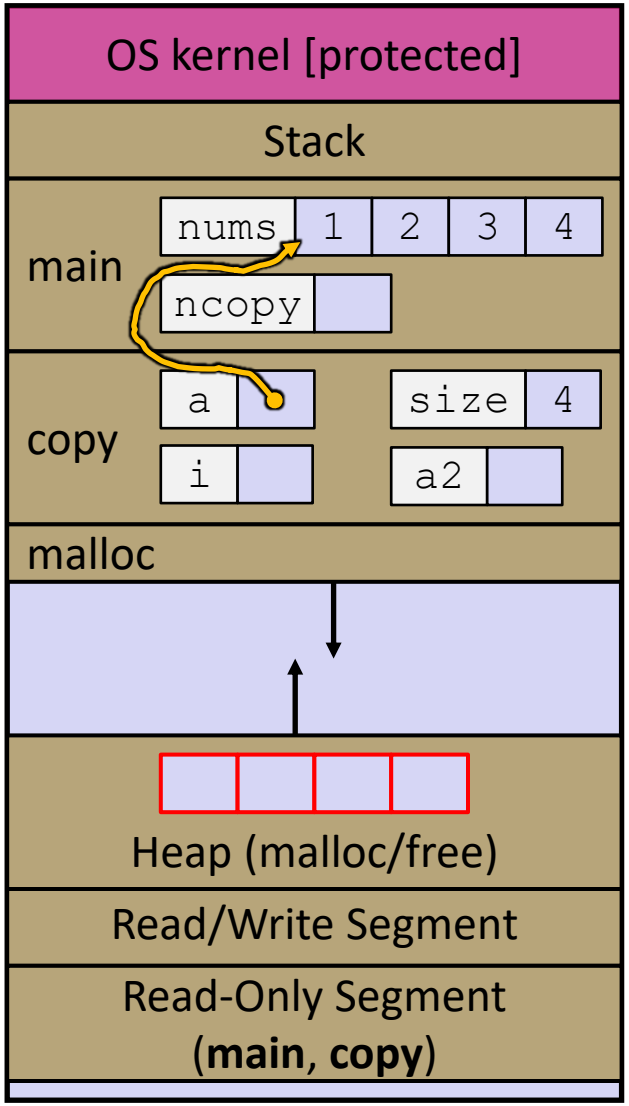


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

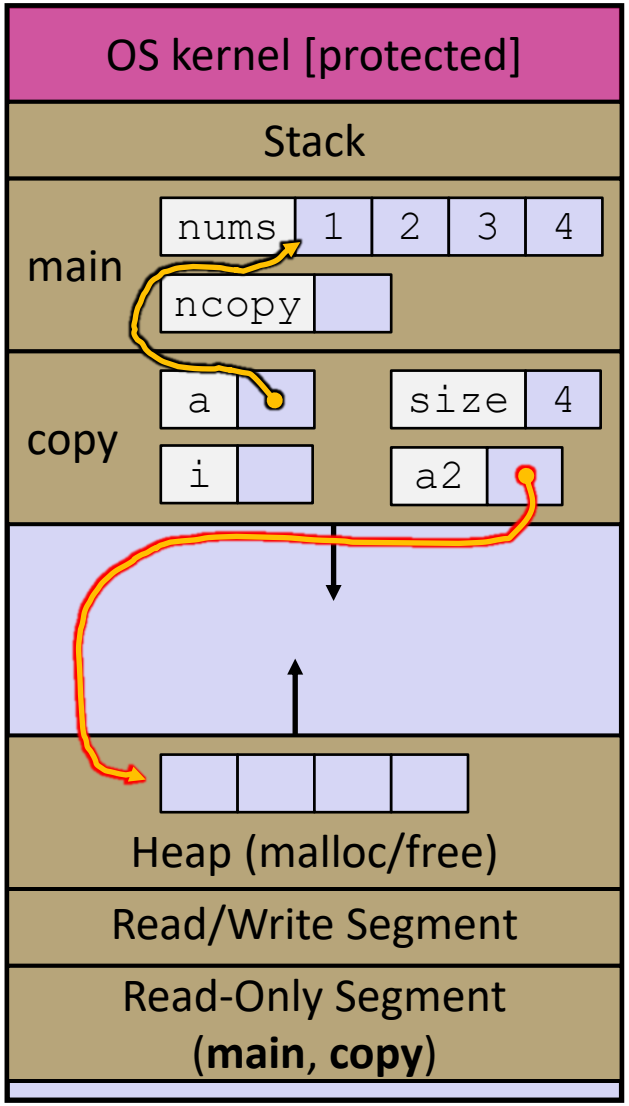


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

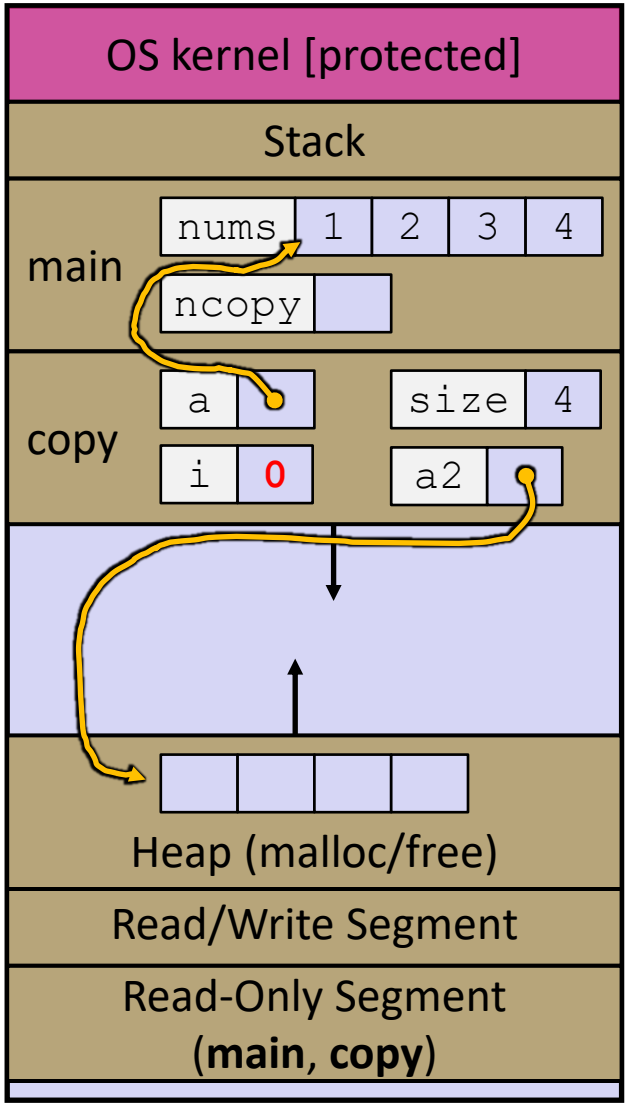


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

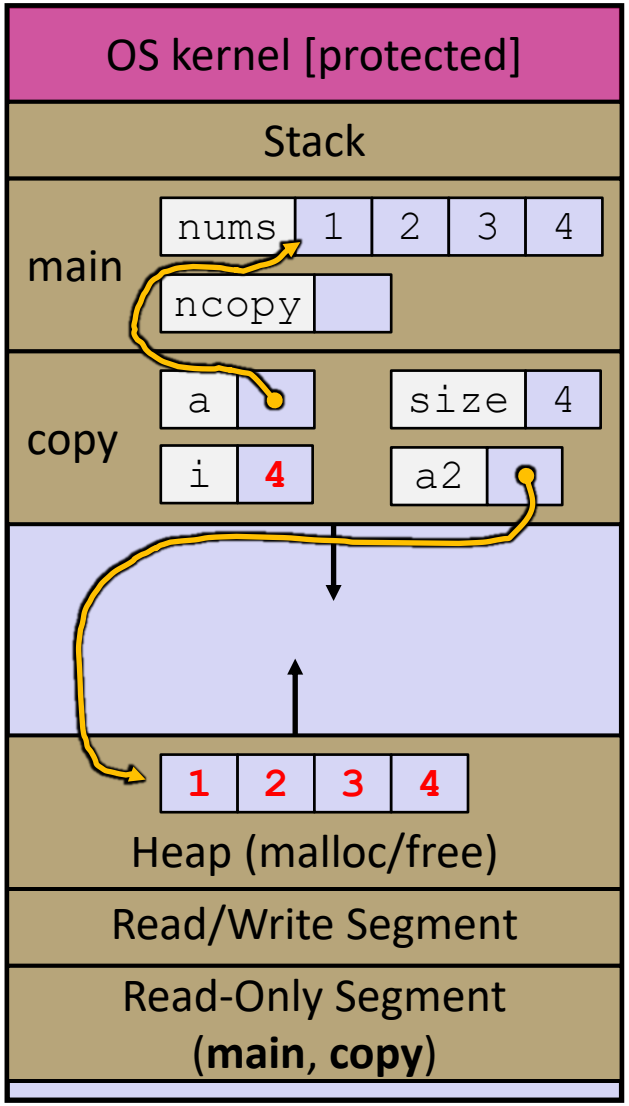


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

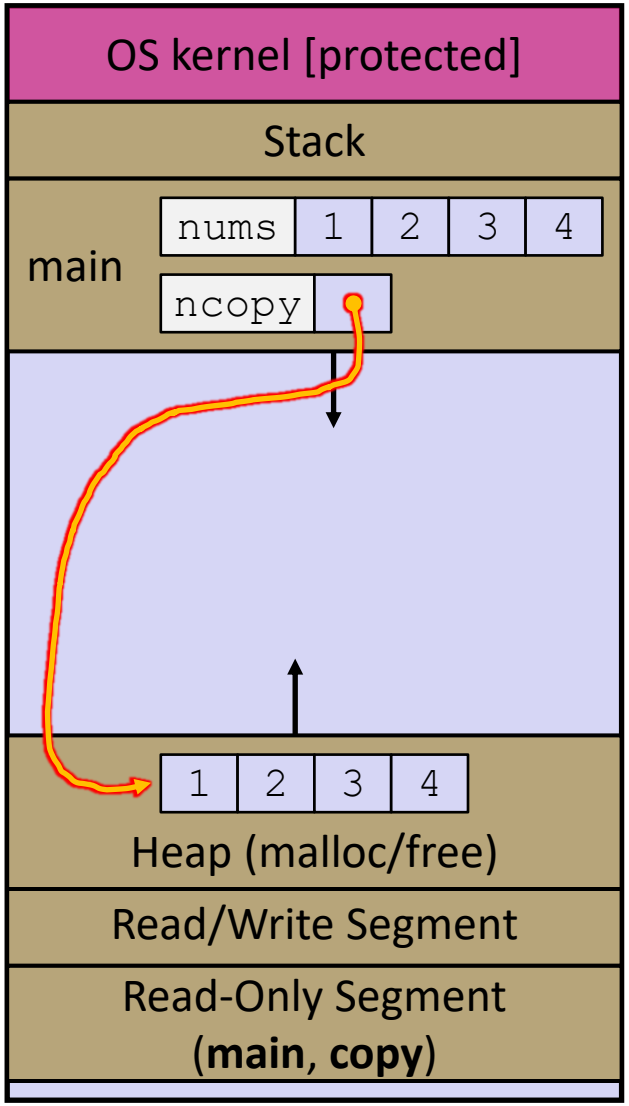


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

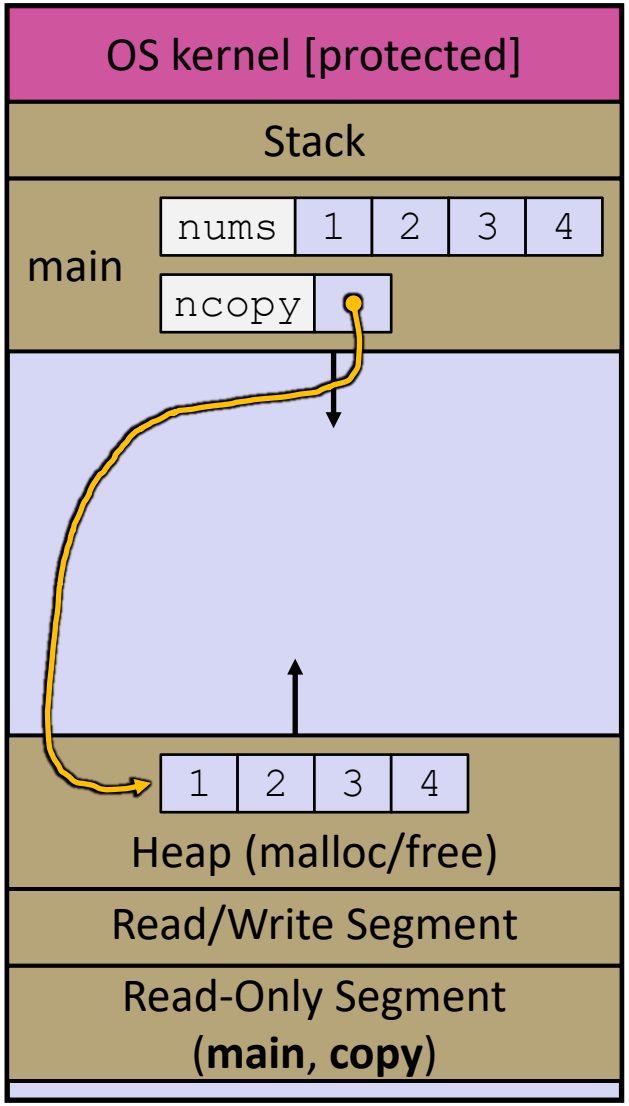


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

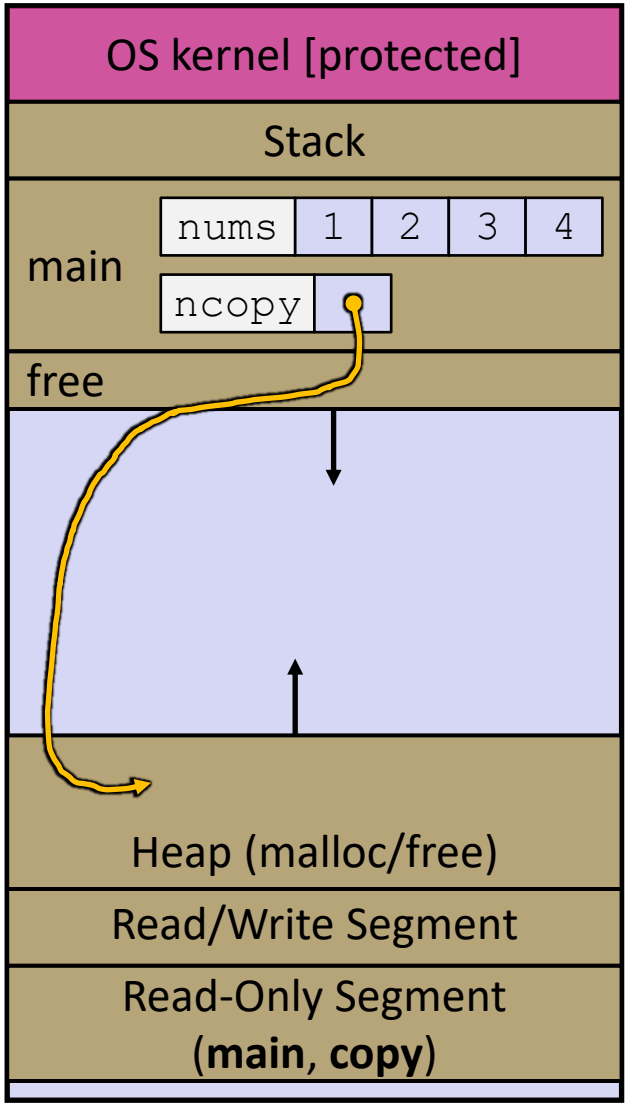


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```

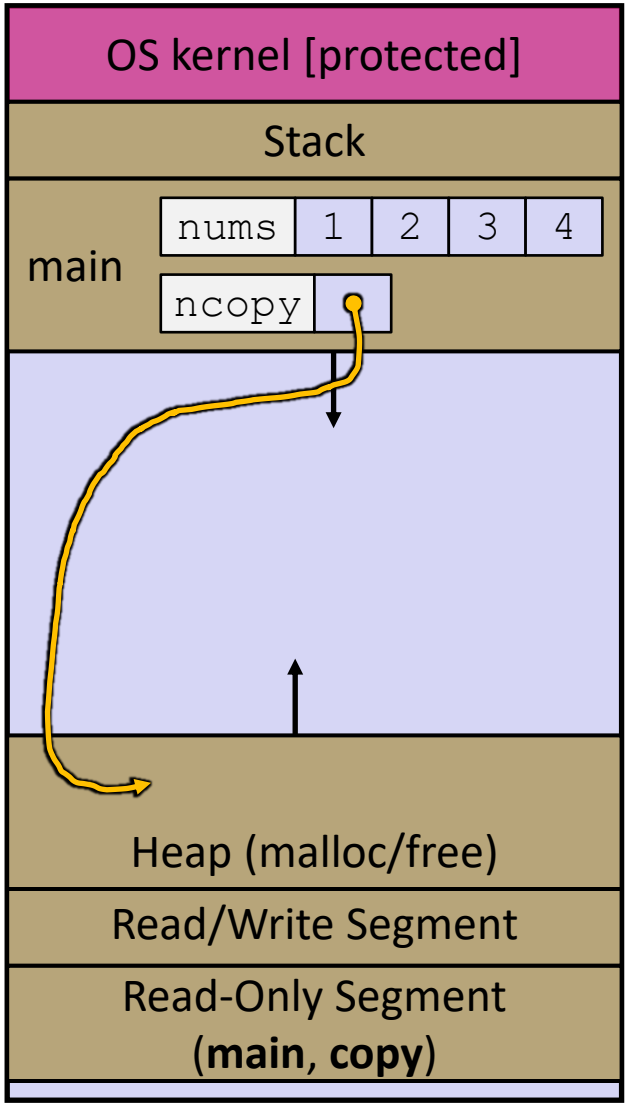


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Dynamic Memory Pitfalls

- Buffer Overflows
  - E.g. ask for 10 bytes, but write 11 bytes
  - Could overwrite information needed to manage the heap
  - Common when forgetting the null-terminator on malloc'd strings
- Not checking for **`NULL`**
  - Malloc returns NULL if out of memory
  - Should check this after every call to malloc
- Giving **`free()`** a pointer to the middle of an allocated region
  - Free won't recognize the block of memory and may crash
- Giving free() a pointer that has already been freed
  - Will interfere with the management of the heap and likely crash
- **`malloc`** does NOT initialize memory
  - There are other functions like **`calloc`** that will zero out memory

Memory Leaks

- The most common Memory Pitfall
- What happens if we malloc something, but don't free it?
  - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **`free`**d
  - If this happens enough, we run out of heap space and program may slow down and eventually crash
- Garbage Collection
  - Automatically "frees" anything once the program has lost all references to it
  - Affects performance, but avoid memory leaks
  - Java has this, C doesn't

Poll Everywhere

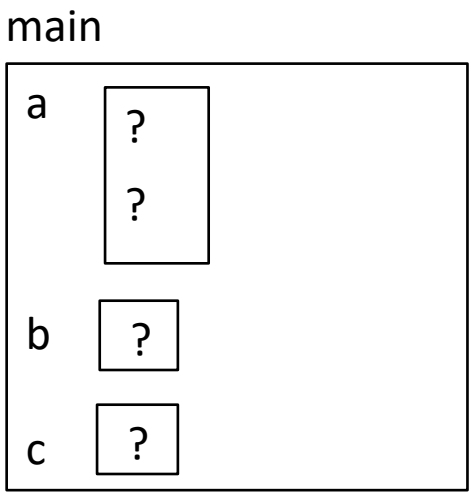
pollev.com/tqm

- Which line below is first to cause a crash?
  - Yes, there are a lot of bugs, but not all cause a crash ☺
  - See if you can find all the bugs!

A. Line 1
B. Line 4
C. Line 6
D. Line 7
E. We're lost…

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

1   a[2] = 5;
2   b[0] += 2;
3   c = b+3;
4   free(&(a[0]));
5   free(b);
6   free(b);
7   b[0] = 5;

  return 0;
}
```

Memory Corruption - What Happens?



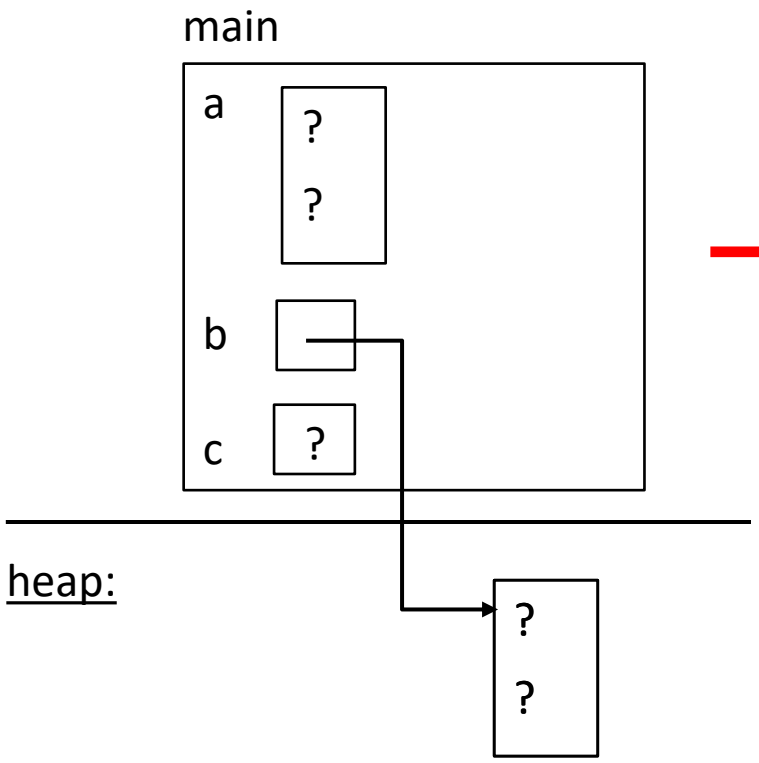
heap:

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

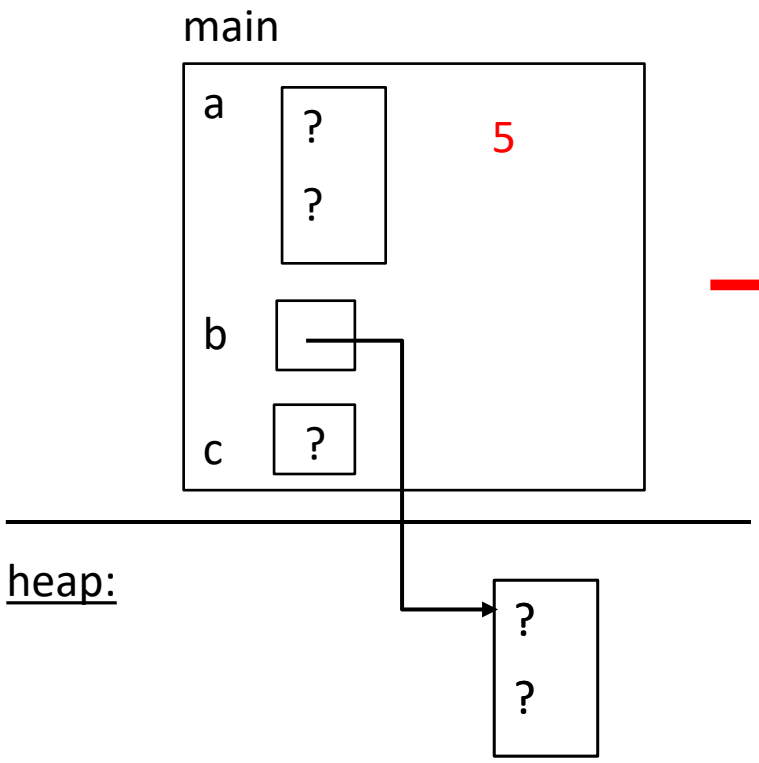



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

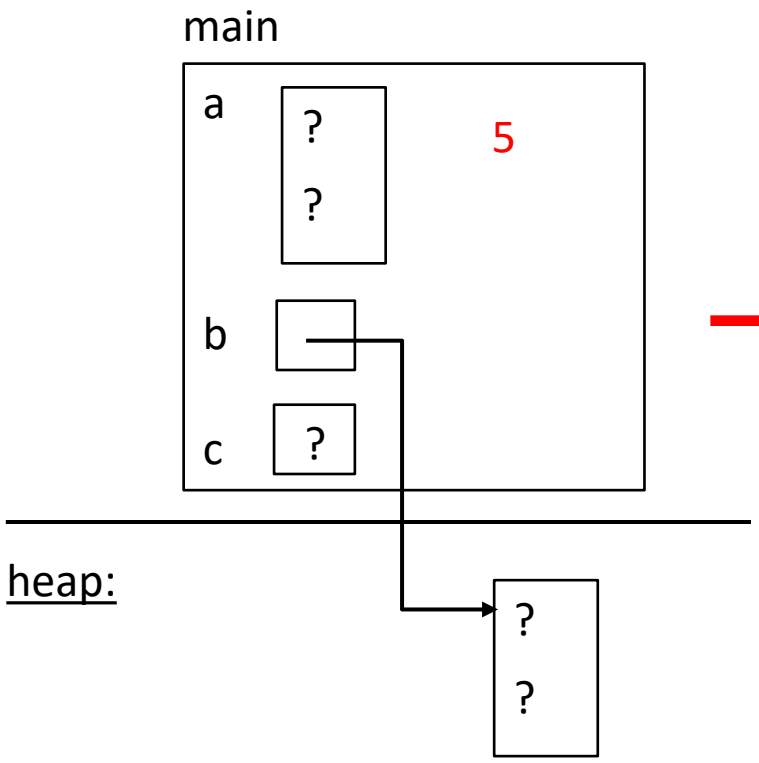



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

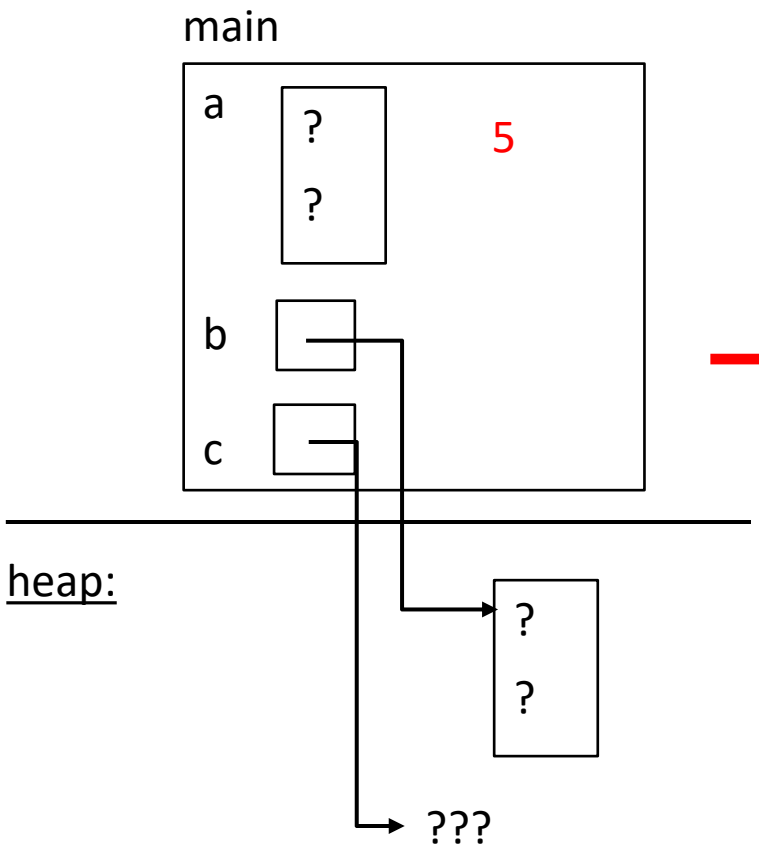



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

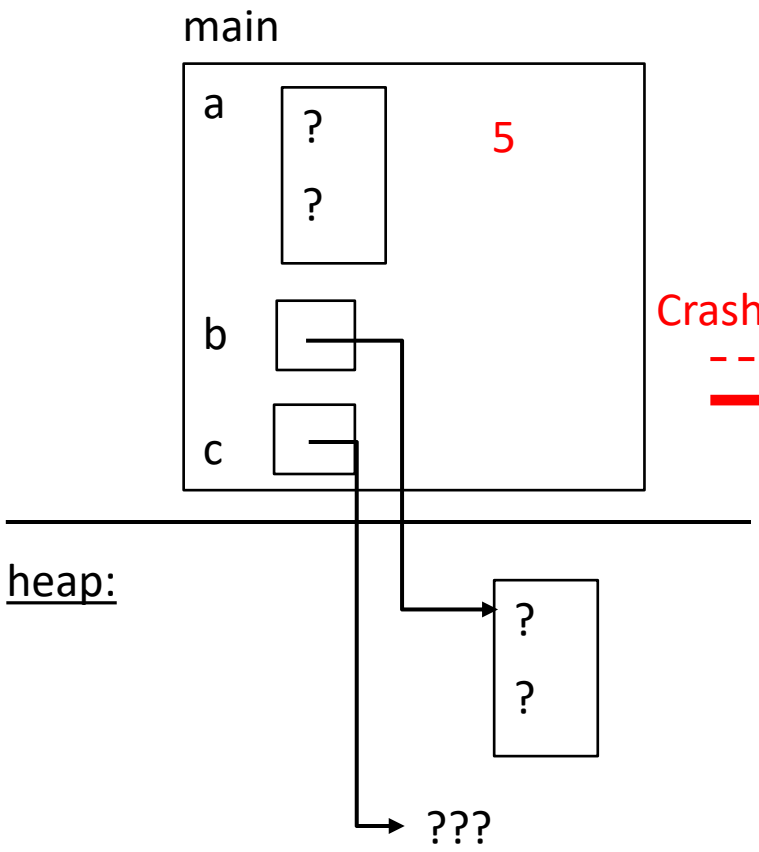



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



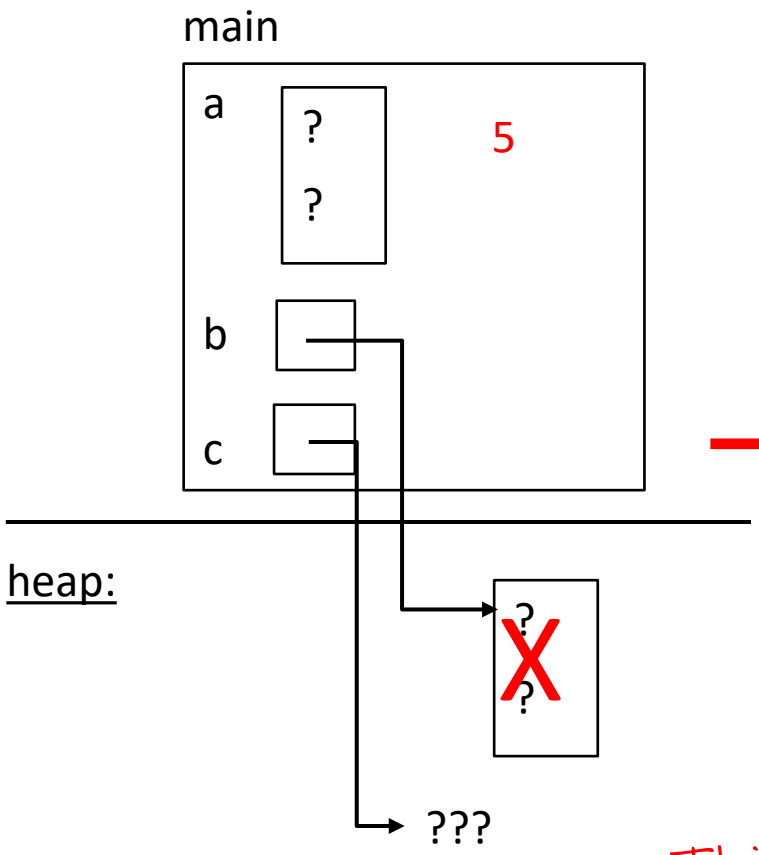
Crash!

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



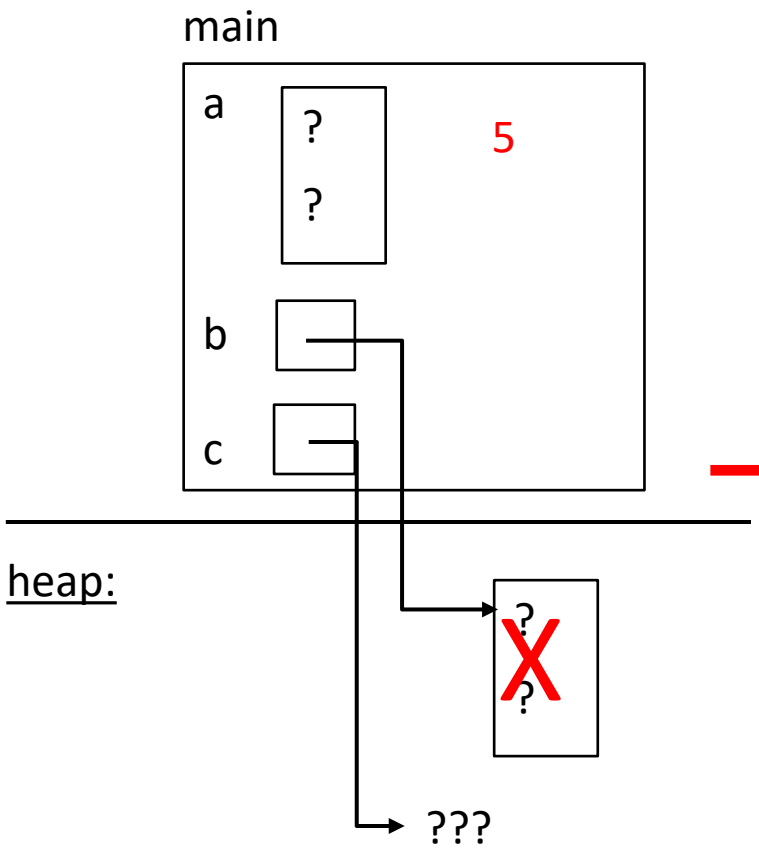

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

This "double free" would also cause the program to crash

memcorrupt.c

Memory Corruption - What Happens?

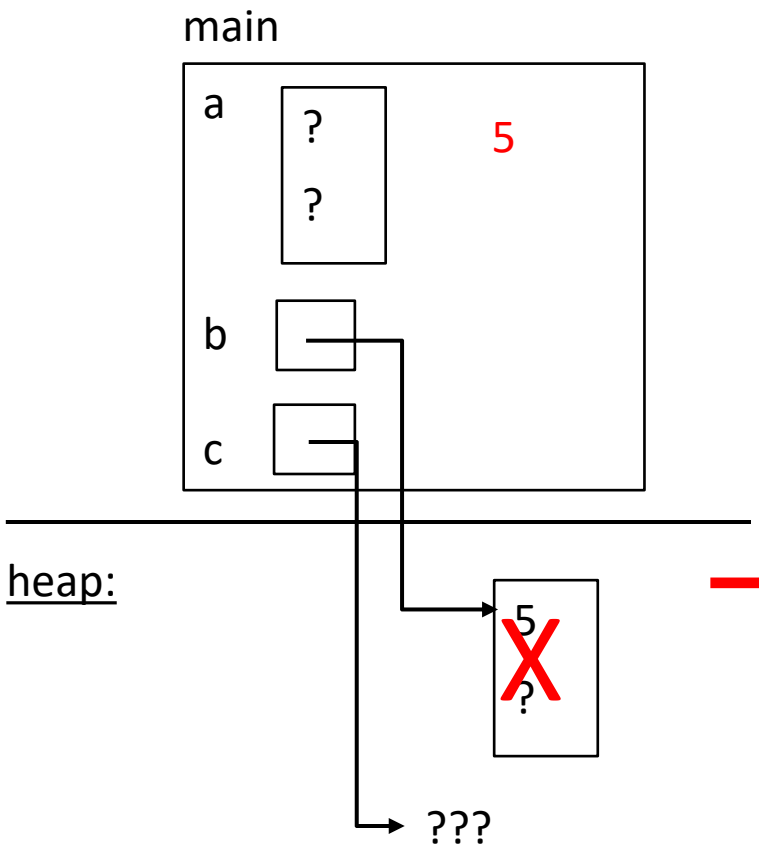



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?




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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Next Time

- More on The Heap!
  - Making our own data structures
- Valgrind! A tool to check for memory errors
- Header Files