Memory Allocation & Structs

Intro to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema, Andrew Rigas, Anisha Bhatia

Audrey Yang, Craig Lee, Daniel Duan

David LuoZhang, Eddy Yang, Ernest Ng

Heyi Liu, Janavi Chadha, Jason Hom

Katherine Wang, Kyrie Dowling, Mohamed Abaker

Noam Elul, Patricia Agnes, Patrick Kehinde Jr.

Ria Sharma, Sarah Luthra, Sofia Mouchtaris

Upcoming Due Dates

- HW06 (Video Game) Due Friday 11/4 @ 11:59 pm
- Midsemester Survey Due Wednesday 11/9 @ 11:59 pm
- Check-in07 Due Monday 11/7 @ 4:59 pm
- HW03 Regrade Requests are open
  - Closes at 11:59 pm Friday 11/4
- Students taking final exams through the Weingarten Center need to schedule by Nov 30th to guarantee extended time for an exam.

Any Logistical Questions? Thoughts? Feelings?

This Lecture:

- The idea with this lecture is to understand where data is stored over the lifetime of a C program

- Three types of data allocation:
  - Static (e.g. Globals)
  - Automatic (e.g. Local Variables & the stack)
  - Dynamic (e.g. stored on the Heap)

LC4 Data Memory So Far

Recall the full LC-4 Memory Map

- We have 2 Program Memories
- And 2 Data Memories

User Region

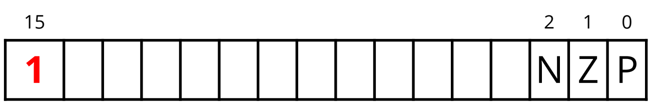
- Programs run by users (e.g.: factorial program)
- Processes run in user mode have **PSR[15]=0**
- ***NOT allowed*** to access OS locations in memory

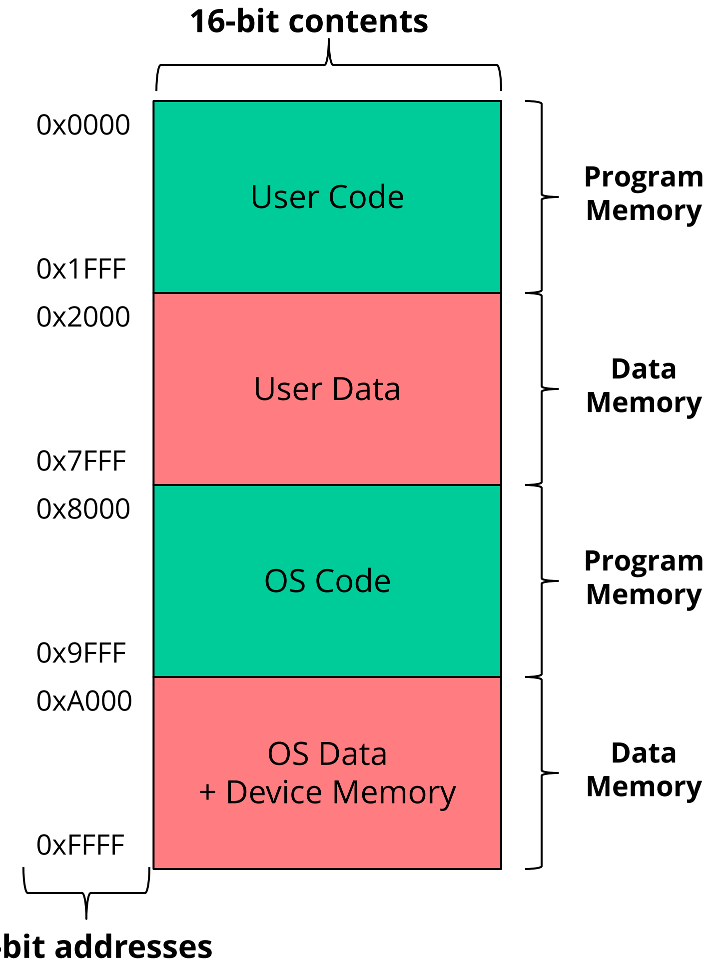
Operating System Region

- Programs run by OS (e.g.: I/O device programs)
- Processes run in OS mode have **PSR[15]=1**
- ***Allowed*** to access OS & User locations in memory

Processor Status Register

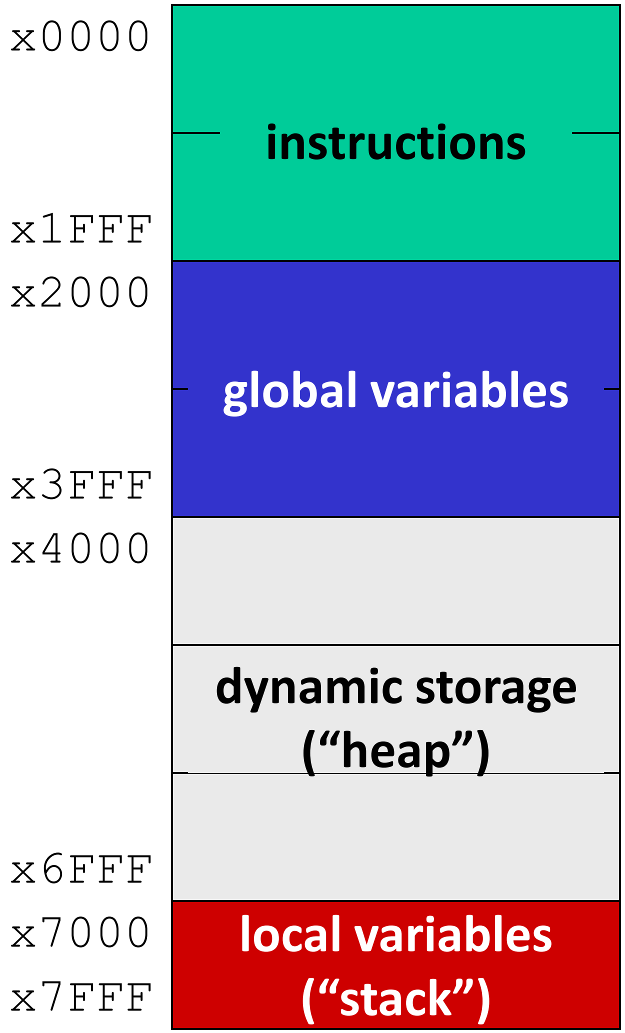
- Contains the privilege bit
- Contains the three NZP bits





LC4 User Memory Layout for C

- LC4 User memory has CODE and DATA portions. But the DATA is split into three parts for running C code
- Global Variables
- Dynamic Storage (the heap)
- Local Variables (the stack)



Lecture Outline

- **Global Memory**
- The Stack
- The Heap
  - malloc() & free()
- Structs & C Data Structures

Global Variables in C

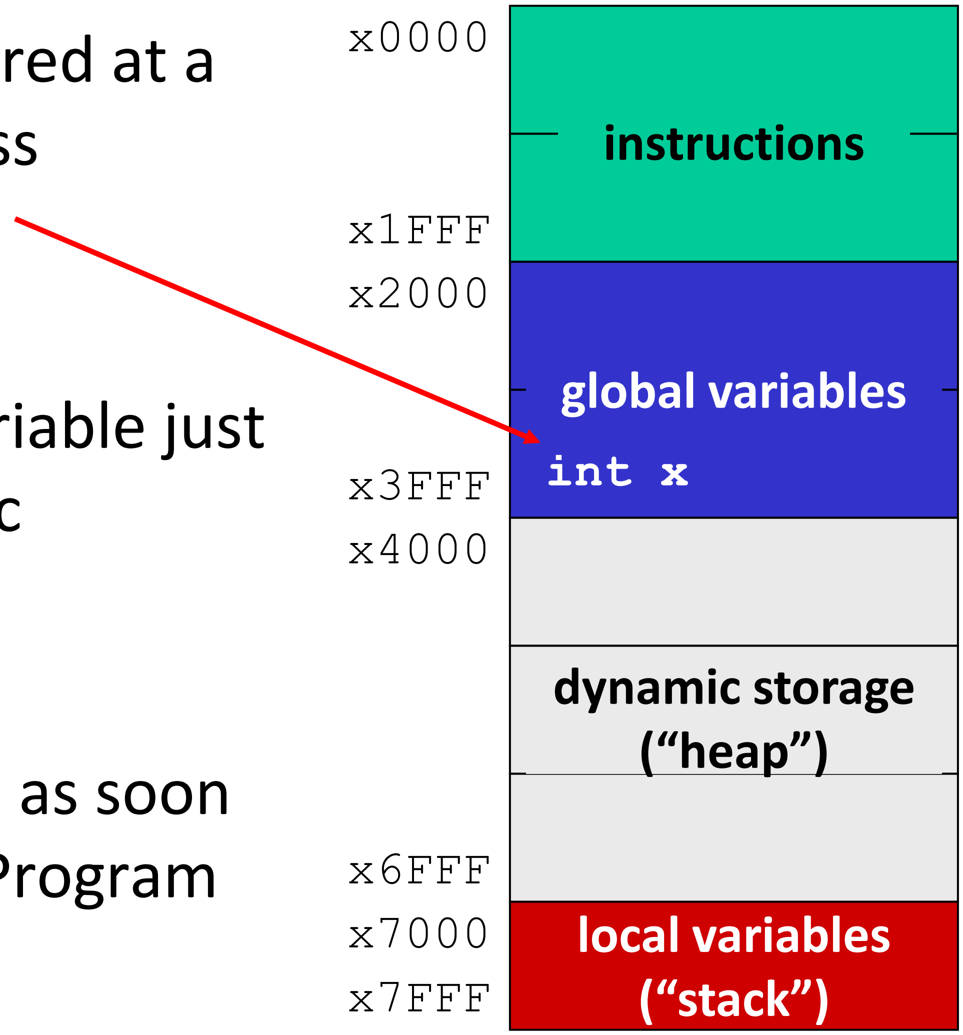
```c
#include <stdio.h>
#include <stdlib.h>

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 (similar to video memory)
- Reading/writing to that variable just involves going to that static memory location.
- The variable are "allocated as soon as the program is loaded. Program exiting will "de-allocate" t



Lecture Outline

- Global Memory
- **The Stack**
- The Heap
  - malloc() & free()
- Structs & C Data Structures

Variables in Functions

- Variables declared outside of functions (global variables) exist over the lifetime of the program

- What about variables in functions?
  - Function parameters, local variables, return values etc.
  - Exist only for the lifetime of an instance of execution of a function
  - There may be multiple instances of a function at a time, needing multiple (but separate) sets of variables (e.g. recursion)
  - **<u>Where do these exist in memory?</u>**

The Stack – <u>short version</u> for now

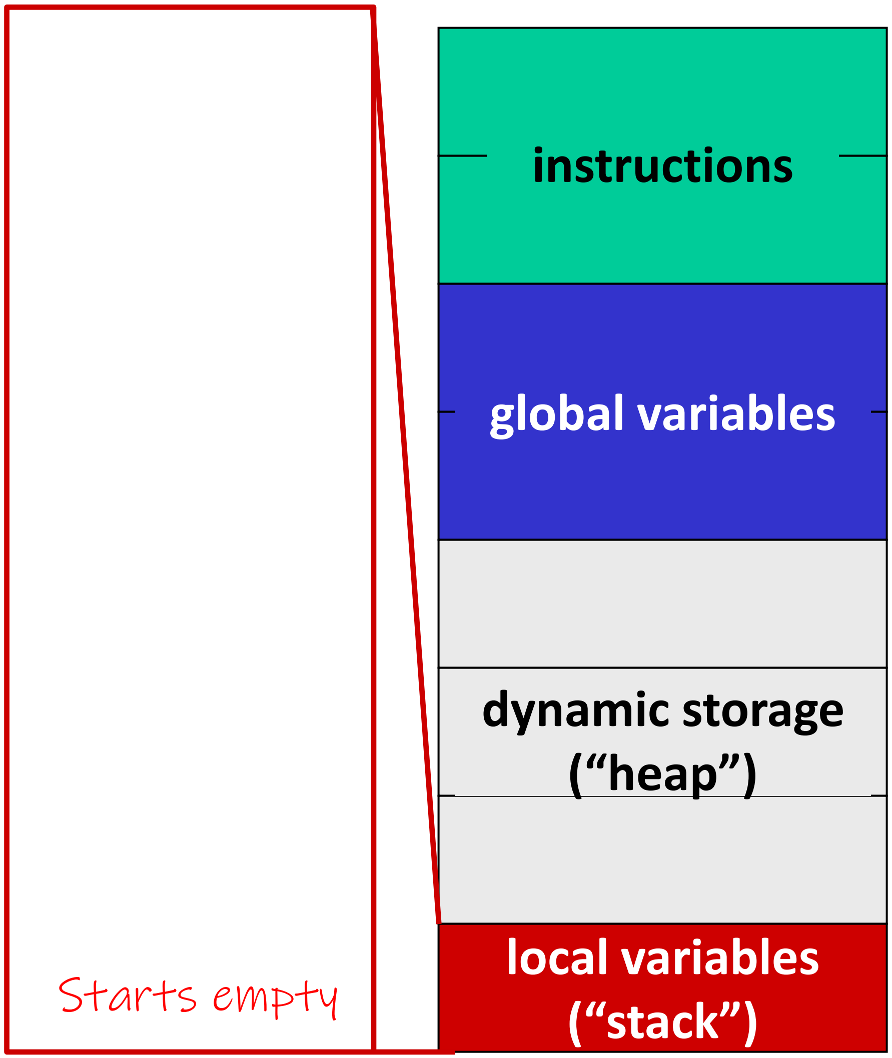
- Local variables are stored in a portion of memory called the “Stack” sometimes called the “Call Stack”.
  - Whenever a function is invoked, we “push” a “stack frame” for that function onto the top of the stack.
  - The stack frame contains important information about the execution of the function and has space for every local variable
  - When a function exits, its stack frame is “popped” and the local variables are “deallocated”

More details on how the stack works in a couple lectures from now

Stack Example 1:

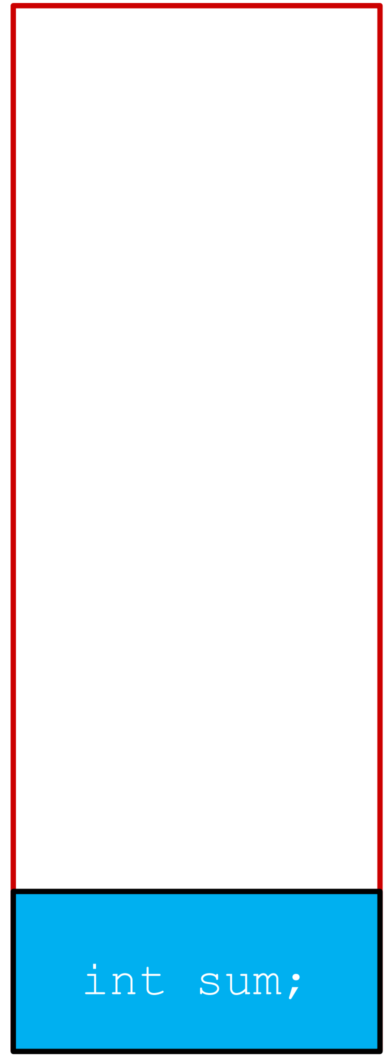
```
#include <stdio.h>
#include <stdlib.h>

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



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;
}
```

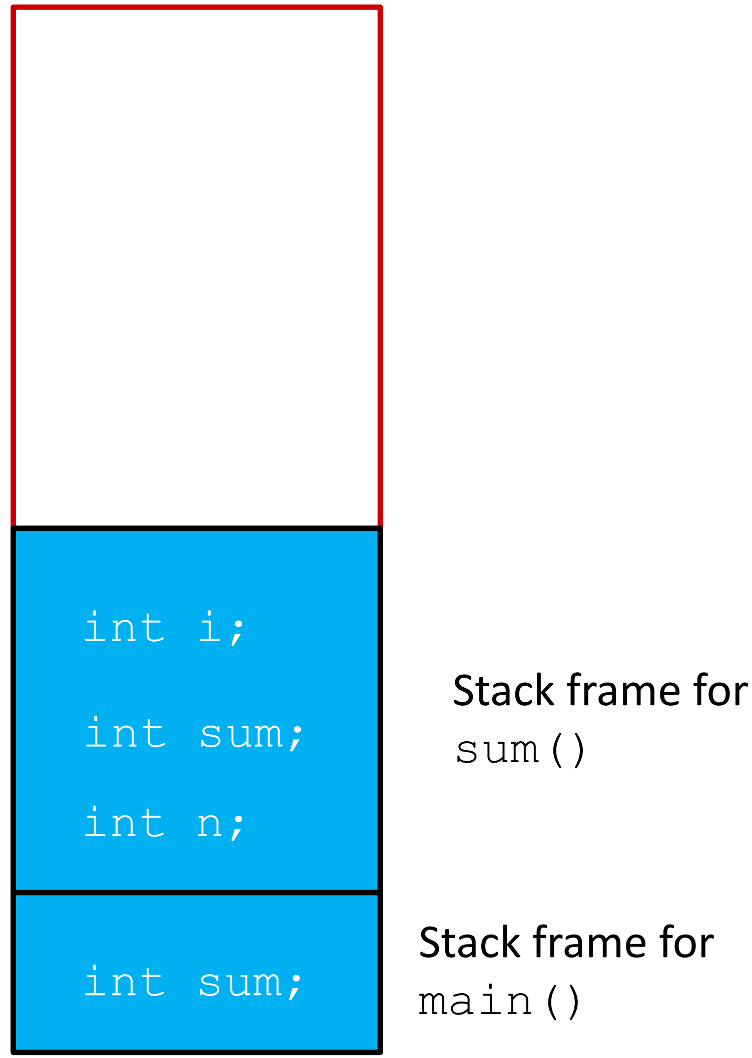


Stack frame for main is created when CPU starts executing it

Stack frame for `main()`

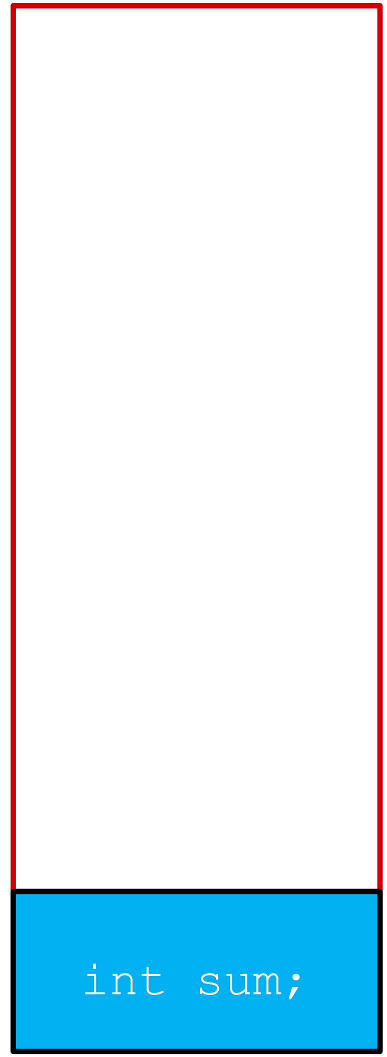
Stack Example 1:

```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 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;
}
```



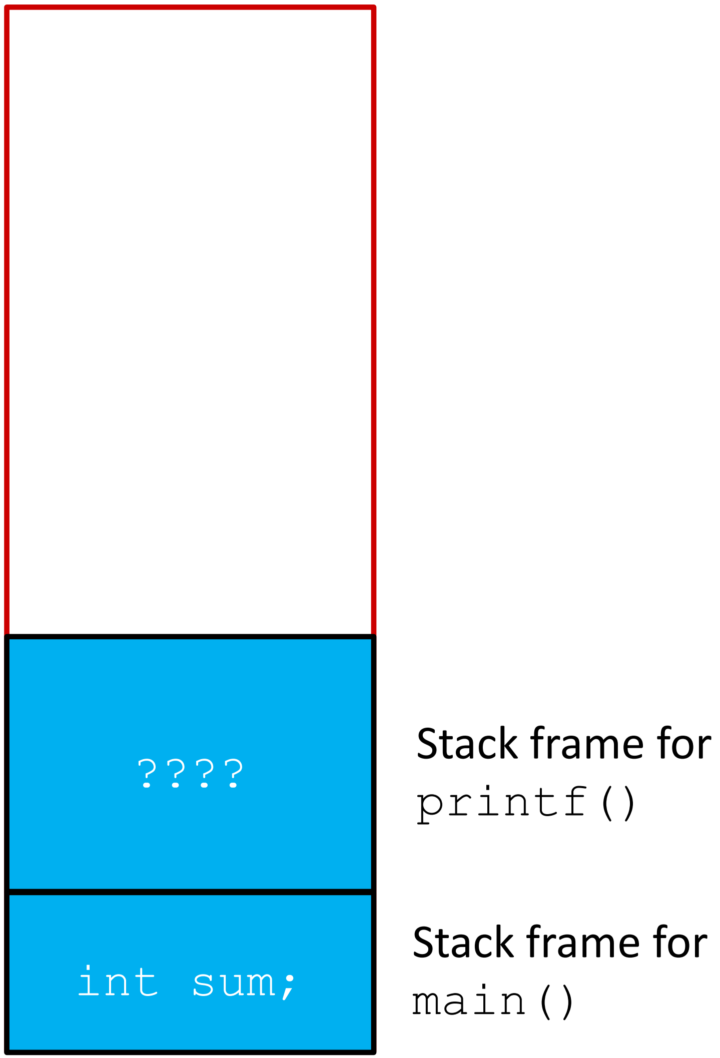
`sum()`'s stack frame goes away after **`sum()`** returns.

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

Stack frame for `main()`

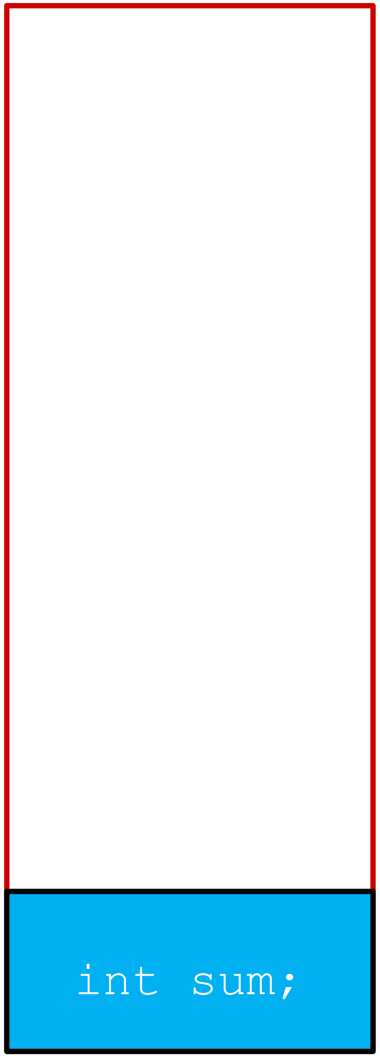
Stack Example 1:

```
#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 Example 2:

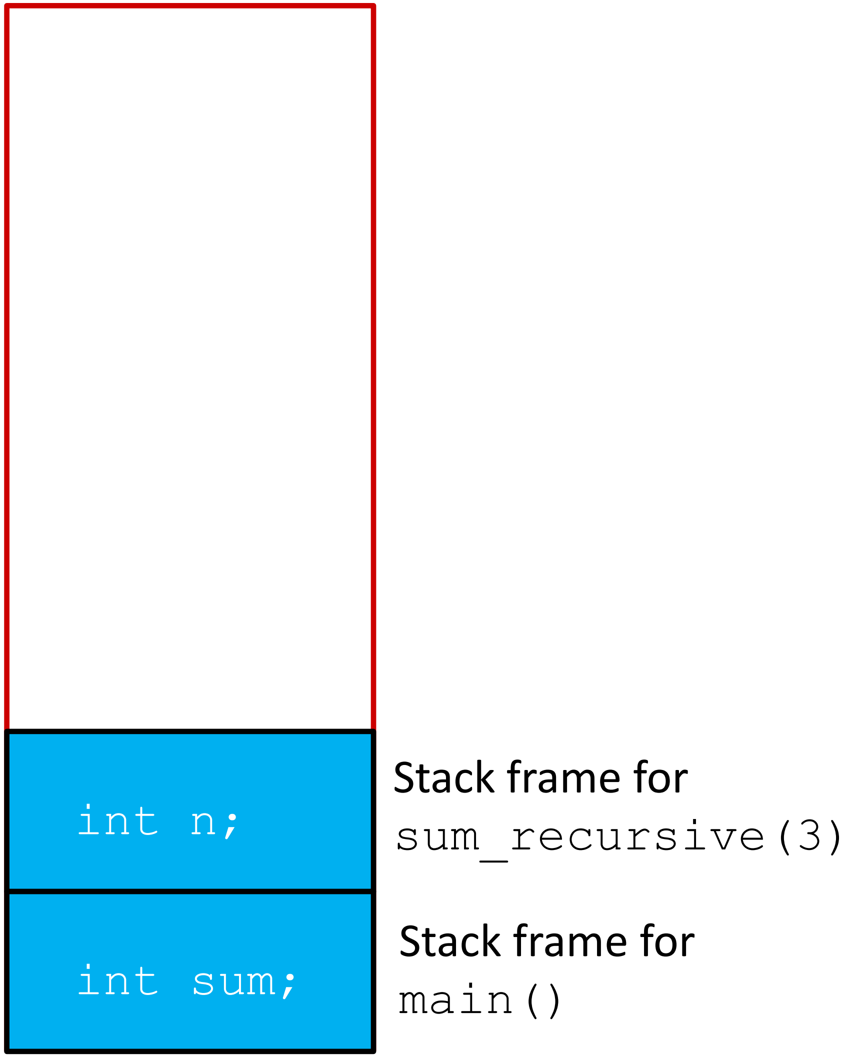
```c
#include <stdio.h>
#include <stdlib.h>

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()`

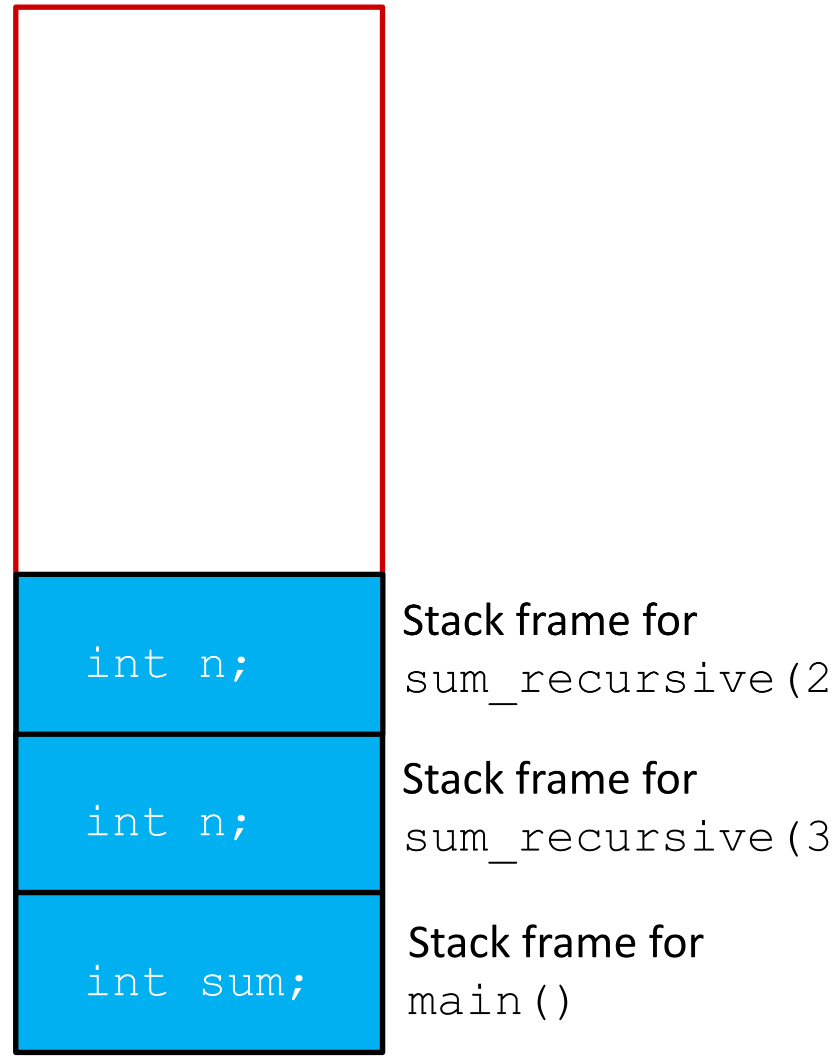
Stack Example 2:

```
#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;
}
```



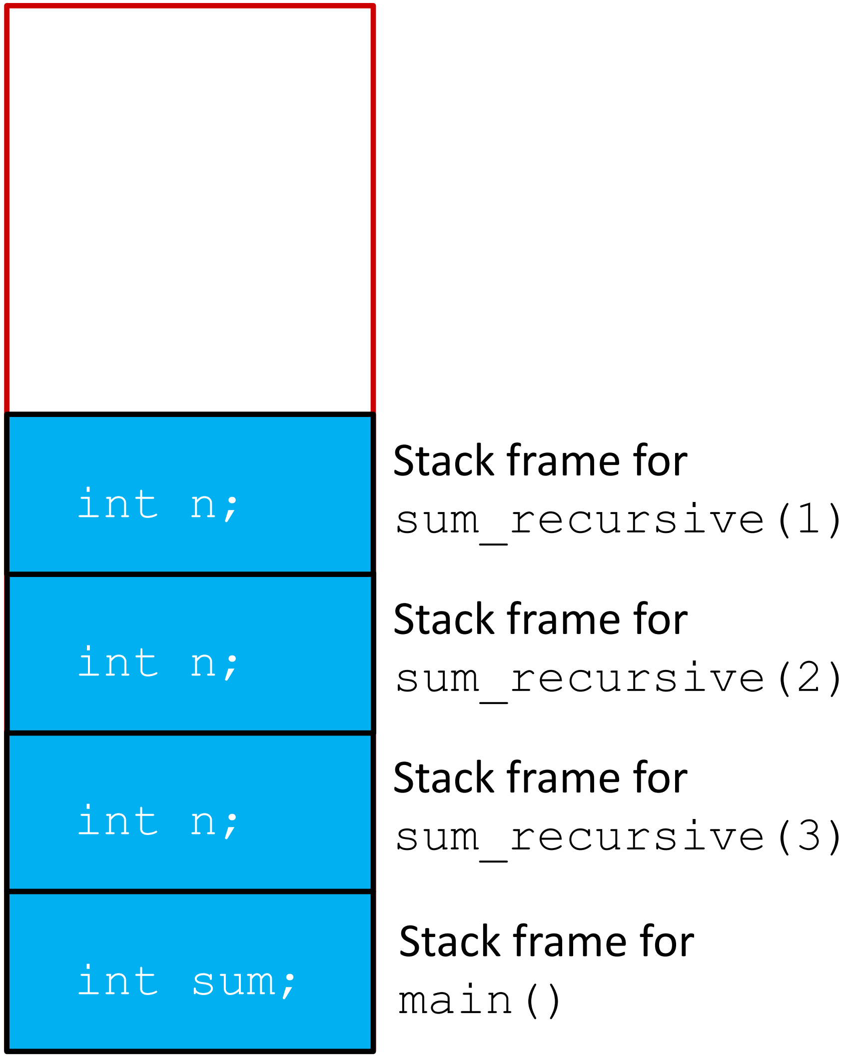
Stack Example 2:

```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;
}
```



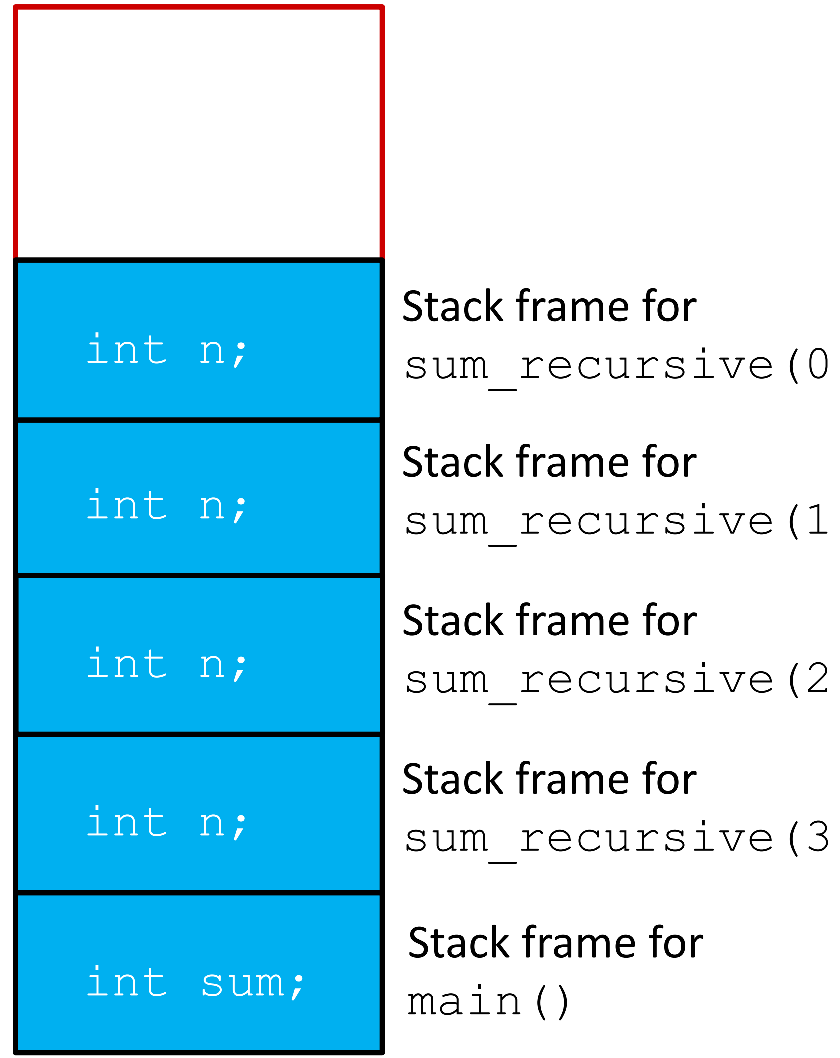
Stack Example 2:

```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;
}
```



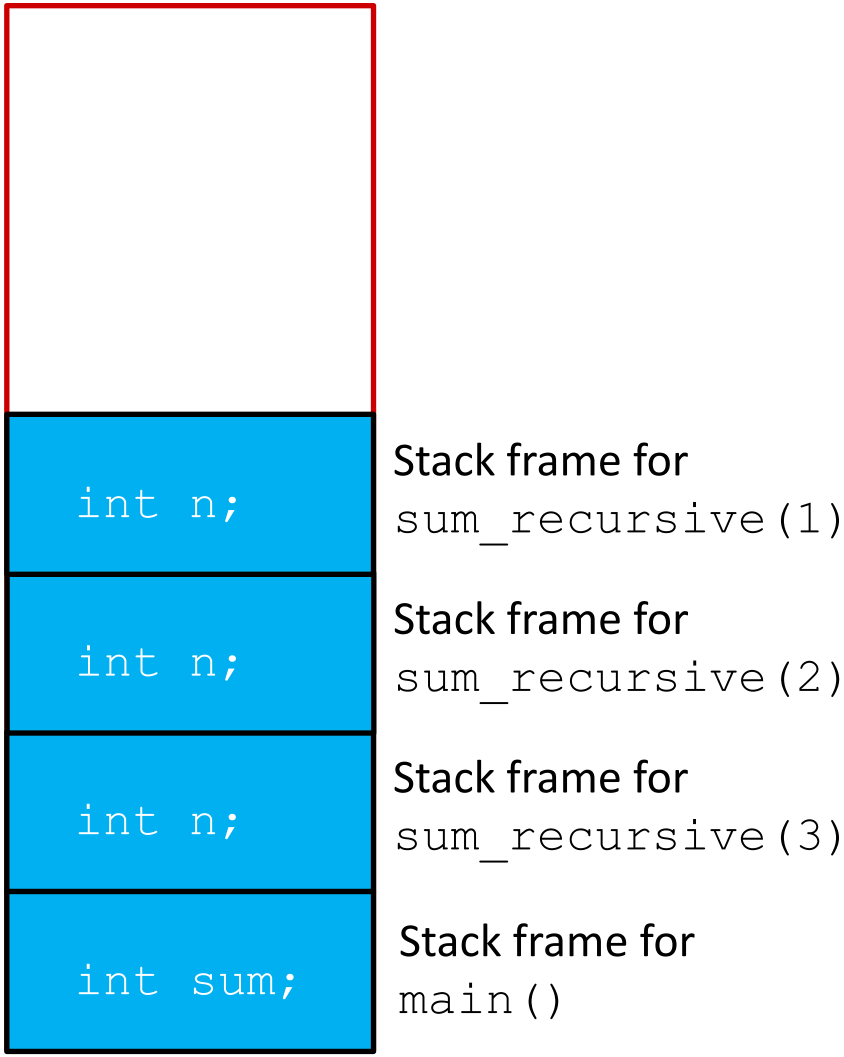
Stack Example 2:

```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;
}
```



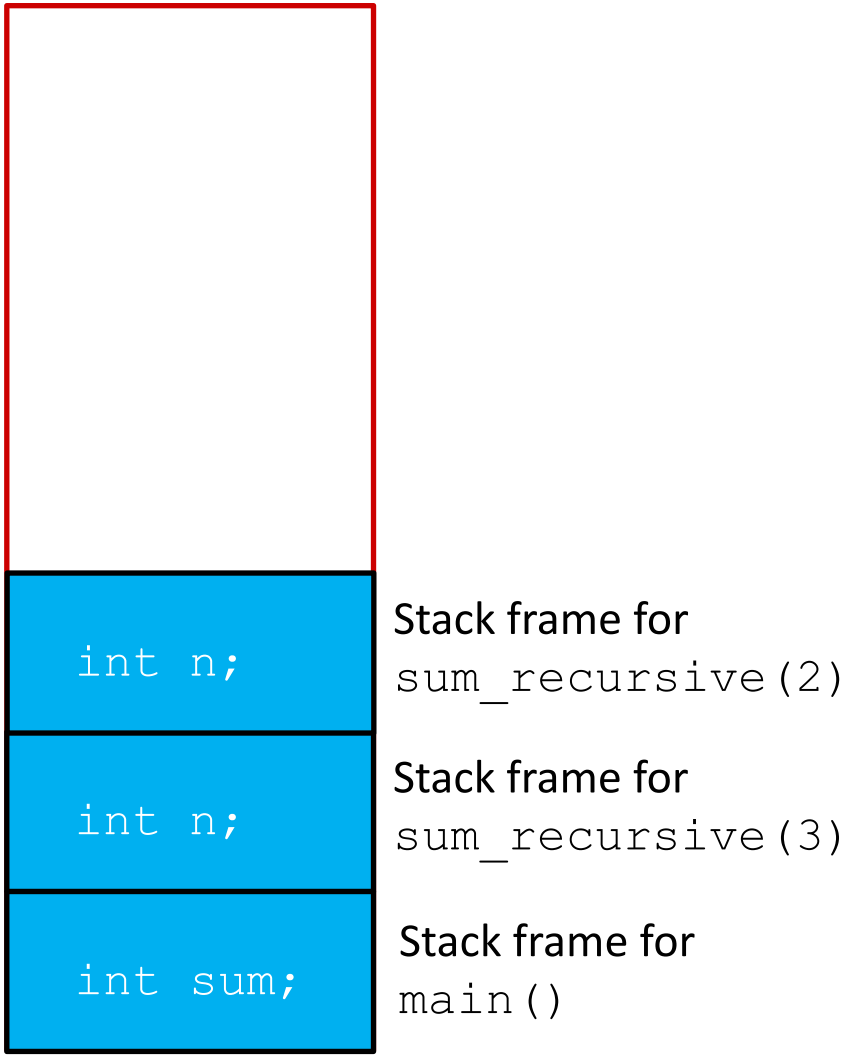
Stack Example 2:

```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;
}
```



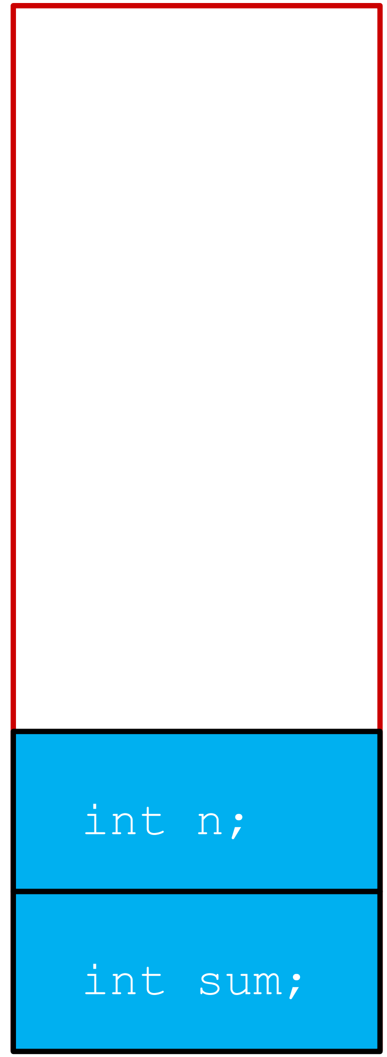
Stack Example 2:

```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 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;
}
```

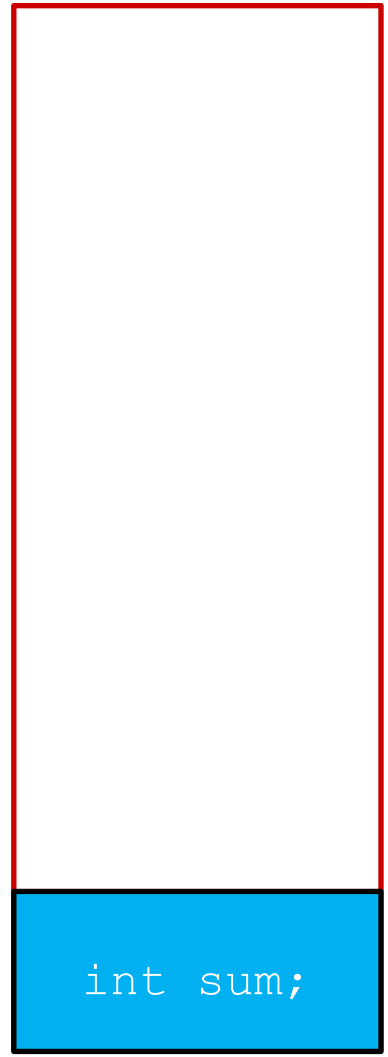


Stack frame for `sum_recursive(3)`

Stack frame for `main()`

Stack Example 2:

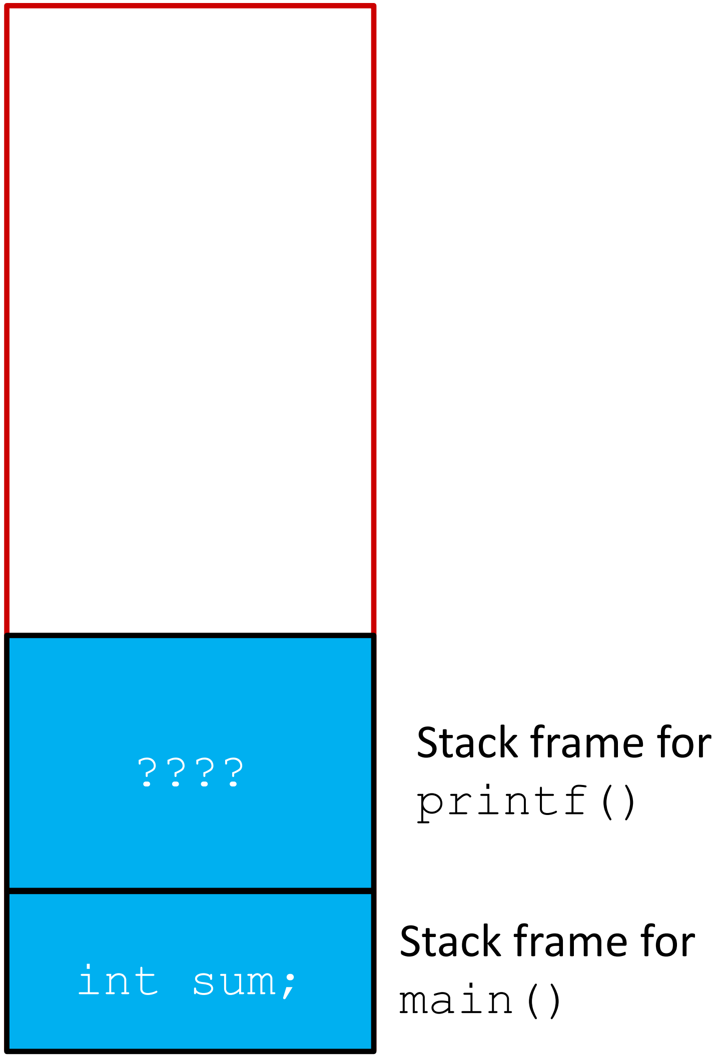
```
#include <stdio.h>
#include <stdlib.h>

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;
}
```



Memory Allocation So Far

- So far, we have seen two kinds of memory allocation:

```
int counter = 0;      // global var

int main() {
  counter++;
  printf("count = %d\n",counter);
  return 0;
}
```

- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when program exits

```
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main() {
  int y = foo(10);    // local var
  printf("y = %d\n",y);
  return 0;
}
```

- `a`, `x`, `y` are ***automatically***-allocated
  - Allocated when function is called
  - Deallocated when function returns

Poll Everywhere

pollev.com/tqm

- The following program compiles without errors. Does it work as seemingly intended though?

A. Yes

B. No

C. I'm not sure

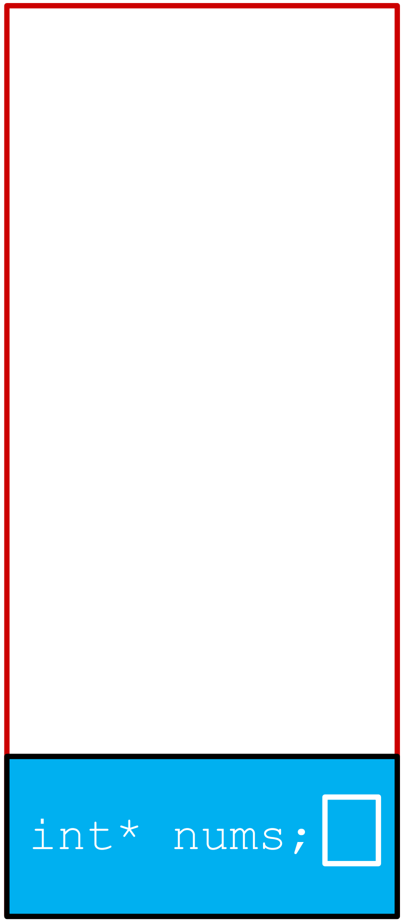
```
#include <stdio.h>
#include <stdlib.h>

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- The following program compiles without errors. Does it work as seemingly intended though?

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

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```



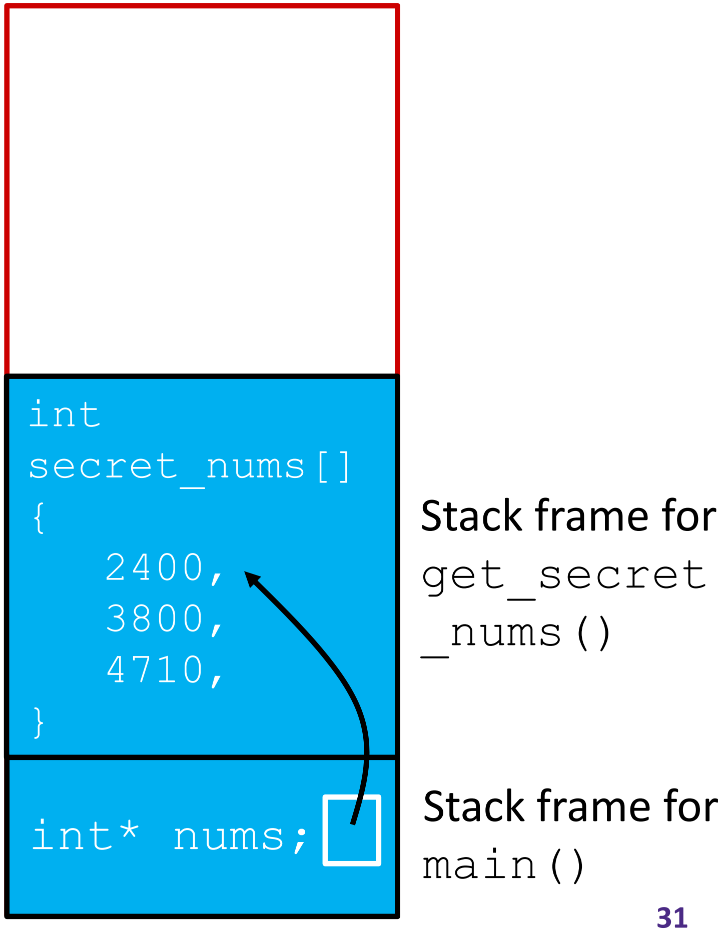
Stack frame for `main()`

Poll Everywhere

pollev.com/tqm

- The following program compiles without errors. Does it work as seemingly intended though?

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

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```

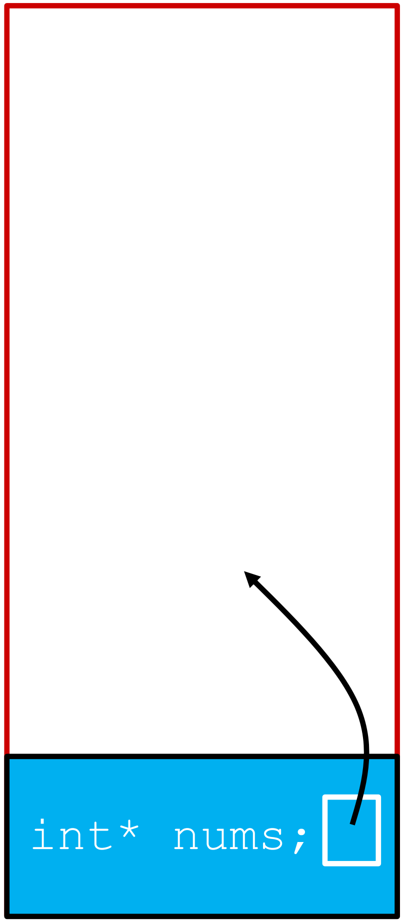


Poll Everywhere

pollev.com/tqm

- The following program compiles without errors. Does it work as seemingly intended though?

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

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```



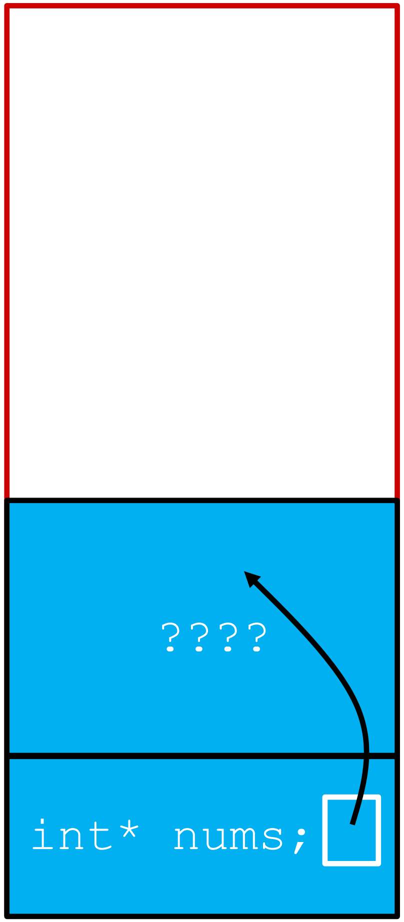
Stack frame for `main()`

Poll Everywhere

pollev.com/tqm

- The following program compiles without errors. Does it work as seemingly intended though?

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

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```



Stack frame for `printf()`

Stack frame for `main()`

B. No

Lecture Outline

- Global Memory
- The Stack
- **The Heap**
  - **malloc() & free()**
- Structs & C Data Structures

Aside: NULL

- `NULL` is a memory location that is guaranteed to be invalid
  - In C on Linux, `NULL` is `0x0` and an attempt to dereference `NULL` *causes a segmentation fault*
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It's better to cause a segfault than to allow the corruption of memory!

```
int main(int argc, char** argv) {
  int* p = NULL;
  *p = 1;  // causes a segmentation fault
  return EXIT_SUCCESS;
}
```

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.

Dynamic Memory Example

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  if (ptr == NULL)
    return EXIT_FAILURE;
  ...            // do stuff with ptr
  free(ptr);
}
```

addr	var	value
0x2001	ptr	--
...	...	--
0x4000	HEAP START	USED
0x4001		USED
0x4002		
0x4003		
0x4004		
0x4005		
0x4006		
0x4007		
0x4008		USED
0x4009		USED

Dynamic Memory Example

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  if (ptr == NULL)
    return EXIT_FAILURE;
  ...               // do stuff with ptr
  free(ptr);
}
```

addr	var	value
0x2001	**ptr**	**0x4002**
...	...	--
0x4000	**HEAP START**	USED
0x4001		USED
0x4002		**USED**
0x4003		**USED**
0x4004		**USED**
0x4005		**USED**
0x4006		
0x4007		
0x4008		USED
0x4009		USED

Dynamic Memory Example

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  if (ptr == NULL)
    return EXIT_FAILURE;
  ...           // do stuff with ptr
  free(ptr);
}
```

addr	var	value
0x2001	ptr	0x4002
...	...	--
0x4000	HEAP START	USED
0x4001		USED
0x4002		
0x4003		
0x4004		
0x4005		
0x4006		
0x4007		
0x4008		USED
0x4009		USED

Revisiting get_secret_nums Poll

- If we want to return a pointer to newly created data, that data should exist on the heap
  - Create the array with malloc
  - Check for NULL

```
int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}
```

```
int* get_secret_nums() {
  int* secret_nums = malloc(3 * sizeof(int));
  if (secret_nums == NULL) {
    return NULL;
  }
  secret_nums[0] = 2400;
  secret_nums[1] = 3800;
  secret_nums[2] = 4710;
  return secret_nums;
}
```

Revisiting get_secret_nums Poll

- Main
  - Check for NULL
  - free dynamically allocated data once done with it

```
int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```

```
int main() {
  int* nums = get_secret_nums();
  if (nums == NULL) {
    return EXIT_FAILURE;
  }
  printf("%d\n", nums[0]);
  free(nums);
  return EXIT_SUCCESS;
}
```

Fixed Poll Code

- Put all together

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

int* get_secret_nums() {
  int* secret_nums = malloc(3 * sizeof(int));
  if (secret_nums == NULL)
    return NULL;
  secret_nums[0] = 2400;
  secret_nums[1] = 3800;
  secret_nums[2] = 4710;
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  if (nums == NULL)
    return EXIT_FAILURE;
  printf("%d\n", nums[0]);
  free(nums);
  return EXIT_SUCCESS;
}
```

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 probably 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

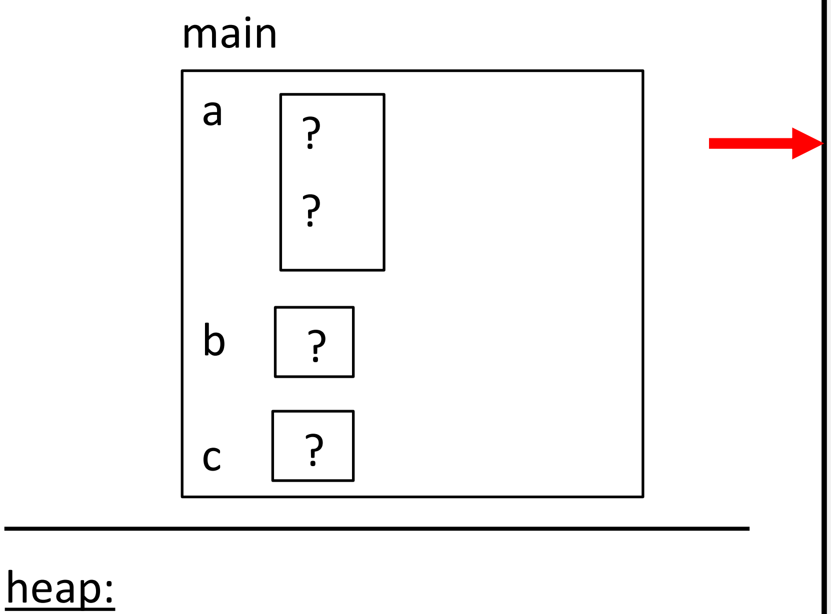
Poll Everywhere

pollev.com/tqm

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

```c
#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?

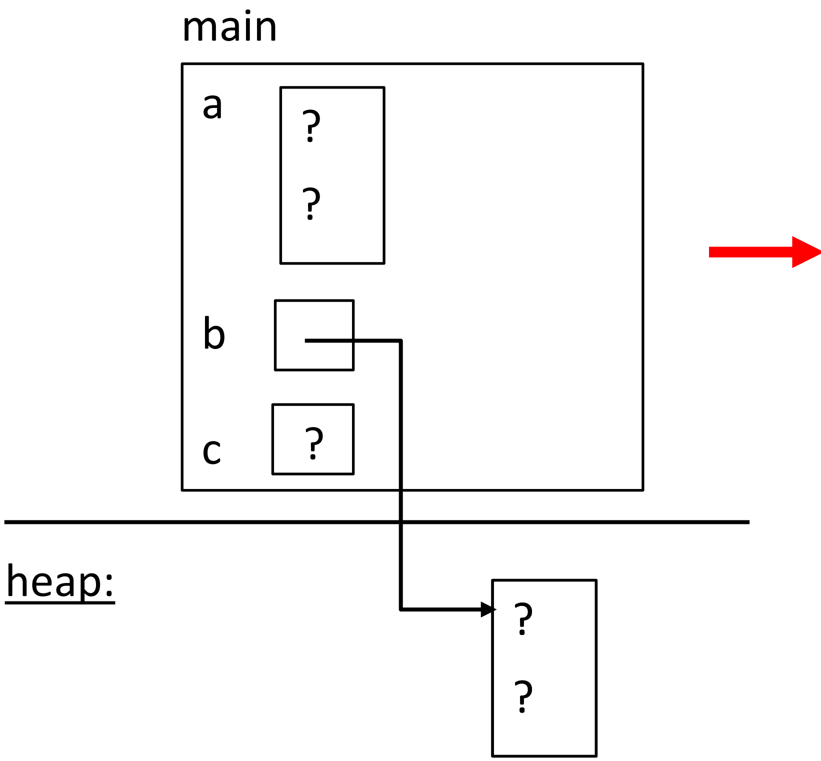



```c
#include <stdio.h>
#include <stdlib.h>

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?

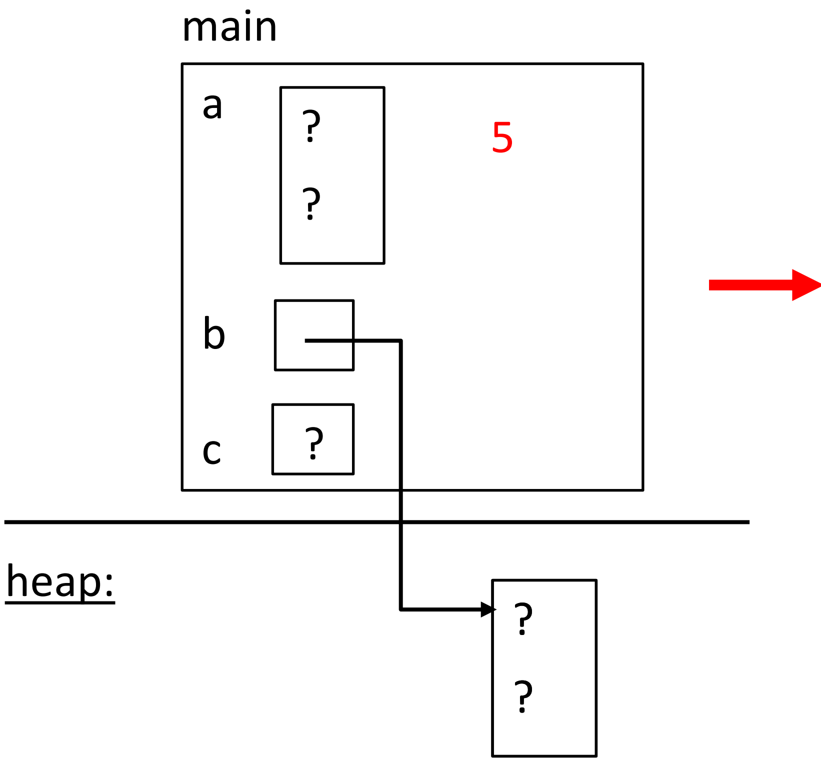



```c
#include <stdio.h>
#include <stdlib.h>

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?

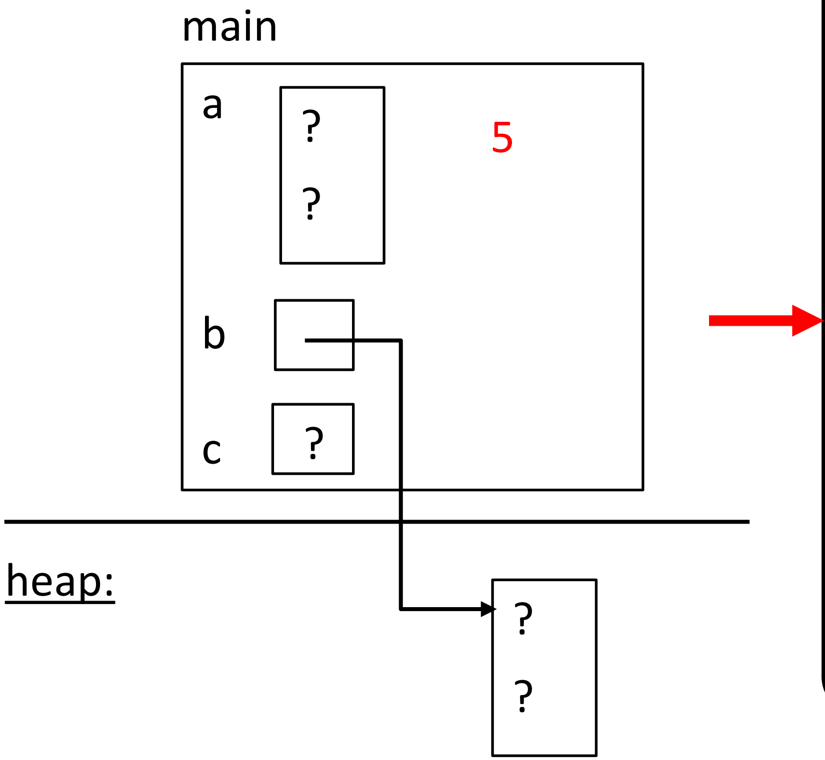



```c
#include <stdio.h>
#include <stdlib.h>

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?

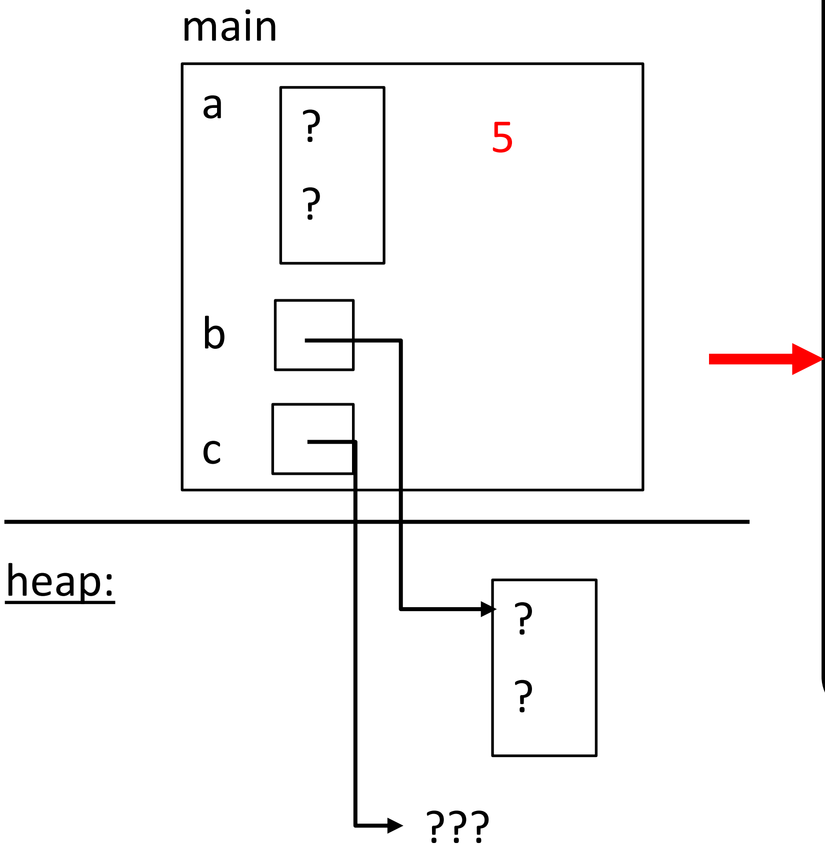



```c
#include <stdio.h>
#include <stdlib.h>

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?

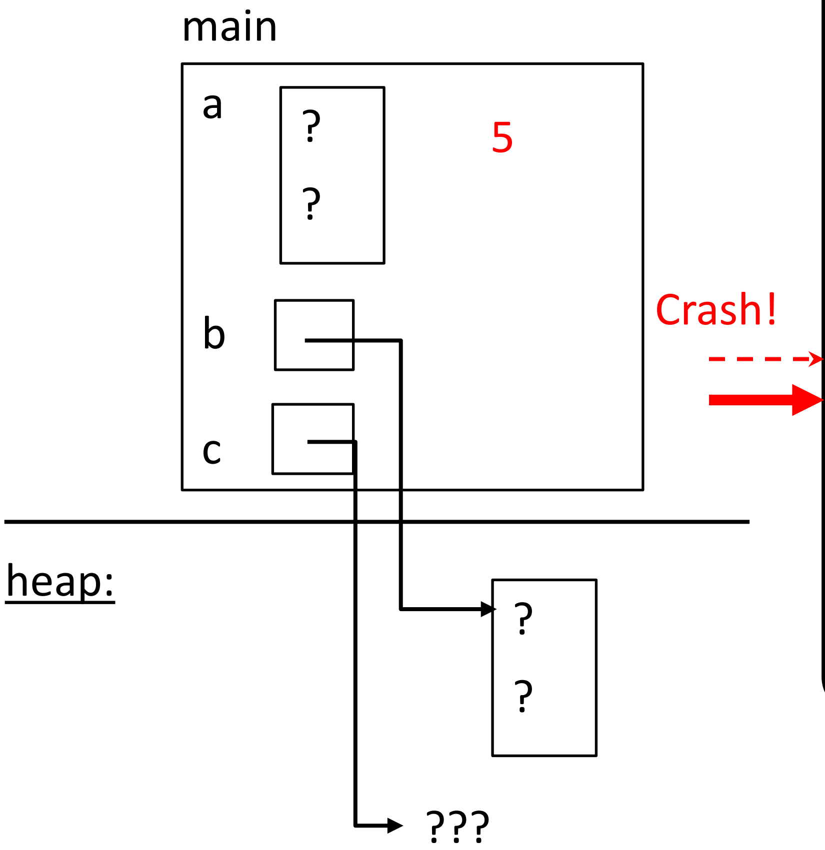



```c
#include <stdio.h>
#include <stdlib.h>

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.

memcrupt.c

Memory Corruption - What Happens?

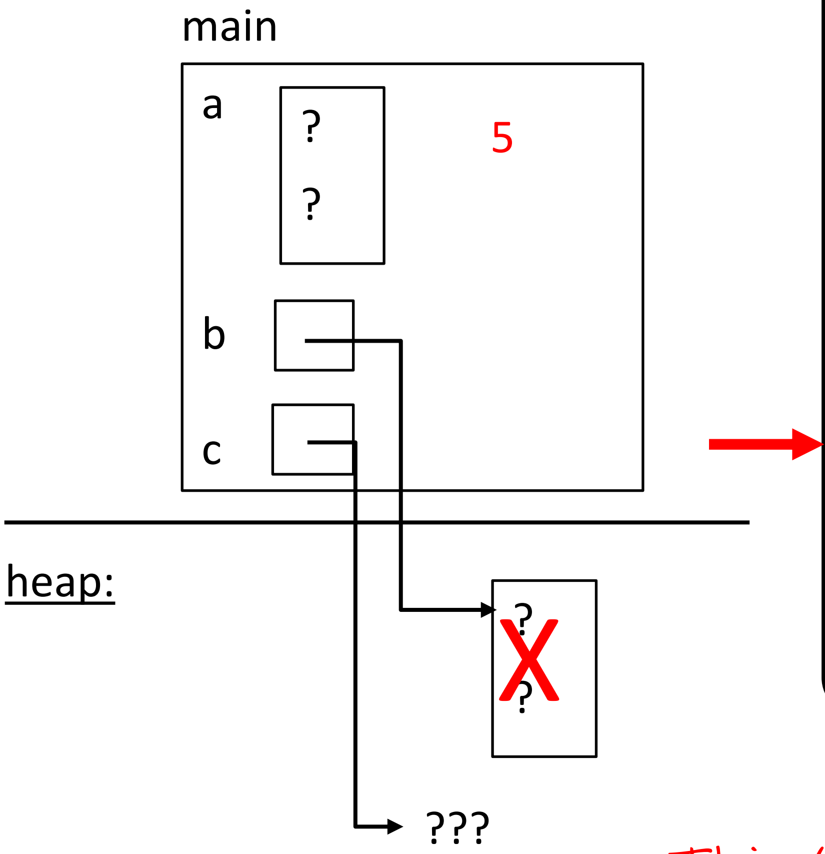



```c
#include <stdio.h>
#include <stdlib.h>

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?



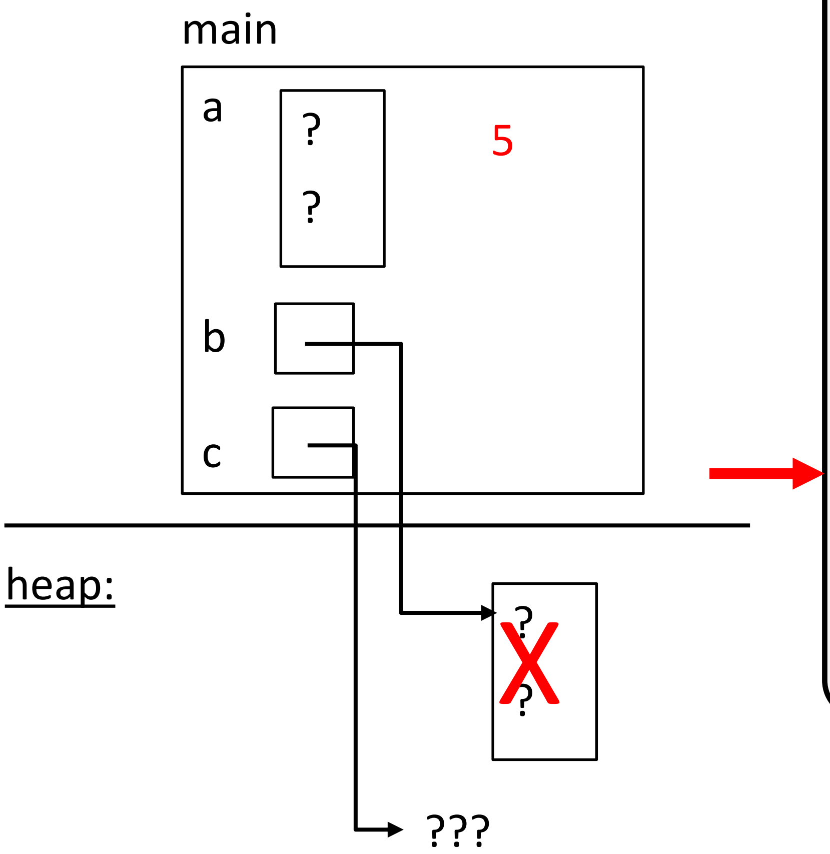

```c
#include <stdio.h>
#include <stdlib.h>

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.

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

memcorrupt.c

Memory Corruption - What Happens?

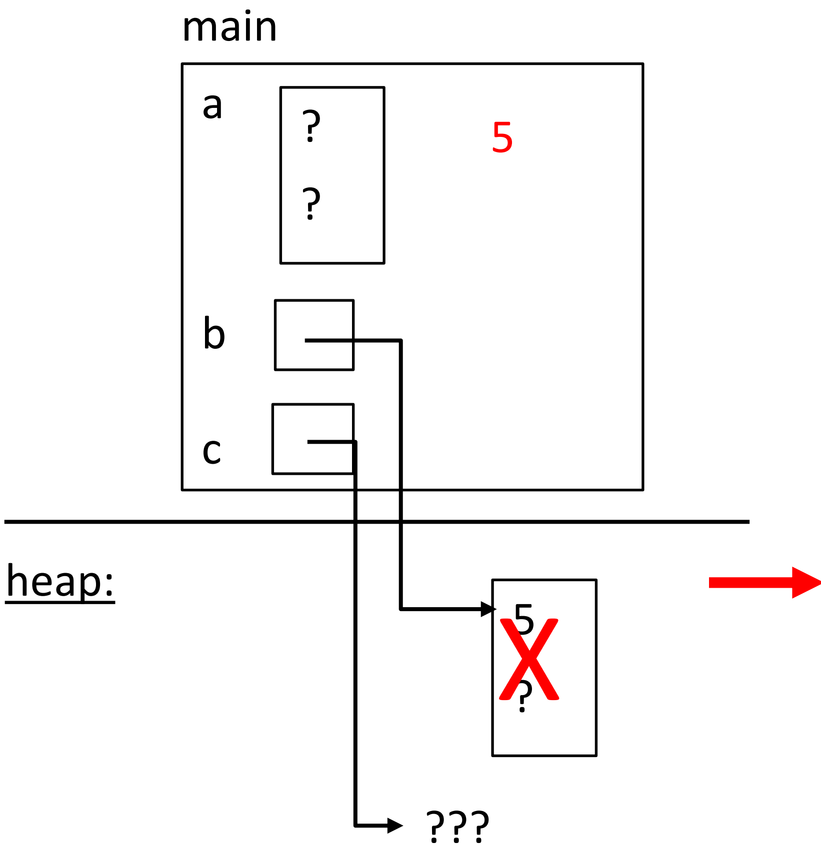



```c
#include <stdio.h>
#include <stdlib.h>

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

Aside: Casting

- In older implementations of the C language, malloc returned a (char*) instead of a (void*) and you would have to employ casting to convert the returned value to the appropriate type
  - double *ptr = (double*) malloc (sizeof(double) * 10);

- Casting also used for casting between non-pointer types.
  - Needed when casting from larger data representation to smaller ones.
    - E.g. casting to convert from double -> float or long -> short

Lecture Outline

- Global Memory
- The Stack
- The Heap
  - Motivation
  - malloc() & free()
- **Structs & C Data Structures**

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:

```
typedef struct point_st {
  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.

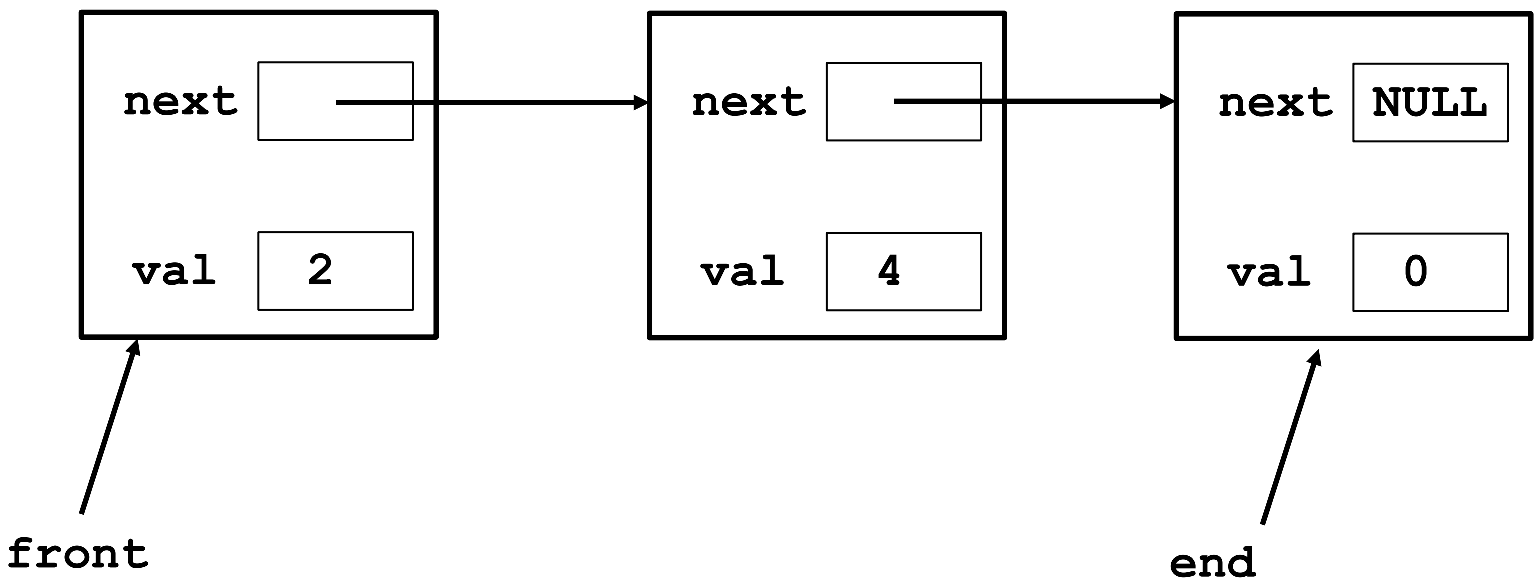
Accessing struct Fields

- 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 point_st {
  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;
}
```

Queue Example

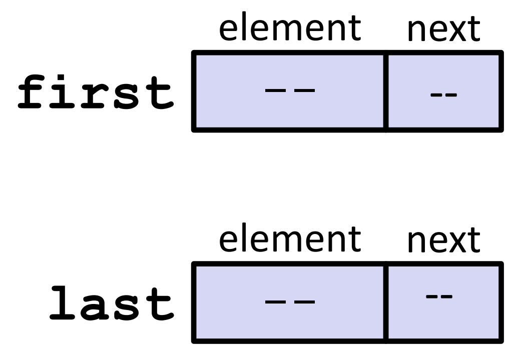
- Simple Data structure modeling a queue
  - Implemented with a singly linked list
- Items added to the end and removed from the front.
- We maintain a list of queue elements chained together with pointers.



Queue Implementation Demo

- Let’s create a naïve implementation for our queue

```c
#include <stdio.h>

typedef struct node_st {
  struct node_st* next;
  int val;
} Node;

int main(int argc, char** argv) {
  Node first, last;

  first.val = 2;
  first.next = &last;
  last.val = 0;
  last.next = NULL;
  return 0;
}
```

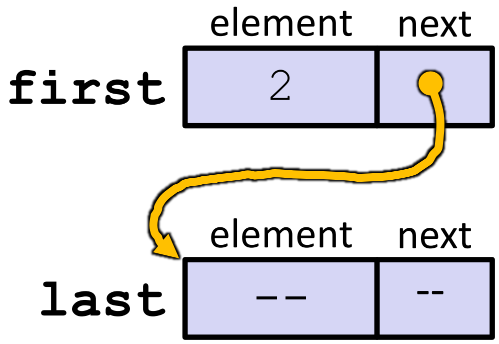


naive_queue.c

Queue Implementation Demo

- Let's create a naïve implementation for our queue

```
#include <stdio.h>

typedef struct node_st {
  struct node_st* next;
  int val;
} Node;

int main(int argc, char** argv) {
  Node first, last;

  first.val = 2;
  first.next = &last;
  last.val = 0;
  last.next = NULL;
  return 0;
}
```



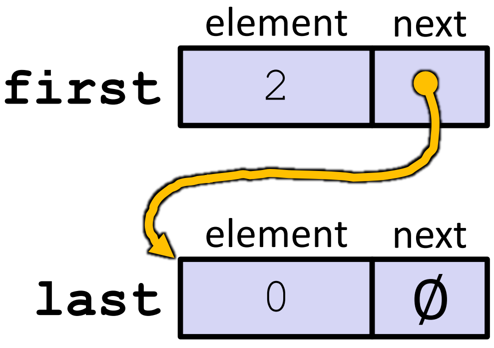
naive_queue.c

Queue Implementation Demo

- Let's create a naïve implementation for our queue

```
#include <stdio.h>

typedef struct node_st {
  struct node_st* next;
  int val;
} Node;

int main(int argc, char** argv) {
  Node first, last;

  first.val = 2;
  first.next = &last;
  last.val = 0;
  last.next = NULL;
  return 0;
}
```

naive_queue.c

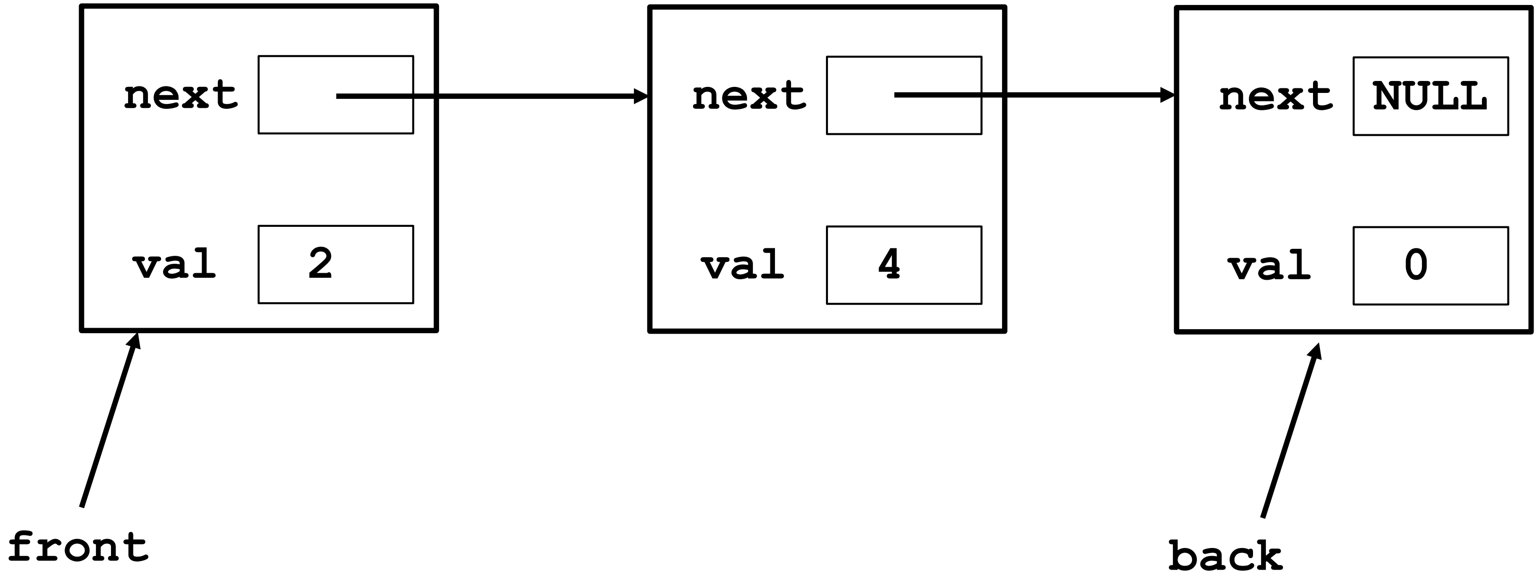


What happens if we want more than two elements?

What happens if we don't know the size we need until run-time?

Revisiting the Queue Example

- Simple Data structure modeling a queue
  - Implemented with a singly linked list
- Items added to the end and removed from the front.
- We maintain a list of queue elements chained together with pointers.
- We can use Dynamic Allocation to create new elements

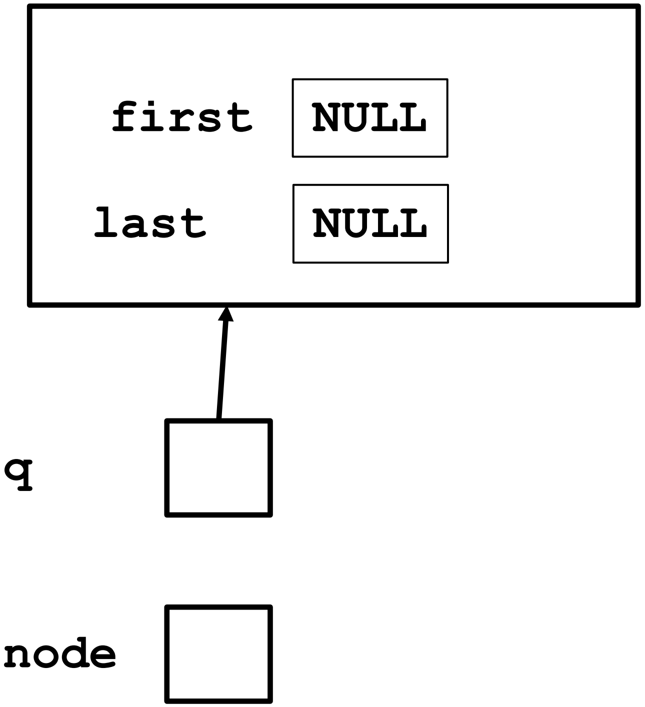


Dynamically Allocated Queue Demo

- See code on course website:
  - main_queue.c
  - queue.h
  - queue.c
  - Makefile

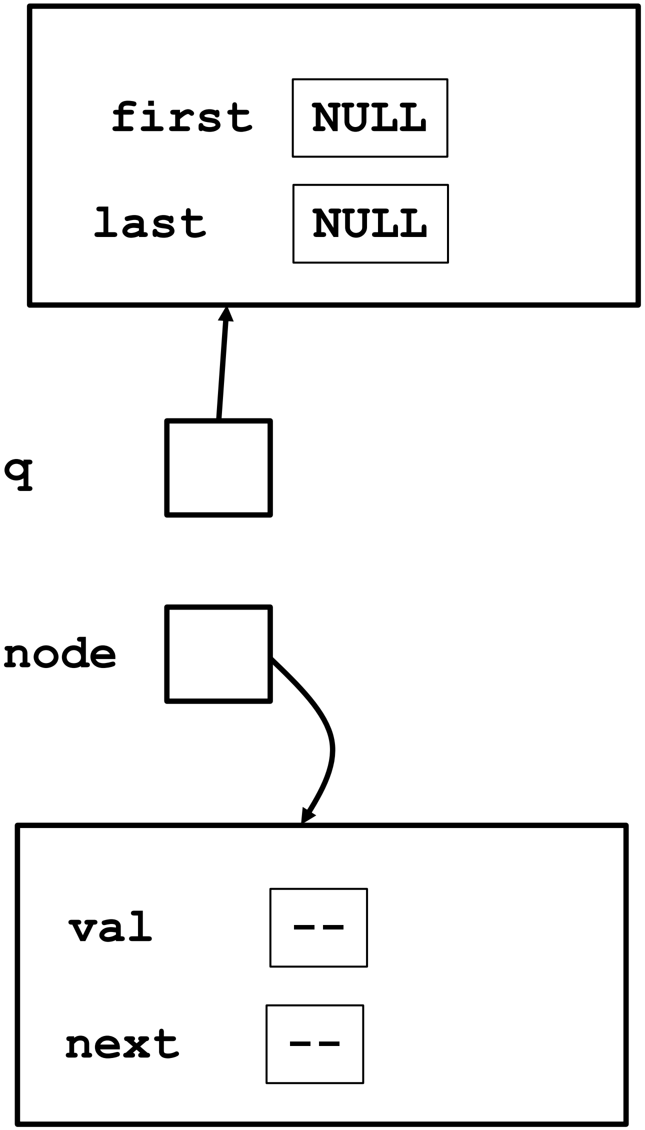
Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```



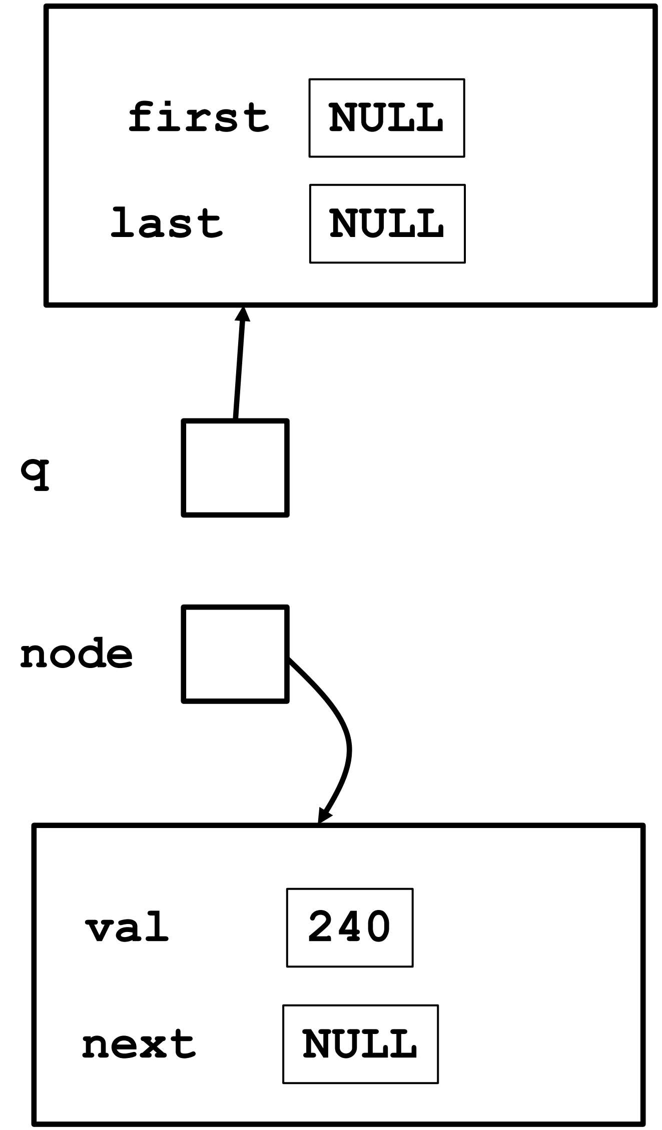
Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```



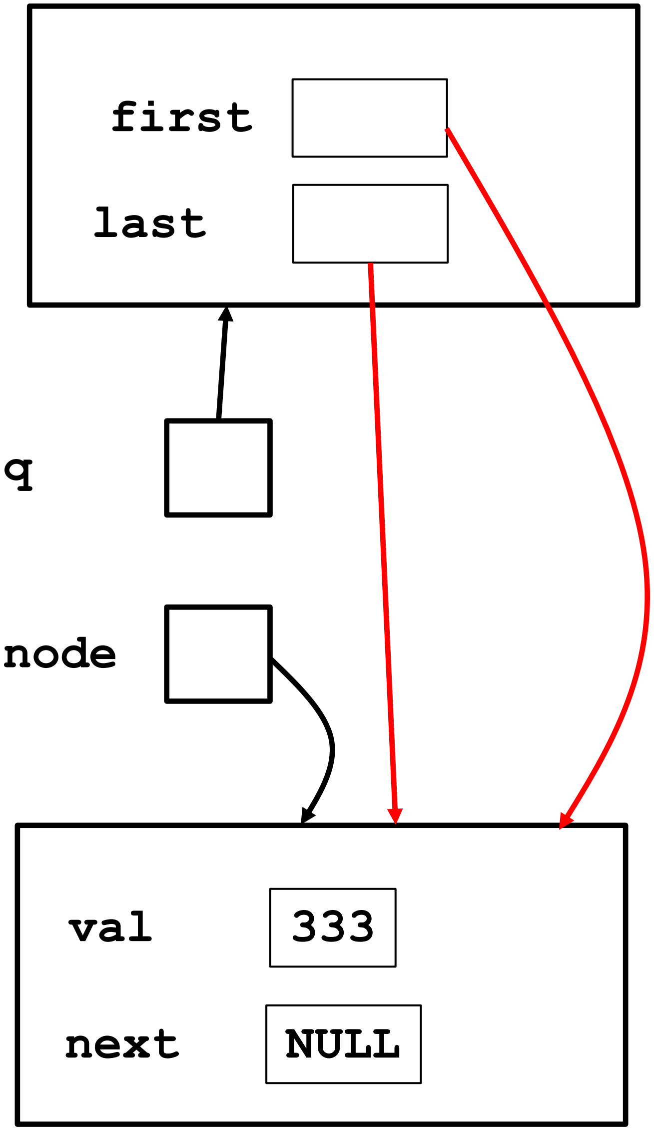
Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```



Queue_Add

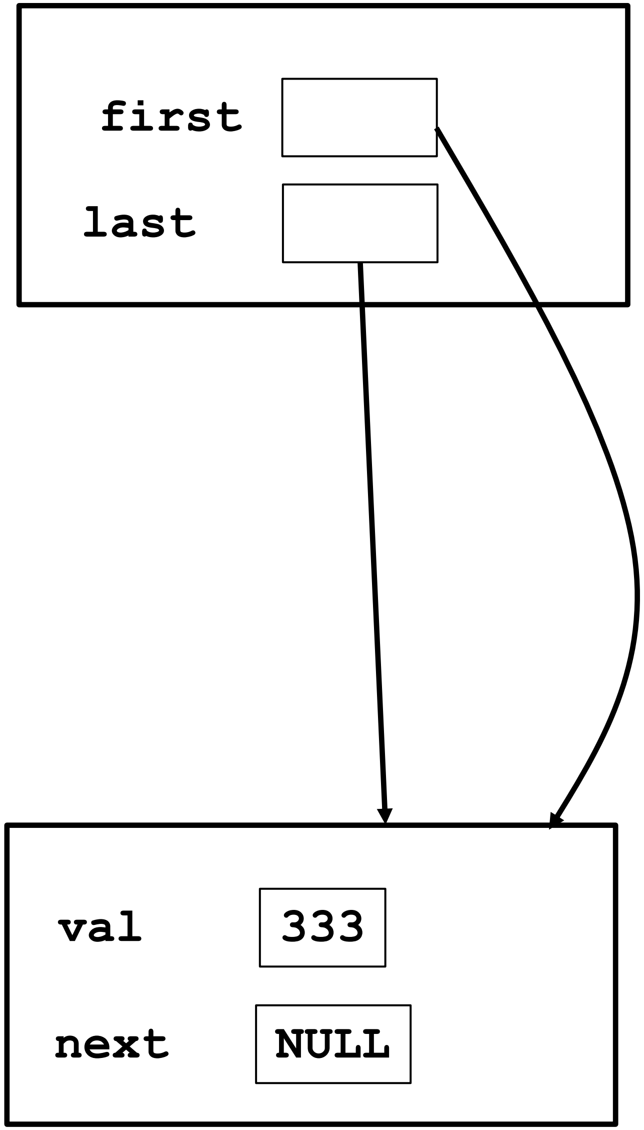
```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```



Queue_Add

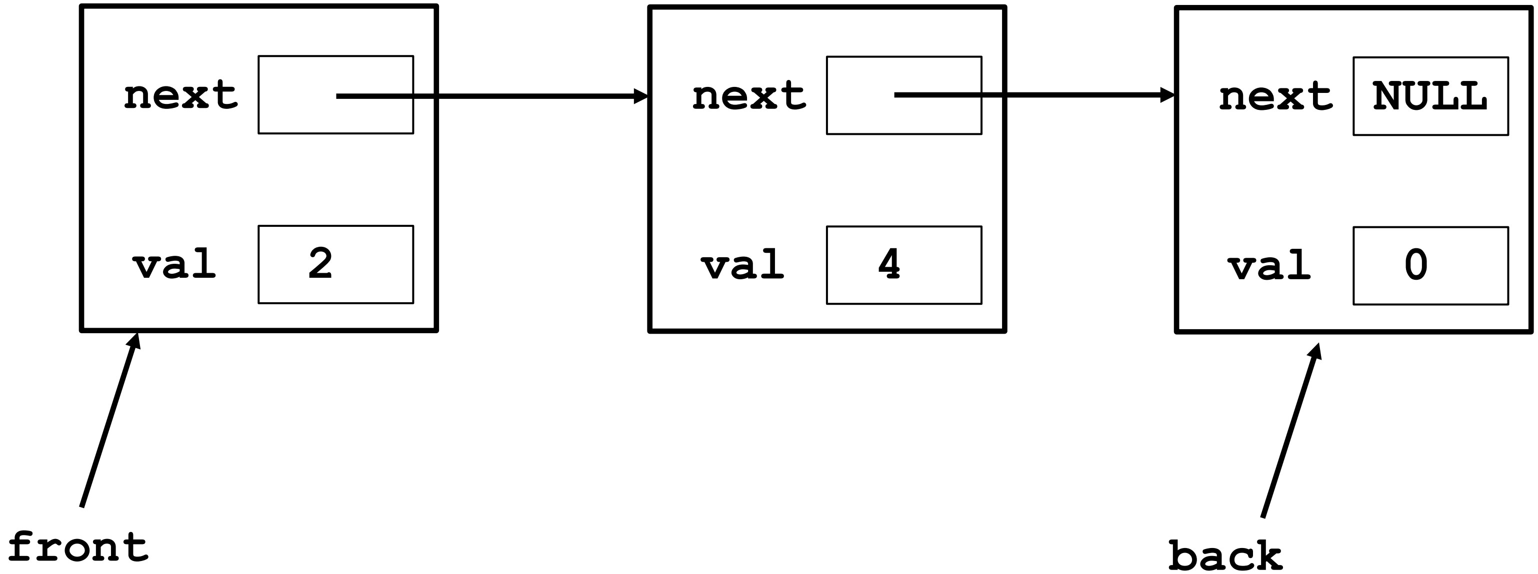
```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```

Since node is dynamically allocated, it persists after the function returns



Revisiting the Queue Example

- Simple Data structure modeling a queue
  - Implemented with a singly linked list
- Items added to the end and removed from the front.
- We maintain a list of queue elements chained together with pointers.
- **We can use Dynamic Allocation to create new elements**



Next Time

- Deeper explanation on:
  - Organizing a C program across multiple files (.h and .c)
  - How Makefiles work
  - The C Pre-processor (#include, #define, etc)
  - Valgrind
- The C Pre-processor
- Command line arguments
- File I/O in C