Week 7 - Friday

COMP 2400

Last time

- What did we talk about last time?
- Practice using **`malloc()`**
- Allocating multi-dimensional arrays

Questions?

Project 4

Quotes

In theory, theory and practice are the same. In practice, they're not.

Yoggi Berra

Allocating 2D Arrays

Allocating 2D arrays

- We know how to dynamically allocate a regular array
- How would you dynamically allocate a 2D array?
- In C, you can't do it in one step
  - You have to allocate an array of pointers
  - Then you make each one of them point at an appropriate place in memory

Ragged Approach

- One way to dynamically allocate a 2D array is to allocate each row individually

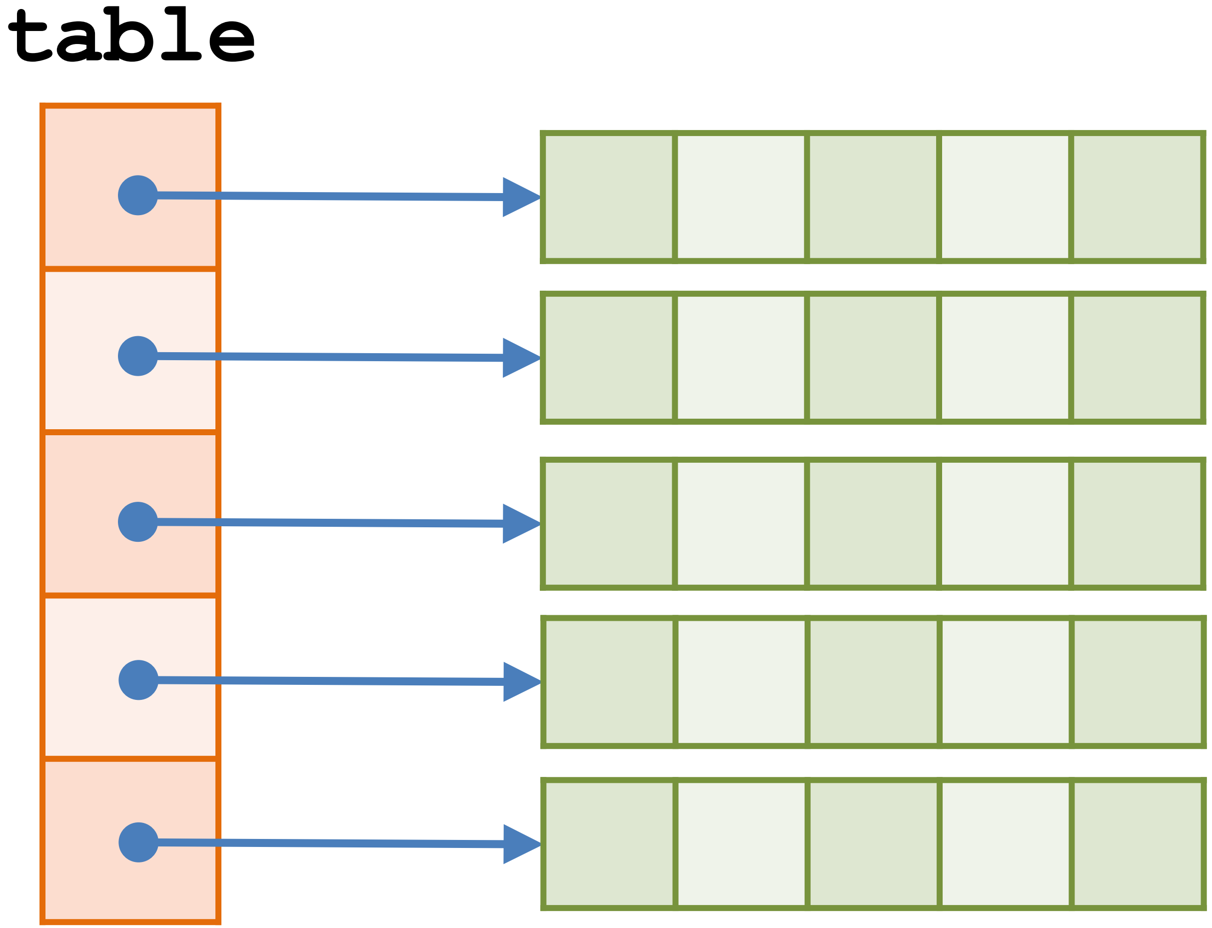
```
int** table = (int**)malloc (sizeof(int*)*rows);

for (int i = 0; i < rows; ++i)
  table[i] = (int*)malloc (sizeof(int)*columns);
```

- When finished, you can access **table** like any 2D array

```
table[3][7] = 14;
```

Ragged Approach in memory



Chunks of data that could be anywhere in memory

Freeing the Ragged Approach

- To free a 2D array allocated with the Ragged Approach
  - Free each row separately
  - Finally, free the array of rows

```
for (int i = 0; i < rows; ++i)
  free (table[i]);

free (table);
```

Contiguous Approach

- Alternatively, you can allocate the memory for all rows at once
- Then you make each row point to the right place

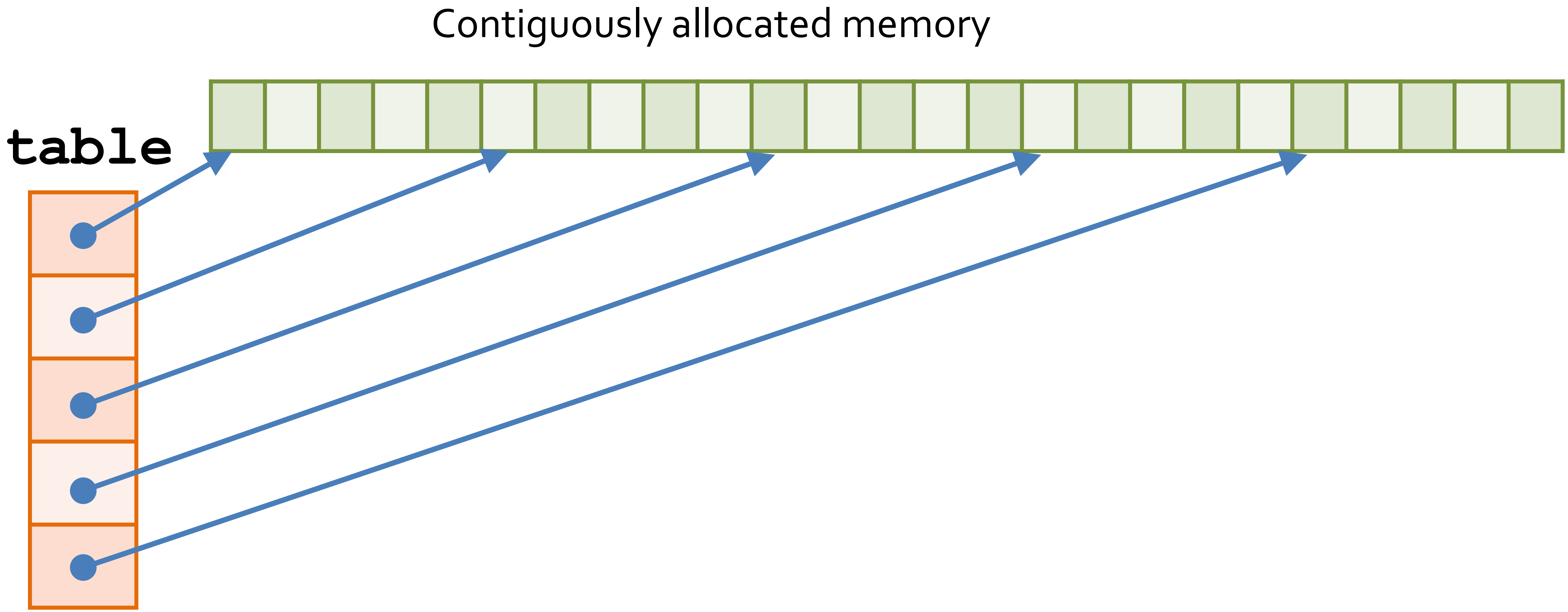
```
int** table = (int**)malloc (sizeof(int*)*rows);
int* data = (int*)malloc (sizeof(int)*rows*columns);

for(int i = 0; i < rows; ++i)
  table[i] = &data[i*columns];
```

- When finished, you can still access **`table`** like any 2D array

```
table[3][7] = 14;
```

Contiguous Approach in memory



Freeing the Contiguous Approach

- To free a 2D array allocated with the Contiguous Approach
  - Free the big block of memory
  - Free the array of rows
  - No loop needed

```
free (table[0]);
free (table);
```

Memory Allocation (System Side)

Memory allocation as seen from the system

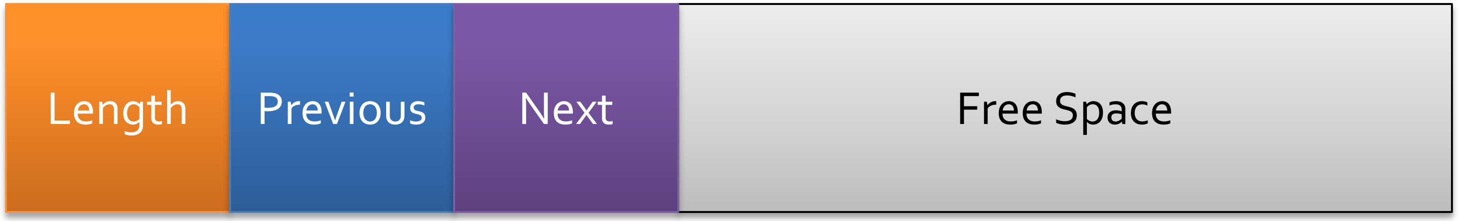
- There are really low level functions **`brk()`** and **`sbrk()`** which essentially increase the maximum size of the heap
- You can use any of that space as a memory playground
- **`malloc()`** gives finer grained control
  - But also has additional overhead

How does `malloc()` work?

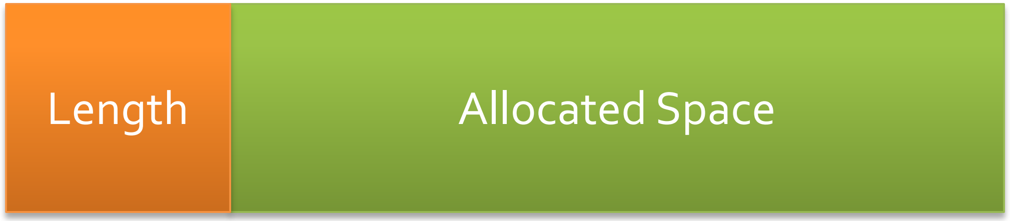
- `malloc()` sees a huge range of free memory when the program starts
- It uses a doubly linked list to keep track of the blocks of free memory, which is perhaps one giant block to begin with
- As you allocate memory, a free block is often split up to make the block you need
- The returned block *knows* its length
  - The length is usually kept **before** the data that you use

Free and allocated blocks

- The free list is a doubly linked list of available blocks of memory
- Each block knows its length, the next block in the list, and the previous block
- In a 32-bit architecture, the length, previous, and next data are all 4 bytes
  - Free block



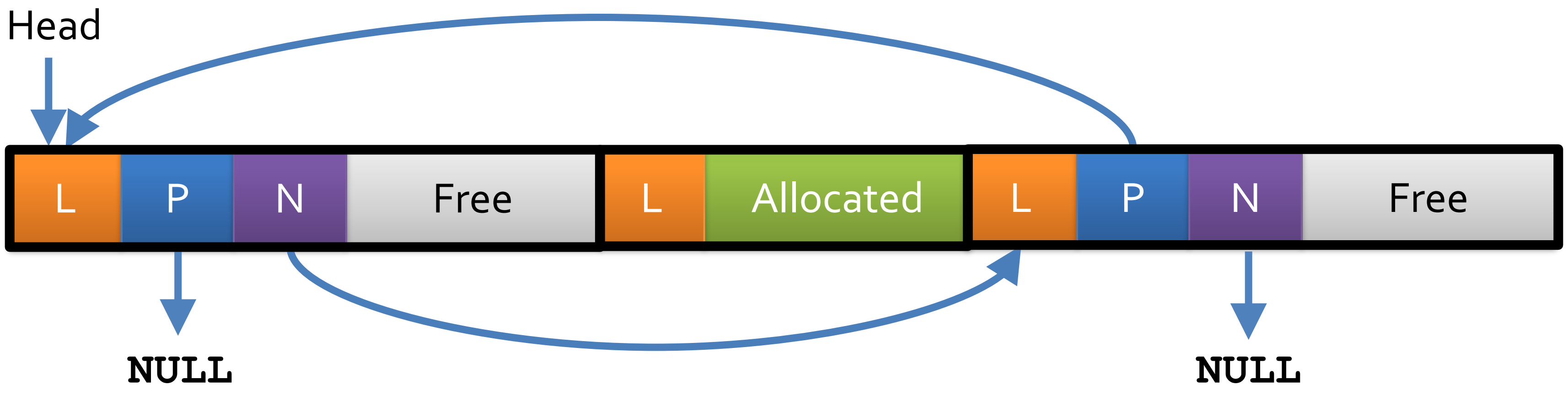
  - Allocated block



- In 64-bit, they're probably all 8 bytes

Free list

- Here's a visualization of the free list
- When an item is freed, most implementations will try to coalesce two neighboring free blocks to reduce fragmentation
  - Calling **`free()`** has some time overhead

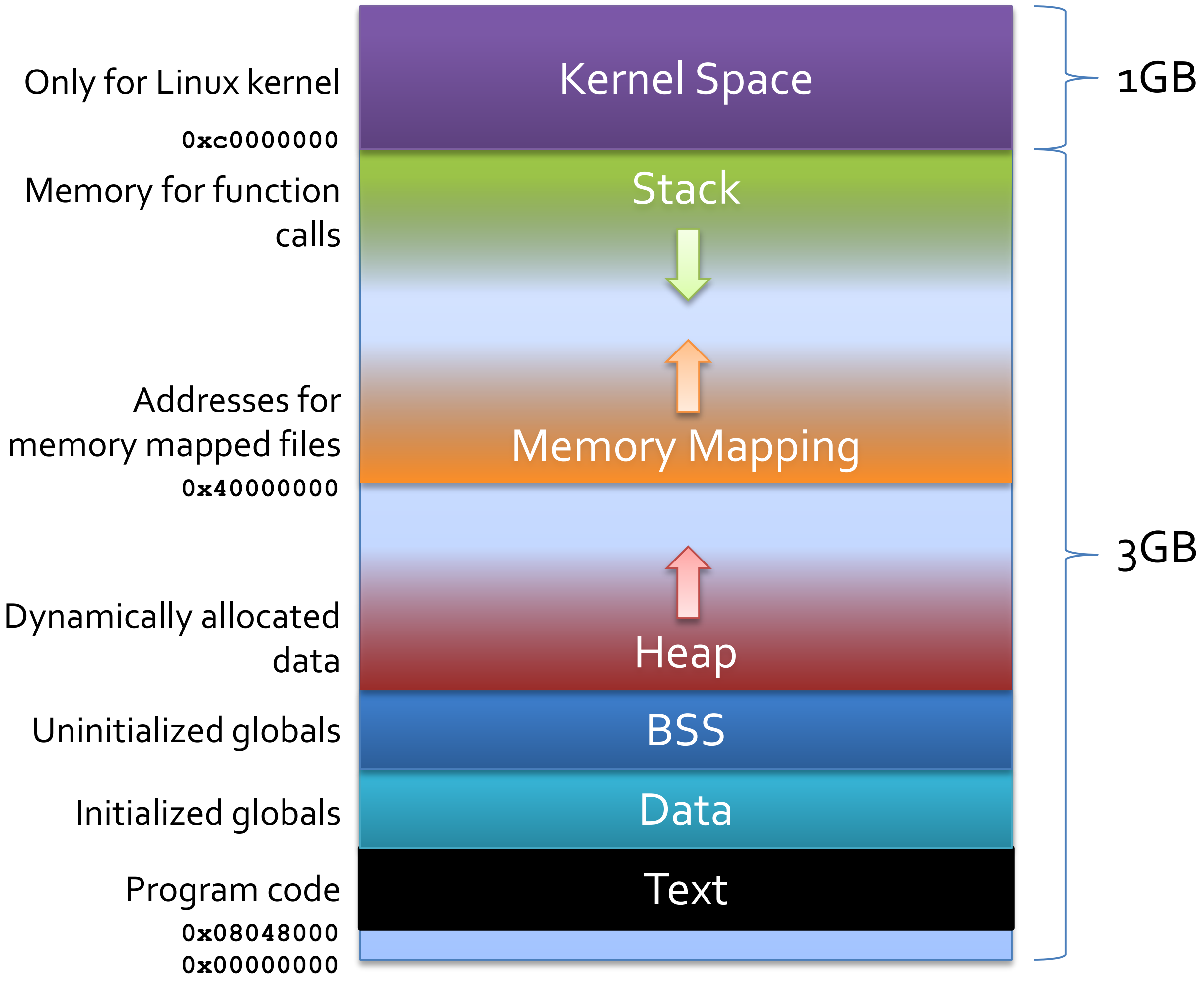


Other memory functions

- **`void* calloc(size_t items, size_t size);`**
  - Clear and allocate **`items`** items, each with size **`size`**
  - Memory is zeroed out
- **`void* realloc(void* pointer, size_t size);`**
  - Resize a block of memory pointed at by pointer, usually to be larger
  - If there is enough free space at the end, **`realloc()`** will tack that on
  - Otherwise, it allocates new memory and copies over the old
- **`void* alloca(size_t size);`**
  - Dynamically allocate memory on the stack (at the end of the current frame)
  - Automatically freed when the function returns
  - You need to **`#include <alloca.h>`**

Process memory segments

- Layout for 32-bit architecture
  - Could only address 4GB
- Modern layouts often have random offsets for stack, heap, and memory mapping for security reasons



Why aren't I showing the 64-bit version?

- The Linux machines in this lab use 64-bit processors with 64-bit versions of Ubuntu
- Our version of **gcc** supports 64-bit operations
  - Our pointers are 8 bytes in size
- But 64-bit stuff is confusing
  - They're still working out where the eventual standard will be
  - 64-bit addressing allows 16,777,216 terabytes of memory to be addressed (**far** beyond what anyone needs)
- Current implementations only use 48 bits
  - User space (text up through stack) gets low 128 terabytes
  - Kernel space gets the high 128 terabytes

Let's see those addresses

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


int global = 10;

int main()
{
      int stack = 5;
      int *heap = (int*)malloc (sizeof(int)*100);
      printf ("Stack:  %p\n", &stack);
      printf ("Heap:   %p\n", heap);
      printf ("Global: %p\n", &global);
      printf ("Text:   %p\n", main);
      return 0;
}
```

Random Numbers

Random numbers

- C provides the **`rand()`** function in **`stdlib.h`**
- **`rand()`** uses a **linear congruential generator (LCG)** to generate pseudorandom numbers
- **`rand()`** generates an **`int`** in the range 0 to **`RAND_MAX`** (a constant defined in **`stdlib.h`**)

Linear congruential generators

- LCGs use the following relation to determine the next pseudorandom number in a sequence
  - $x_{i+1} = (ax_i + c) \bmod m$
- I believe our version of the **`glibc`** uses the following values for **`rand()`**
  - $a = 1103515245$
  - $c = 12345$
  - $m = 2^{31} = 2147483648$

How do I use it?

- If you want values between **0** and **n** (not including **n**), you usually mod the result by **n**

```
//dice rolls
int die = 0;
for (int i = 0; i < 10; ++i)
{
  die = rand () % 6 + 1; //[0,5] + 1 is [1,6]
  printf ("Die value: %d\n", die);
}
```

Wait …

- Every time I run the program, I get the same sequence of random numbers
  - **Pseudo**random, indeed!
- This problem is fundamental to LCGs
- The pseudorandom number generated at each step is computed by the number from the previous step
  - By default, the starting point is 1

Seeding `rand()`

- To overcome the problem, we call **`srand()`** which allows us to set a starting point for the random numbers

```
int random = 0;
srand (93);
random = rand (); //starts from seed of 93
```

- But, if I always start with **93**, I'll still always get the same sequence of random numbers each time I run my program
- I need a random number to put into **`srand()`**
- I need a random number to get a random number?

Time is on our side

- Well, time changes when you run your program
- The typical solution is to use the number of seconds since January 1, 1970 as your seed
- To get this value, call the **time()** function with parameter **NULL**
  - You'll need to include **time.h**

```
int die = 0;
srand (time(NULL));
for (int i = 0; i < 10; ++i)
{
  die = rand () % 6 + 1; //[0,5] + 1 is [1,6]
  printf ("Die value: %d\n", die);
}
```

Rules for random numbers

- Include the following headers:
  - **`stdlib.h`**
  - **`time.h`**
- Use **`rand() % n`** to get values between **0** and **`n – 1`**
- Always call **`srand(time(NULL))`** **before** your first call to **`rand()`**
- Only call **`srand()`** **once** per program
  - Seeding multiple times makes no sense and usually makes your output much **less** random

Example

- Dynamically allocate an 8 × 8 array of `char` values
- Loop through each element in the array
  - With 1/8 probability, put a `'Q'` in the element, representing a queen
  - Otherwise, put a `' '` (space) in the element
- Print out the resulting chessboard
  - Use | and – to mark rows and columns
- Print out whether or not there are queens that can attack each other

Upcoming

Next time…

- Debugging
- Structs

Reminders

- Finish Project 3
  - Due tonight!
- Keep working on Project 4