Week 7 - Wednesday

CS222

Last time

- What did we talk about last time?
- **`scanf()`**
- Memory allocation
  - **`malloc()`**

Questions?

Project 3

Quotes

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald E. Knuth

free()

- C is not garbage collected liked Java
- If you allocate something on the stack, it disappears when the function returns
- If you allocate something on the heap, you have to deallocate it with **free()**
- **free()** does not set the pointer to be **NULL**
  - But you can afterwards

```
char* things = (char*)malloc(100);
free(things);
```

Who is responsible?

- Who is supposed to call **`free()`**?
- You should feel fear in your gut every time you write a **`malloc()`**
  - That fear should only dissipate when you write a matching **`free()`**
- You need to be aware of functions like **`strdup()`** that call **`malloc()`** internally
  - Their return values will need to be freed eventually
- Read documentation closely
  - And create good documentation for any functions you write that allocate memory

Using dynamic allocation

- Prompt the user for an integer giving the size of a list of numbers
- Dynamically allocate an array of the appropriate size
- Read each of the numbers into the array
- Sort the array
- Print it out
- Free the memory

Double freeing

- If you try to free something that has already been freed, your program will probably crash
- If you try to free a **`NULL`** pointer, it doesn't do anything

- Life is hard

Memory leaks

- Everything gets freed at the end of your program
- So, you can just hope you don't run out of space
- However, if you are constantly allocating things and never freeing them, you will run out of space

Memory leak example

- Let's see this in action

```
char* buffer;

while( 1 )
{
  buffer = (char*)malloc(1024);
  buffer[0] = 'a';
}
```

- On some machines, you'll run out of space pretty quickly
- On these, the system will try hard to make enough space for you

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

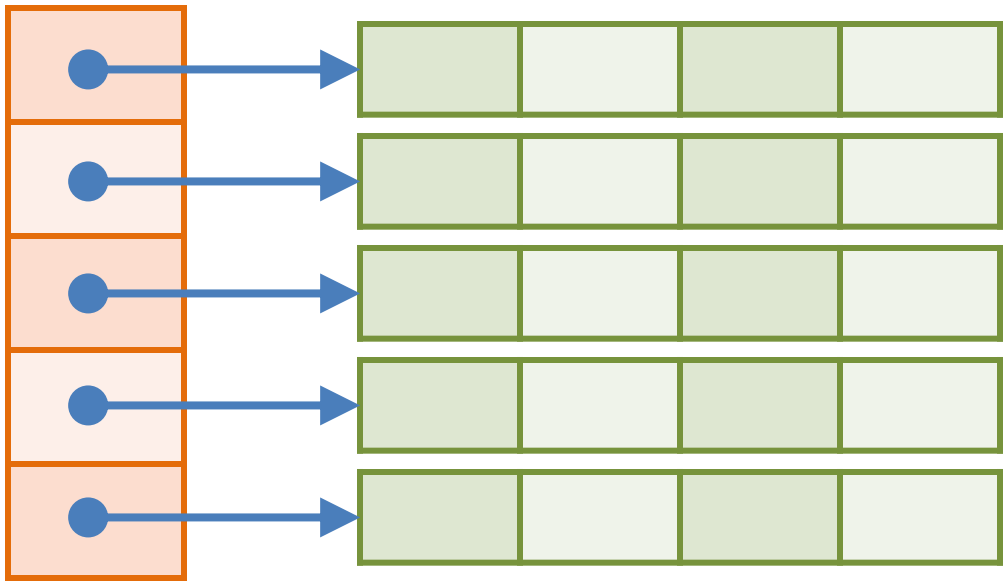
```
int** table = (int**)malloc(sizeof(int*)*rows);
int i = 0;

for( 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

`table`



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

free( table );
```

Contiguous Approach

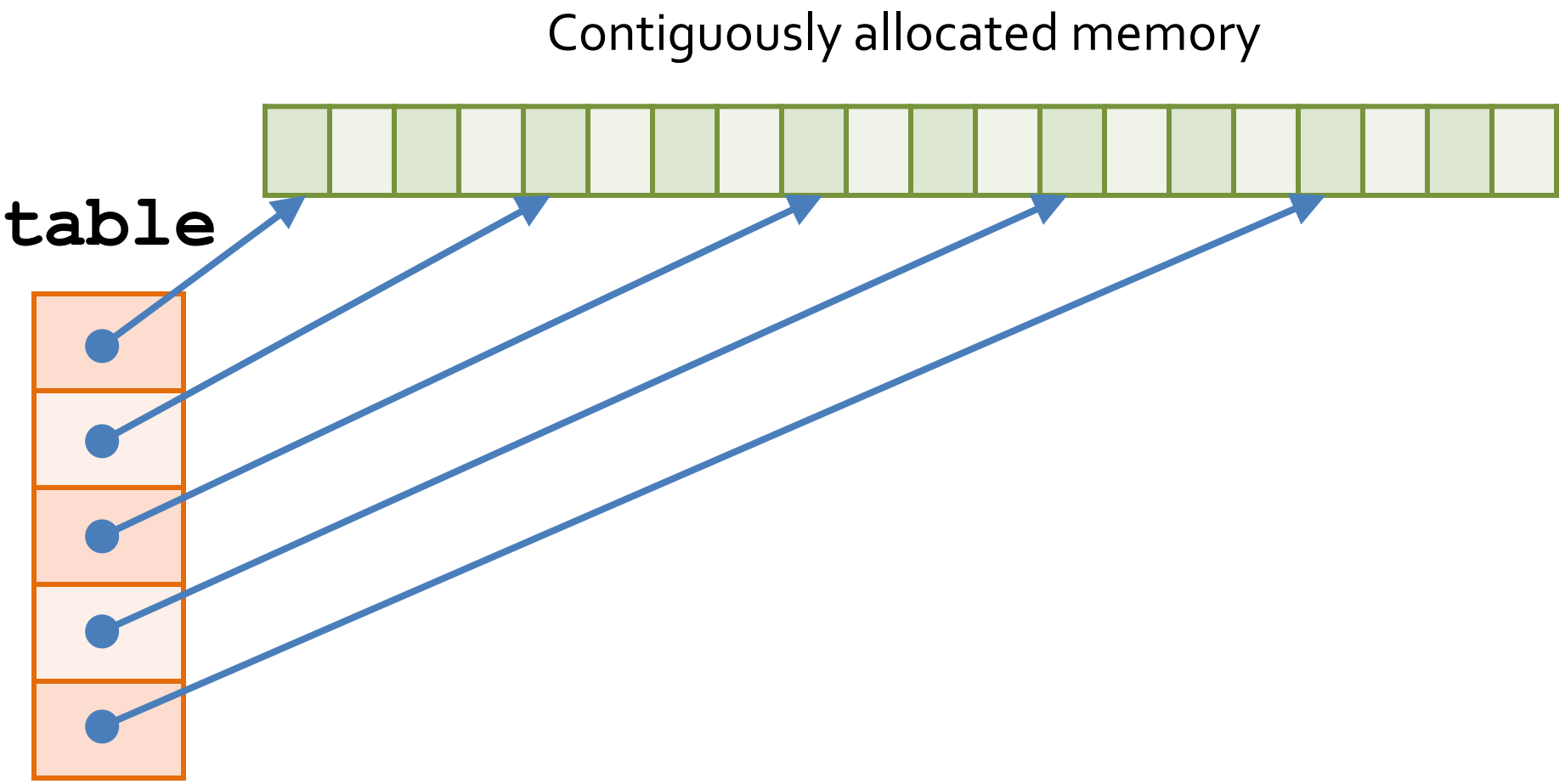
- Alternatively, you can allocate the memory for all rows at once
- Then you make each row point to the right place

```
int** table = (int**)malloc(sizeof(int*)*rows);
int* data = (int*)malloc(sizeof(int)*rows*columns);
int i = 0;

for( 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 );
```

Comparing the approaches

RAGGED

- Pros
  - Each row can be allocated and freed independently
  - Rows can be shuffled in order with only pointer changes
  - Rows can be different lengths
- Cons
  - Fragmented memory
  - Less locality of reference
  - Requires a loop to free

CONTIGUOUS

- Pros
  - Better locality of reference
  - Can free the entire thing with two `free()` calls
  - Shuffling rows with pointers is possible, but you also have to keep track of the beginning
- Cons
  - Large allocations are more likely to fail (out of memory)
  - Can't free individual rows

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;
int i = 0;
for( 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;
int i = 0;
srand(time(NULL));
for( 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 x 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

Quiz

Upcoming

Next time…

- Memory allocation from the system's perspective
- Lab 7

Reminders

- Read LPI chapter 7
- **Finish Project 3**
  - **Due Friday by midnight!**