Week 7 - Wednesday

COMP 2400

Last time

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

Questions?

Project 3

Project 4

Project 4

Quotes

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

Donald E. Knuth

Dynamic Memory Allocation

malloc()

- Memory can be allocated dynamically using a function called **malloc()**
  - Similar to using **new** in Java or C++
  - **#include <stdlib.h>** to use **malloc()**
- Dynamically allocated memory is on the heap
  - It doesn't disappear when a function returns
- To allocate memory, call **malloc()** with the number of bytes you want
- It returns a pointer to that memory, which you cast to the appropriate type

```
int* data = (int*)malloc(sizeof(int));
```

`free()`

- C is not garbage collected like 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 (and should) afterwards

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

Who is responsible?

- Who is supposed to call **`free()`**?
- You should feel fear in your gut every time you type **`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

Double freeing

- If you try to free something that has already been freed, your program will probably crash
- If you use data that's already been freed, your program *might* crash
- If you try to free a **`NULL`** pointer, it's fine

- Life is hard.

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

`realloc()`

- There are situations (like on the next slide) where you want to grow the amount of memory that you've allocated
- When that happens, you can call **`realloc()`**
  - It takes the old pointer and a size in bytes
  - It returns a pointer to new memory with all the old values copied over and the old memory freed
  - You almost always store the return value in the old pointer
- You could do this with a temporary pointer and a loop, but **`realloc()`** is easier and faster

```
int *data = malloc(sizeof(int) * 100);
// Do some work
// It turns out that we need twice as much space
data = realloc(data, sizeof(int) * 200);
```

What if we keep adding stuff to a list?

- You probably did an array-backed list in Java in COMP 2100
- Strategy:
  - Allocate an initial array of, say, 10 elements in size
  - Keep track of the capacity (starts at 10)
  - Keep track of the number of elements actually used (starts at 0)
  - When the array is full, we'll use **`realloc()`** to double the capacity of our memory

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're constantly allocating things and never freeing them, you will run out of space
  - Eventually, **`malloc()`** will return **`NULL`** when it can't allocate more space
  - More likely, your program will get slower and slower as the OS tries to make more memory for you, probably copying memory to the hard drive so that you can exceed your physical RAM

Memory leak example

- Let's see this in action (but don't do this in a real program!)

```
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

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

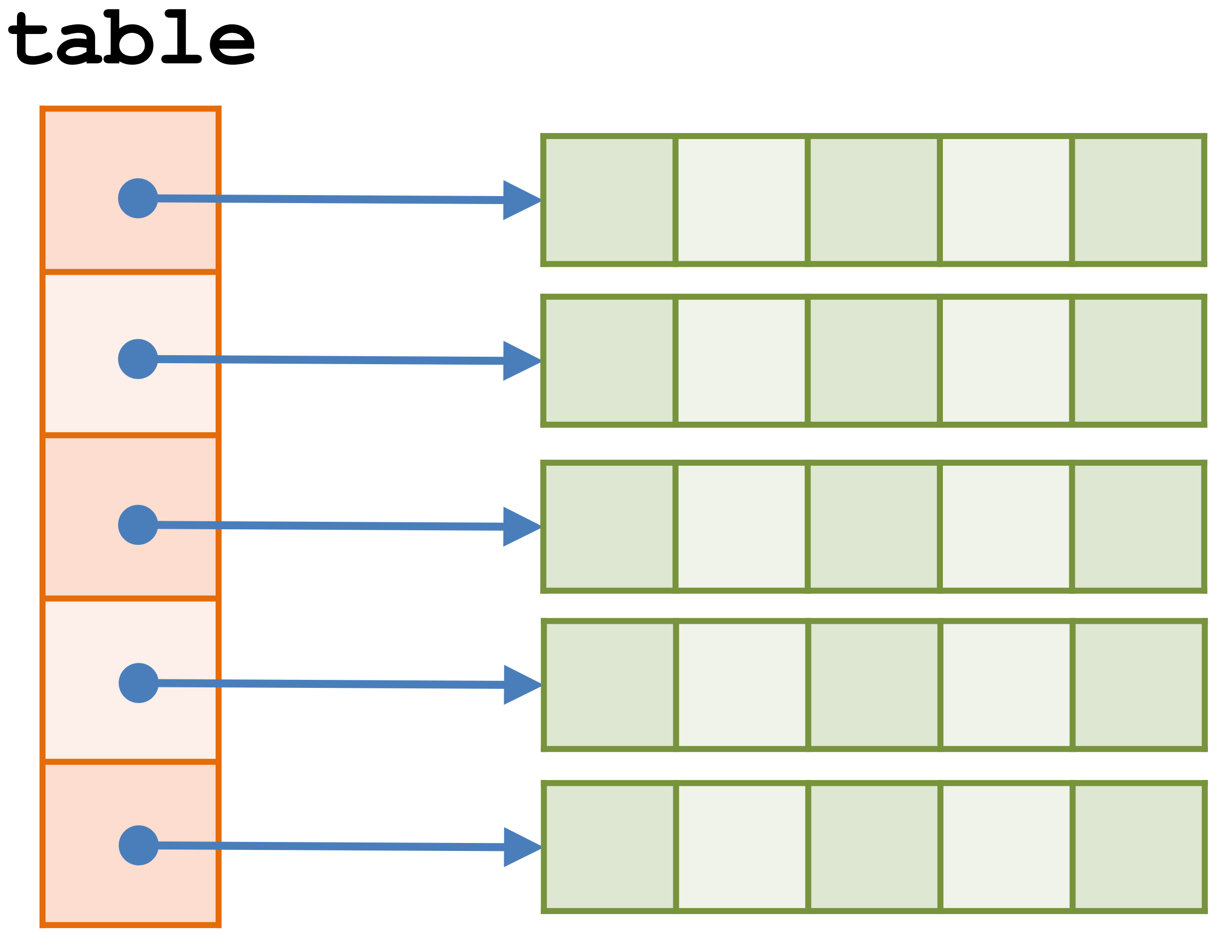
```
int** table = (int**)malloc (sizeof(int*)*rows);

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

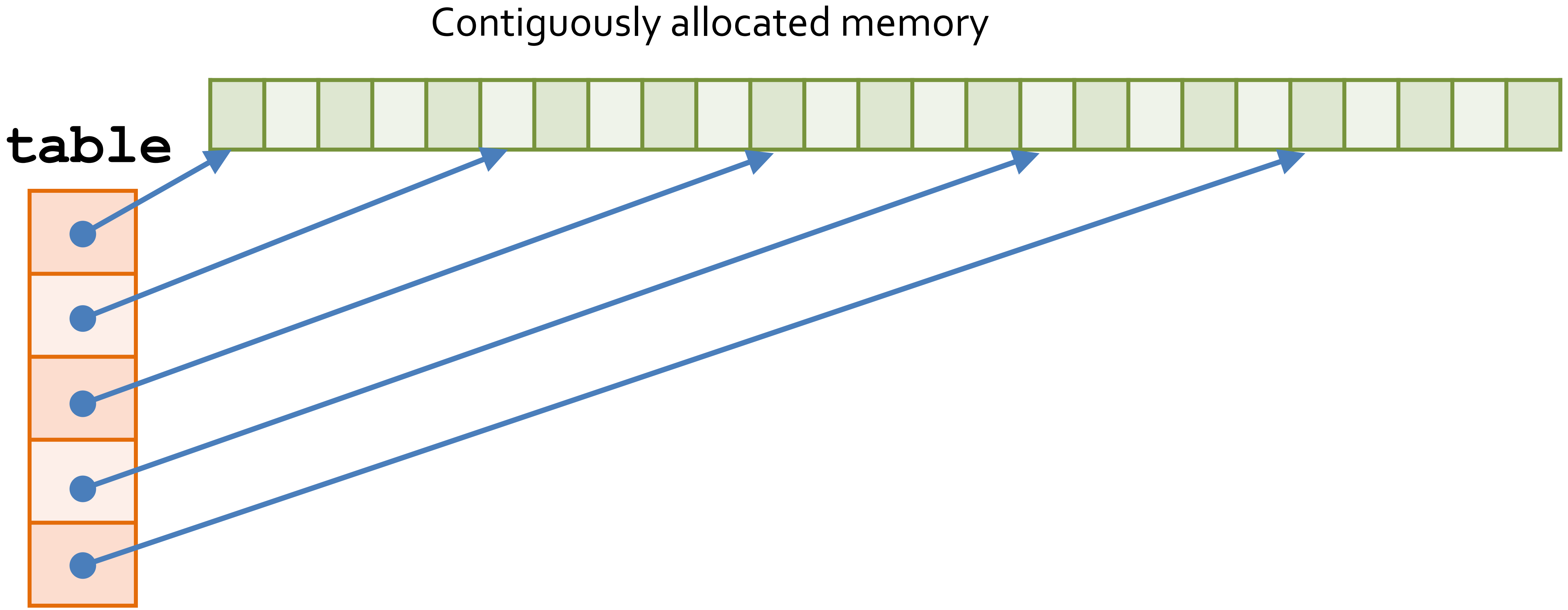
```
int** table = (int**)malloc (sizeof(int*)*rows);
int* data = (int*)malloc (sizeof(int)*rows*columns);

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);
```

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

Ticket Out the Door

Upcoming

Next time…

- Random numbers
- Memory allocation from the system's perspective

Reminders

- Read LPI chapter 7
- Finish Project 3
- Start working on Project 4
  - It's tricky!