Week 15 - Wednesday

CS222

Last time

- What did we talk about last time?
- Review up to Exam 1

Questions?

Project 6

Programming Practice

- Write a function that converts an **`int`** to a string representation in hexadecimal
- Write a function that converts a string representation of an integer in hexadecimal to an **`int`**

Review up to Exam 2

Pointers

- A **pointer** is a variable that holds an address
- Often this address is to another variable
- Sometimes it's to a piece of memory that is mapped to file I/O or something else
- Important operations:
  - Reference (**&**) gets the address of something
  - Dereference (*****) gets the contents of a pointer

Declaration of a pointer

- We typically want a pointer that points to a certain kind of thing
- To declare a pointer to a particular type

```
type * name;
```

- Example of a pointer with type **`int`**:

```
int * pointer;
```

Reference operator

- A fundamental operation is to find the address of a variable
- This is done with the reference operator (**&**)

```
int value = 5;
int* pointer;
pointer = &value;
//pointer has value's address
```

- We usually can't predict what the address of something will be

Dereference operator

- The reference operator doesn't let you do much
- You can get an address, but so what?
- Using the dereference operator, you can read and write the contents of the address

```
int value = 5;
int* pointer;
pointer = &value;
printf("%d", *pointer); //prints 5
*pointer = 900; //value just changed!
```

Pointer arithmetic

- One of the most powerful (and most dangerous) qualities of pointers in C is that you can take arbitrary offsets in memory
- When you add to (or subtract from) a pointers, it jumps the number of bytes in memory of the size of the type it points to

```
int a = 10;
int b = 20;
int c = 30;
int* value = &b;
value++;
printf("%d", *value); //what does it print?
```

Arrays are pointers too

- An array **is** a pointer
  - It is pre-allocated a fixed amount of memory to point to
  - You can't make it point at something else
- For this reason, you can assign an array directly to a pointer

```
int numbers[] = {3, 5, 7, 11, 13};
int* value;

value = numbers;
value = &numbers[0]; //exactly equivalent

//What about the following?
value = &numbers;
```

Surprisingly, pointers are arrays too

- Well, no, they aren't
- But you can use array subscript notation (**[ ]**) to read and write the contents of offsets from an initial pointer

```
int numbers[] = {3, 5, 7, 11, 13};
int* value = numbers;

printf("%d", value[3] ); //prints 11
printf("%d", *(value + 3) ); //prints 11
value[4] = 19; //changes 13 to 19
```

void pointers

- What if you don't know what you're going to point at?
- You can use a **void***, which is an address to….something!
- You have to cast it to another kind of pointer to use it
- You can't do pointer arithmetic on it
- It's not useful very often

```
char s[] = "Hello World!";
void* address = s;
int* thingy = (int*)address;
printf("%d\n", *thingy);
```

Functions that can change arguments

- In general, data is passed **by value**
- This means that a variable cannot be changed for the function that calls it
- Usually, that's good, since we don't have to worry about functions screwing up our data
- It's annoying if we need a function to return more than one thing, though
- Passing a pointer is equivalent to passing the original data **by reference**

Pointers to pointers

- Just as we can declare a pointer that points at a particular data type, we can declare a pointer to a pointer
- Simply add another star

```
int value = 5;
int* pointer;
int** amazingPointer;
pointer = &value;
amazingPointer = &pointer;
```

Change `main()` to get command line arguments

- To get the command line values, use the following definition for **`main()`**

```
int main(int argc, char** argv)
{

  return 0;
}
```

- Is that even allowed?
  - Yes.
- You can name the parameters whatever you want, but **`argc`** and **`argv`** are traditional
  - **`argc`** is the number of arguments (argument count)
  - **`argv`** are the actual arguments (argument values) as strings

scanf()

- So far, we have only talked about using **`getchar()`** (and command line arguments) for input
- As some of you have discovered, there is a function that parallels **`printf()`** called **`scanf()`**
- **`scanf()`** can read strings, **`int`** values, **`double`** values, characters, and anything else you can specify with a `%` formatting string

```
int number;
scanf("%d", &number);
```

Format specifiers

- These are mostly what you would expect, from your experience with **`printf()`**

Specifier	Type
`%d`	**`int`**
`%u`	**`unsigned int`**
`%o %x`	**`unsigned int`** (in octal for **`o`** or hex for **`x`**)
`%hd`	**`short`**
`%c`	**`char`**
`%s`	null-terminated string
`%f`	**`float`**
`%lf`	**`double`**
`%Lf`	**`long double`**

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));
```

Allocating arrays

- It is common to allocate an array of values dynamically
- The syntax is exactly the same as allocating a single value, but you multiply the size of the type by the number of elements you want

```
int i = 0;
int* array = (int*)malloc(sizeof(int)*100);
for( i = 0; i < 100; i++ )
  array[i] = i + 1;
```

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);
```

Ragged Approach

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

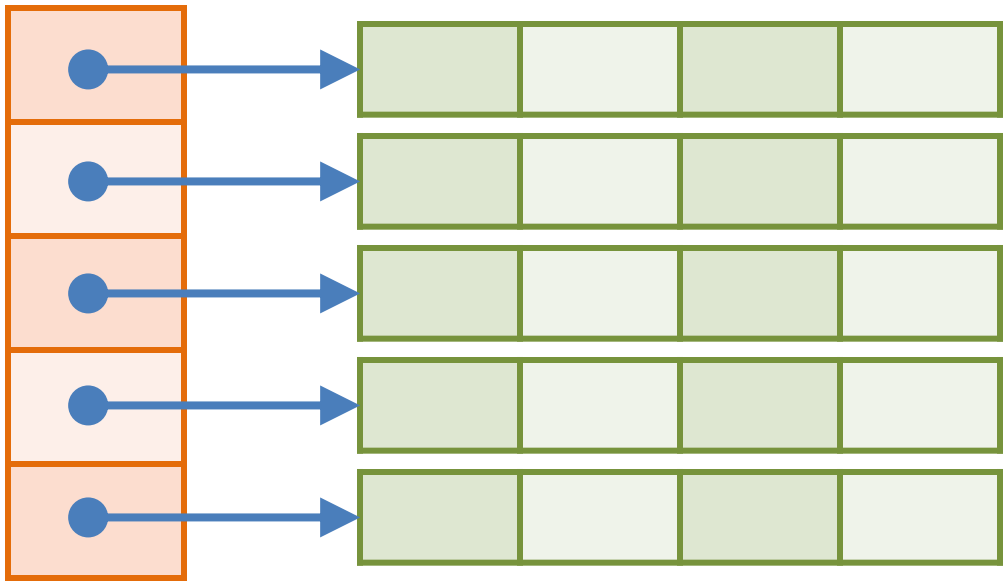
```
int** table = (int**)malloc(sizeof(int*)*rows);
int i = 0;

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

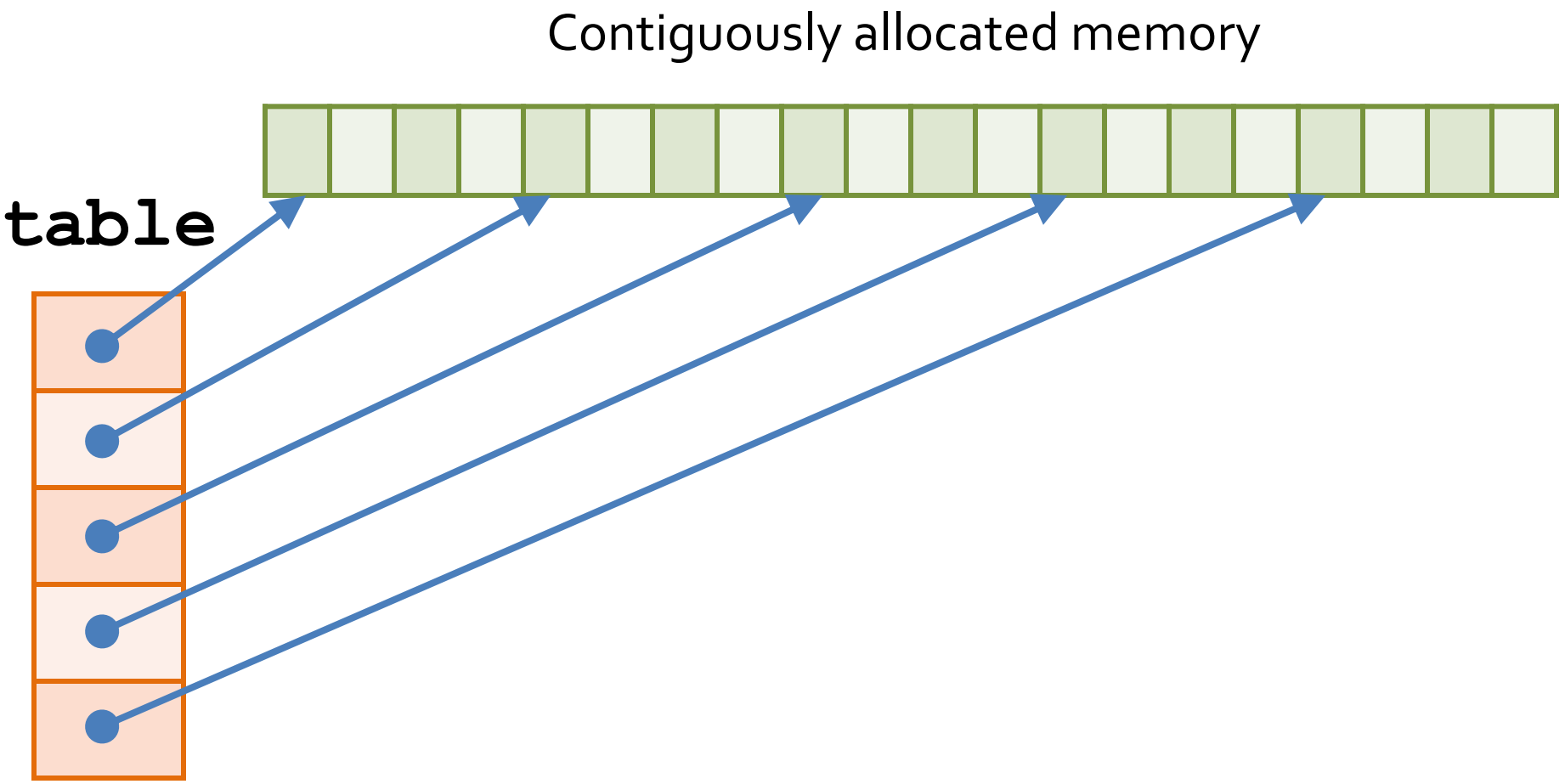
- Alternatively, you can allocate the memory for all rows at once
- Then you make each row point to the right place

```
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 Ragged 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

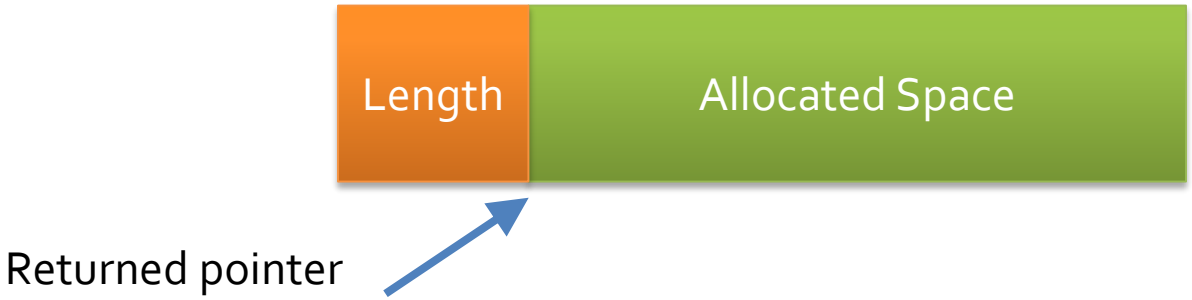
- Pros
  - Better locality of reference
  - Can free the entire thing with two `free()` calls
  - Shuffling rows with pointers is possible, but you also have to keep track of the beginning
- Cons
  - Large allocations are more likely to fail (out of memory)
  - Can't free individual rows

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

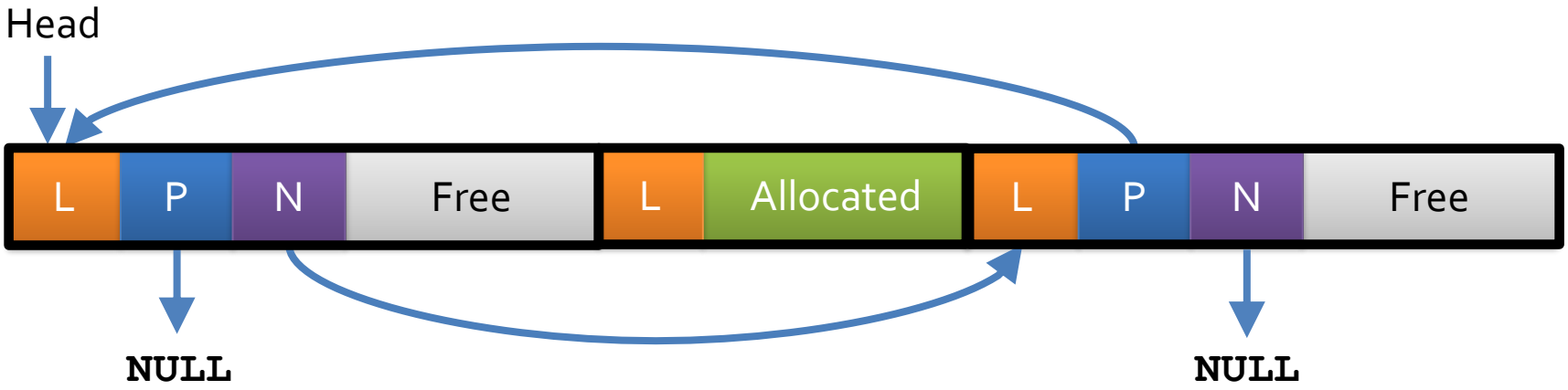
How does `malloc()` work?

- **`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 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()`** can be time consuming



Other memory functions

- **`void* calloc(size_t items, size_t size);`**
  - Clear and allocate **`items`** items 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>`**

Unit testing

- **Unit tests** are tests for a single piece of code in isolation for the rest of your program
- A unit test could test
  - A single method or function
  - A whole class

Test case selection

- Positive test cases
  - Cases that should work, and we can easily check the answer
- Boundary condition test cases
  - Special cases, like deleting the first or last item in a list
  - Sometimes called "corner cases"
- Negative test cases
  - Cases that we know should fail

Regression testing

- When you find a bug, keep the test case that demonstrated the bug
- Include it as part of your test suite
- Always run all your tests when you make a change to your code
- Sometimes a "fix" can break something that was working
  - Especially if you didn't fix the bug correctly the first time

Black box testing

- One philosophy of testing is making **black box tests**
- A black box test takes some input **A** and knows that the output is supposed to be **B**
- It assumes nothing about the internals of the program
- To write black box tests, you come up with a set of input you think covers lots of cases and you run it and see if it works
- In the real world, black box testing can easily be done by a team that did not work on the original development

White box testing

- **White box testing** is the opposite of black box testing
  - Sometimes white box testing is called "clear box testing"
- In white box testing, you can use your knowledge of how the code works to generate tests
- Are there lots of if statements?
  - Write tests that go through all possible branches
- There are white box testing tools that can help you generate tests to exercise all branches
- Which is better, white box or black box testing?

String to integer

- In C, the standard way to convert a string to an **`int`** is the **`atoi()`** function
  - **`#include <stdlib.h>`** to use it

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

int main()
{
      char* value = "3047";
      int x = atoi(value);
      printf("%d\n", x);
      return 0;
}
```

Integer to string

- Oddly enough, this is a stranger situation
- Many systems have a non-standard function (also in **stdlib.h**) called **itoa()**
  - It takes the **int**, a buffer to hold the resulting string, and the base

```
char value[10];  //has to be big enough
int x = 3047;
itoa( x, value, 10 );
```

- The portable way to do this is to use **sprintf()**
  - It's like **printf()** except that it prints things to a string buffer instead of the screen

```
char value[10];  //has to be big enough
int x = 3047;
sprintf( value, "%d", x );
```

Users

- Recall that each user on a Linux system has a unique login name and a unique numerical identifier (the UID)
- Users can belong to one or more groups as well
- Where is this information stored?

Password file

- The system has a password file stored in **`/etc/passwd`**
- Each line of this file corresponds to one user in the system and has seven fields separated by colons:
  - Login name
  - Encrypted password
  - UID
  - GID (group ID of the first group that the user is a member of)
  - Comment
  - Home directory (where you are when you log in)
  - Login shell (which shell you running when you log in)
- Example:

```
wittmanb:x:1000:100:Barry Wittman:/home/wittmanb:/bin/bash
```

The Linux and Unix solution

- Instead of storing actual passwords, Linux machines store the hash of the passwords
- When someone logs on, the operating system hashes the password and compares it to the stored version
- No one gets to see your original password
  - Not even **`root`**!

Groups

- Files are associated with a group as well as a user who is owner
- The groups are listed in the /etc/group file
- Each line of this file corresponds to a group and has four fields separated by colons:
  - Group name
  - Encrypted password
    - Often not used
  - Group ID (GID)
  - User list
    - Comma separated
- Example:

```
users:x:100:
jambit:x:106:claus,felli,frank,harti,markus,martin,mtk,paul
```

Time

- In the systems programming world, there are two different kinds of time that are useful
- Real time
  - This is also known as wall-clock time or calendar time
  - It's the human notion of time that we're familiar with
- Process time
  - Process time is the amount of time your process has spent on the CPU
  - There is often no obvious correlation between process time and real time (except that process time is never more than real time elapsed)

time()

- The **time()** function gives back the seconds since the Unix Epoch
- Its signature is:

```
time_t time(time_t* timePointer);
```

- **time_t** is a signed 32-bit or 64-bit integer
- You can pass in a pointer to a **time_t** variable or save the return value (both have the same result)
- Typically we pass in **NULL** and save the return value
- Include **time.h** to use **time()**

```
time_t seconds = time(NULL);
printf("%d seconds have passed since 1970",
      seconds);
```

gettimeofday()

- The **gettimeofday()** function offers a way to get higher precision timing data
- Its signature is:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

- The struct **timeval** has a **tv_secs** member which is the same as the return value from **time()**
- It also has a **tv_usec** member which gives microseconds (millionths of a second)
- The **timezone** pointer **tz** is obsolete and should have **NULL** passed into it
- Include **sys/time.h** (**not** the same as **time.h**) to use this function

Process time

- For optimization purposes, it can be useful to know how much time a process spends running on the CPU
- This time is often broken down into
  - **User time:** the amount of time your program spends executing its own code
  - **System time:** the amount of time spent in kernel mode executing code for your program (memory allocation, page faults, file opening)

The `time` command

- You can time a program's complete execution by running it with the time command
  - It will give the real time taken, user time, and system time
- Let's say you've got a program called **`timewaster`**
  - Run it like this:

```
time ./timewaster
```

  - Output might be:

```
real 0m4.84s
user 0m1.030s
sys 0m3.43s
```

Structs

- A struct in C is:
  - A collection of one or more variables
  - Possibly of different types
  - Grouped together for convenient handling.
- They were called records in Pascal
- They have similarities to a class in Java
  - Except all fields are public and there are no methods
- Struct declarations are usually global
  - They are outside of **`main()`** and often in header files

Anatomy of a `struct`

```
struct name
{
    type1 member1;
    type2 member2;
    type3 member3;
    ...
};
```

Declaring a struct variable

- Type:
  - **`struct`**
  - The name of the struct
  - The name of the identifier
- You have to put **`struct`** first

```
struct student bob;
struct student jameel;
struct point start;
struct point end;
```

Accessing members of a struct

- Once you have a struct variable, you can access its members with dot notation (**`variable.member`**)
  - Members can be read and written

```
struct student bob;
strcpy(bob.name, "Bob Blobberwob");
bob.GPA = 3.7;
bob.ID = 100008;
printf("Bob's GPA: %f\n", bob.GPA);
```

Initializing structs

- There are no constructors for structs in C
- You can initialize each element manually:

```
struct student julio;
strcpy(julio.name, "Julio Iglesias");
julio.GPA = 3.9;
julio.ID = 100009;
```

- Or you can use braces to initialize the entire struct at once:

```
struct student julio =
{ "Julio Iglesias", 3.9, 100009 };
```

Assigning structs

- It is possible to assign one struct to another

```
struct student julio;
struct student bob;
strcpy(julio.name, "Julio Iglesias");
julio.GPA = 3.9;
julio.ID = 100009;
bob = julio;
```

- Doing so is equivalent to using **`memcpy()`** to copy the memory of **`julio`** into the memory of **`bob`**
- **`bob`** is still separate memory: it's not like copying references in Java

Dangers with pointers in structs

- With a pointer in a struct, copying the struct will copy the pointer but will not make a copy of the contents
- Changing one struct could change another

```
struct person
{
        char* firstName;
        char* lastName;
};
struct person bob1;
struct person bob2;
```

```
bob1.firstName = strdup("Bob");
bob1.lastName = strdup("Newhart");
bob2 = bob1;
strcpy(bob2.lastName, "Hope");
printf("Name: %s %s\n", bob1.firstName, bob1.lastName);
//prints Bob Hope
```

Arrow notation

- We could dereference a struct pointer and then use the dot to access a member

```
struct student* studentPointer = (struct
student*) malloc(sizeof(struct student));

(*studentPointer).ID = 3030;
```

- This is cumbersome and requires parentheses
- Because this is a frequent operation, dereference + dot can be written as an arrow (**->**)

```
studentPointer->ID = 3030;
```

Passing structs to functions

- If you pass a struct directly to a function, you are passing it by value
  - A copy of its contents is made
- It is common to pass a struct by pointer to avoid copying and so that its members can be changed

```
void flip(struct point* value)
{
    double temp = value->x;
    value->x = value->y;
    value->y = temp;
}
```

Gotchas

- **Always** put a semicolon at the end of a struct declaration
- Don't put constructors or methods inside of a struct
  - C doesn't have them
- Assigning one struct to another copies the memory of one into the other
- Pointers to struct variables are usually passed into functions
  - Both for efficiency and so that you can change the data inside

typedef

- The **typedef** command allows you to make an alias for an existing type
- You type **typedef**, the type you want to alias, and then the new name

```
typedef int SUPER_INT;

SUPER_INT value = 3; //has type int
```

- Don't overuse **typedef**
- It is useful for types like **time_t** which can have different meanings in different systems

typedef with structs

- The **typedef** command is commonly used with structs
  - Often it is built into the struct declaration process
- It allows the programmer to leave off the stupid **struct** keyword when declaring variables

```
typedef struct _wombat
{
        char name[100];
        double weight;
} wombat;
```

- The type defined is actually **struct _wombat**
- We can refer to that type as **wombat**

```
wombat martin;
```

An example linked list node struct

- We'll use this definition for our node for singly linked lists

```
typedef struct _node
{
      int data;
      struct _node* next;
} node;
```

- Somewhere, we will have the following variable to hold the beginning of the list

```
node* head = NULL;
```

Using enum

- To create named constants with different values, type **enum** and then the names of your constants in braces

```
enum { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY };
```

- Then in your code, you can use these values (which are stored as integers)

```
int day = FRIDAY;
if( day == SUNDAY )
     printf("My 'I don't have to run' day");
```

Specifying values

- You can even specify the values in the **enum**

```
enum { ANIMAL = 7, MINERAL = 9, VEGETABLE = 11 };
```

- If you assign values, it is possible to make two or more of the constants have the same value (usually bad)
- A common reason that values are assigned is so that you can do bitwise combinations of values

```
enum { PEPPERONI = 1, SAUSAGE = 2, BACON = 4,
MUSHROOMS = 8, PEPPER = 16, ONIONS = 32, OLIVES
= 64, EXTRA_CHEESE = 128 };

int toppings = PEPPERONI | ONIONS | MUSHROOMS;
```

An example doubly linked list node struct

- We'll use this definition for our node for doubly linked lists

```
typedef struct _node
{
      int data;
      struct _node* next;
      struct _node* previous;
} node;
```

- Somewhere, we will have the following variables to hold the beginning and ending of the list

```
node* head = NULL;
node* tail = NULL;
```

Bit fields in a struct

- You can define a struct and define how many bits wide each element is
  - It only works for integral types, and it makes the most sense for **`unsigned int`**
  - Give the number of bits it uses after a colon
  - The bits can't be larger than the size the type would normally have
  - You can have unnamed fields for padding purposes

```
typedef struct _toppings
{
        unsigned pepperoni : 1;
        unsigned sausage   : 1;
        unsigned onions    : 1;
        unsigned peppers   : 1;
        unsigned mushrooms : 1;
        unsigned sauce     : 1;
        unsigned cheese    : 2;
        //goes from no cheese to triple cheese
} toppings;
```

Struct size and padding

- Remember that structs are always padded out to multiples of 4 bytes (well, of **`int`** size)
  - Unless you use compiler specific statements to change byte packing
- After the last bit field, there will be empty space up to the nearest 4 byte boundary
- You can mix bit field members and non-bit field members in a struct
  - Whenever you switch, it will pad out to 4 bytes
  - You can also have **0** bit fields which also pad out to 4 bytes

Declaring unions

- Unions look like structs
  - Put the keyword **`union`** in place of **`struct`**

```
union Congressperson
{
      int district;    //representatives
      char state[15]; //senators
};
```

- There isn't a separate district and a state
  - There's only space for the larger one
  - In this case, 15 bytes (rounded up to 16) is the larger one

Practice

- Write a function that finds the median of an array
  - You'll have to sort it
- Write a function that, given a string, creates a dynamically allocated chunk of memory containing the string reversed
- Write a function that will delete an element from the doubly linked list struct given in earlier slides
- Write a program that counts the **total** number of characters in all the arguments passed in through the command line
  - Ignore **`argv[0]`**
- Write a program to "encrypt" a file by writing a new file with exactly the same contents, except that each byte in the file is inverted
  - Old byte: ***x***
  - New byte: 255 - ***x***

Quiz

Upcoming

Next time…

- Review after Exam 2
- Lab 15

Reminders

- Study for Final Exam
  - 11:00am - 2:00pm, Tuesday, 5/08/2018 (Section A)
  - 11:00am - 2:00pm, Monday, 5/07/2018 (Section B)

- **Finish Project 6**
  - **Due this Friday before midnight**