

Week 10 - Friday

COMP 2400

Last time

- What did we talk about last time?
- Unions
- Trees
- Time

Questions?

Project 4

Back to Time

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);
```

Time structures

- Many time functions need different structs that can hold things
- One such struct is defined as follows:

```
struct timeval
{
    time_t tv_sec;           // Seconds since Epoch
    suseconds_t tv_usec;    // Extra microseconds
};
```

gettimeofday ()

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

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

- From the previous slide, **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- The **timezone** pointer **tz** is obsolete and should have **NULL** passed into it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

Timing with `gettimeofday()`

- `gettimeofday()` is a reliable way to see how long something takes
- Get the start time, the end time, and subtract them

```
double start;
double end;
struct timeval tv;
gettimeofday(&tv, NULL);
start = tv.tv_sec + tv.tv_usec/1000000.0;
someLongRunningFunction();
gettimeofday(&tv, NULL);
end = tv.tv_sec + tv.tv_usec/1000000.0;
printf("Your function took %.3f seconds", end - start);
```

Back to Unions

Unions

- What if you wanted a data type that could hold any of three different things
 - But it would only hold one at a time...
- Yeah, you probably wouldn't want that
- But, back in the day when space was important, maybe you would have
- This is exactly the problem that unions were designed to solve

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

What's in the union?

- How can you tell what's in the union?
 - You can't!
- You need to keep separate information that says what's in the union
- Anonymous (unnamed) unions inside of structs are common

```
struct Congressperson
{
    bool senator;           // Which one?
    union
    {
        int district;      // Representatives
        char state[15];    // Senators
    };
};
```

Operands and operators

- We could use such a struct to store terms in an algebraic expression
- Terms are of the following types
 - Operands are double values
 - Operators are char values: +, -, *, and /

```
typedef enum { OPERATOR, OPERAND } Type;
typedef struct
{
    Type type;
    union
    {
        double operand;
        char operator;
    };
} Term;
```

Stack

- A stack is a simple (but useful) data structure that has three basic operations:
 - **Push** Put an item on the top of the stack
 - **Pop** Remove an item from the top of the stack
 - **Top** Return the item currently on the top of the stack
- This kind of data structure is sometimes referred to as an **Abstract Data Type (ADT)**
- We don't actually care how the ADT works, as long as it supports certain basic operations

Stack of double values

- We can implement a stack of **double** values

```
typedef struct
{
    double* values;
    int size;
    int capacity;
} Stack;
```

Stack initialization

- Initializing the stack isn't hard
 - We give it an initial capacity (perhaps 5)
 - We allocate enough space to hold that capacity
 - We set the size to 0

```
Stack stack;  
stack.capacity = 5;  
stack.values = (double*)malloc(sizeof(double) * stack.capacity );  
stack.size = 0;
```

Push, pop, and top

- We can write simple methods that will do the operations of the stack ADT

```
void push(Stack* stack, double value);
```

```
double pop(Stack* stack);
```

```
double top(Stack* stack);
```

Postfix notation

- You might recall postfix notation from COMP 2100
 - It's an unambiguous way of writing mathematical expressions
- Whenever you see an operand, put it on the stack
- Whenever you see an operator, pop the last two things off the stack, perform the operation, then put the result back on the stack
- The last thing should be the result
- Example: $5 \ 6 \ + \ 3 \ -$ gives $(5 + 6) - 3 = 8$

Evaluate postfix

- Finally, we have enough machinery to evaluate an array of postfix terms
- Write the following function that does the evaluation:

```
double evaluate(Term terms[], int size);
```

- We'll have to see if each term is an operator or an operand and interact appropriately with the stack

Review

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? (not defined)
```

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

// The following is not allowed!
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 + 2;  
  
printf("%d", value[0] ); // Prints 7  
printf("%d", value[-2] ); // Prints 3  
value[2] = 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
- **malloc()** returns a **void***, but our compiler casts it for us

```
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 ()

- Before, we only talked about using `getchar ()` (and command line arguments) for input
- 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
- You must pass in a pointer for the memory you want to read into

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

Format specifiers

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

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

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 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 afterwards

```
char* things = (char*)malloc (100);  
free (things); // Should have used things first  
things = NULL;
```

Ragged Approach

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

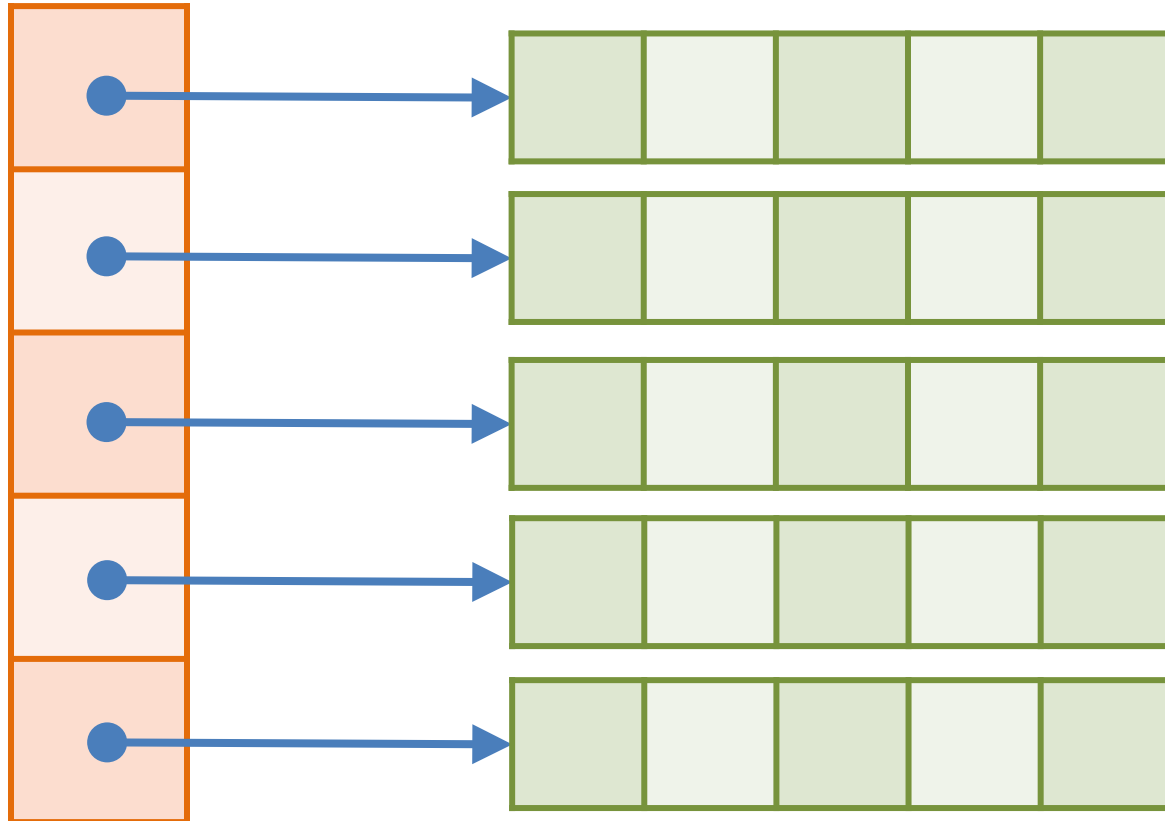
```
int** table = (int**)malloc (sizeof(int*) *rows) ;  
int i = 0 ;  
  
for (i = 0; i < rows; ++i)  
    table[i] = (int*)malloc (sizeof(int) *columns) ;
```

- 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

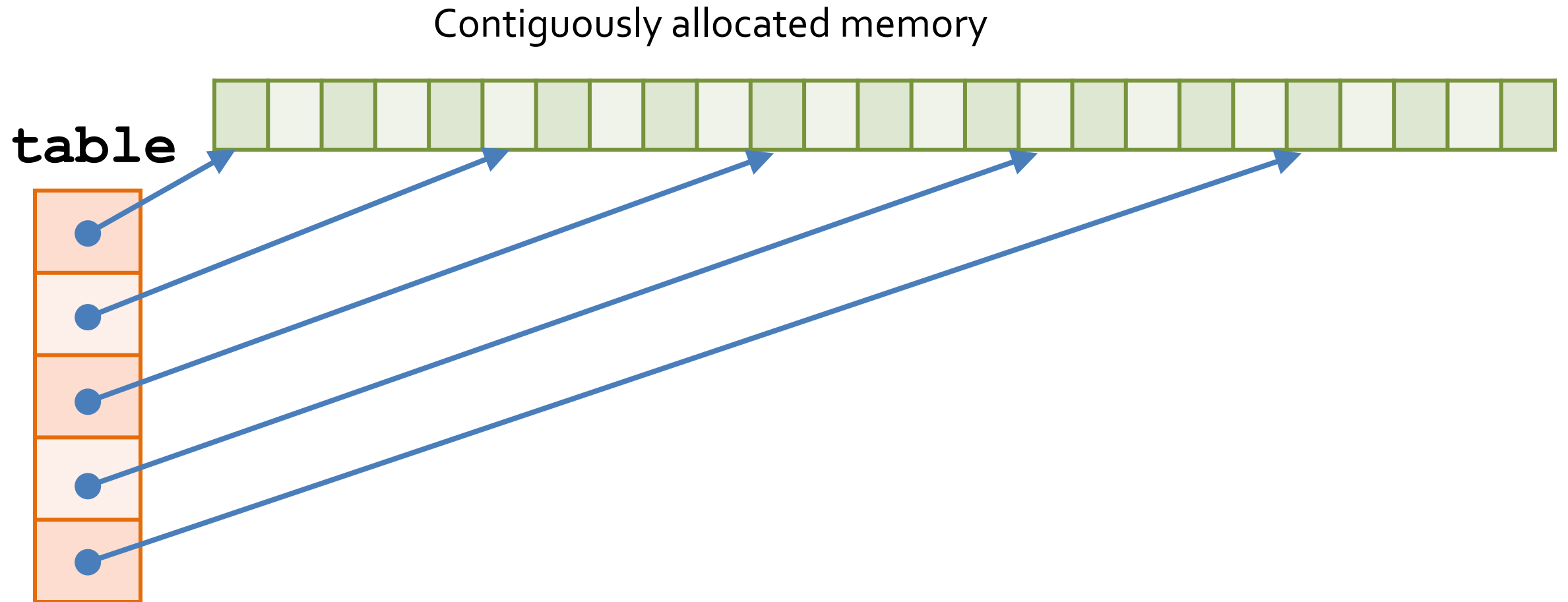
```
int** table = (int**)malloc (sizeof(int*)*rows);  
int* data = (int*)malloc (sizeof(int)*rows*columns);  
int i = 0;
```

```
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

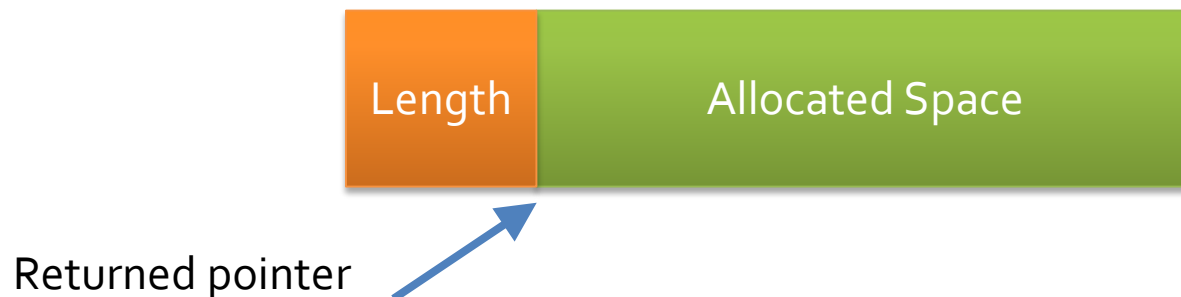
```
free (table[0]) ;  
free (table) ;
```

Rules for random numbers

- Include the following headers:
 - `stdlib.h`
 - `time.h`
- Use `rand() % n` to get `int` 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

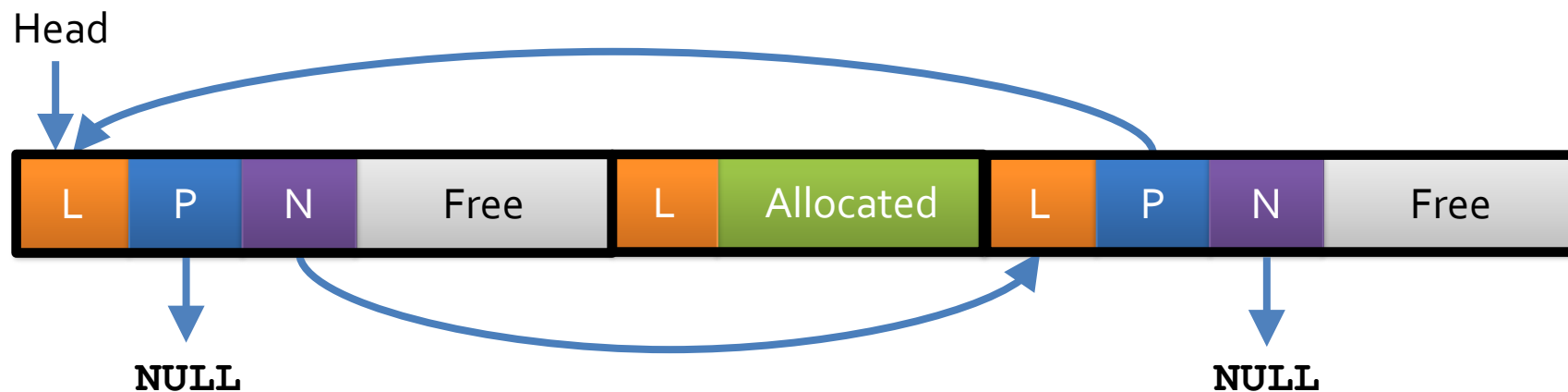
How does `malloc()` work?

- `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



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

- The portable way to convert an integer (or other numerical types) to a string to use **sprintf()**
 - It's like **printf()** except that it prints things to a string buffer instead of the screen

```
char value[12]; // Has to be big enough
int x = 3047;
sprintf( value, "%d", x );
```

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 (which I do not encourage):

```
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;
```

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 linked list node struct

- We can 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;
```

Example BST node struct

- We can use this definition for our node for binary search trees

```
typedef struct _Tree
{
    int data;
    struct _Tree* left;
    struct _Tree* right;
} Tree;
```

- Somewhere, we will have the following variable to hold the root of the tree

```
Tree* root = NULL;
```

Unions

- What if you wanted a data type that could hold any of three different things
- Back in the day when space was important, people wanted such things
- That's why they created unions, which look like structs but only have enough room for the largest thing inside of them
- They're only designed to store one thing at a time

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

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);
```

Upcoming

Next time...

- Exam 2!

Reminders

- Finish Project 4
 - **Due Monday before midnight!**
- Review for Exam 2