CMSC 15200

Oliver Fong

Autumn 2022

Contents

1 Introduction

I'm a student in the class of 2025 in CMSC 15200. These are my notes that I took in Professor Wachs's course. I added a little fun and such to my notes, but hopefully that doesn't distract from the content present. I wrote all these in Vim using Latex. The main package I use is listings in Latex, and the most important plugin I use in Vim is Ulti-Snips which is mainly for boilerplate .tex file headings and to begin environments.

1.1 Missing Content

I missed a couple Friday classes here and there, so there might be some holes in these notes. I will try to fill them in before the final. The main days are Friday of week 3, Friday of week 6 and of week 8. I also totally spaced out during the Friday's class on week 8 which was on Stacks and Queues.

1.2 For loop

```c
unsigned long long int factorial (unsigned int n) {
    unsigned long long res = 1;
    unsigned int i = n;
    while (i > 7) {
        res *= i;
        i--;
    }
    return res;
}
```

OR a more readable version:

```c
unsigned long long int factorial (unsigned int n) {
    unsigned long long res = 1;
    unsigned int i = n;
    for (i = n; i > 1; i--) {
        res *= i;
    }
    return res;
}
```

2 Week 3

2.1 Switch

```c
void print_days_in_month(unsigned char month, unsigned short year){
    if (month == 9 || month == 3 || month == 6)
}
```

Note 1. This format is a little tedious to type out: 9, 3, 6...

Idea 1. Try 'or-ing' the numbers together instead of the conditions...

1. It's not that this way is a 'bad' way, but there is a better way.
2. It's kind of like pattern matching in typed-racket.
3. This is called 'switch'.

```c
void print_days_in_month(unsigned char month, unsigned short year){
    switch (month) {
    case 9:
    case 4:
    case 6:
    case 11:
        printf("30");
        break;
    case 2:
        if (is_leap_year(year)) {
            printf("29");
        } else {
            printf("28");
        }
        break;
    default:
        printf("31");

    }
}
```

Note 2. Notice there is no curly braces on the first tabbed-in area on printf(”30”).

1. There are different cases the conditions can ’jump’ to.
2. They are targets for the computer
3. It’s not bothered by seeing other cases that also apply to a certain return.
4. This ’cleverness’ allows us to lump together these cases for a certain return. But we do have to tell the computer to stop reading with the break;.
5. When it hits the break statement, the computer jumps to the ending brace. and skips the rest of the switch.
6. If the variable doesn’t match any of the cases, the computer will jump down to the default: and do whatever that entails. It’s similar to an ’else’ statement.
7. There needs to be a break before the default case or else the computer will run that code anyways.
8. Try to imagine the returns and instructions nested in the switch statement as flat as there is no curly braces around them. They are not really ’inside’ a

certain case or whatnot, but simply contained inside the switch statement and is placed after different cases.

Note 3. The printf function really only takes strings, so the numbers in the string are a little easier to use instead of using some percent sign and plugging that into the string.

Note 4. If there was a return statement somewhere in the switch, there would be no need for a break as the whole function (days_in_month) would immediately be returned.

1. If some number was entered in the parameter of 'month' that was outside the domain of months (e.g. 0 or 100), there would be some need for an error return.

2. Let's re-analyze the default statement...

```
default:
    if (month == 0 || month > 12) {
        fprintf(stderr, "days_in_month: invalid month\n");
        exit(1);
    }
    return 31;
```

Term 1. fprintf stands for 'file print file'. stderr is the file error, and the string is the text you want to print. Really fancy could be including "%u invalid month" or something. stderr is the file at which you want to print the error message. They create some sort of 'dummy' file that doesn't go to the disk but goes to the screen. stderr is a fake 'dummy' file that sends what you're printing to the screen instead of the file. When you print with stderr instead of stdout, it is labelled as an error message. There are some advantages to using the error instead of the stdout: it might be more likely to print on the screen before the program crashes. There is more urgency with stderr. The user is able to configure the error message as different things, such as making the message bright and terrifying red. Just in case someone wants to. tldr: just use stderr when you want an error message.

Term 2. exit(1); just ends the program. Since there is no valid month that is used, there's no point in running the program anymore... so just stop running it with exit(1); Just stop. 0 is the indication of success while 1 is the error return.

2.2 Loops

1. We saw that factorial uses some type of induction, or recursion. This tends not to be done in typed-racket or functional programming language as much.
2. The culture of C is not so much in line with lab 2's method of writing factorial...
3. With loops, it is more that when you finish it for the first time, you go back and do it again from the top.

```c
unsigned long long int factorial(unsigned int n) {
    unsigned long long int res = 1;
    while (n > 0) {
        res = res * n;
        n = n - 1;
    }
    return res;
}
```

Note 5. Iterations: 1) n = 4, res = 1 . 2) n = 3, res = 4. 3) n = 2 res = 12. 4) n = 1, res = 24. 5) n = 0. Return 24.

2.3 More while loops

Time to critique some of this code...

1. One thing he doesn't like is this (n ¿ 0) condition. Kind of a waste because res = res * (n = 1) doesn't affect res at all. Change to (n ¿ 1).
2. Imagine the case when the initial condition isn't true (such as n !¿ 1). This notion of the 'base case' doesn't really exist in this as it's not recursive. In the case of n = 1 or 0, the function returns 1 which is right.
3. 'just to reiterate' - 'there's probably a joke in there.'
4. Change 'res = res * n' to 'res *= n'
5. Change 'n = n - 1' to 'n -= 1' to 'n−−'. Note: 'n−−' does not equal 'n - 1'

6. Change the two lines of 'res *= n;
   n n−−;' to 'res *= n−−;'

Note 6. You don't necessarily have to write code this condensed if you don't want to, but you have to be able to read code like this from other programmers.

Note 7. '−−n' (prefix) will first decrement n and then multiply res by n. Same as 'n = n - 1;
n res *= n'. Improved function...

```
unsigned long long int factorial(unsigned int n) {
    unsigned long long int res = 1;
    while (n > 1) {
        res *=  n--;
    }
    return res;
}
```

Example 1. 'res *= n = n - 2' - totally fine syntax.

On the topic of infinite loops:

1. What to do if your computer gets stuck in an infinite loop that doesn't stop? It doesn't crash as it just keeps running.
2. "ctrl + c" in the terminal will immediately stop a running program.

2.4 Do... While Loop

Idea 2. Have a calculator that stops if the user presses Q.

```
char operator;
while (operator != 'Q'){
    /* Do the operation */;
    operator = get.key();
}
```

Question 1. Well what are the problems with this program?

1. What is operator when the loop starts?? Should not be using the operator uninitialized.

2. There are solutions, but we're caught in an inelegant solution as the while loop asks for the condition at the beginning...

Do this instead:

```
char operator;
do {
    operator = get.key();
    if (operator == '+') {
        /* ... */
    }
} while (operator == 'Q');
```

This allows the loop to run at least once before the while condition is checked.

Note 8. At the beginning, the operator is uninitialized and is just some gibberish stored in memory.

3 Week 4

3.1 Ampersand gives the address of a variable in memory!

```
int a = 7, int b = 7, int c = 13;
int *ap = &a, *bp = &b, *cp = &c;
```

1. Pointers essentially store the location of data of different values.
2. For example, if "int a" was stored at location 123, "ap" would be equal to 123.

```
printf("\%d \%d \%d\\n", a, b, c); /* 7 7 13 */
printf("\%d \%d \%d\\n", ap, bp, cp); /* 123 456 789 */
```

1. a == b? Yes
2. b == c? No.
3. ap == bp? No.

- ”ap” stores 123, and ”bp” stores 456.

4. *ap == *bp? Yes.
   - Essentially the same question as a == b?
   - The asterisk informs the value stored, and not the storage location.
   - Using the star value is called dereferencing the value.

1. If you want ”bp” to instead point to ”c”, you could either do:
   - bp = &c;
   - bp = cp;
2. You can change the value through dereferencing as well, such as: *ap = 1;

3.2 Use pointers to change variables in other functions!

```c
void vector_add(double x1, double y1, double x2, double y2, double
    xres, double yres) {
    xres = x1 + x2;
    yres = y1 + y2;
}
```

This function is not correct because of the way parameters work. It will not have the intended effect.
Example:

```c
void foo() {
    double x, y;
    vector_add(12, 20, 1, 2, x, y);
}
```

Note 9. Each function has it’s own independent variables and parameters.

1. There’s space in memory for the function vector_add, and space in memory for foo. They are not the same spaces in memory.
2. The local variables for each respective functions are stored within that space allocated for the function.

3. Parameters are essentially local variables that are just initiated to the value that they're passed.

4. These spaces are called stack frames.

When you call vector_add with the parameters x and y, the local variables inside the vector_add stack frames for parameters "xres" and "yres" will store the variable location (the pointer essentially).

When the vector_add function overwrites the local variable values for xres and yres, they will not change the local variables x and y in the foo stack.

```
void foo(){
    double x, y;
    vector_add(10, 20, 1, 2, &x, &y);
```

Note 10. Now the local variable xres and yres in the vector stack will be pointers pointing towards x and y in the foo stack.

```
    *xres = x1 + x2;
    *yres = y1 + y2;
```

Note 11. Used the ampersand to pass the address of "x" and "y", and used the asterisk to change the value of the variables in the foo stack.

Note 12. Topic of next class: lists!

3.3 Lists and Arrays! (Finally)

Note 13. Arrays are one of two ways to implement a list. Can't grow or shrink. The other method is less efficient in some ways and is more complicated.

1. If the array is storing 4 elements and each element is 8 bytes, then the array is logically 32 bytes.

2. The "addresses" of the numbers in the array (in memory) would be sequential. For example, if the first element is stored at location 1000, then element two would be at 1008, and then 1016 for the third element and so forth. If you were looking for a later spot in the array, you might be able to track them through address location in memory.

3. In other words, pointers will be how we work with arrays. How we keep track of where they are and how we will parse through them.

4. To access a certain element of the array, you don't have to go through the whole array to get to a certain element. You can just skip over them straight to the later element. Whether the fourth or billionth element, there will be no efficiency difference, only memory.

    - We just can't grow it or shrink it.

How do we keep track of an array?

1. We will be using a pointer to the first element

2. The first element in the array is technically called the (zero-ith element).

3. The exact reason for this is that if the zero-ith element is in address 1000, then the "first" element (the one right after) would be in $1000 + 8 \cdot 1$, and the "second" would be at $1000 + 8 \cdot 2$ etc.

```
double* a;
```

This is just a pointer to the first element. How do we know it's a pointer to a whole array than just the first double? How do we tell the difference? You can't. It's just context. There's no specific type for the pointer other than the first element's type.

Note 14. We don't really use ampersands for arrays because there's already built in infrastructure for that in arrays.

Pointers store the address.

Example: (Note: the parentheses are just for note-taking the address)

```
[1.23 (1000), 4.56 (1008), 3.14 (1016), 0.68 (1024)]
double* a = 1000;
/* some syntax:
*a -> 1.23
a -> 1000
*(a + 1) -> 4.56 */
```

```c
void foo() {
    /* Stores 4 elements of type: double */
    double my_array[4];
    /* Sets the first element to 1.23 */
    my_array[0] = 1.23;
    printf("%lf", my_array[0]);
}
```

Note 15. Valid indexes for this array would be : 0, 1, 2, 3. Not 4! If you try to access spots outside the array, the computer will try and read the memory which would be sequentially following the array (1032 or something) and will process whatever gibberish is there. Or, if you try and write something to that memory, it would override whatever is there. Try not to do this.

```c
double my_arra[] = {1.23, 4.56, 3.14, 0.68}
```

After C creates the space in memory for the array when you create the array, it totally forgets what the array is. I guess.

Note 16. There is no built in function to track the length of an array, it really doesn't know. You have to keep track of the length and the storage.

Mapping, folding, filtering, are all things we are going to cover.

Example:

```c
/* a is an array of unspecified length. */
/* Should really pass in the length of the array.*/
double sum(double a[], unsigned int alen) {
    double res = 0;
    /* Convention in C is to i,j,k for loop counters... */
    unsigned int i;
    /* i < alen - Important to be strict here because the length is
   one larger than the valid last index. */
    for (i = 0; i < alen; i++) {
        res += a[i];
    }
    return res;
}
```

```c
void boo() {
    /* When I create this array, C sets aside some memory for the
   values */
    double my_a[] = {1,2,3,4};
    double asum = sum(my_a, 4);
}
```

4 Week 5

Note 17. Have to pass in a ”out” parameter for the length of the array - because functions can only return one value.

```c
int* evens(int* a, unsigned int alen, unsigned int* reslen) {
    unsigned int n = 0, i, j = 0;
    for (i = 0; i < alen; i++) {
        if (a[i] % 2 == 0) {
            n++;
        }
    }
    int* res = (int*)malloc(n * sizeof(int));
    if (res == null) {
        fprintf(stderr, "...");
    }
```

Note 18. Need multiple values! one to work its way through a, and another to work its way through res.

```c
    for (i = 0; i < alen; i++) {
        if (a[i] % 2 == 0) {
            res[j] = a[i];
            j++;
        }
        *reslen = n;
        return res;
    }
}
```

Improper to have some type of for loop for both the i and the j, it’s good to only have j increment when the certain condition is satisfied...

Note 19. Cooler way to do the same thing:

```c
res[j++] = a[i];
```

```c
void foo() {
    int nums[] = {1, 2, 3, 4};
    unisgned int elen;
    int* e = evens(nums, 4, &elen);
    for (unsigned int i = 0; i < elen; i++) {
        printf("%d", e[i]);
    }
    free(e);
}
```

Other topics:

4.1 Sentinels

Keeping track of lengths of lists. In C, you cannot ask it what the length of the list is. One solution to this is to have an extra parameter in your function as an ”out” parameter. There is another way! You still have to do it, C doesn’t do it for you. It’s called a SENTINEL. (ooooh).

It’s a value in an array to denote the last number of the array. For example, in a list of student ID numbers, there is no negative value. You might be able to put in a negative value as the last value and when the computer found a negative it would know it has ended. You do have to use more memory to store the value. Another problem is that there is no one set value that sentinels are. For example, in an array of ints, which int are you never going to use? Who knows. Generally speaking, for all possible lists of integers, there are no safe values. It’s a difficult question. There is a special type of array where you can use a sentinel and have a universal sentinel value)

This is an array of characters. A string is just an array of characters right? There are 256 different values for characters, but it turns out they didn’t need all 256 different symbols and letters for our entire system.

The flavour of working with strings is similar to working with arrays. String lengths are based on hitting the sentinel...

4.2 Char and Arrays and Strings

We use single quotes to designate a single character -

```
"hello" //String - array
'P' //Character
"P" //String - array
```

The length of the string array ”hello” is not actually 5, but 6 as the terminal character is there too. This ”end” character is called the *NULL terminator*.

```
"hello" -> {'h', 'e', 'l', 'l', 'o', '\0'}
"hello" -> {'h', 'e', 'l', 'l', 'o', 0} \\Also works
```

If you're creating a string without using double quotes, if you're going some other way, you need to be careful to allocate enough memory to the proper size of the quote, and you need to be careful to put in the null terminator at the end.

Behind the scenes in storing a character, each character is actually some number from 0 - 255 where 0 is actually the null terminator. For example, 'h' might be 104 or something.

```
void foo() {
    char* s1 = "hello";
    /* Strings created this way are actually read-only */
    char s2[] = {'h', 'i', 0};
    char* s3 = (char*)malloc(6 * sizeof(char));
    s2 = "bye"; // DOESN'T WORK
    s3[0] = 'h';
    s3[1] = 'i';
    s3[2] = '\0';
```

When you use "=" with arrays, you're creating a pointer to the array and not a separate whole new array. It is fine sometimes, but there are implications.

In some versions of C, you can declare array lengths with variables, but in others you can only include hard coded numbers.

```
unsigned int strlen(char* s) {
    unsigned int i = 0;
    while (s[i] != '\0') {
        i++;
    }
    return i;
}
```

Little Trick!

```
while (s[i] != '\0') ~~ (s[i] != 0) ~~ (s[i])
```

Second verison of strlen() - travelling pointer

```
unsigned int strlen(char* s) {
    char* t = s;
    while (*t != '\0') {
```

```c
        t++;
    }
    return t - s;
}
```

Little shorthand for this now:

```c
while (*t != '\0') { t++; } ~ while (*t++) {}
/* But *t++ will iterate one too many times, so we have to
   compensate in the return statement such that it is : */
return t - s - 1;
```

4.3 Copying Strings, and String Functions

Copying strings: why can't I just say s2 = s1? It copies the pointer.

```c
void strcpy(char* dest, char* source) { //Bad idea!
    unsigned int i;
    for (i = 0; i < strlen(source); i++) {
        dest[i] = source[i];
    }
}
```

This would be an example of a bad idea. You will call strlen() over and over again which will iterate over the string over and over, which would ultimately take a long time. Next step would be to store the length as a variable so strlen() only runs once. But, this isn't typically the way we write this.

Maybe a better idea:

```c
void strcpy(char* dest, char* source) {
    unsigned int i;
    while (source[i] != '\0') {
        dest[i] = source[i];
        i++;
    }
}
```

Warning! Is the terminating character 0 (the sentinel) copied into dest? No. We terminated the loop as source[i] hit 0 so it didn't copy into dest.

Again, we can use that little shorthand to make the loop less legible...

```c
void strcpy(char* dest, char* source) {
    unsigned int i;
    while (source[i]) {
        dest[i] = source[i];
        i++;
    }
    dest[i] = '\0' //The final task for the loop, after the loop
   finished.
}
```

strlen() is a library function in C. A similar library function...

```c
char* strdup(char* src);
```

But Ooooooh, let's look at the travelling pointer version of strcpy():

```c
void strcpy(char* dest, char* source) {
    while (*source) {
        *dest = *source;
        source++;
        dest++;
    }
    *dest = '\0'
}
```

Some funky code you can write:

```c
i = n = n - 2;
```

Initially would assign n = n - 2, and then assign i to the new n. Reads right to left in these assignment operations. Using this logic, we can shorten the code a little more.

```c
void strcpy(char* dest, char* source) {
    while (*dest = *source) {
        source++;
        dest++;
    }
}
```

This version will also copy the terminating character into dest inside the loop.

Again, you don't need to write in this manner, but you need to be able to understand when other people write code like this.

```c
void strcpy(char* dest, char* source) {
    while (*dest++ = *source++) {
    }
}//Need the curly braces still to tell C that the loop is empty
   inside, and so it won't interpret the next line it sees as part
   of the loop
```

How the computer reads this line: Looks at everything at the right-hand side first in the assignment (*src++), it goes to course and it retrieves the value src is pointing to because of the star, because of the ++ it will say, I'll increment src later, then it will look at dest and write the value at the place dest is pointing to (because of the star) and then it looks at the ++ and says it will do that at the end, and then it will increment both dest and src and then it will look at t he while loop and it will check the condition and if it's the null terminating character it will stop.

Something else - try and feel the overall flow of this. An example of some function:

```
char* emphasize(char* s) {
    char* res = (char*)malloc((strlen(s) + 1) * sizeof(char)); //The
    + 1 is for the terminating character
    unsigned int = 0;
    while (s[i]) {
        if (s[i] = '.') {
            res[i] = '!';
        } else {
            res[i] = s[i];
        }
        i++;
    }
    res[i] = '\0';
    return res;
}
```

5 Week 6

Note 20. Recap: Last week we talked about sentinels and the null terminator in strings. We talked a lot about travelling pointer function, and tracking through arrays like that. Allocation of memory to arrays. strcopy(). strlen(). All built-in functions.

Let's look at another string function today: strcmp() (string compare). This function compares all symbols and characters at least in the ASCI listings...

Note 21. So all characters are differentiated. There's an overall scheme: numbers come before letters, they go in order, space comes before numbers, uppercase A comes before lowercase a, Z comes before a.

```c
int strcmp(char* a, char* b) {
    /* strcmp() returns:
       a < b -> negative
       a == b -> 0
       a > b -> positive */
}
```

When you want to return more than one value, you can use out-parameters to alter variables out of function scope...

5.1 Structs

We've used structs and such in typed-racket, and you might've seen this as objects in python... The header file is kind of the glue that ties the evidence and hw.c files together, similarly, we put the structs in the header files and this will let C know where and how we're using them. A point has an x value and a y value.

In racket, we often used structs as a way to return more than one value from a function by burying more variables inside a larger struct that contains multiple variables under a single overarching name.

```c
struct point {
    double x; // Can also write as double x, y;
    double y; // These are called "fields" - The stuff clarified in
   the struct.
};
```

These fields don't have to be the same type, and there can be as many or as little as you so desire.

```c
void foo() {
    struct point p = {1.0, 2.0};
    p = {3.0, 4.0} // Wrong!
    p.x = 3.1; // Correct!
    printf("%lf\n", p.y);
    struct point q;
    q.x = 10;
    q.y = 12;
    double d = distance(p, q); // Imagine this is below the following functions in the program...
    struct point r = midpoint(p, q);
}
```

Let's write a distance function that returns the distance, and another that returns a point halfway between two others.

```c
double distance(struct point p1, struct point p2) {
    double dx = p1.x - p2.x;
    double dy = p1.y - p2.y; // Wachs like the SYMMETRY of this code and how it LINES UP BABY! Do it like Wachs does it and make it pretty.
    return sqrt(dx * dx + dy * dy);
}

struct point midpoint(struct point p1, struct point p2) {
    struct point res = {(p1.x + p2.x) / 2, (p1.y + p2.y) / 2}; // You could also just say res.x = and res.y equals separately, but this is valid too.
    return res; // If you were thinking you could remove one line and just return the curly braces instead of having to put it in a variable, you were wrong. That's not valid C syntax! C doesn't know what the type of the curly braces even is, so you need to clarify that.
}
```

5.2 Struct Memory Allocation

So why don't we have to manually allocate memory for this data type? With larger data types, we should allocate some memory to store the data (aka using malloc for arrays). It seems that with structs (e.g. midpoint that returns two variables) they return more information than something like a long or int or char with those amounts of bytes. Machinery for arrays and structs are different.

Note 22. When you pass in a struct as a parameters, it makes a copy of that struct in local memory. They work similar to something like ints or floats or doubles when they are passed in as a parameter. There are no pointers here, when we're using structs.

It's weird because we were stressing using pointers and such with larger data types. Imagine structs with something like a dozen fields so copying that information into another place in memory might not be efficient. If you don't want to copy a whole big struct, then you can choose to use a pointer to the struct instead of copying it into local memory. For arrays, C forces you to use pointers and doesn't give you an option to copy it into local memory.

5.3 Pointers with Structs

There's some extra syntax here we should be aware of.

Let's see with, say for example, midpoint() if we wanted to use pointers instead.

```c
struct point* midpt(struct point p1, struct point p2) {
    struct point* res = (struct point*)malloc(sizeof(struct point));
    // You might think that because struct point is two doubles, and
    doubles are something like 8 bytes, maybe you could do 2 *
   sizeof(double). That would be clever but stupid. Don't do that
   because it's not always right. Different hardware has different
   rules and it might not follow exactly the size of that.
    \\ Res is a pointer to a struct now, so we have to dereference
   it first.
    (*res).x = (p1.x + p2.x) / 2;
    res -> y = (p1.y + p2.y) / 2; // Syntax that is almost
   universally used.
    return res;
}
```

5.4 Structs with Pointers: Syntax Fun

Let's talk a little more about structs! Something something about a cooking show and stuff being up on the board already.

```c
// Some helper functions that might be useful.
struct point* point_new(double x, double y) {
    struct point* res = (struct point*)malloc(sizeof(struct point));
    // Does res == NULL?
    res -> x = x;
    res -> y = y;
    return res;
```

```c

void point_free(struct point* p) {
    free(p);
}
void point_show(struct point* p) {
    printf("(%lf, %lf)\n", p->x, p->y);
}

// If you have a pointer to a struct.
struct point* point_copy(struct point* p) {
    return point_now(p->x, p->y);
}
```

You can't ask C to find the fields of the struct and do something like loop over all the different fields of the struct. Can only access fields by specifying a field of the struct.

```c
// Let's declare some structs:
struct point {
    double x, y;
};

// Declaring a field inside a struct as another struct.
struct circle {
    struct point center; // No star, no pointer.
    double radius;
};

// Accessing the nested structs:
void foo() {
    struct circle c; // Right now just a struct. No pointer.
    c.radius = 10; // Should use the dot because there's no pointer.
    c.center.x = 1;
    c.center.y = 3;
}

void foo2() {
    // Sometimes initializing structs this way will be too complex.
    struct circle c = {{1, 3}, 10};
    // A little more overhead to create an excess struct for this
   circle initialization.
    struct point p = {1, 2};
    c.center = p;
}

// Suppose there were pointers now...
// How to use struct syntax with pointers.
void foo3() {
    struct circle* c = (struct circle*)malloc(sizeof(struct circle))
    ;
    c->radius = 10;
    c->center.x = 1;
    c->center.y = 2;
}
```

Imagine for a slightly different declaration of the circle struct:

```c
struct circle {
    struct point* center;
    double radius;
}

void foo() {
    struct circle* c = (struct circle*)malloc(sizeof(struct circle);
    c->radius = 10;
    c->center->x = 1; // TWO ARROWS YO
    (*(*c).center).y = 2; // Same thing as the above line with
    different syntax
}
```

The final permutation in the pointer-struct combo:

```c
struct circle {
    // Two field struct:
    struct point* center; // This means that it is storing the
    address of the point, and not the point itself.
    double radius;
}

void foo() {
    struct circle c;
    c.radius = 10;
    c.center->x = 1; // ONE ARROW
    c.center->y = 2;
}
```

Let's talk about related stuff!

ARRAYS OF STRUCTS. Maybe we have a bunch of points and we want to make a graph of them. We'd do something like this:

```c
void foo() {
    struct point points[] = {{1, 2}, {3, 4}, {5, 6}};
    points[1].x = 10;
    // Uses malloc to create memory for an array for 3 structs.
    struct point* points2 = (struct point*)malloc(3 * sizeof(struct
   point));
    points2[0].y = 12;
}
```

GOOD DAY!

6 Week 7

6.1 typedef

typedef is a way to rename a data type for future use in the code. If it's in the header file, it's everywhere that the header file accesses. Sometimes for naming convention there might be an underscore _s for structs or and underscore _t for type, but this isn't really a universal preference.

```c
struct measurement {
    ~~~~~~
};

typedef struct measurement measurement;

//OR (to declare inline)

typedef struct measurement {
    ~~~~~~~~
} measurement;
```

The last word on this line of code is the new name, and the "struct measurement" is the old name. It's just a re-naming, or creating a synonym.

Let's say you don't like how c uses "int" for data declarations. You want to type something like integer.

```c
typedef int integer;
```

6.2 Matrix

What's the difference between what we're doing with concatenating strings and matrixes? Matrixes are more a mathematical idea where every row has the same length, and it's very structured and such.

Note 23. Two different ways to make a matrix in C. They are not compatible with each other.

Note 24. 1st method (Really just a regular array with fancy syntax making it appear 2D): Let's say I want to make the matrix: 1,2,3, 4,5,6, 7,8,9.

```c
// 1D array:
int l[] = {1,2,3,};
// 2D array (matrix):
int m[][] = {{1,2,3},
             {4,5,6},
             {7,8,9}};
// On different lines for legibility, C doesn't really care.
int m2[3][5];
// Would create a 3 row, 5 column matrix. 3x5.
m2[0][2] = 7; //Initialize This 0 row 2nd column value.

// mtx parameter needs 3 columns, but can have n amount of rows.
void print_matrix(int mtx[][3], unsigned int nrows) {
    unsigned int i, j;
    for (i = 0; i < nrows; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d ", mtx[i][j]);
        }
    }
}
```

The way matrixes are stored in memory is really just a single line array.
1 2 3
4 5 6
7 8 9
Is really just stored as: 1 2 3 4 5 6 7 8 9. When you want to access row 2, you want to skip row 1 and row 0. Computer will look at the number of columns and find that in the matrix there are 3 elements per row (3 columns) and will need to skip 6 elements ahead in total in the flat inversion to find the 2nd row. The "formula" for finding elements in code would be as such:

$columns * row + c$

This is essentially why you need to hardcode the amount of columns in memory.

This whole thing can be imagined as just a single array for which syntax makes it look 2D. In reality, it's not structured differently than regular arrays.

Note 25. 2ND method: Array of arrays.

```c
int** m3 = (int**)malloc(3 * sizeof(int*)); // 3 -> rows
unsigned int i;
for (i = 0; i < 3; i++) {
    m3[i] = (int*)malloc(5 * sizeof(int)); // 5 -> columns
}
m3[0][2] = 0;
```

Visualization of how the data looks where the black square is initialized to zero. The arrows are pointer visualization.
□ → □□■□□
□ → □□□□□
□ → □□□□□

NEW TOPIC!!!!

6.3 Trees

Hierarchical data structure. An example of a tree data structure is our computer's directory!

In a kind of layout where there's parents, and there are children, and the children have children and so on so forth.

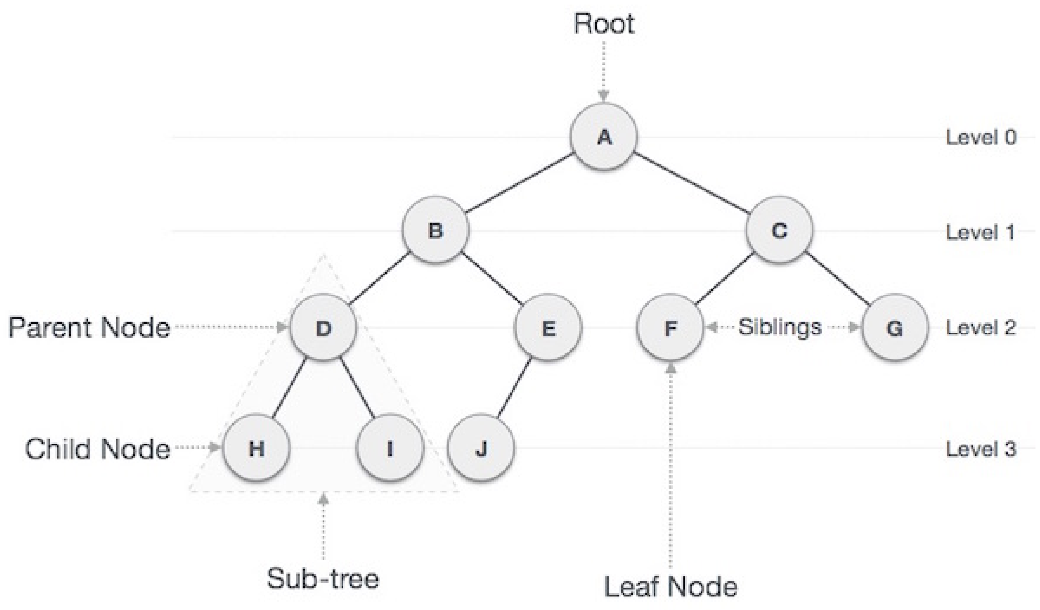
As a memory structure, there is a main pointer that stores the information on the parent node. The parent node itself will be represented as a struct, and will store pointer information (addresses) to the children of that node.

Let's design a struct for the structure of a binary tree: a tree where each node has at most two children.

```c
// For the storage of ints only.
// No polymorphism (ability to store different types willy-nilly)
   for this.
struct int_tree {
    int val; // Value of that certain node.
    struct int_tree* left; // Want a pointer to another node of the
   same struct.
};
// There's a problem with the code above: using the struct
   definition inside the struct defintion: there's a circular
   dependency.

// There's a get around that basically involves tricking the
   compiler
typedef struct int_tree int_tree;

struct int_tree {
    int val;
    int_tree *left, *right; // Works
};
```

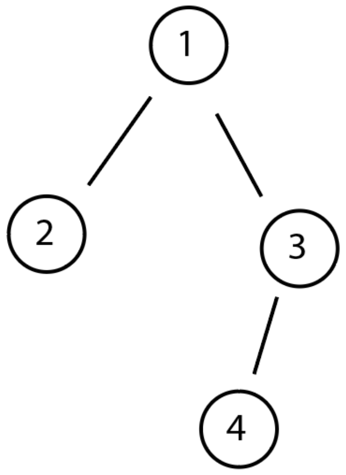


NULL is essentially the equivalent of none in Python. Null pointers are ones that don't point to anything. Note that Nodes H, I, J, F, G, and E all contain null pointers.

Let's write some functions with these "trees"

```
int_tree* make_node(int val, int_tree* left, int_tree* right) {
    int_tree* res = (int_tree*)malloc(sizeof(int_tree));
    res -> val = val;
    res -> left = left;
    res -> right = right;
    return res;
}
```

int_tree* t:



```
void foo() {
    int_tree* t = make_node(1,
                            make_node(2, NULL, NULL),
                            make_node(3, make_node(4, NULL, NULL),
   NULL));
}
```

Note 26. Trees are a naturally recursive data structure.

Let's look at some function that finds if a certain value is within the tree.

```
int member(int_tree* t, int val) {
    // 2 base cases:
    // The node has a NULL child
    if (t == NULL)
        return 0;
    // If the node has the value you're looking for
```

```c
    if (t - > val == val)
        return 1;
    return member(t -> left , val) || member(t -> right , val);

}
```

QUICKLY TALK ABOUT ANOTHER TOPIC IN THE LAB:

It turns out that a useful type of tree is one of which we put an artificial hierarchy on the data. Let's say we're storing name and phone number. How long will this take? We'd have no pattern or method of searching if the tree is unorganized.

6.4 Binary Search Tree (BST)

Everything to the left of a node has to be less than it, and everything to the right of it has to be greater than it. Strictly. This is consistent with every single node.

(Trying this depiction of a tree out)

```
// Example BST:
        7
    6
        5
4
        3
    2
        1
```

6.5 Linked List

This whole idea of not being able to expand arrays to something larger than it is already declared and initialized to in memory is frustrating. We can't rely on there not being anything in the memory after the array.

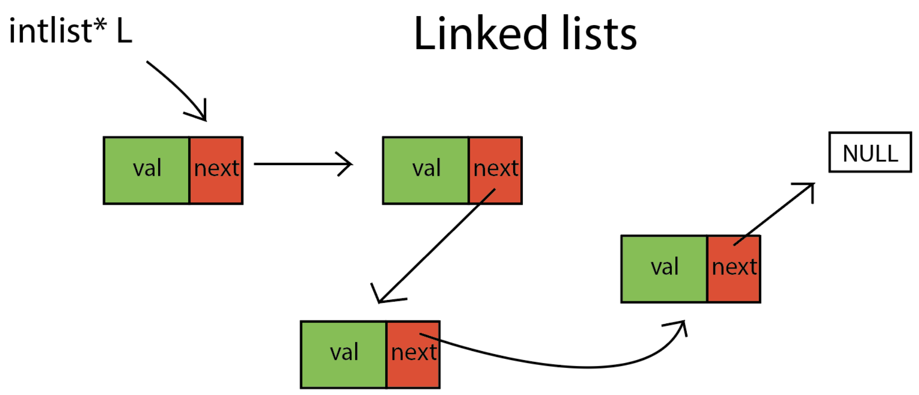
Idea 3. Let's create an array of a struct where each struct stores a value and a pointer towards another struct of the same type. This struct will essentially create a type of 'array' where each value is in some random place in memory but they're all linked through a pointer chain.

This is called a: *Linked List.*

The problem with these is that if you want to access the thousandth value in the list, you have to go through every single value before that to reach that 1000th index. But, they can grow and shrink.

Note 27. No free lunch for python either where it seems that they have the best of both worlds. Because behind the scenes there are tradeoffs that they have to make.

```c
typedef struct intlist intlist;
struct intlist {
    int val;
    intlist* next;
};
```



everyone loves the cons() function from 15100 - RIGHT?

```c
// Some helper functions:
intlist* cons(int val, intlist* rest) {
    intlist* res = (intlist*)malloc(sizeof(intlist));
    res->val = val;
    res->rest = rest;
    return res;
}
int is_empty(intlist* l) {
    return l == NULL;
}
```

```c
int first(intlist* l) {
    if (is_empty(l)) {
        fprintf(stderr, "first() error: given empty list");
        exit(1);
    } else {
        return l->val;
    }
}
}

void foo() {
    intlist* l = cons(7, cons(13, cons(25, cons(-2, NULL))));
}
```

Note 28. NULL is always the place in memory: 0, and some bytes past that. Computers will always reserve the first couple, ten, hundred bytes to be inaccessible for legit allocation.

Since NULL is always stored in the address 0, it is interpreted as false as a boolean. Everything else would be interpreted as true.

```c
int sum(instlist* l) {
    if (is_empty(l)) {
        return 0;
    }
    return first(l) + sum(rest(l));
}

intlist* squares_copy(instlist* l) {
    if (is_empty(l)) {
        return NULL;
    }
    return cons(first(l) * first(l), squares_copy(rest(l)));
}
```

Idea 4. In real world scenarios where you're outputting a product to billions of people, there might be some debug version that has more overhead but is easier to debug, but there will also be a "deployment" version that would have less overhead but less debugging code too.

7 Week 8

7.1 More Linked Lists

Let's write a function that changes the list - not creates a new list - but changes the values in the indexes.

```c
void squares_change(intlist* l) {
    if (is_empty(l)) [
        return;
    }
    l->val = l->val * l->val;
    squares_change(rest(l));
}
```

Let's now see a function that filters. We will return a new list back with only even numbers.

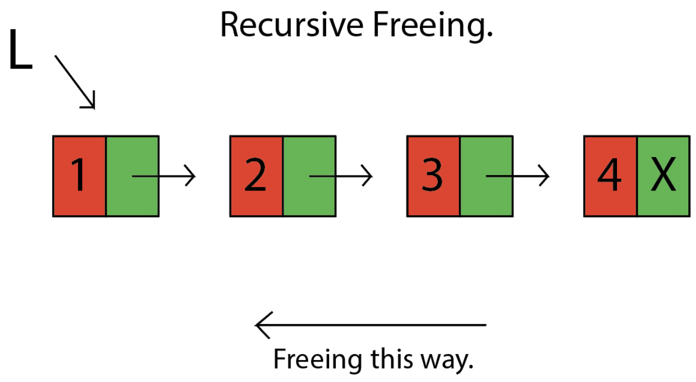
```c
intlist* evens(intlist* l) {
    if (is_empty(l)) {
        return NULL;
    }
    if (first(l) % 2 == 0) {
        // Cons does malloc for the return...
        return cons(first(l), evens(rest(l)));
    }
    // For when I have an odd number
    return evens(rest(l));
}
```

Now, how should I free this linked list? Is there an order in which I should free them? I should kind of free these things backwards. If I free the first element first, then how would I know where the second element was? On the other hand, I could probably make a copy of the first element and free it, while the copy still has the location of the second element. Two ways.

Let's try the method where we go backwards. We'll be using recursion here.

```c
void free_list(intlist* l) {
    if (is_empty(l))
        return;
    free_list(rest(l));
    // Here's how the recursion frees backwards: recursive call is
    BEFORE the action.
    free(l);
}
```

An visual depiction of the recursive free_list();



Now let's try what an "experienced" C program might do with iteration instead of recursion to sum the list.

Cases: You're either going to end up at the end of a list, or start with an empty list.

```
int sum(intlist* l) {
    int res = 0;
    while (l != NULL) {
        res += l->val;
        l = l->next;
    }
    return res;
}
```

It's important to remember that the parameter "l" is a copy of the pointer to the values in the heap, and not the pointer that is passed into the function.

Let's look at some more ways to iterate through linked lists.

```
void squares_change(intlist* l) {
    while (l != NULL) {
        l->val = l->val * l*val;
```

```c
        l = l->next;
    }
}
```

Let's free the list now with iteration. We have to be careful not to use previously freed elements of the list. This method will essentially free forwards with the direction of the linked list, but it will save a temporary copy of the element as to keep the location of the next element intact.

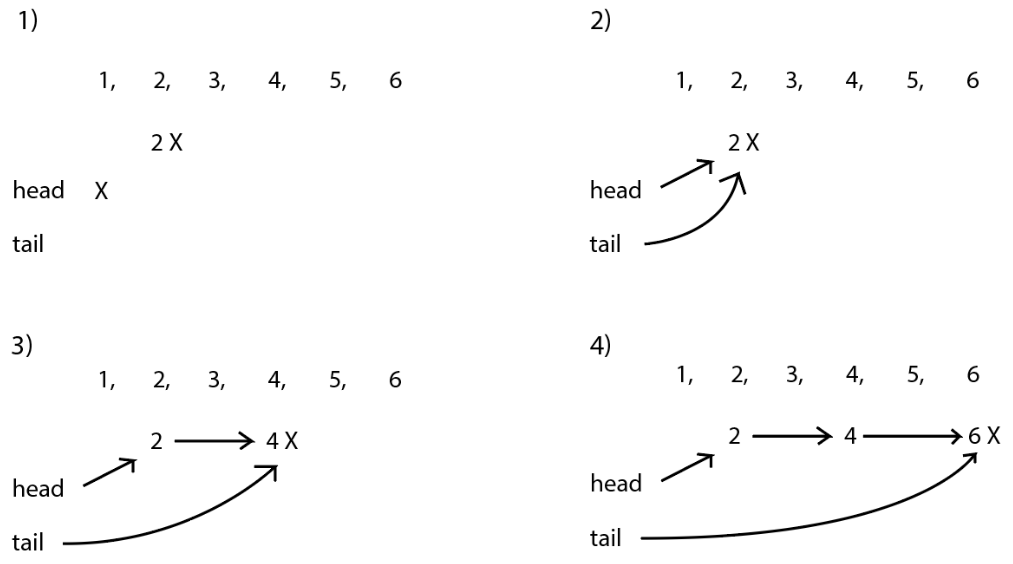
```c
void free_list(intlist* l) {
    intlist* n;
    // Will almost certainly work, but is incorrect because it
   reaches into freed elements.
    while (l != NULL) {
        free(l);
        l = l->next;
    }
}
```

Instead, do this:

```c
void free_list(intlist* l) {
    intlist* n;
    while (l != NULL) {
        n = l->next;
        free(l);
        l = n;
    }
}
```

7.2 Wednesday Linked Lists and Such

A visual depiction of the steps to be taken for evens()
where X = NULL pointer.



```
intlist* evens(intlist* l) {
    intlist *head = NULL;
    intlist *tail, *item;
    while (l != NULL) {
        if (l->val % 2 == 0) {
            item = cons(l->val, NULL);
            if (head == NULL) {
                head = item;
            } else {
                tail->next = item;
            }
            tail = item
        }
    }
    l = l->next;
    return head;
}
```

7.3 Inserting and Removing from Linked Lists

Removing from the front and end of the list is relatively simple. For the first element, it is easy because you start with it. That's where the pointer to the list is pointing. Removing from the end would essentially be looking for an element that has a null pointer. You have a travelling pointer looking at each of the elements until you find the null pointer and then you free it and make the element before have a null pointer. Removing in the middle consists of making copies of pointers and stitching and freeing stuff.

Note 29. Even experienced programmers will really just write out diagrams to figure out what to do. So do it.

7.4 Stacks on Stacks on Stacks!

Push means to add something to the stack.

Pop means to remove things from the stack.

7.5 Queueueueueues

All information has gone through my head.

7.6 HOW DO WE REPRESENT NUMBERS?

Storing data. Different bases in the number system. In computers, there are two main systems - base 2 and base 16.

We use base 10 primarily because we have 10 fingers.

Note 30. HOW MANY FINGERS DO COMPUTERS HAVE? Technically none, but really they have 2.

Computers are assemblies of logic gates essentially. Essentially, there can be two settings that some wire could have: on or off. Essentially, ON would be if there was some voltage running through it, OFF would be an absence of voltage. We have to figure out how to represent everything through these two options. This is why we use BINARY.

Let's convert 152 to binary.

$$152$$

Idea 5. We're going to divide by 2 repeatedly and see what their remainder is.

152	0	1
76	0	2
38	0	4
19	1	8
9	1	16
4	0	32
2	0	64
1	1	128

Essentially, works out to $8 + 16 + 128$

8 Addendum

This is some brief code from some of the classes I missed. The stack-heap memory video notes are from the Simplilearn video on it, and the others are from Charlotte Gilmore.

8.1 For loop

```c
unsigned long long int factorial (unsigned int n) {
    unsigned long long res = 1;
    unsigned int i = n;
    while (i > 7) {
        res *= i;
        i--;
    }
    return res;
}
```

OR a more readable version:

```c
unsigned long long int factorial (unsigned int n) {
    unsigned long long res = 1;
    unsigned int i = n;
    for (i = n; i > 1; i--) {
        res *= i;
    }
    return res;
}
```

8.2 Stack & Heap

All from the Simplilearn video: In writing a program, first the computer will expand the header files for utilization in the C functions. Then the computer will compile the code. This is checking for errors, and then the transformation into Machine level code (010010101011 etc...).

Then the code editor looks at variables and loads it into memory. The memory area is the RAM (Random Access Memory). There are three different sections of RAM: Machine Code, stack, and the heap. The Machine code block stores the machine level code in the IDE and stuff (kinda unimportant right now). The stack block stores variables that are static in nature. The heap stores variables that are dynamic in nature.

Static memory allocation first stores all global-level variables, and then the function calls and local variables. The local variables are the variables written inside the function and are only used in the function. The memory allocation for the stack doesn't change while the program is running.

The stack frame is essentially the part of the stack that is storing the function memories. If main is running, there is an area in the stack memory storing the stack frame containing all variables in main. Then if main calls another function foo(), there will be another stack frame created in the stack storing all the local variables in foo(). After foo() has finished running, that stack frame will be removed from memory. After main() finishes, main()'s stack frame will also be removed.

In imagining the stack, stack frames are placed on top of each other as the function runs such that main() will usually be on the bottom.

The heap can grow or shrink in memory size such and is called dynamic. The functions: malloc(), calloc(), realloc(), and free() all access the heap memory.

```
    int foo() {
        int *z;
        z = (int*)malloc(sizeof(int));
        z = 100;
    }

```

This function will first create a variable z containing a NULL pointer. Then the variable is initialized to a pointer pointing to the value 100 stored in the heap memory. Some code like:

```
int *p = (int*)malloc(1000000000000000 * sizeof(int));
```

This code will likely create a "heap overflow" error because the computer tried to allocate too much memory to the heap.

8.3 Unions, Enums

```
// Idea to have a struct only storing one or the other.
enum shape_tag {
    CIRCLE,
    RECTANGLE
};
struct shape {
    struct circle c;
    struct rectangle r;
```

```c
    enum shape_tag;
};
// Unions are better! Takes only one.
union shape {
    struct circle;
    struct rectangle r;
}
struct tagged_shape {
    union shape s;
    enum shape_tag tag;
};

void bar() {
    struct tagged_shape myc;
    myc.shape.c.radius = 10;
    myc.shape.c.center.x = 5;
    myc.tag = CIRCLE; // Order for this doesn't matter.
}
double shape_even(shape_tag s.tag) {
    // Switch works with enums:
    switch(s.tag) {
    case CIRCLE:
        return 2;
    }
}
```