This week is a fully practical lecture, so let’s get straight to it.

Imagine that we had a list of names that we would like to give different values to, but we don’t care too much what the values are. The easiest way to do this is to list the names in order and then give numbers to them (so the first would be 0, the second would be 1, and so on.) This is exactly what an enumeration (enum for short) does. For example:

enum direction { U, L, D, R};

This defines the direction enumeration and creates the four values U,L,D, and R. These have the values 0,1,2, and 3 respectively. Since the values of an enumeration are numbers it is possible to assign them values. For example:

enum direction {U=‘U’,L=‘L’,D=‘D’,R =‘R’};

Remember, C treats chars and ints the same. So the above gives U,L,D, and R the values 85,76,68,82 (which are the ascii codes of U,L,D and R).

Here’s a weird bit of syntax. In C we must end our enum definition with a ;
This is because the { and } are not functioning as a block (or scope), they are functioning as an initializer (e.g. int myArray[2]={0,1};)

In C defining an enum does not create a new type. In order to use an enum it is necessary to preface it with the ‘enum’ keyword. If we had already defined the direction enum we would use it as follows:

enum direction UP = U;

You’ll notice in the above example we don’t need to say direction.U
In C enums actually define values, so we can refer to U directly. In fact, we can even use U as an int value independent of the enum direction.

Another strange bit of C syntax is the typedef keyword. It allows us to define a type. First let’s typedef an enum as an example.

typedef enum {U,L,D,R} direction;

Note that now we put the name at the end. This typedef syntax will allow us to use the enum direction without the enum keyword. For example:

direction UP = U;

Of course C has other stranger things than enums, a great example of this is unions. A union is effectively a block of memory that can hold one of a number of different types. Allow me to elaborate with an example.

```
union number{
	int intVal;
	float floatVal;
};
union number myNumber;
```

Here we've declared a variable named myNumber that can be EITHER an int, or a float, but NOT BOTH. We decide which type by accessing either the int member or the float member with the member operator(.) For example, to write to the int we would use:

```
myNumber.intVal=5;
```

When we define a union C asks the OS for just enough memory to hold the largest type that it can take on. Then it we change which type we're using C simply overwrites the memory that held the old type with the new type.

Much like with enums we have to put the 'union' keyword in front any time we want to talk about a number union. Of course we can solve this with typedef as well. Here is the sample syntax:

```
typedef union {
	int intVal;
	float floatVal;
} number;
```

Unions though are very limiting. If we need to use both members then we are better suited by a struct. Here is a sample:

```
struct number{
	 int intPart;
	 float floatPart;
};
struct number myNumber;
```

When we define a struct C asks the OS for enough memory to store each one of the members sequentially. So if in our example &myNumber ==&myNumber.intPart, and &myNumber.floatPart is just four addresses down.

We can apply the typedef keyword to structs as well.

Now that we've gone over all of the structures in C that have members it is time to discuss the indirection operator. Since C passes by value we can't pass a union or a struct into a function and expect its members to change, instead we have to pass a reference. This is done in the following example:

```
void updateFloat(struct number * input, float update){
	*input.floatVal=update;
}
```

Instead of dereferencing input and then accessing it's floatVal member we can use the indirection operator to reference it's floatVal directly from the struct number reference. That would change our example to look like:

```
void updateFloat(struct number * input, float update){
	input->floatVal=update;
}
```

Enums, unions, and structs are often used together in the form of a discriminated union. A discriminated union is a struct that contains a union which stores the "data" and an enum which represents a "flag" that tells you what type of data is contained in the union. Here's what it looks like in code:

```
enum dataType {INT,FLOAT,CHAR};

union data{
	int intVal;
	float floatVal;
	char charVal;
}

typedef struct{
	enum dataType flag;
	union data dataVal;
} discriminatedData;
```

Now if we are sure to carefully update the flag whenever we update the members of our data union we can work with our union safely without sacrificing much space. (This is especially useful when you have unions of structs, for example if you were trying to emulate generics from Java.