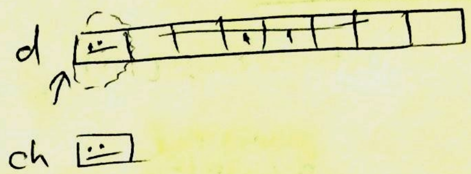
(1)

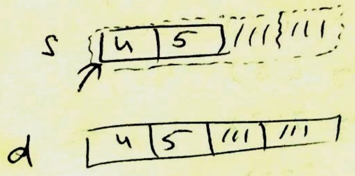
CS 107 Lecture 3

```
ex double d = 3.1416;
   char ch = *(ch *) &d;
   cout << ch << endl;
```



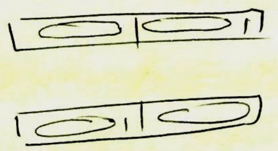
[redacted] dangerous recasting outside

```
double d = *(double *) &s;
```



big endian, little endian

1 in short



} big endian

} little endian

big: lowest byte stores the largest contribution

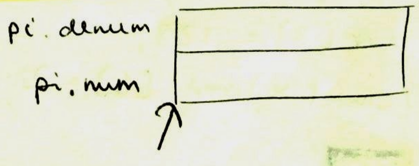
little: [redacted] bytes in the reverse order

structs:

```
struct fraction {
    int num ;
    int denum;
};
```

in memory

- two 4 byte blocks stacked on top of each other
- pointer to struct is pointer to the first field

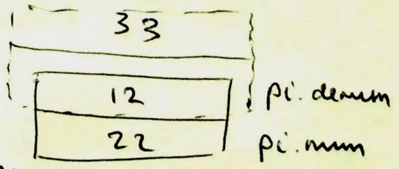


```
fraction pi ;
pi.num = 22;
```
← stores [redacted] 22 to the field that is at offset of 0 from the pointer to the struct

```
pi.denum = 7;
```
← based on definition, [redacted] stores 7 at offset of 4 bytes from the pointer to the struct

casting with struct pointer

```
fraction pi;
pi.num = 22;
pi.denum = 7;
((fraction *) & (pi.denum)) -> num = 12;
((fraction *) & (pi.denum)) -> denum = 33;
```



to access 33

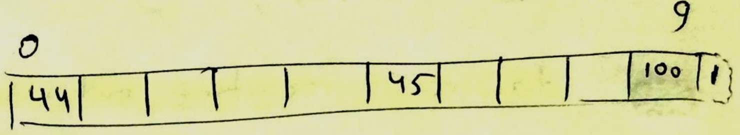
(&pi)[1].num

↖ [redacted] (fraction *) type

arrays :

$\underbrace{\text{int array}}_{\text{address of 0th entry}}\text{[10];}$

allocates 40 bytes of memory for 10 ints



array[0] = 44; $\boxed{77}$

array[9] = 100;

array[5] = 45;

array ≡ &array[0] $\boxed{25}$

array[10] = 1; ← in C/C++ no bounds checking on arrays; go 10 int quantums from &array[0] and store 1 [redacted] as int

array[25] = 25;

array[-4] = 77;

*(array - 4) = 77;

pointer arithmetic

$\underbrace{\underbrace{\text{array}}_{\text{type (int *)}} + \underbrace{\text{k}}_{\text{integer automatically scaled by type}}}_{\text{pointer arithmetic}} \equiv \text{\&array[k]}$

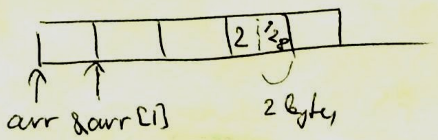
*array ≡ array[0]

*(array + k) ≡ array[k]

ex

```
int arr [5];
arr[3] = 128;
((short *) arr) [6] = 2;
```

$\boxed{100}$



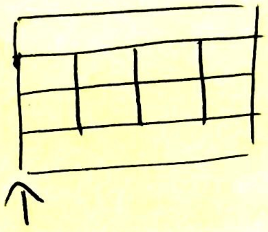
[redacted]

$\text{((short*)}\underbrace{\text{(((char*)(\&arr [1])) + 8))}}_{\text{forward from \&arr[1] by 8 chars}}\underbrace{\text{[3]}}_{\text{forward by 3 shorts}} \text{= 100;}$

CS 107 lecture 3

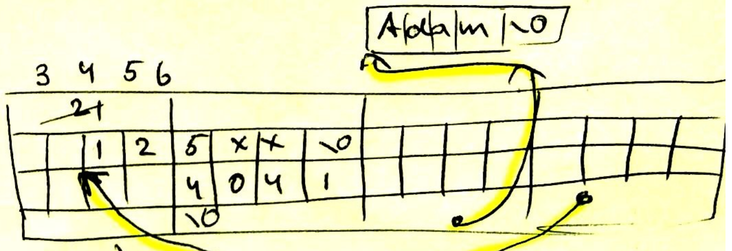
structs with arrays inside:

```
struct student {
    char * name ;
    char  suid [8];
    int   numUnits ;
};
```



```
student pupils [4];
pupils [0]. numUnits = 21;
pupils [2]. name = strdup ("Adam");
pupils [3]. name =
    pupils [0]. suid + 6 ;
```

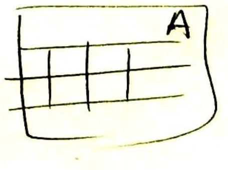
↖ dynamically allocates space for the string writes it, and returns the address of the first character



```
strcpy (pupils [1]. suid, "40415xx");
```

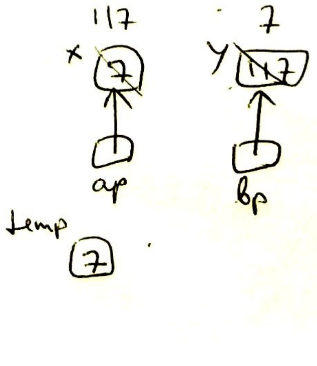
↖ no allocation, assumes the pointer to space is provided by the first argument

```
strcpy (pupils [3]. name, "123456")
pupils [7]. suid [11] = "A"
```



generics (start)

```
void swap (int * ap, int * bp)
{ int temp = * ap;
  * ap = * bp;
  * bp = temp;
}
```


117
7
x
7
y
117
ap
bp
temp
7