Pointers, Strings, Arrays

→ Pointers are adresses to memory.

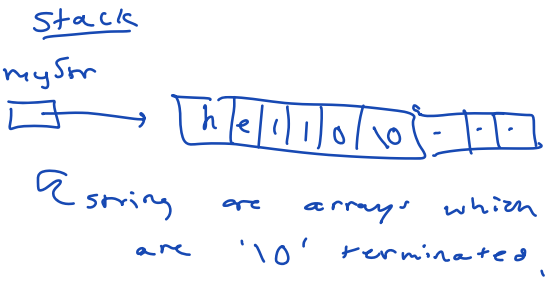
→ Pointers can be also called references.

→ Strings and ~~arrays~~ are pointers to contingous blocks of memory.

→ For example

```
char *myStr = "hello"
```

type



↖ string are arrays which are '\0' terminated.

→ To dynamically allocate arrays:

```
char *str = (char *) malloc ( amount of bytes );
```

↖ cast

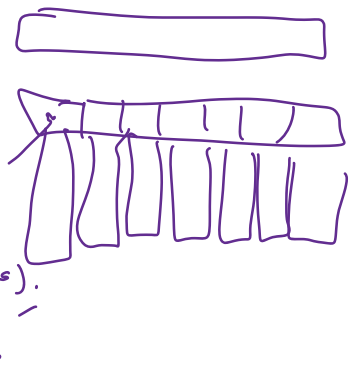
When you allocate, at some point you should also free.

Example

how do we allocate a matrix.? ↗ **

array is just pointer (int *)

matrix is pointer array (int **)

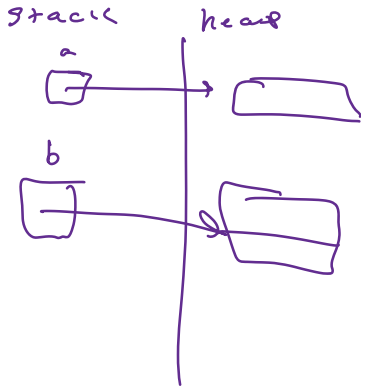


```
int **matrix = (int **) malloc(sizeof(int*) . rows)
for (int i=0; i < rows; i++) {
    matrix[i] = (int *) malloc(sizeof(int) cols);
}
```

String don't have to keep trace of length because strlen ← give you the length, strcpy ← copy

memcpy

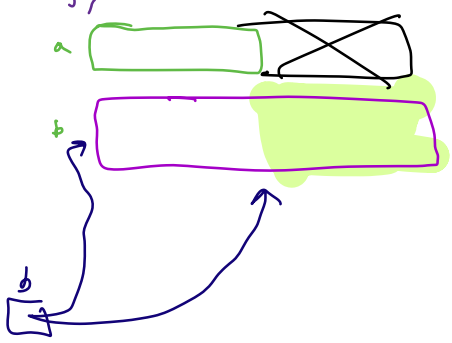
```
int *a = malloc(10 * sizeof(int));
int *b = malloc(20 * sizeof(int));
```




```
.....
int alen = 10, blen = 20
//suppose copy all the values
        from a to b:
for(int = 0; i < alen; i++) {
        b[i] = a[i];
memcpy(b, a, alen * sizeof(int));
```

"copy alen*sizeof(int) of a to b"

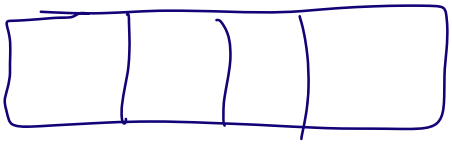
copy a to the second half b?




```
for(int = 9, i < a len + 9; i++) {
        b[i] = a[i];
}
memcpy(b + 9, a, alen * sizeof(int))
memcpy(a, b+9, alen * sizeof(int))
```

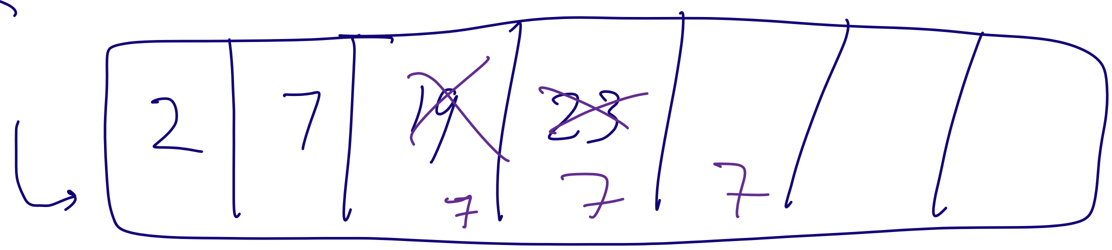
```
void insert_in_array(int *arr, int cur_size, int total, int value)
{
        int pos_to_go = ....
        memcpy(arr+pos_to_go, arr + pos_to_go, cur_size - pos_to_go)
        arr[pos_to_go] = value
}
```

moves everything from pos_to_go → pos_to_go + 1



ex)

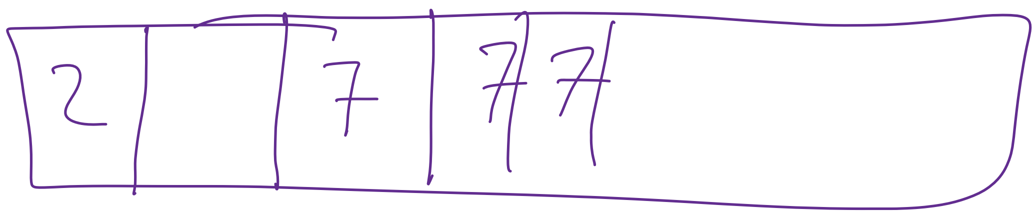
arr



insert_to_array(arr, 4, 7, 3)

pos_to_go = 1;

memcpy(2, 1, 3·sizeof(int))



*might works

---

strcpy → except don't give a length

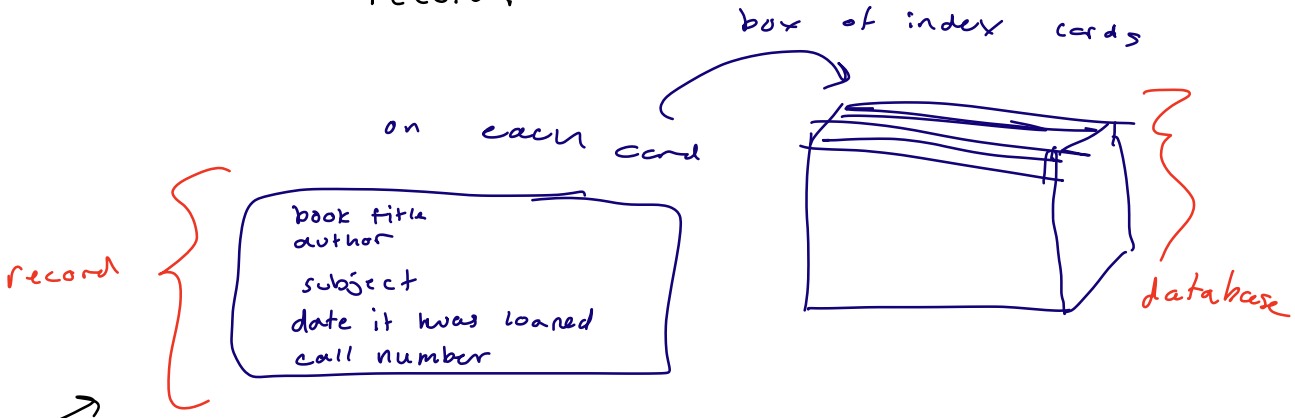
```
char * myStr = "hello my ␣ 7 spaces";
char * name = "Austin";

strcpy( myStr+9, name );
```

^ length be using strlen when to

```
nstrcpy( myStr+9, name, 1 );
```

STRUCTS

Database analogy → a collection of info. stored in records.



Structed data type ≡ a specification for a record.

ex)

```
char * title;
char * author;
char * subject;
int date;
char * callNumber;
```

Specifies the data types for a record.

Records at a local planetary observatory

Structed data type

```
char* name
int distance
int moons
double years
double rotation
```

Jupiter
142,800 km
16 moons
orbit 11.9 years
9.925 h

"object" ≈ "record"

"class diagrams" → another struct ⇒ class
↑ objects

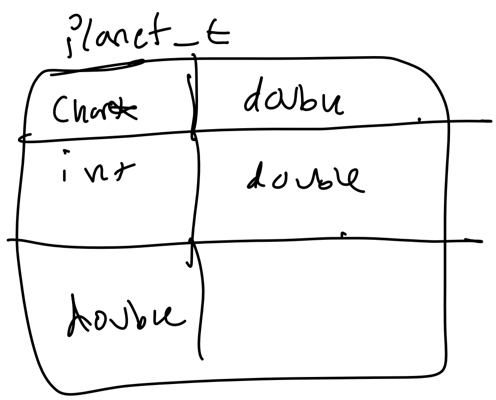
In C

```
typedef struct {
    char*   planetName;
    double  distance;
    int     num_moons;
    double  orbit_years;
    double  rot_time;
} planet_t;
```

.h files

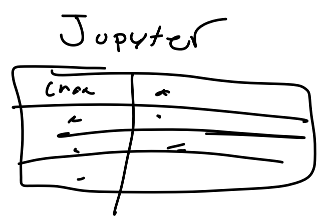
planet_t; ← good naming

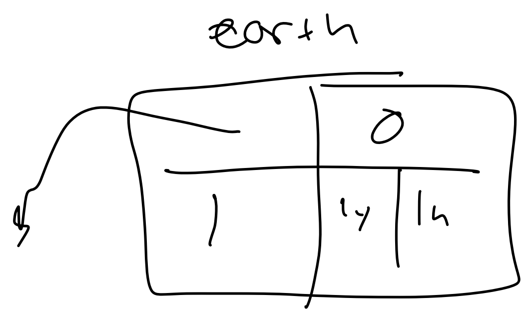
name_t ← "refer to this is a type"!

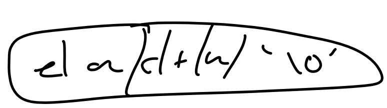


example

```
int main() {
    planet_t jupiter;
    jupyter.name = "hello";
    jupyter.distance = 142800;
    jupyter.rot = ....;

    planet_t earth;
    earth.name = "earth";
    earth.distance = 0;
    earth.moon = 1
}
```







ex)

```
typedef struct c
    double x;
    double y;
3 point_t
point_t mc_point.
```

```
void FillinPoint (point_t *point) {
    scanf("%lf", &(*point).x);
    scanf("%.1f", &(*point).y);
}
```

special operator

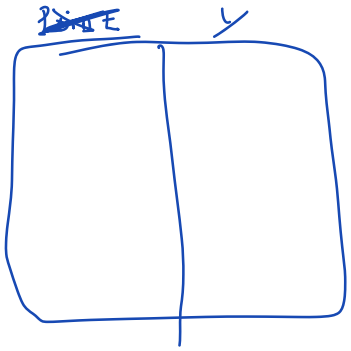
3

&(*point).x (== point)

Special operator

called indirect component selection operator

Point



&(*point.x) == &point->x

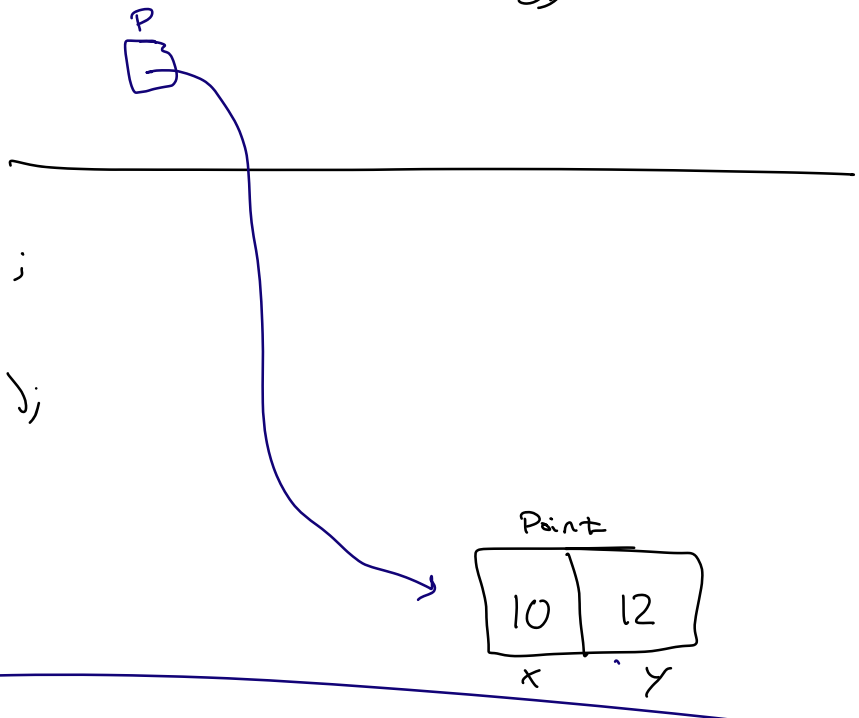
(*object).component == object->component

examples

```
void printPlanet(point_t *planet) {
    printf("%s", planet->name);
    printf("rot %f", planet->rot);
}
```

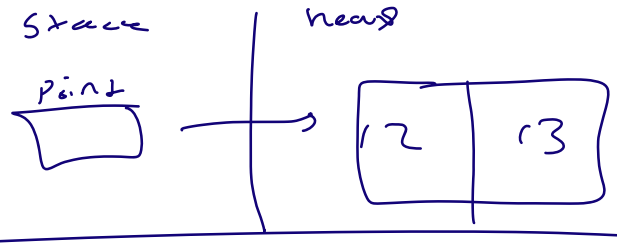
ex)

```
double magnitude(point_t *p) {
return ( sqrt ( ( (p->x * p->x) + (p->y * p->y) ) ) );
}
```



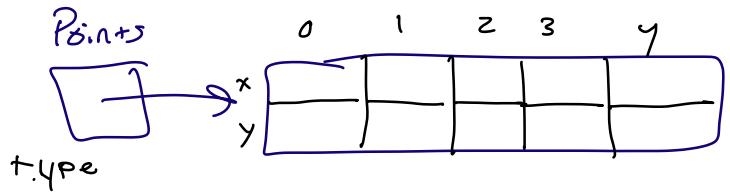

```
int main()  {

    point_t   point;
    point.x = 10;
    point.y = 12;
    magnitude(&point);
}
```

```
point_t *point = (*point_t) malloc(sizeof(point_t));
point->x = 12;
point->y = 13;
```



~~int x[10];~~
~~int y[10];~~

```
typedef struct {
        int  x, y;
} point_t;

Point_t  points[5];

for (int i = 0; i < 5; i++) {
        printf("%d", points[0].x);
}
```



points[0] point_t

points pointer to an array of point_ts

(points + 2)->x (int) x value of index 2 point

==

(*(points + 2)).x

==

points[2].x (int) x value of index 2 point

---

*(points + 3) point_t one its the fourth point

Union Types

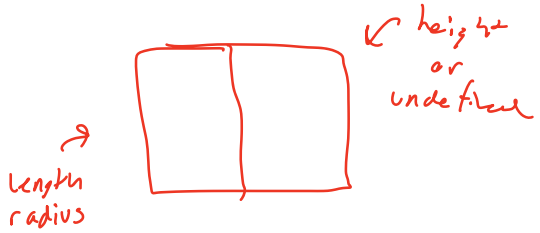
radius

```
typedef struct {
    double radius;
} circle_t;
```

length, height

```
typedef struct {
    double length, height;
} rectangle_t;
```

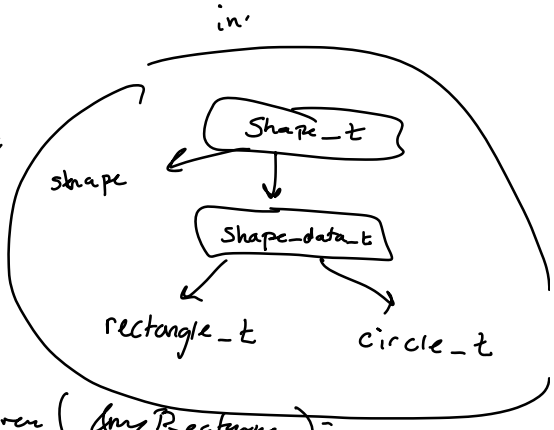
```
typedef union {
    circle_t circle;
    rectangle_t rectangle;
} shape_data_t;

typedef struct {
    char shape;
    shape_data_t data;
} shape_t;
```

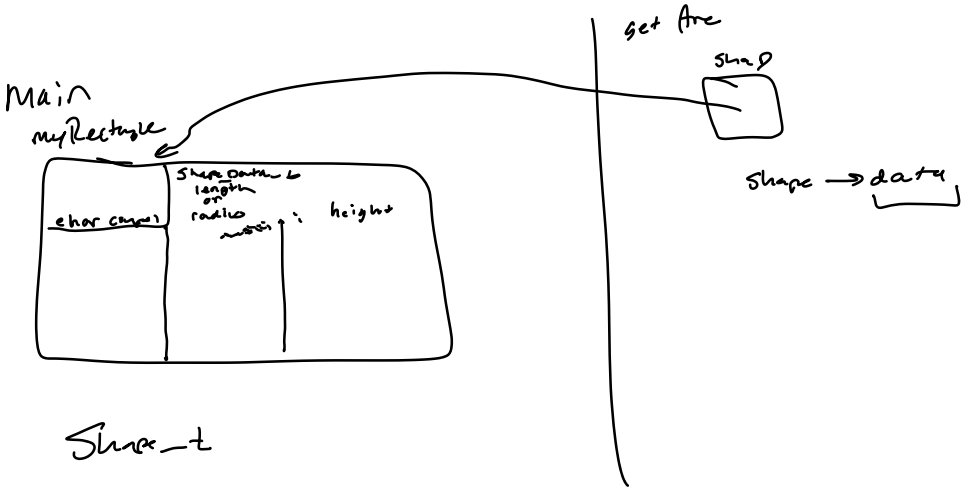


Shape_data_t → get enough memory for either a circle_t or rectangle

max(sizeof(circle_t), sizeof(rectangle_t));

```
int main() {

    Shape_t myRectangle;
    mySquare.shape = 'R';
    mySquare.data.rectangle.length = 7;
    mySquare.data.rectangle.width = 8;

    Shape_t myCircle;
    myCircle.shape = 'c';
    myCircle.data.circle.radius = 2;

    getArea(&myCircle) or getArea(&myRectangle);
}

double getArea(Shape_t *shape) {

    switch (shape->shape) {
        case 'c':
            return pi * shape->data.circle.radius * shape->data.circle.radius
        case 'r':
            return shape->data.rectangle.width *
                   shape->data.rectangle.height;
    }
}
```





```
typedef struct {
    shape_t* shapeP;
} example_t;

example_t *example = malloc(sizeof(example_t));
```

example → shapp → data. circle. radius

```
shape_t shape;
shape.shape = 'R';
shape.data.rectangle.height = 7;
shape.data.rectangle.length = 9;
```

shape.data.circle.radius = ? union