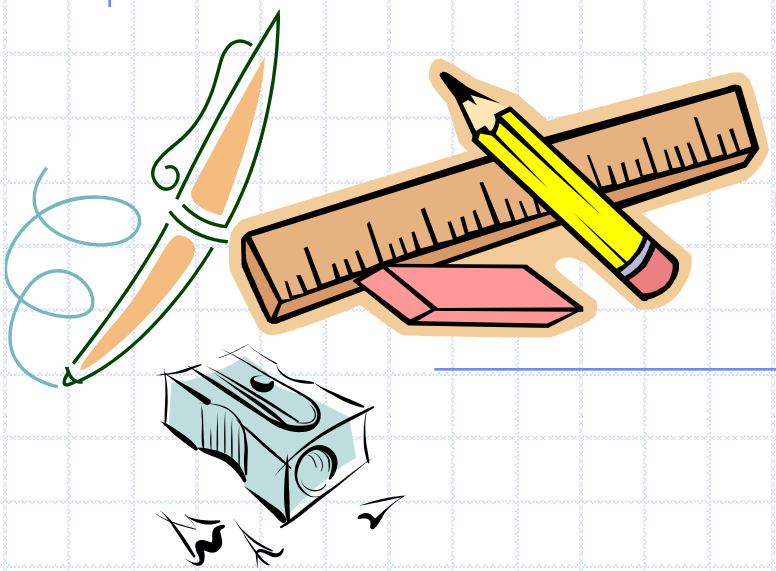
ESC101: Introduction to Computing

Structures



Motivation

- Till now, we have used data types int, float, char, arrays (1D, 2D,...) and pointers.
- What if we want to define our own data types based on these?
- A geometry package – we want to define a point as having an x coordinate, and a y coordinate.
- Student data – Name and Roll Number
  - array of size 2?
  - two variables:
    - int point_x , point_y;
    - char *name; int roll_num;

Motivation

- A geometry package – we want to define a point as having an x coordinate, and a y coordinate.
- Student data – Name and Roll Number
  - array of size 2? (Can not mix TYPES)
  - two variables:
    - int point_x , point_y;
    - char *name; int roll_num;
  - There is no way to indicate that they are part of the same point!
  - requires a disciplined use of variable names
- Is there any better way ?

Motivation: Practical Example

- Write a program to manage customer accounts for a large bank.
- Customer information as well as account information, for e.g.:
  - Account Number — int
  - Account Type — int (enum – not covered)
  - Customer Name — char*/char[]
  - Customer Address — char*/char[]
  - Signature scan — bitmap image (2-D array of bits)

Example: Enumerated types

- Account type via Enumerated Types.
- Enumerated type allows us to create our own symbolic name for a list of related ideas.
  - The key word for an enumerated type is **enum**.
- We could create an enumerated type to represent various "account types", by using the following C statement:

```
enum act_Type { savings, current, fixDeposit, minor };
```

Example: Enumerated types

- Account type via Enumerated Types.

```
enum act_Type { savings, current, fixDeposit, minor };

enum act_Type a;

a = current;

if (a==savings)
    printf("Savings account\n");

if (a==current)
    printf("Current account\n");
```

Enumerated types provide a symbol to represent one state out of several constant states.

Structures

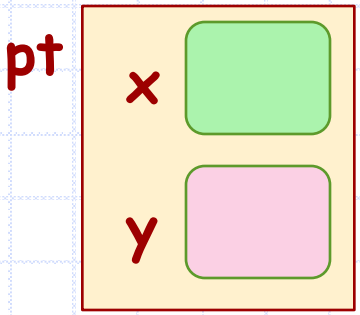
- A structure is a collection, of variables, under a common name.
- The variables can be of different types (including arrays, pointers or structures themselves!).
- Structure variables are called fields.

```
struct point {
        int x;
        int y;
};
```

```
struct point pt;
```

This defines a structure called point containing two integer variables (fields), called x and y.

struct point pt defines a variable pt to be of type struct point.



memory depiction of pt

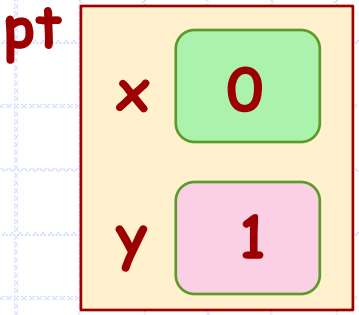
Structures

- The x field of pt is accessed as pt.x.
- Field pt.x is an int and can be used as any other int.
- Similarly the y field of pt is accessed as pt.y

```
struct point {
        int x;
        int y;
};
```

```
struct point pt;
```

```
pt.x = 0;
pt.y = 1;
```



memory depiction of pt

Structures

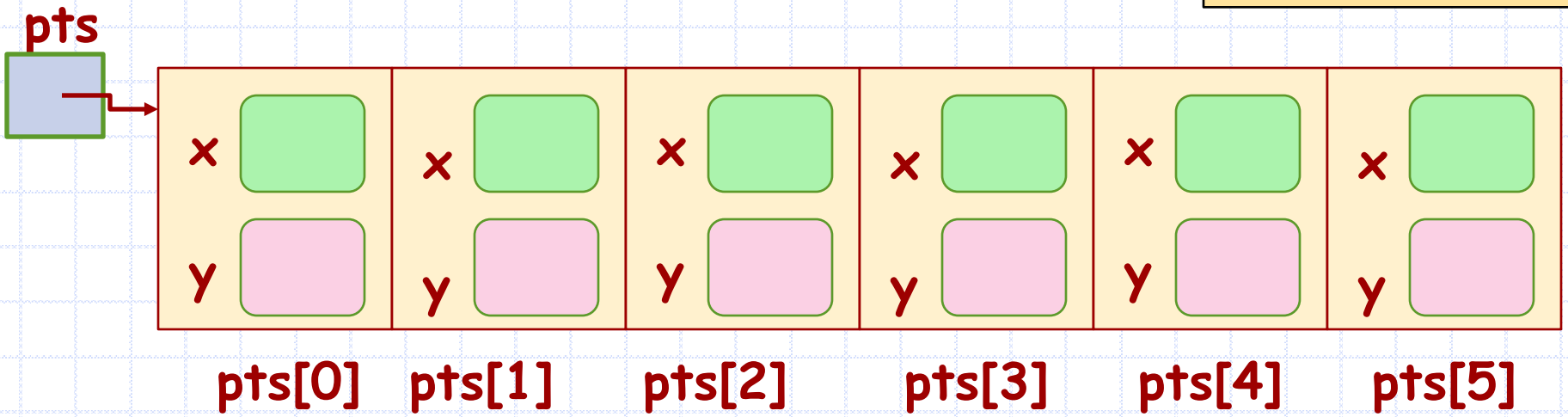
```
struct point {
        int x; int y;
}
```

struct point is a type. It can be used just like int, char etc..

For now, define structs in the beginning of the file, after #include.

```
struct point pt1,pt2;
struct point pts[6];
```

We can define array of struct point also.



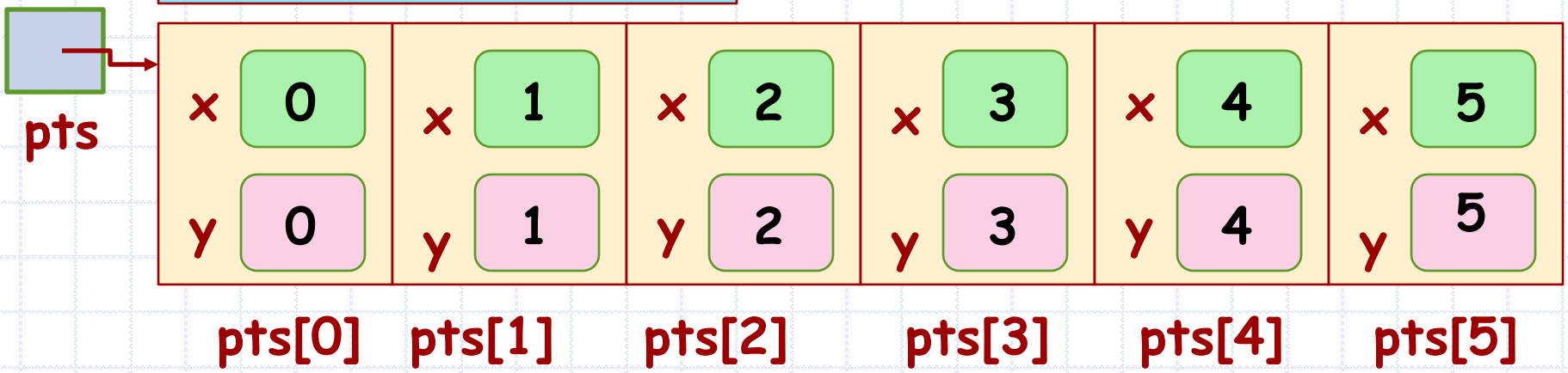

```
int i;
for (i=0; i < 6; i=i+1) {
        pts[i].x = i;
        pts[i].y = i;
}
```

Read pts[i].x as (pts[i]).x
The . and [] operators have same precedence. Associativity: left-right.

Structures

```
struct point {
        int x; int y;
};
struct point pts[6];
int i;
for (i=0; i < 6; i=i+1)
{
        pts[i].x = i;
        pts[i].y = i;
}
```

State of memory after the code executes.



Reading structures (scanf ?)

```
struct point {
      int x; int y;
};
```

```
int main() {
    int x, y;
    struct point pt;
    scanf("%d%d", &(pt.x),&(pt.y));
    return 0;
}
```

1. You can not read a structure directly using scanf!
2. Read individual fields using scanf (note the &).
3. A better way is to define our own functions to read structures
   - to avoid cluttering the code!

Functions returning structures

```
struct point {
        int x; int y;
};
```

```
struct point make_point
                (int x, int y)
{
        struct point temp;
        temp.x = x;
        temp.y = y;
        return temp;
}
int main() {
        int x, y;
        struct point pt;
        scanf("%d%d", &x,&y);
        pt = make_point(x,y);
        return 0;
}
```

1. make_point(x,y) creates a struct point given coordinates (x,y).
2. Note: make_point(x,y) returns struct point.
3. Functions can return structures just like int, char, int *, etc..
4. We can also pass struct parameters. struct are passed by copying the values.

Given int coordinates x,y, make_point(x,y) creates and returns a struct point with these coordinates.

Functions with structures as parameters

```
# include <stdio.h>
# include <math.h>
struct point {
        int x; int y;
};
double norm2( struct point p)  {
    return sqrt ( p.x*p.x + p.y*p.y);
}
int main() {
        int x, y;
        struct point pt;
        scanf("%d%d", &x,&y);
        pt = make_point(x,y);
        printf("distance from origin
                is %f ", norm2(pt) );
        return 0;
}
```

The norm2 or Euclidean norm of point (x,y) is

$$\sqrt{x^2 + y^2}$$

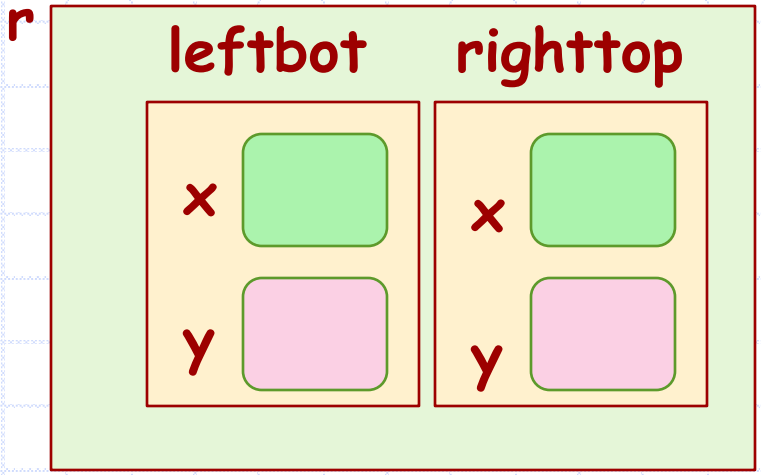
norm2(struct point p) returns Euclidean norm of point p.

Structures inside structures

```
struct point {
        int x; int y;
};
```

```
struct rect {
    struct point leftbot;
    struct point righttop;
};
struct rect r;
```

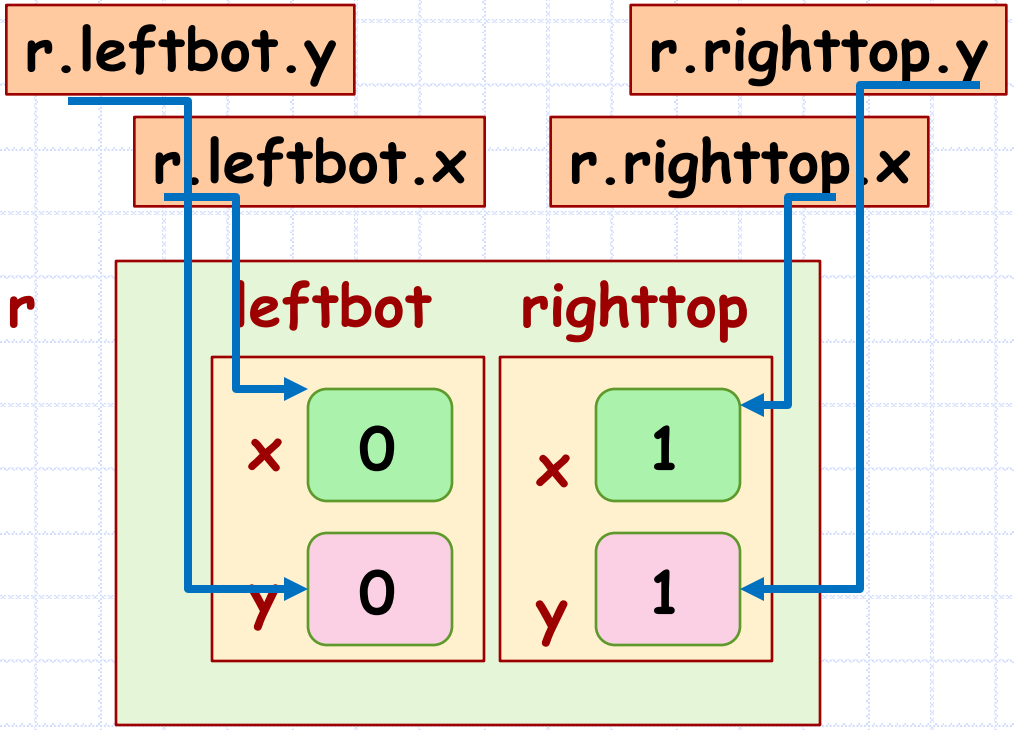
1. Recall, a structure definition defines a type.
2. Once a type is defined, it can be used in the definition of new types.
3. struct point is used to define struct rect. Each struct rect has two instances of struct point.



r is a variable of type struct rect. It has two struct point structures as fields.

So how do we refer to the x of leftbot point structure of r?

```
struct point {
        int x;
        int y;
};
struct rect {
    struct point leftbot;
    struct point righttop;
};
int main() {
        struct rect r;
        r.leftbot.x = 0;
        r.leftbot.y = 0;
        r.righttop.x = 1;
        r.righttop.y = 1;
        return 0;
}
```



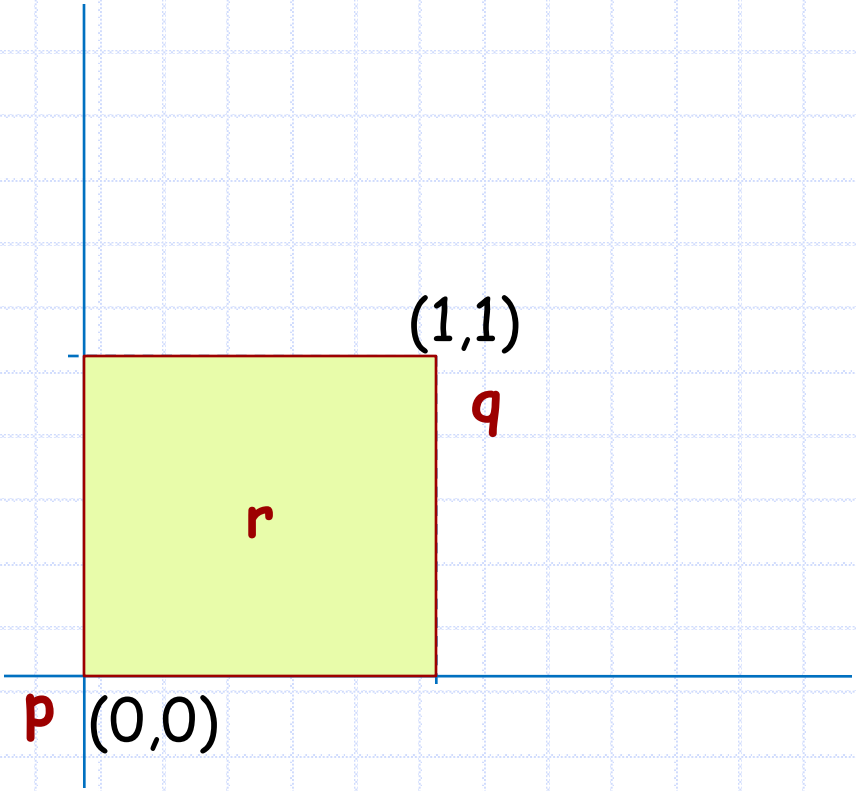
Addressing nested fields unambiguously

Initializing structures

```
struct point {
        int x; int y;
};
```

1. Initializing structures is very similar to initializing arrays.
2. Enclose the values of all the fields in braces.
3. Values of different fields are separated by commas.

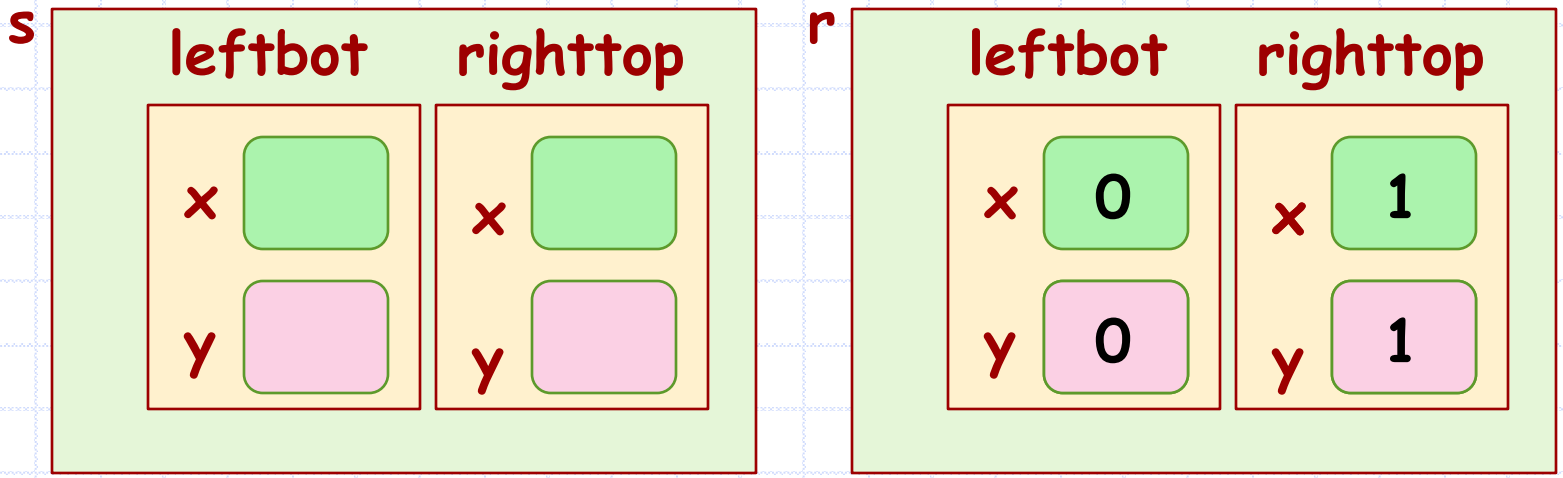
```
struct rect {
    struct point leftbot;
    struct point righttop;
};
struct point p = {0,0};
struct point q = {1,1};
struct rect r = {{0,0}, {1,1}};
```



Assigning structure variables

```
struct rect r,s;
r.leftbot.x = 0;
r.leftbot.y = 0;
r.righttop.x = 1;
r.righttop.y = 1;
s=r;
```

1. We can assign a structure variable to another structure variable
2. The statement `s=r;` does this
3. Structures are *assignable* variables, unlike arrays!

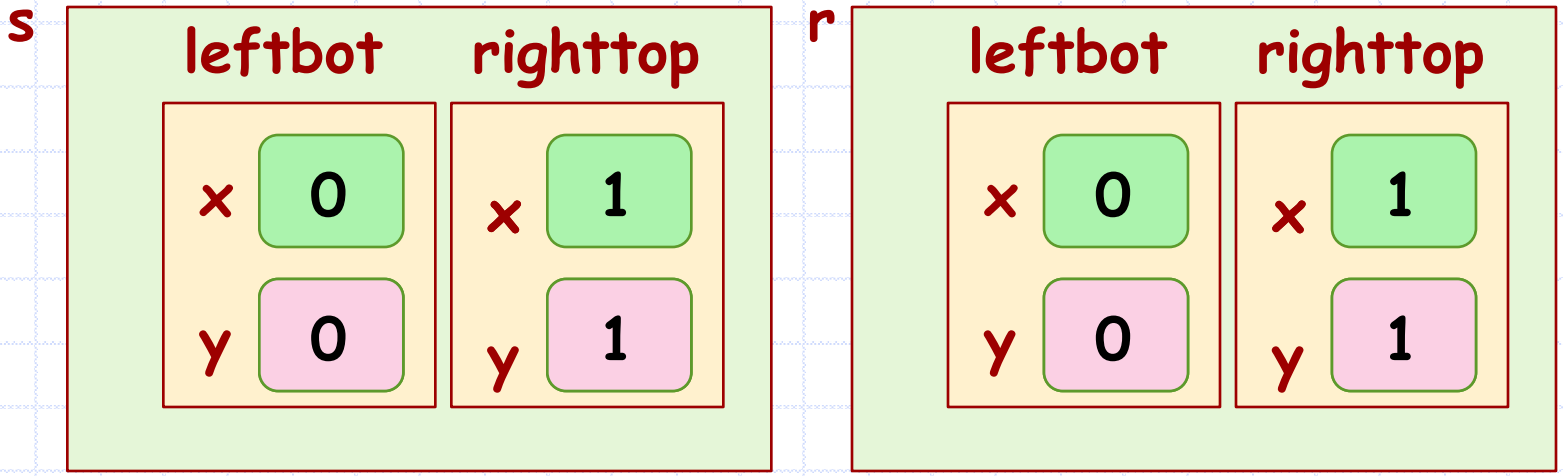


Before the assignment

Assigning structure variables

```
struct rect r,s;
r.leftbot.x = 0;
r.leftbot.y = 0;
r.righttop.x = 1;
r.righttop.y = 1;
s=r;
```

1. We can assign a structure variable to another structure variable
2. The statement `s=r;` does this.
3. Structures are *assignable* variables, unlike arrays!



After the assignment