# Composite Data Types:

# Structures

ESC101: Fundamentals of Computing

Nisheeth

# Composite Data

- Case 1: A geometry package – we want to define a variable for a two-dimensiona point to store its x coordinate and y coordinate.
- Case 2: Student data – Name and Roll Number
- First strategy: Array of size 2?
  - Will work for case 1 but not for case 2 since we can not mix TYPES
- Another strategy: Use two variables,

  int point_x , point_y ;  char *name; int roll_num;
  - No way to indicate that both variables are part of the same “big” variable
  - We need to be very careful about variable names.
- Is there any better way ?

# 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!).
- Each variable within a structure is called a field.

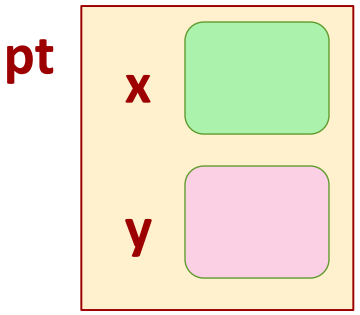
name of this structure

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

```
struct point pt;
```

Defines a structure named 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

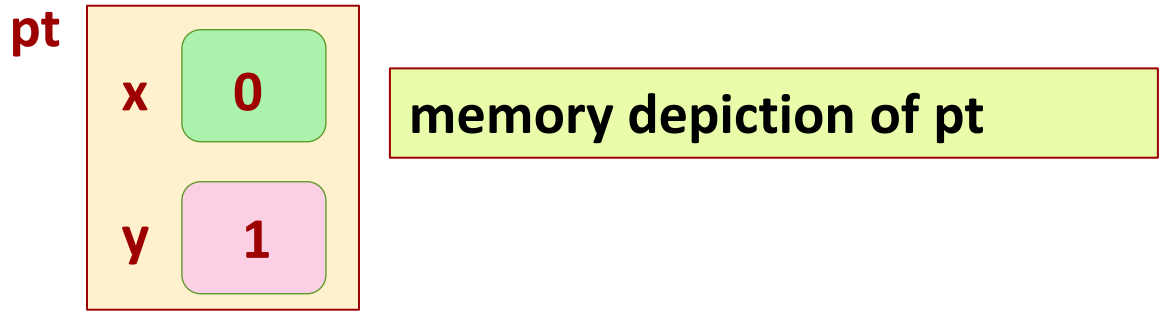
# Structures

- 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;
```



# Structures

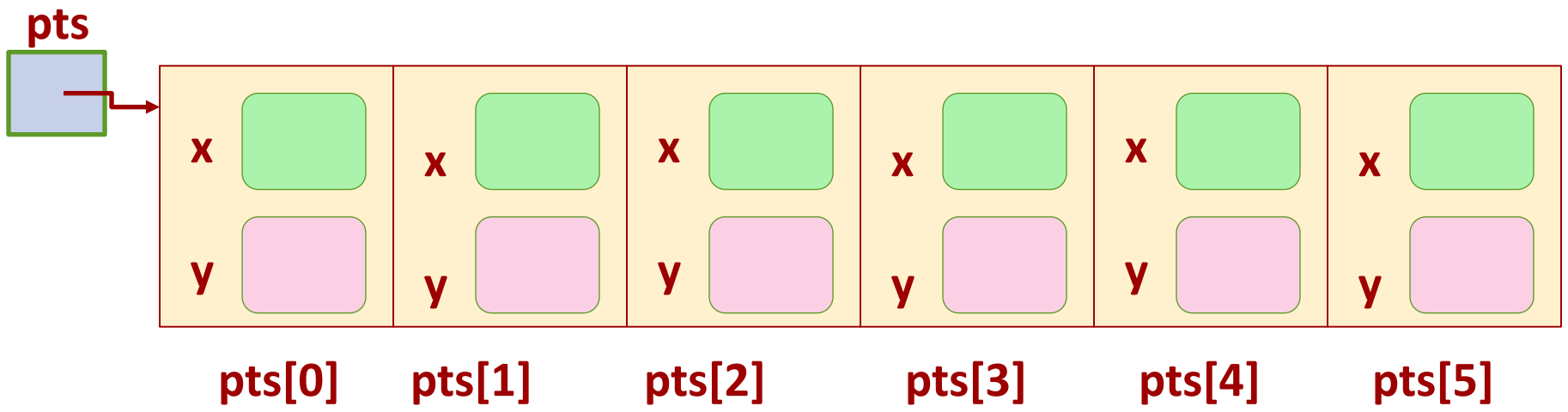
```
struct point {
        int x; int y;
}
```

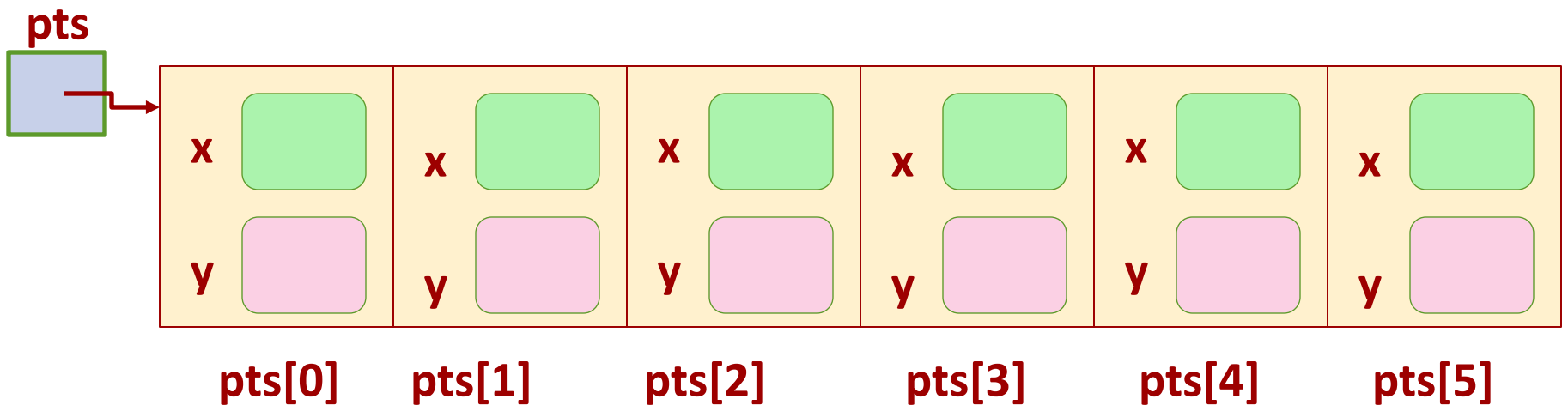
**struct point** is a type.
It can be used just like int, char etc..

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

We can even define an array of struct point

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





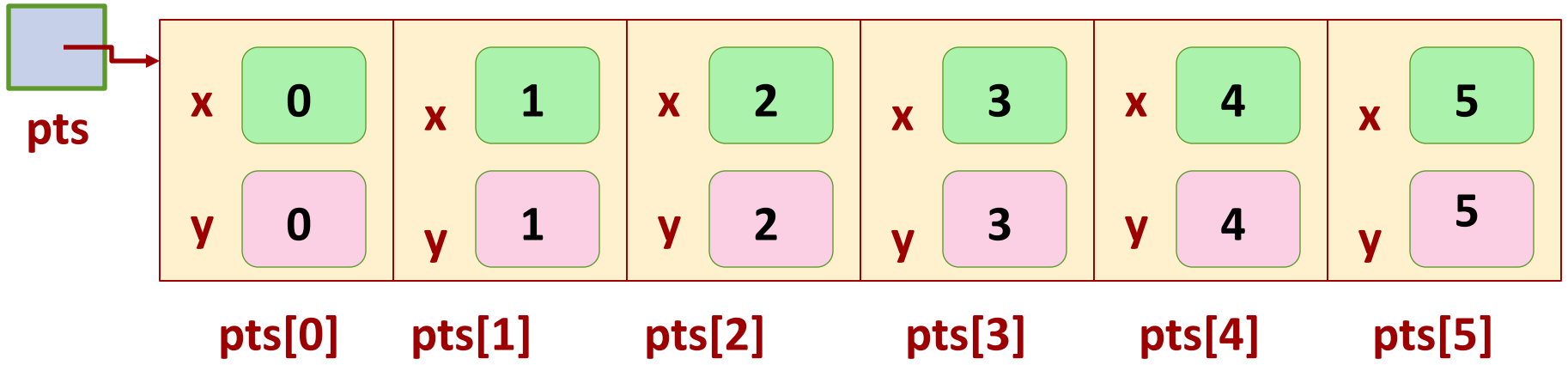

```
int i;
for (i=0; i < 6; i=i+1) {
        pts[i].x = i;
        pts[i].y = i;
}
```

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_pt(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_pt(x,y);
    return 0;
}
```

**make_pt(x,y):**
creates a struct point with coordinates (x,y), and returns a struct point.

Functions can return structures just like int, char, int *, etc..

struct can be passed as arguments (pass by value).

Given int coordinates x,y, make_pt(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

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

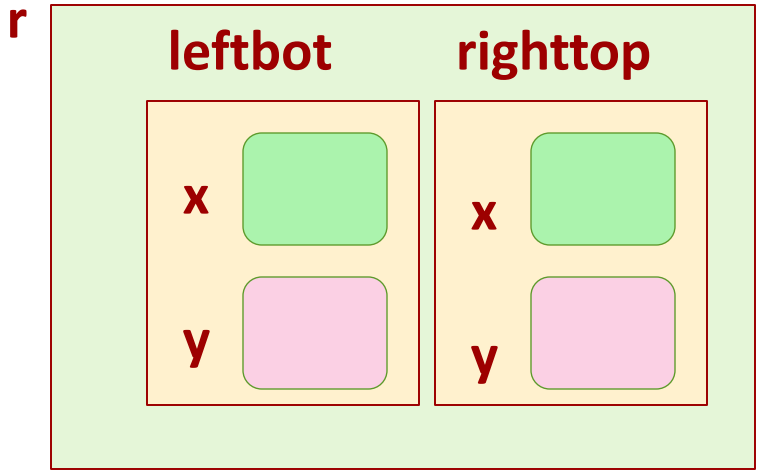
norm2(struct point p) returns Euclidean norm of point p.

# Structures inside structures

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

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

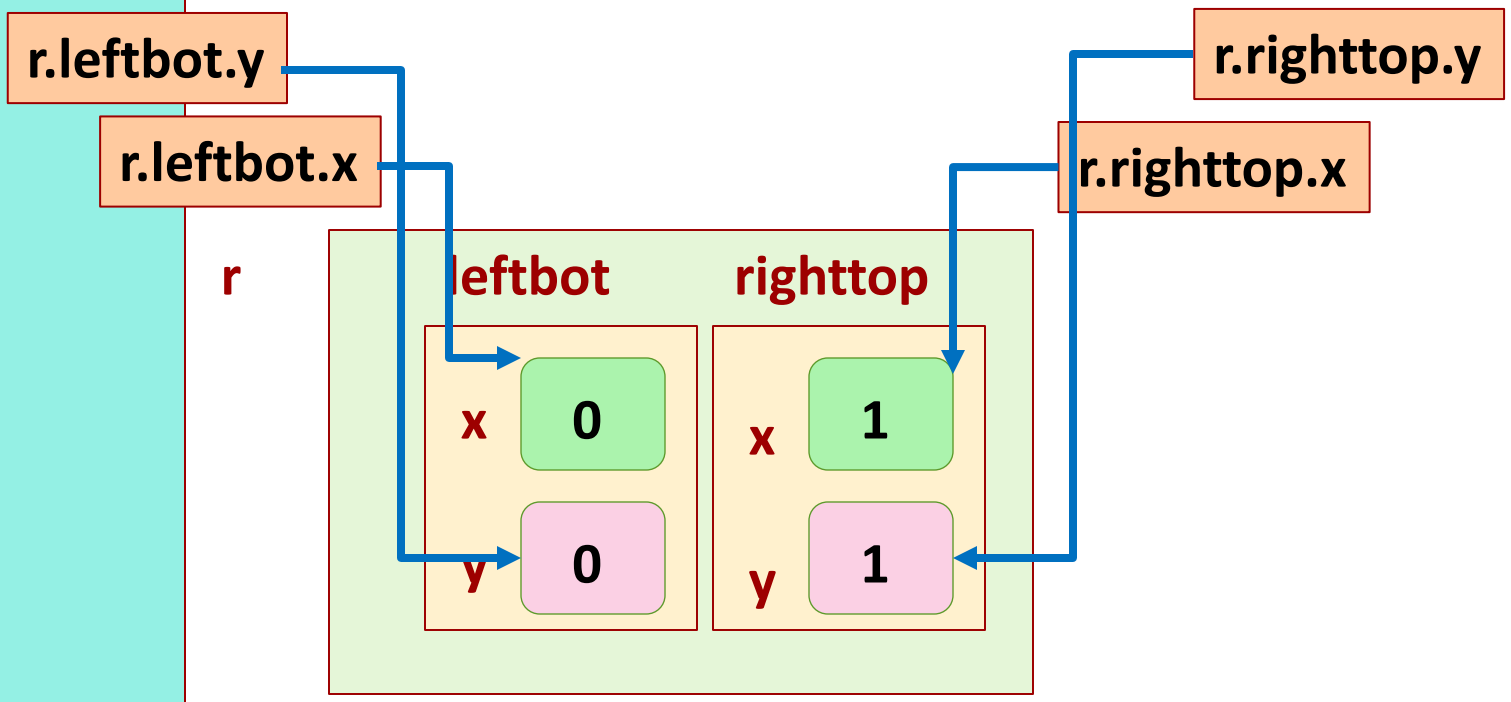
1. Recall, a structure definition defines a type.
2. Once a type is defined, it can be used in the definition of new types.
3. struct point is used to define struct rect. Each struct rect has two instances of struct point.




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;
}
```



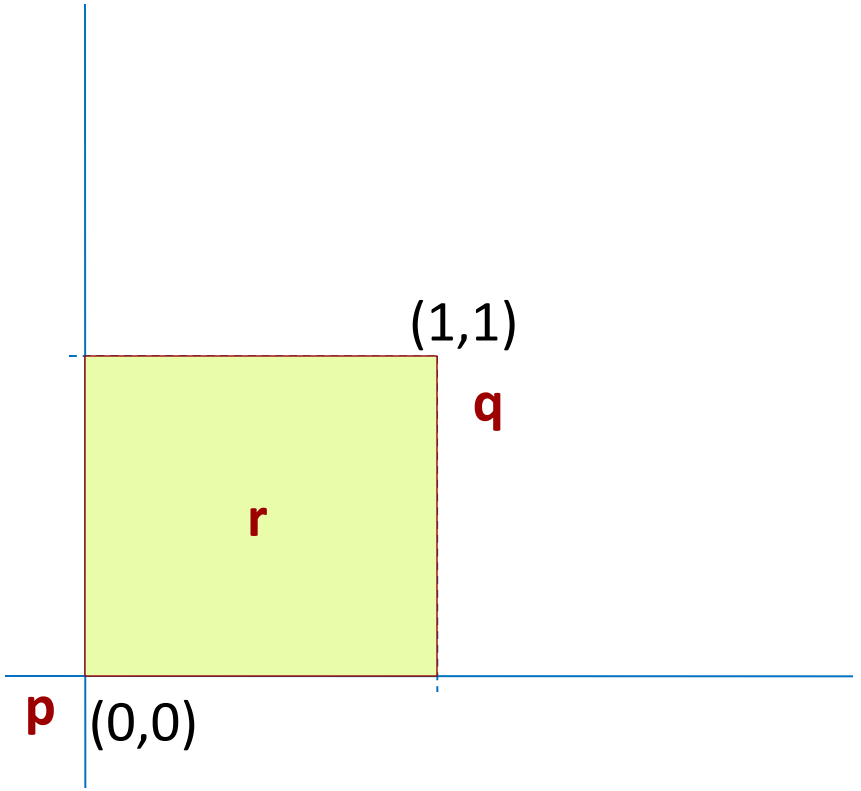

Addressing nested fields unambiguously

# 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.

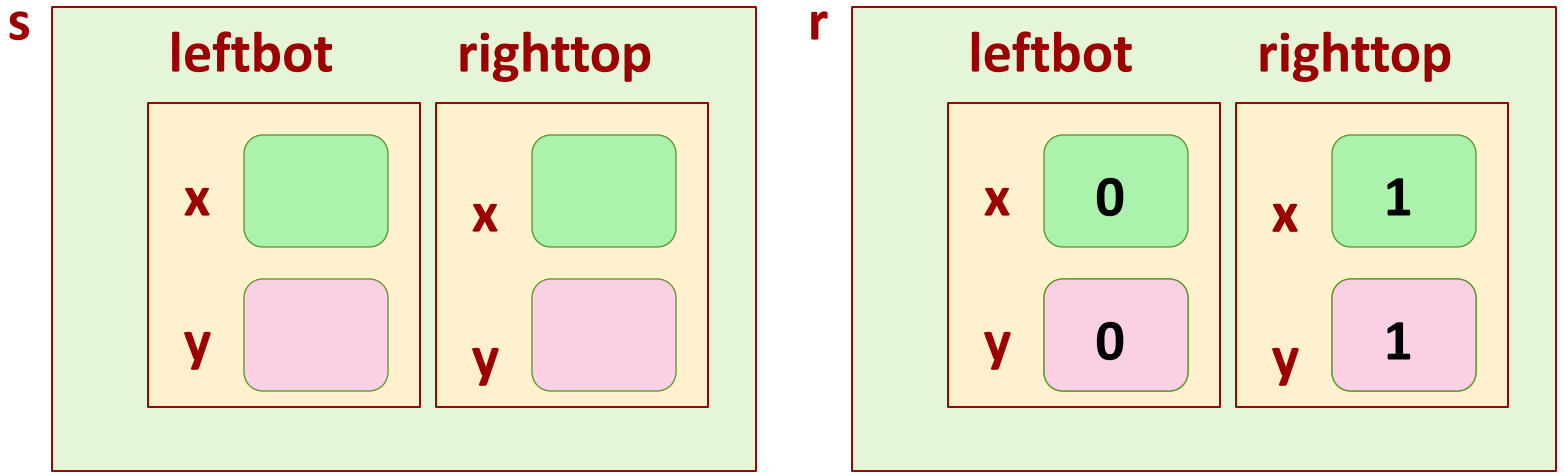
```
struct rect {
  struct point leftbot;
  struct point righttop;
};
struct point p = {0,0};
struct point q = {1,1};
struct rect r = {{0,0}, {1,1}};
```



# 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!

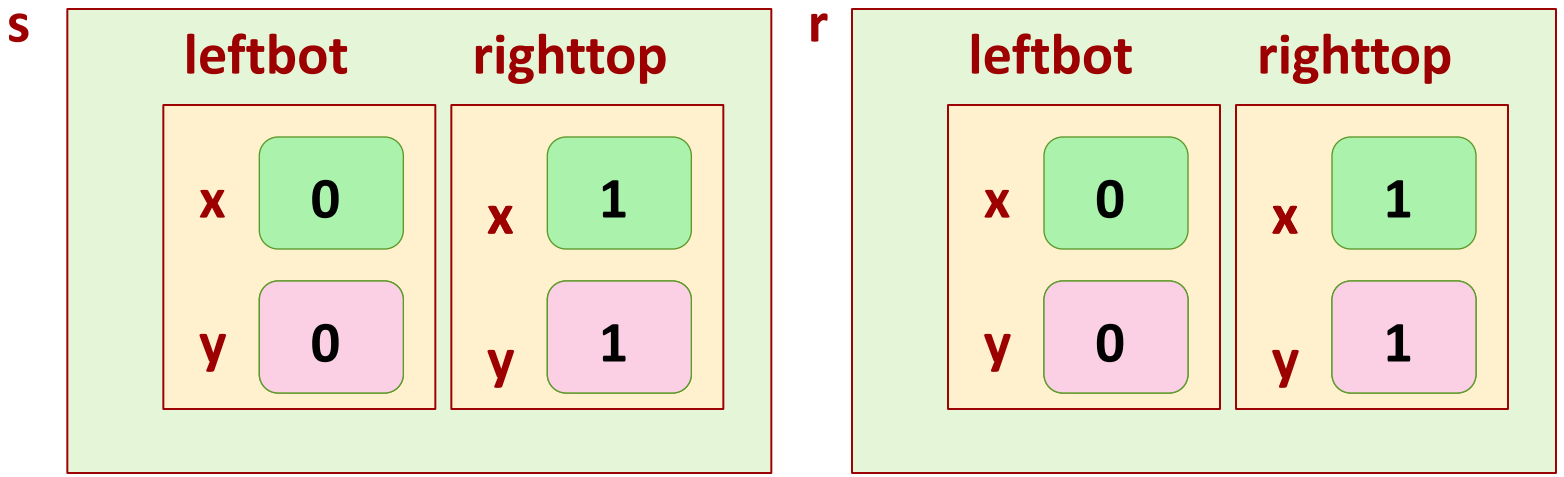



Before the assignment

# 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!

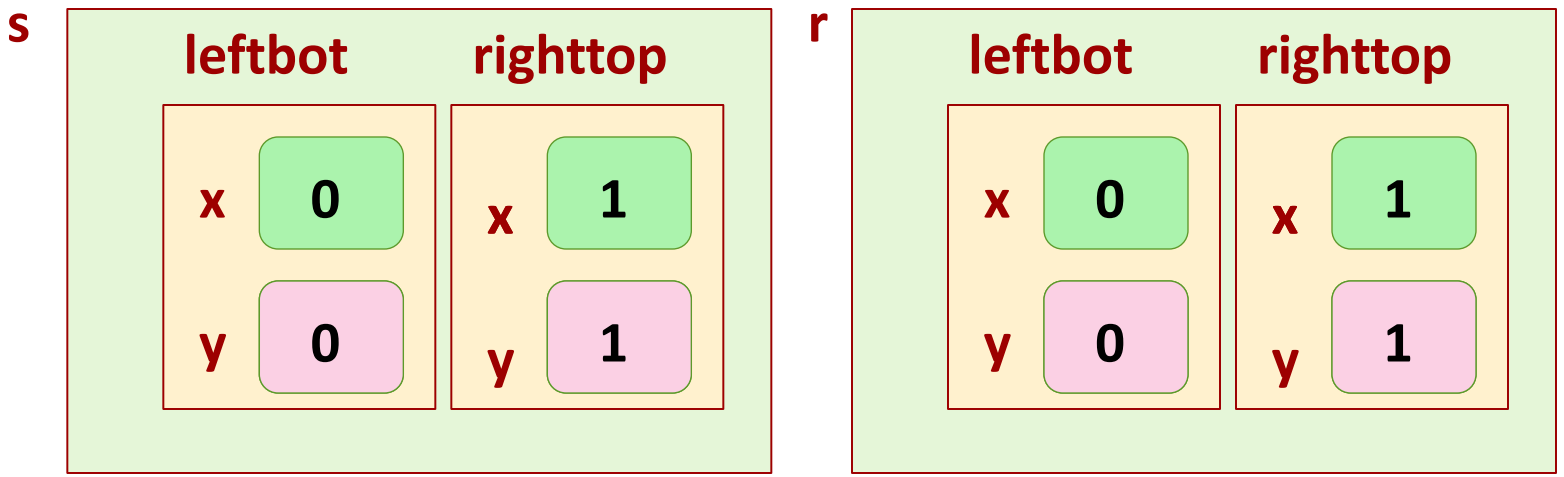



After the assignment

# 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!
4. Structure name is *not* a pointer, unlike arrays.



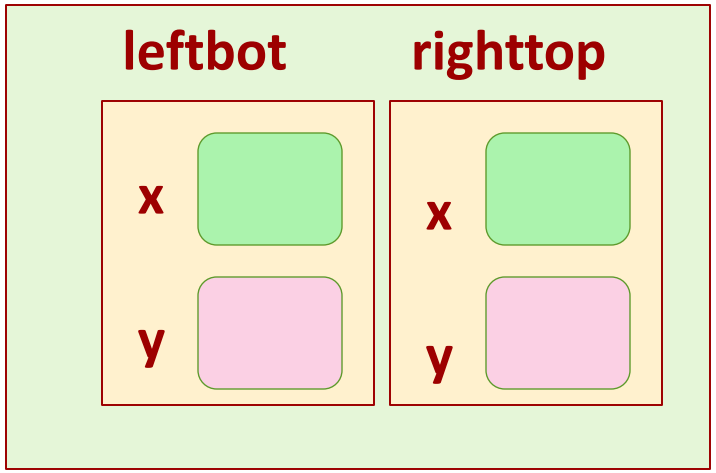

After the assignment

# Passing structures..?

```
struct rect { struct point leftbot;
              struct point righttop; };
int area(struct rect r) {
   return
      (r.righttop.x – r.leftbot.x) *
      (r.righttop.y – r.leftbot.y);
}
void fun() {
   int ar;
   struct rect r1 ={{0,0}, {1,1}};
   ar = area(r1);
}
```

We can pass structures as parameters, and return structures from functions, like the basic types int, char, double etc..

But is it efficient to pass structures or to return structures?




Usually NO. E.g., to pass struct rect as parameter, 4 integers are copied. This is expensive.

Same for returning structures

*So what should be done to pass structures to functions?*

# Passing structures..?

```
struct rect { struct point leftbot;
              struct point righttop;};
int area(struct rect *pr) {
  return
 ((*pr).righttop.x – (*pr).leftbot.x) *
 ((*pr).righttop.y – (*pr).leftbot.y);
}
void fun() {
   int ar;
   struct rect r ={{0,0}, {1,1}};
   ar = area (&r);
 }
```

Instead of passing structures, pass pointers to structures.

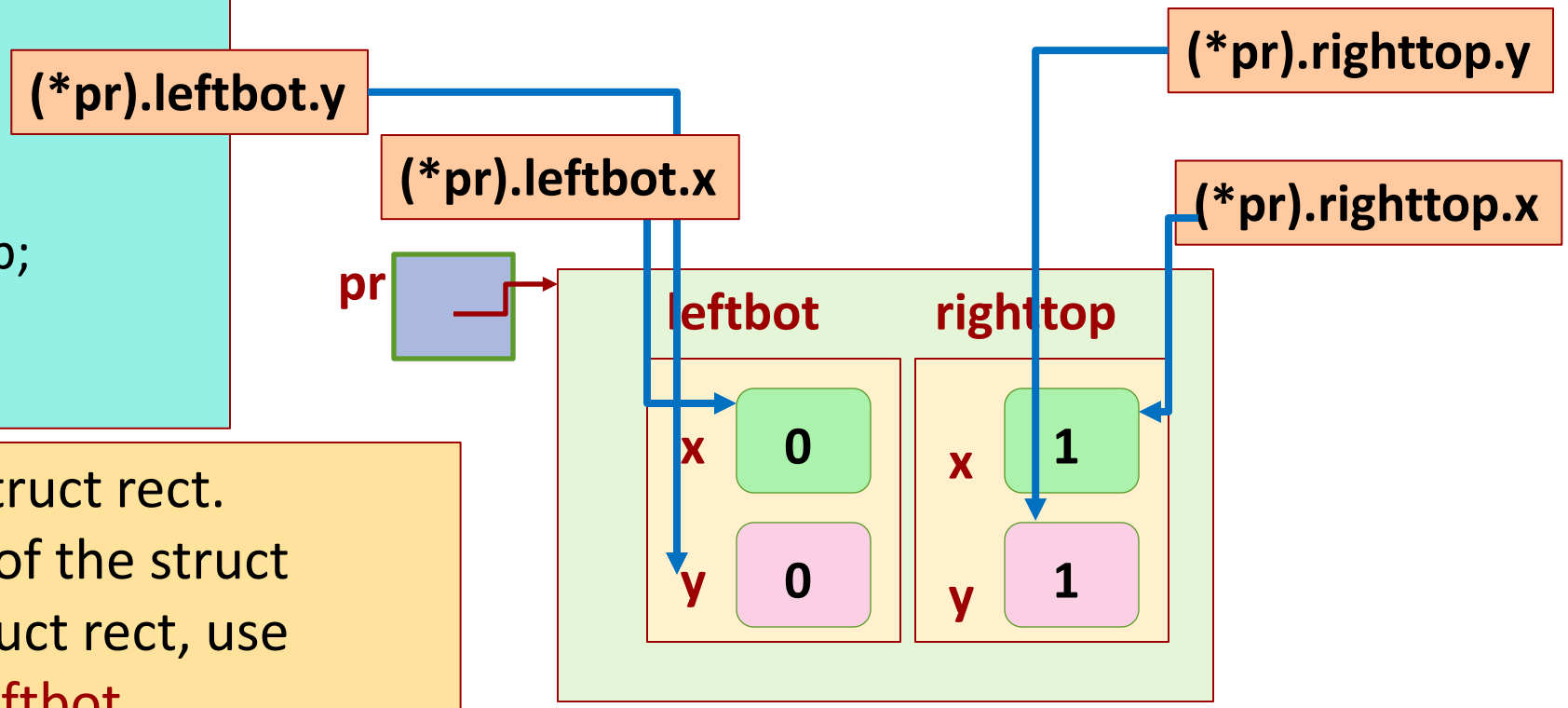
**area() uses a pointer to struct as a parameter, instead of struct rect itself.**

Only one pointer instead of large struct.

**Same for returning structures**

# Structure Pointers

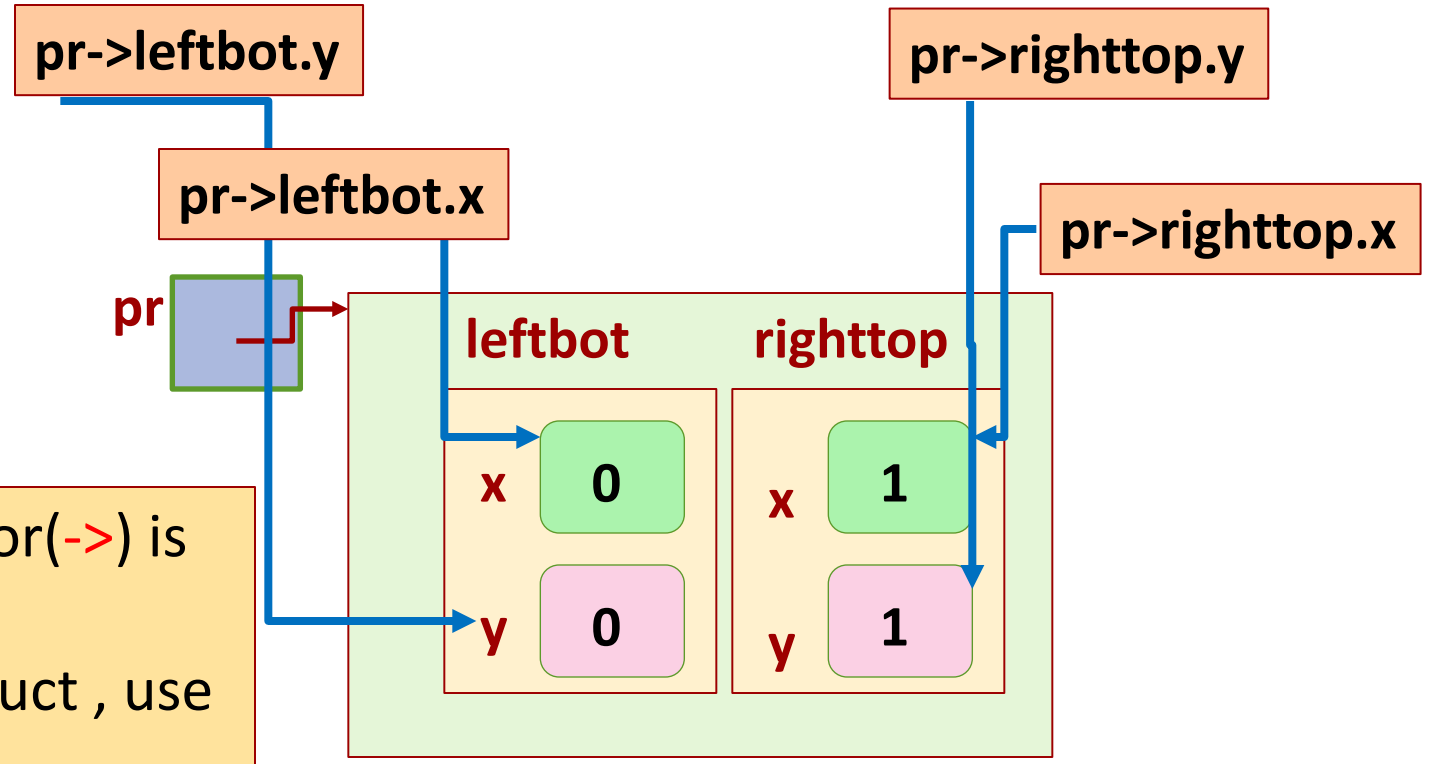
```
struct point {
   int x; int y;
};
struct rect {
   struct point leftbot;
   struct point righttop;
};
struct rect *pr;
```




1. pr is pointer to struct rect.
2. To access a field of the struct pointed to by struct rect, use
   (*pr).leftbot
   (*pr).righttop
3. Bracketing (*pr) is essential here. * has lower precedence than .
4. To access the x field of leftbot, use (*pr).leftbot.x

Addressing fields
via the structure's pointer

# Addressing fields via the pointer (shorthand)




1. Shorthand: arrow operator(->) is provided.
2. To access a field of the struct , use pr->leftbot
3. -> is one operator. To access x field of leftbot, pr->leftbot.x
4. -> and . have same precedence and are left-associative. Equivalent to (pr->leftbot).x

pr->leftbot is equivalent to (*pr).leftbot

# Passing by value or reference

- When a struct is passed directly, it is passed by copying its contents
  - Any changes made inside the called function are lost on return
  - This is same as that for simple variables
- When a struct is passed using pointer
  - Change made to the contents using pointer dereference are visible outside the called function

# Functions Returning Structures

```
struct point make_pt (int x, int y) {
        struct point temp;
        temp.x = x;
        temp.y = y;
        return temp;    }

void print_pt (struct point pt) {
        printf("%d  %d\n", pt.x, pt.y); }

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

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

Even though not returning anything, make_pt is still able to do the job using pointers

# Functions Returning Structures

```
void make_pt(int x, int y, struct point *temp) {
        temp->x = x;
        temp->y = y;
}

void print_pt(struct point *pt) {
        printf("%d  %d\n", pt->x, pt->y);
}

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

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