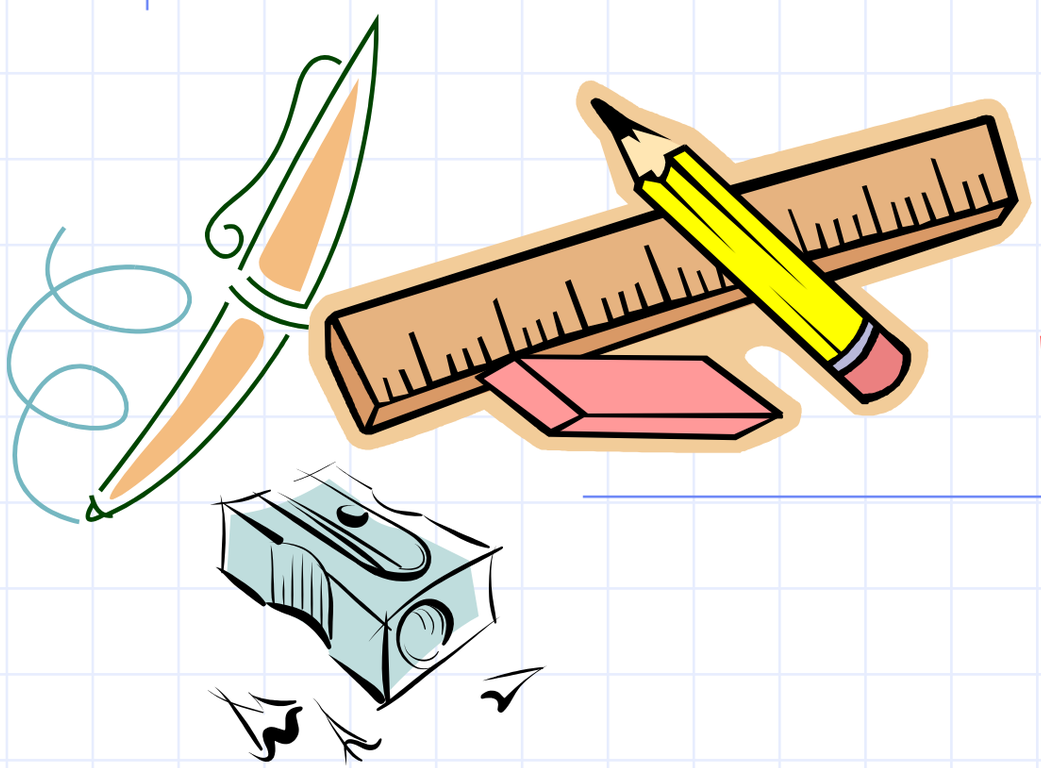
ESC101: Introduction to Computing

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
  - Account Type
  - Customer Name
  - Customer Address
  - Signature scan

int

int (enum – not covered)

char*/char[]

char*/char[]

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.

Wrong Code

```
#include <stdio.h>

int main()
{
   enum color ={black, blue, red, yellow, white};
   scanf("%d",&color);
   switch(color)
   {
      case black: printf("black\n"); break;
      case blue: printf("blue\n"); break;
      case red: printf("red\n"); break;
      case yellow: printf("yellow\n"); break;
      default: printf("white\n"); break;
   }
   return 0;
}
```

```c
#include <stdio.h>

int main()
{
  enum color {black, blue, red, yellow, white};
  enum color col;
  scanf("%d",&col);
  switch(col)
  {
    case black: printf("black\n"); break;
    case blue: printf("blue\n"); break;
    case red: printf("red\n"); break;
    case yellow: printf("yellow\n"); break;
    default: printf("white\n"); break;
  }
  return 0;
}
```

Structures

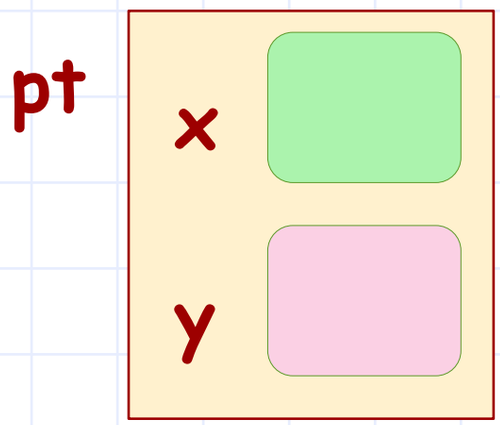
- A structure is a collection, of **variables**, under a common name.
- The variables can be of **different** types (including arrays, pointers or structures themselves!).
- Structure variables are called fields.

```
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

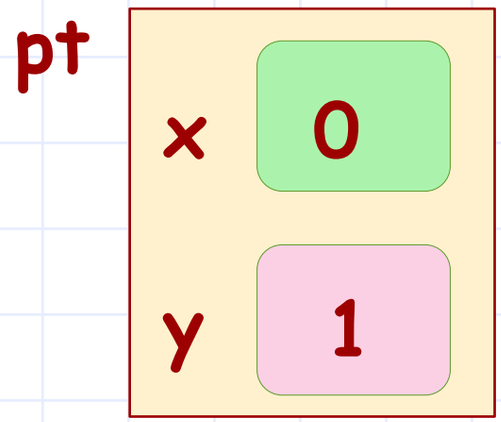
Structures

- 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

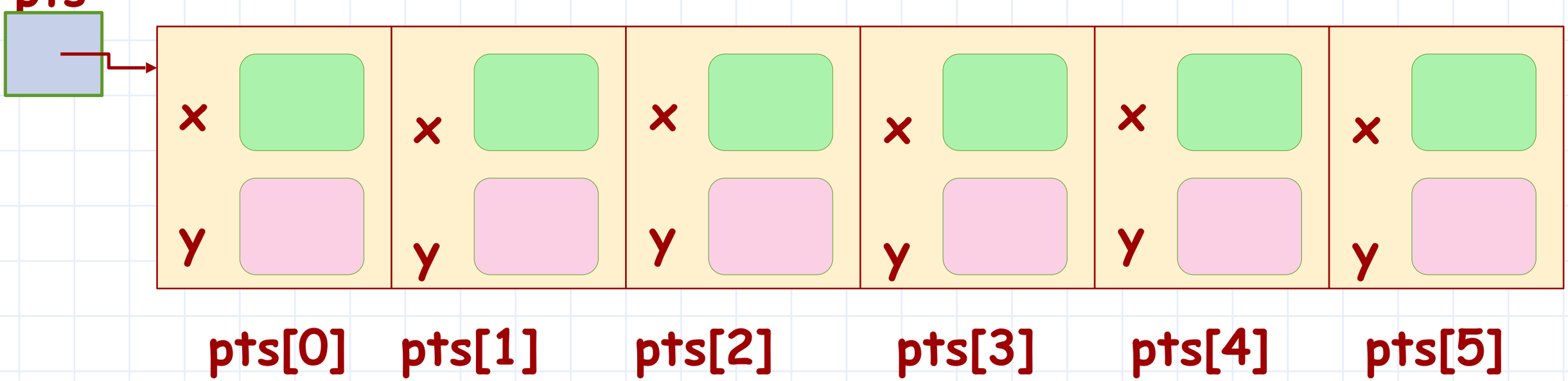
```
struct point {
        int x; int y;
}
```

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.



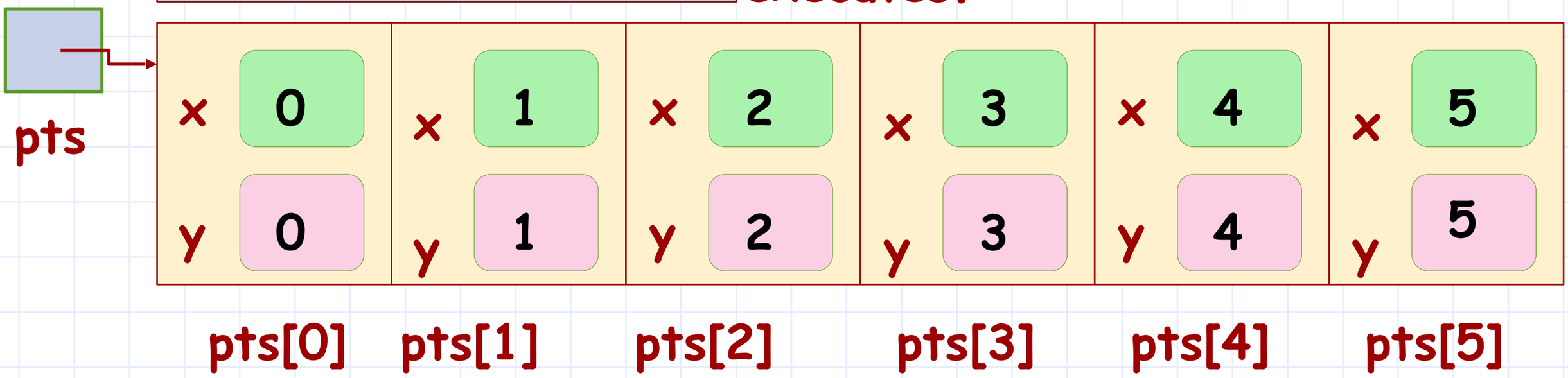

```
int i;
for (i=0; i < 6; i=i+1) {
        pts[i].x = i;
        pts[i].y = i;
}
```

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 &).

Exercise

- Write a program to read in *n* points with x and y coordinates. Find the (axis aligned) rectangle that contains all the points. Assume that maximum number of points will not exceed 100

```
#include <stdio.h>

struct Point
{
   int x, y;
};

int main()
{
   int n;
   struct Point ptarr[100];
  int minx, maxx, miny, maxy;
  scanf("%d",&n);
  for( int i=0; i<n; i++)
     scanf("%d %d”,&(ptarr[i].x), &(ptarr[i].y) );
  return 0;
}
```

```
#include <stdio.h>
// Struct Point definition
int main()
{
   int n;
   struct Point ptarr[100];
   int minx, maxx, miny, maxy;
   //input the points

   minx = maxx = ptarr[0].x; miny = maxy = ptarr[0].y;
   for( int i=1; i<n; i++)
   {
      if(ptarr[i].x < minx)
         minx = ptarr[i].x;
      if(ptarr[i].x > maxx)
         maxx = ptarr[i].x;
      if(ptarr[i].y < miny)
         miny = ptarr[i].y;
      if(ptarr[i].y > maxy)
         maxy = ptarr[i].y;
   }
   printf("Rectangle is left = %d right = %d, top = %d bottom = %d\n",minx, maxx,
miny, maxy);
  return 0;
}
```

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);
        r
}
```

1. make_point(x,y) creates a struct point given coordinates (x,y).

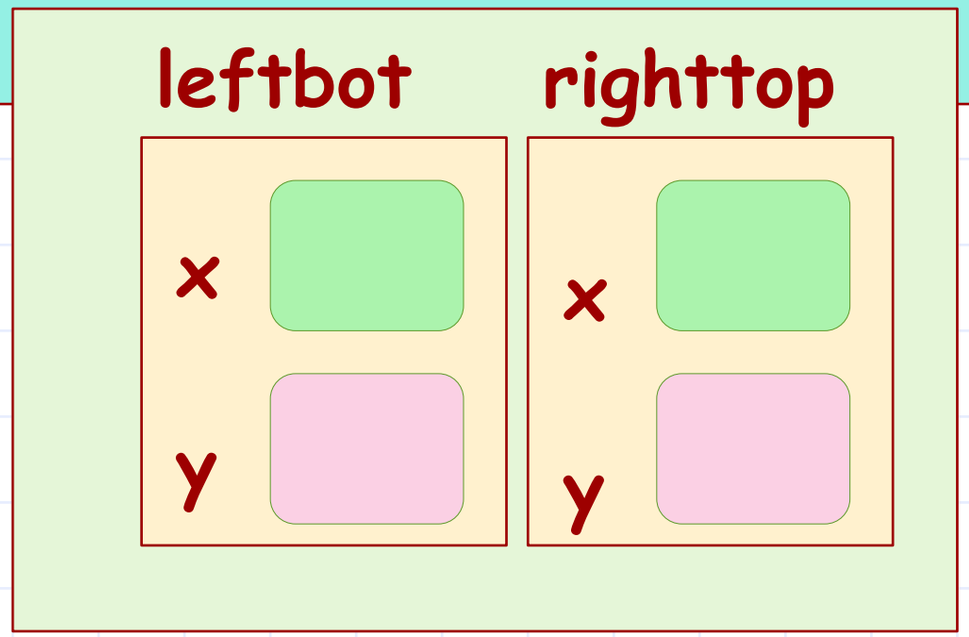
2. Note: make_point(x,y) returns struct point.

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

4. We can also pass struct parameters. struct are passed by copying the

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

```
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() {
    struct rect r1 ={{0,0}, {1,1}};
    area(r1);
} r
```

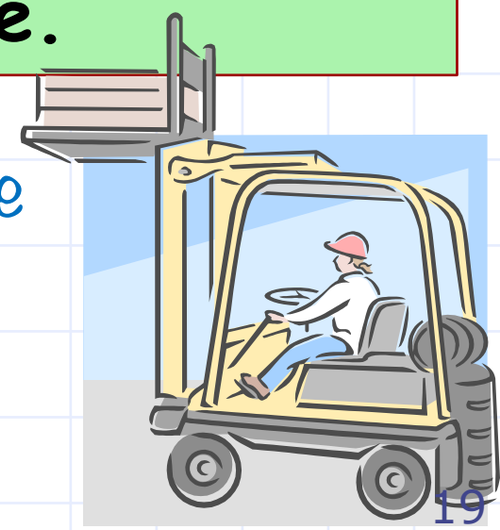


Passing structures..?

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.

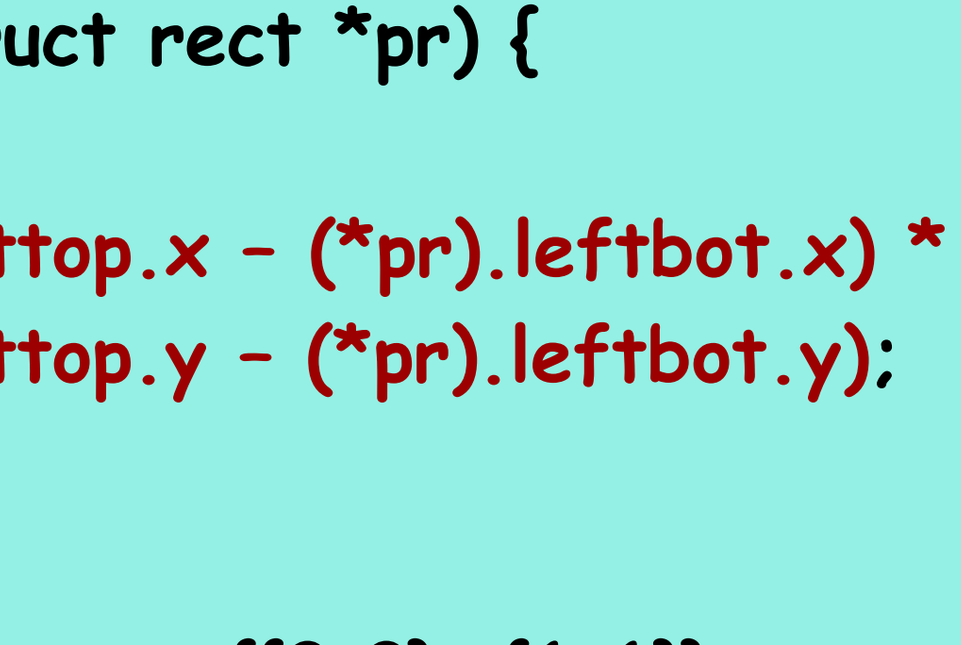


So what should be done to pass structures to functions?

Same for returning 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() {
   struct rect r ={{0,0}, {1,1}};
   area (&r);
}
```

r



Passing structures..?

Instead of passing structures, pass pointers to structures.

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

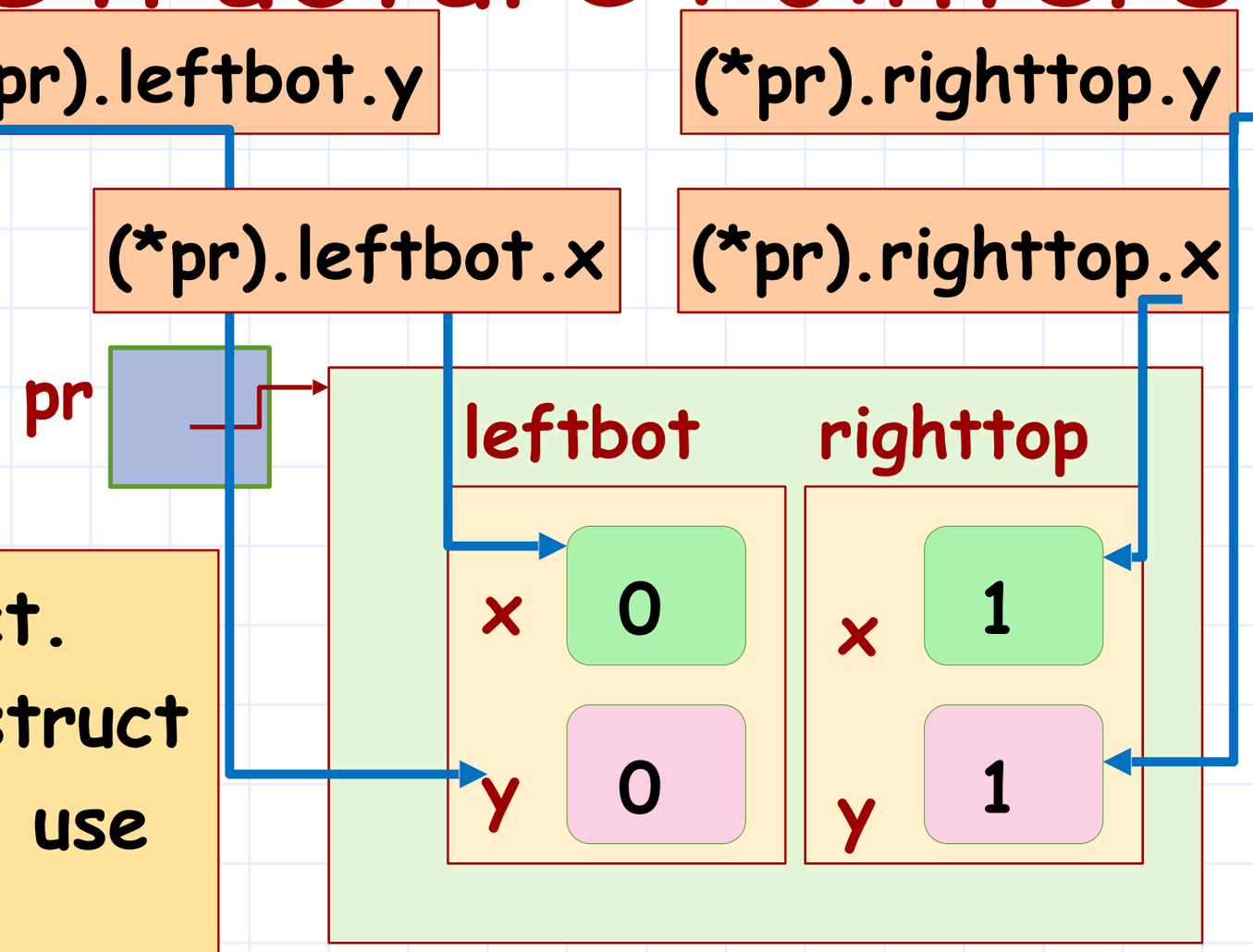
Now only one pointer is passed instead of a large struct.

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

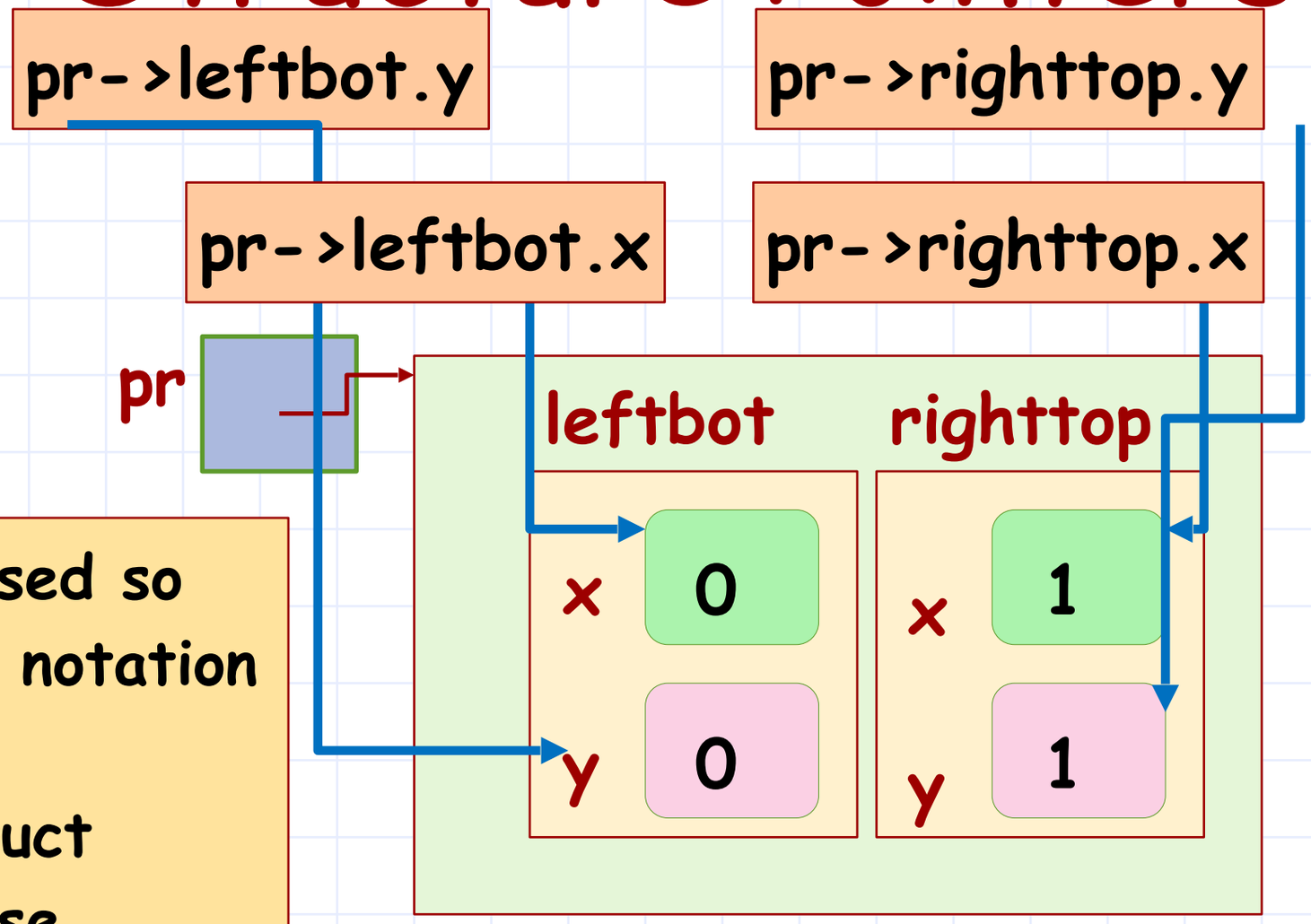


Addressing fields
via the structure's pointer

Structure Pointers

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

1. Pointers to structures are used so frequently that a shorthand notation (->) is provided.
2. To access a field of the struct pointed to by struct rect, use
   `pr->leftbot`
3. -> is one operator. To access the x field of leftbot, use
   `pr->leftbot.x`
3. -> and . have same precedence and are left-associative. Equivalent to
   `(pr->leftbot).x`



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

Addressing fields via the structure's pointer (shorthand)

Passing struct to functions

- 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