```c
#include <stdio.h>
#include <stdlib.h>

/**********
Structures are neat ways of collecting two or more variables into a nice
package so as to create a new datatype
**********/

// The structure Point has two feilds
// The first field is an integer called x
// The second field is another integer called y
struct Point{
    int x;
    int y;
};

// Structures can have any number of fields of any types
struct Misc{
    int a;
    float b;
    char c;
    double d;
};

// The structure Point3D has three feilds
// All three feilds are integers and are named x, y, and z
struct Point3D{
    int x, y, z;
};

// Structure fields can be structures themselves
// The structure Rectangle has two fields. Both the fields are Point
// variables. The first Point variable is called UR, the next is called LL
struct Rectangle{
    struct Point UR;
    struct Point LL;
};

// The structure Cuboid has two Point3D fields with names Corner1, Corner2
struct Cuboid{
    struct Point3D Corner1, Corner2;
};

// scanf cannot read user-defined structures directly
// We need to write a function to read structures ourselves
// scanPoint takes the address of a Point variable, reads data from
// input and writes it into the fields of that Point variable
void scanPoint(struct Point *ptr){
    scanf("(%d, %d)", &(ptr->x), &ptr->y);
    // The above is a shortcut for
    // scanf("(%d, %d)", &((*ptr).x), &((*ptr).y));
    // In general, the -> operator is a neat shortcut to first
    // dereference an address, then access a certain field i.e.
    // ptr->x is a shortcut for (*ptr).x

    // WARNING: (*ptr).x is not the same as *ptr.x since the . operator
    // has a higher precedence than the * operator
    // *ptr.x would be interpreted as *(ptr.x) which may cause a segfault
}

// Can write beautiful code to read a Rectangle variable from the input
// by reading two Point variables. Can reuse the scanPoint function here.
void scanRectangle(struct Rectangle *ptr){
    scanf("[");
    scanPoint(&ptr->LL);
    // Recall that the -> operator is a shortcut
    // The above statement is exactly equivalent to
    // scanPoint(&((*ptr).LL));
    scanf(":");
```

```
    scanPoint(&ptr->UR);
    scanf("]");
}

// Find the area of a Rectangle
int findArea(struct Rectangle r){
    int area = (r.UR.x - r.LL.x) * (r.UR.y - r.LL.y);
    r.UR.x = 100;
    return area;
}

int main(){
    // Can declare structure variables just as simple int variables
    int a, b;
    struct Point p, q;
    struct Rectangle r;
    // We have decided that a Point variable must be formatted as
    // (x, y)
    // in the input. Try giving input (3, 4) and see what happens.
    // Also try violating this format and give input as (3 4) and see.
    scanPoint(&p);
    printf("Point p = [%d %d]", p.x, p.y);
    // We have decided that a Rectangle variable must be formatted as
    // [(x1, y1):(x2, y2)]
    // Give various input and see what happens
    scanRectangle(&r);
    printf("Rectangle [%d %d %d %d]\n", (r.LL).x, r.LL.y, r.UR.x, r.UR.y);
    printf("Area is %d\n", findArea(r));
    printf("Rectangle [%d %d %d %d]\n", (r.LL).x, r.LL.y, r.UR.x, r.UR.y);
    // sizeof operator can work a bit funny with structures but it will
    // always give num of bytes used to store a variable of type Cuboid
    printf("size of rect is %d", sizeof(struct Cuboid));

    // Can declare arrays of structures too
    // Can declare static arrays
    int n = 20;
    int intArr[n];
    struct Rectangle rectArrStatic[n];
    // ... as well as dynamic arrays. The syntax is exactly the same
    // as that used to declare a simple integer array
    int *intPtr = (int*)malloc(n * sizeof(int));
    struct Rectangle *rectPtr = (struct Rectangle*)malloc(n * sizeof(struct Rectangle));
    // Can access ith element of an array of structures using []
    struct Rectangle temp = rectPtr[2];
    // Rule 5 of pointers applies here too - array name is just a
    // pointer to first element. Thus, rectArrStatic points to the
    // Rectangle variable rectArrStatic[0]
    temp = *(rectArrStatic + 5);
    return 0;
}
```