```c
#include <stdio.h>

// Use the typedef shortcut in case you wish to avoid writing
// struct Point again and again. Once you do this, you can just
// write Point as a bonafide datatype as you can write int
typedef struct Point{
    char c;
    int x, y;
}Point;
// Note that the name of the structure i.e. "Point" is repeated
// twice. This is not necessary and the above is just a very common
// idiomatic use of the typedef command. Look up books or websites
// in case you are interested in learning more.

// A line is made up of two points. We can use the typedef command
// to avoid writing struct Line again and again. After this, we just
// have to write Line and not struct Line everywhere.
typedef struct Line{
    Point p1, p2;
}Line;

// The function takes in a Line variable (note we did not have to write
// struct Line l as the input to the function) and prints one of its
// subfields and then changes. However, this does nothing to the original
// Line variable that was passed to the function (Rule 4 of functions)
void fun(Line l){
    printf("The char variable is %c\n", l.p2.c);
    l.p2.c = 'x';
}

int main(){
    // Can initialize structure variables at time of declaration
    Point p = {'a',10,30};
    // Can do so even if some fields are structures themselves
    Line l1 = {{'a',1,2},{'b',4,5}};

    // The sizeof operator can sometimes give a size of a structure
    // variable that is larger than the sum of the sizes of the fields
    // in that structure - this is due to padding done by the compiler
    // to obey demands made by the microprocessors. However, if you use
    // sizeof operator during malloc/calloc/realloc, you never have to
    // worry about this weird feature.
    printf("Size of Point is %d and that of Line is %d\n", sizeof(Point), sizeof(Line));


    // The assignment operator, when used with structure variables,
    // copies every field individually. If those fields happen to be
    // structure variable themselves, their fields are also copied one
    // by one - it is quite a thorough process :)
    Line l2 = l1;
    // I can change l1 but l2 will not get affected since I copied l1 to l2
    l1.p2.c = 'p';
    // Rule 4 of functions applies as is to structure variables
    fun(l2);
    printf("p2 looks like {%c %d %d}\n", l2.p2.c, l2.p2.x, l2.p2.y);
    // Can declare an array of Line variables just as we could declare
    // an array of int variables
    Line lineArr[5];
    // The [] and . operators are both left associative. Moreover, the two
    // operators [] and . have the same precedence.
    lineArr[2].p1.x = 90;
    // The above expression is equivalent to
    // ((lineArr[2]).p1).x = 90;
}
```