CS 162C++
More Data Structures

As with any other data type, you can create pointers to structures, group them in arrays, and dynamically allocate memory for them on the heap.

Pointers to Structs

Lets consider our data structure **t_move**:

```
struct t_move {
     int row;
     int col;
};
```

Previously, we created a variable of type **t_move** and named it **playerMove**:

```
t_move playerMove;
```

Now we can create a pointer to a **t_move** and call it **movePtr**:

```
t_move *movePtr;
```

As shown before, you can access the contents of **playerMove** using a combination of the ***variable name*** and the two ***member names***, ***using dot notation***.

```
playerMove.row = 12;
playerMove.col = 22;
```

It is very similar when you are using a pointer, but you have to dereference the pointer before ***using dot notation***.

```
(*movePtr).row = 12;
(*movePtr).col = 22;
```

Note that you have to use parentheses to indicate that you are dereferencing **movePtr** and not **movePtr.row**. In other words, dot notation has higher precedence than dereferencing.

There is an alternate notion that is equivalent, but uses **->** instead of ***dot notation***.

```
*movePtr->row = 12;
*movePtr->col = 22;
```

As with any other pointer type, you can use them to pass by reference, you can pass pointers to functions and you can return pointers from functions as shown below in the section on dynamic allocation of data structures.

Arrays of Structs

As with any other data type, you can create an array of structs:

```
const int SIZE = 10;
t_move playerMoves[SIZE];
```

Then you access the elements of an item in the array using a combination of ***dot notation*** and ***indexing***.

```
playerMoves[3].row = 12;
playerMoves[3].col = 22;
```

Example

For a program that works with arrays of rectangles, passing their dimensions with a struct

```
#include <iostream>

using namespace std;

    // global definition of rectangle for later use
struct t_rect {
    int width;
    int length;
};

    // function declarations, definitions below
t_rect getRectangle();
void showArea(t_rect aRect);

    // main program, does example work
int main()
{
    const int SIZE = 10;
    t_rect myRects[SIZE];

    for(int i = 0; i < SIZE; i++)
        myRects[i] = getRectangle();

    for(int i = 0; i < SIZE; i++)
    {
        int area = myRects[i].length * myRects[i].width;
        cout << "For rectangle " << i << " the area is " << area << endl;
    }

    return 0;
}

    // get the length and width of a rectangle
t_rect getRectangle()
{
    t_rect data;
    cout << "Enter width: ";
    cin >> data.width;
    cout << "Enter length: ";
    cin >> data.length;

    return data;
}
```

Dynamic Allocation of Structs

As with any other data type, you can allocate memory on the heap for a struct

```
t_move *movePtr = new t_move;
```

Then you access the elements of this new t_move instance using pointer notation as above.

```
(*movePtr).row = 12;
(*movePtr).col = 22;
```

or

```
*movePtr->row = 12;
*movePtr->col = 22;
```

When you are done, you must free up the allocated memory using delete:

```
delete movePtr;
```