Lecture 4.2

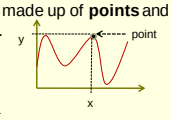
C-Style Structs

What are structures?

- A structure is a complex data type built by using elements of other types.
- A structure provides a way of grouping variables under a single name for easier handling and identification.
- Structures may be copied to and assigned. They are also useful in passing groups of logically related data into functions.

Why use structures?

- Consider a program to manipulate the x and y coordinates of a graph.
- Two known ways:
  - Create variables for each x and y coordinate; i.e. `float x1; float y1; float x2; float y2; float x3; float y3;…float x100; float y100;`
  - Use parallel arrays to hold x and y coordinates; i.e. `float x[100]; float y[100];`
- It could be easily noted that an `x` and `y` value makes up a ***point*** on the graph.
- Hence, it could be said that a graph is made up of **points** and each **point** has an **x** and **y** coordinate.



How to declare a structure?

- A structure is declared by using the keyword ***struct*** followed by a ***structure tag*** followed by the body of the structure containing the structure ***members***. E.g.

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

struct keyword

structure tag – defines the name of the structure type

structure members – defines the characteristics or attributes of the structure; i.e. a point has an x and y coordinate part

How can I use this?

- The preceding structure definition does not reserve any space in memory; rather the definition just creates a new data type that is used to declare variables.
- The **struct** declaration is a ***user-defined data type***.
- This means that with a `struct`, YOU can, define YOUR OWN data types.
- Structure variables are declared like variables of other types:
  - `point left, right;` *// left and right are **objects (or // instances)** of type **point***
- is analogous to
  - `float rate;` *//rate is an instance (object)of type **float***

Accessing Members of the Structure

- The individual members of a structure can be accessed using the **dot operator** ".", which is a special type of a **member access operator**.

Syntax

object.membername;

Object or instance of structure

Dot operator

Name of the variable member to be accessed

E.g.

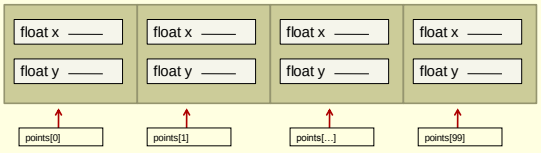
```
point location;
location.x = 2.5;
location.y = 3.5;
cout << "The x coordinate is "<< location.x;
cout << "The y coordinate is "<< location.y;
```

Arrays of Structures

- Let's refer to our program to manipulate the x and y coordinates of a graph.
- Instead of creating two arrays for the x and y coordinates, we could create an array of points, since, we now have a data structure that could hold two floating point number in one structure.
- That is declare

```
point points[100]; //this creates an array of size 100 of
                   //type point; i.e. 100 point elements
```


Memory representation of Struct



Accessing Members of Array

- Arrays are grouping of objects of the same type. Thus, and array of points, e.g. `point points[100];` will have a `point` **object** in each element of the array, and each `point` object has an **`x`** and **`y`** data member.
- E.g. to display all points

```
for (int i=0; i < 100; i++){
   cout << "x" << i << " is "<< points[i].x << endl;
   cout << "y" << i << " is "<< points[i].y << endl;
 }
```

`points[i]` reference the point object stored in the i^{th} index of `points` array. "`.y`" accesses the `y` data member

Nesting structures

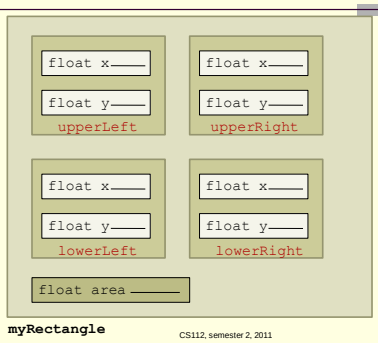
- A structure could also be composed of another structure
- A rectangle could be represented as follows.

```
struct Rect
{
    point upperLeft;
    point upperRight;
    point lowerLeft;
    point lowerRight;
    float area;
 };
```

  - This means that each `Rect` object will have 5 components
- To instantiate a `Rect` object we do:

```
Rect myRectangle;
```


Memory representation of nested Struct



Accessing nested structs

- To access the members of the struct within a struct, we use the "dot" operator in the nested manner:

```
myRectangle.upperLeft.x = 1.0;
myRectangle.lowerRight.y = 3.0;
```

`myRectangle` is the object that we want to access. Remember each `Rect` object has 5 components, thus we need to use the dot operator

`lowerRight` is the member object that we want to access within the `myRectangle` object. Remember `lowerRight` is a `point` object and each `point` object has 2 components

`y` is the member object that we want to access within the `lowerRight` object. Remember `y` is a `float` object and, thus, has no parts.

Pointers to Structs

- Structure instances could be manipulated via pointers to the structure type.
  - E.g.
```
point location;
point *ptrloc = &location; //ptrloc pointing
                           //to object location
```
- Two ways to access data members:
  - **Dot operator (.)**
```
(*ptrloc).x = 1.0;
```
Dereferenced object | Dot operator | Data member
  - **Arrow operator (->)**
```
ptrloc->y = 2.0;
```
Pointer | Arrow operator | Data member

Pointers to Structs

- E.g.

```
point location;
point *ptrloc = &location;
location.x = 2.5;
ptrloc->y = 3.5;
cout << "The x coordinate is "<< location.x << endl;
cout << "The y coordinate is "<< (*ptrloc).y << endl;
```


Structs to Functions

- Since structs are data types (even though user-defined), they could be passed to functions in the same manner as the primitive data types (such as int or float) are passed to functions.
- Pointers to structs are passed to functions in the similar fashion as with the primitive types

Structs to Functions

```
void print_point(point p){ //receiving object
   cout << "The x coordinate is "<< p.x << endl;
   cout << "The y coordinate is "<< p.y << endl;
}

void print_point(point *ptr){ //receiving pointer
   cout << "The x coordinate is "<< ptr->x << endl;
   cout << "The y coordinate is "<< ptr->y << endl;
}

int main(){
   point location;
   point *ptrloc = &location;
   location.x = 2.5;
   ptrloc->y = 3.5;

   print_point(location); //object passed
   print_pointptr(ptrloc); //pointer passed
   print_pointptr(&location); //address of object location passed
}
```