Structure

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
- It is like class in C++/Java, without methods.

```c
struct point {
    int x;
    int y;
};
```
- Define a structure type:

```c
struct point { int x, y; } first, second;
```
- Define variables:

```c
struct point first, second;
```
- Define struct type and declare variables:

```c
struct point {
    int x, y;
};
```
- Use Structure

- Similar to Java objects, we use dot (.) to access structure members:

```c
struct point coordinate;
coordinate.first, second;
```
- In some compilers, and all C++ compilers, you can usually simply say just:

```c
point first, second;
```
- Initialize a struct variable:

```c
struct point first = { 20, 50 };```
- You can even use a typedef if your don’t like having to use the word “struct”

```c
typedef struct point coordinate;
```
- typedef coordinate first, second;

- Can not compare two struct variables using ==

```c
if ( first == second) //compile time error
```

Use Structure

- Point x=20
- Point y=50
- Point x=20
- Point y=50
- In some compilers, and all C++ compilers, you can usually simply say just:

```c
point first, second;
```
- Initialize a struct variable:

```c
struct point first = { 20, 50 };```
- You can even use a typedef if your don’t like having to use the word “struct”

```c
typedef struct point coordinate;
```
- typedef coordinate first, second;

- Can not compare two struct variables using ==

```c
if ( first == second) //compile time error
```
Nested Structure

- Structures can be nested. For example,
  ```c
  struct rectangle {
    struct point topleft;
    struct point bottomright;
  };
  ```
- Define a rectangle struct variable:
  ```c
  struct rectangle mybox;
  ```
- Initializing the points:
  ```c
  mybox.topleft.x = 0;
  mybox.topleft.y = 10;
  mybox.bottomright.x = 20;
  mybox.bottomright.y = 0;
  ```

Function Returning Array

- Functions **cannot** return local arrays variables, which have **local scope**.
- The following won’t work!!
  ```c
  char *itoa(int n) {
    char retbuf[25];
    //char *retbuf = malloc(25);
    //if(retbuf == NULL) return NULL;
    sprintf(retbuf, "%d", n);
    return retbuf;
  }
  ```

Function Returns Structures

- Functions **can** return local **struct variable**, which can be assigned to a variable:
  ```c
  struct point a = makepoint(10,10);
  ```
- This is because a new copy of the local struct variables is returned from the function.
  ```c
  struct point makepoint(int x, int y) {
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
  }
  ```

Passing Structure as Function Parameter

- Structure parameters are passed by values:
  - The entire struct argument is copied over to the struct parameter.
  - The changes made in the struct argument are not visible outside the function.
  ```c
  struct point addpoints(struct point p1, struct point p2) {
    p1.x += p2.x; p1.y += p2.y; return p1;
  }
  ```
**Pointer to Structures**

- If a large structure is to be passed to a function, it is more efficient to pass a **pointer** than to copy the whole structure.
- Structure pointers are just like pointers to ordinary variables (not array).

```c
struct point first, *pp;
pp=&first;
int value=(*pp).x;
//same as first.x
value=pp->x  //same as (*pp).x
```

**Function Return Struct Pointer**

- The local variable goes out of scope when exits the function.
- Compiler would flag a warning.
- The result is unreliable.

```c
struct point* a = makepointer(10, 10);
```

**Passing Structure Pointer as Function Parameter**

- All changes made on the struct parameters inside the function are carried over outside of the function:

```c
struct point w={2,3};
struct point v={1,1};
struct point *p;
addpointers(&w, &v);
//w.x?, w.y?
struct point *q;
addpointers(&w, &v);
//w.x?, w.y?
```

```c
//addpointers: add two points
struct point * addpointers (struct point* p1, struct point* p2)
{
 p1->x += p2->x;
p1->y += p2->y;
return p1;
}
```

**Self-referential Structures**

```c
struct node{
    int data;
    struct node *next;
};
struct node add(struct node* p, int value)
{    
 if (p==NULL) {
    p=(struct node*) malloc(sizeof(struct node));
p->data=value;
p->next=NULL;
    }
else{
    struct node* newNode=(struct node*) malloc(sizeof(struct node));
    newNode->data=value;
    newNode->next=p;
    p=newNode;
    return p;
    }
}
main(){
    struct node* head;
    head=add(head, 10);
    head=add(head, 20);
    head=add(head, 30);
    }
```