Part V  Structures  (Summary)

**Structure** = a collection of data items that may not be all of the same type

**Simple Structure**

 tanımlayıcı yapılar

Structure Template

```
struct struct_name
{
    type item_name1;
    ...
    type item_name_n;
};
```

Example:

```
struct car_template
{
    int year;
    char number[9];
    char owner[20];
    char colour[10];
    char brand[10];
};
```

OR

Typedef Structure Type

```
typedef struct struct_name
{
    type item_name1;
    ...
    type item_name_n;
}struct_type_name;
```

Example:

```
typedef struct car
{
    int year;
    char number[9];
    char owner[20];
    char colour[10];
    char brand[10];
} car_type;
```
Declaring Structures

Once the structure is defined, you can declare a structure variable.

Structure Template

```
struct struct_name variable_name;
```

Example:
```
struct car_template my_car;
```

OR

Structure Type

```
struct_type_name variable_name;
```

Example:
```
car_type my_car;
```

Addressing an Item of a Structure

An item from a structure can be referenced by following the name of the structure by a dot and the name of the item.

```
variable_name.item_name;
```

Example:
```
my_car.year
```
✧ **Initialisation of the Structure**

- **at declaration**:
  
  *complete initialisation or partial initialisation*:

  ```c
  car_type my_car = {2005, "05D201", "John Joe", "silver", "BMW"};
  ```

- **during execution**

  *with / without user input => complete initialisation or partial initialisation*:

  ```c
  scanf("%d", &my_car.year);
  scanf("%s", my_car.owner);
  ...
  
  OR
  ```

  ```c
  my_car.year = 2006;
  strcpy(my_car.owner, "Paul");
  ...
  ```
Nested Structure

Structures can contain other structures as items
In principle, structures can be nested indefinitely

typedef struct no_car
{
    char year[2];
    char county[2];
    int no;
} car_no_type;

typedef struct car
{
    int year;
    car_no_type number;
    char owner[20];
    char colour[10];
    char brand[10];
} car_type;

diagram:

<table>
<thead>
<tr>
<th>car</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
</tr>
<tr>
<td>number</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>owner</td>
</tr>
<tr>
<td>colour</td>
</tr>
<tr>
<td>brand</td>
</tr>
</tbody>
</table>

Usage

car_type my_car;
my_car.year = 2005;
strcpy (my_car.number.year, “05”);
strcpy (my_car.number.county, “D”);
my_car.number.no = 2200;
strcpy (my_car.owner, “John Doe”);
strcpy (my_car.colour, “silver”);
strcpy (my_car.brand, “BMW”);
Arrays of Structures

- **Declaration**

  The structure must be defined before an array of structures is declared.

**When a Structure Template definition is used**

```
struct struct_name array_name[size];
```

Example:

```
struct car_template cars[10];
```

OR

**When a Structure Type definition is used**

```
struct_type_name array_name[size];
```

Example:

```
car_type cars[10];
```
Dynamic Memory Allocation

- Declaring the pointer (reference)

```c
struct_type_name* ref_name;
```

Example:
```
car_type* ptr;
```

- Memory Allocation

```c
ref_name = (struct_type_name*) malloc(sizeof(struct_type_name));
```

Example:
```
ptr = (car_type*) malloc(sizeof(car_type));
```

- Accessing Info from the Memory Location

```c
(*ref_name).item_name;

OR

ref_name->item_name;
```

Example:
```
ptr ->year   or   (*ptr).year
```

- Free the Allocated Memory

```c
free(ref_name);
```