By SAHIL

Structures and Unions

INTRODUCTION

1. We note that an array is the collection of data items of same type, such as int, float or char.
2. Arrays cannot be used to represent the data items of different types.
3. As a solution to this C supports another constructed or user defined data type called as structure.
4. The concept of structure is analogous to that of a record in many other languages.
5. Structures help to organize complex data in a more convenient way.

STRUCTURE

It is a convenient tool for handling a group of logically related data items.

Examples:

student	:	student_name, roll_number, dob, marks
time	:	seconds, minutes, hours
date	:	day, month, year
book	:	author, title, price, year
city	:	name, country, population
address	:	name, house_number, street, city
customer	:	name, telephone, city, category

DEFINING A STRUCTURE

1. Unlike arrays, structures must be defined first for their template that may be used later to declare structure variables.
2. It takes the general form as:

```
struct tag_name
{
    data_type       member1;
    data_type       member2;
    data_type       member3;
    ……..……..       ……..……..
    ……..……..       ……..……..
    data type       memberM;
};
```

3. The above definition of structure tells the compiler following.
   - The keyword **struct** tells the compiler that **tag_name** (structure name) is a **structure.**
   - The defined structure (**tag_name**) has the **members** or **elements** as **member1, member2, member3….memberM** with the **type** as declared.

3. The individual member can be an ordinary variable, pointer, array or other structure.
4. The members name within a particular structure must be **distinct** from one another.

5. The name of a member is local to the structure i.e. a member name can be same as the name of another variable which is declared outside the structure.

Example:

```
struct student
{
    char    name[30];
    int     roll_number;
    int     total_marks;
    char    dob[10];
};
```

Important to note about the definition of structure:

- The template is terminated with a semicolon.
- The member declaration does not declare any variable i.e. the members of a structure do not occupy any memory until they are associated with the structure variables.

DECLARING STRUCTURE VARIABLES

1. After defining structure format, the variables of that type can be declared.
2. The declaration of a structure variable includes the following,
   a. The keyword **struct**.
   b. The structure **tag name**.
   c. List of variable names separated by commas.
   d. A terminating semicolon.

Example:

```
struct    student    student_1, student_2, student_3;
```

3. It is allowed to combine both the structure definition and variable declaration in one statement as,

```
struct     student
{
     char    name[30];
     int     roll_number;
     int     total_marks;
     char    dob[10];
} student_1, student_2, student_3;
```

ACCESSING STRUCTURE MEMBERS

1. We note that the structure members themselves do not represent variables. To represent meaningful variables they must be linked to the structure variables.
2. The link between a structure member and a structure variable is established using the **member operator** ' . ' which is also known as the **dot operator** or **period operator**.

Example:

```
student_1.name
```

Let us assign the values to each member of student_1.

```
strcpy(student_1.name, “WASEEM”);
student_1.roll_number = 49;
student_1.total_marks = 520;
strcpy(student_1.dob, “18\12\1989”);
```

STRUCTURE INITIALIZATION

There are several possible ways to initialize a structure as follows.

1

```
main( )
{
    struct st_record
    {
       int    weight;
       fioat  height;
    } student_1 = {60, 180.75};
    ……..
    ……..
}
```

2

```
main( )
{
    struct
    {
       int    weight;
       fioat  height;
    } student_1 = {60, 180.75};
    ……..
    ……..
}
```

3

```
main( )
{
    struct st_record
    {
        int      weight;
        fioat    height;
    };
    struct st_record student_1 = {60, 180.75};
    struct st_record student_2 = {53, 170.60};
}
```

4

```
struct st_record
{
    int      weight;
    fioat    height;
}student_1 = {60, 180.75};
main( )
{
    struct st_record student_2 = {53, 170.60};
    ..............
    ..............
}
```

Important to note about initialization of structure:

- Individual structure members cannot be initialized within the structure template.
- The order of values present in braces must match with the order of members in structure.
- It is permitted to partially initialize a structure. The uninitialized members should be at the end of the list in structure definition.
- The uninitialized members will be assigned default values as follows:
  - Zero for integer and floating point numbers.
  - ‘/0’ for characters and string

COPYING AND COMPARING STRUCTURE VARIABLES

1. Similar to an ordinary variables , two variables of the same structure type can be copied.

Example:

If person1 and person2 belong to the same structure then the following statements are valid:

```
person1 = person2;
```

```
person2 = person1;
```

2. Any logical operation on structure is not permitted in C. For examples following are the invalid statements:

```
person1 == person2;
```

```
person2 != person1;
```

OPERATIONS ON INDIVIDUAL MEMBERS

Any structure member associated with its variable through dot operator can be manipulated using expressions and operators.

Examples:

1

```
if (student1.number == 111)
    student1.marks += 10.00;
```

2

```
fioat sum = student1.marks + student2.marks;
student2.marks *= 0.5;
```

3

```
student1.number++;
++student1.number;
```

ARRAY OF STRUCTURES

1. We note that after defining a template for a structure, structure variables of that type can be declared, and using member operator with them different structure members can be accessed.
2. Similarly, it is permitted in C to declare an array variable of a particular structure type.
3. When an array variable of a particular structure type is declared, each of its element represent a structure variable.

Marks analysis with different subjects of a class of 100 students.

Example:

```
struct class student[100];
```

Consider the following declaration,

```
struct marks
{
    int   subject1;
    int   subject2;
    int   subject3;
};
main( )
{
    struct marks student[3] = {{45, 68, 81}, {75,
                                        53, 69}, {57, 36, 71}};
}
```

The above declaration declares the student as an array of three elements student[0], student[1], and student[2] and initializes their members as follows.

```
student[0].subject1 = 45;
student[0].subject2 = 65;
……..……..
……..……..
student[2].subject3 = 71;
```

ARRAY WITHIN STRUCTURES

C permits the declaration of an array as a structure member.

Example:

```
struct marks
{
    int number;
    float subject[3];
}student[2];
```

Here, the member **subject** contains three elements, **subject[0]**, **subject[1]** and **subject[2]**. These elements can be accessed using appropriate subscripts. For example,

```
student[1].subject[2];
```

would refer to the marks obtained by second student in subject 3.

STRUCTURES WITHIN STRUCTURES

1. C permits nesting of structures i.e. to declare a structure as a member of a structure.

Consider the structure defined to store information about the salary of employees:

```
struct salary
{
    char name[20];
    char department[20];
    int basic_pay;
    int dearness_allowance;
    int house_rent_allowance;
    int city_allowance;
}employee;
```

Now all the items related to allowance can be grouped together and can be declared under a substructure as follows.

```
struct salary
{
    char name[20];
    char department[20];
    int basic_pay;
    struct
    {
        int dearness;
        int house_rent;
        int city;
    }allowance;
}employee;
```

- Here the structure salary contains one of its members as allowance, which itself is a structure with three members.
- The members contained in inner structure can be referred to as:

```
employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city
```

Note:

To access the members of innermost structure it is must to chain all t concerned structure variables (from outermost to innermost) with t member using dot operator.

For above nested structures the following are invalid:

```
employee.allowance
```

(actual member is missing)

```
employee.house_rent
```

(inner structure variable is missing)

INTRODUCTION TO UNIONS

1. The concept of unions in C is similar to that of structures except a key difference in terms of storage.
2. We know that each structure member has its own memory location but all the members of a union share the same allocated memory location of largest type size.
3. Therefore unlike structures only one union member can be handled at a time.
4. As one union member can be handled at a time, only one union member can be assigned a data value at time.

5. When a different member is is assigned a new value, the new value supersedes the previous member's value.

DEFINING A UNION

Similar to structures, unions must be defined first for their template as follows that may be used later to declare structure variables.

```
union item
{
    int number;
    char name[10];
    float price;
}code;
```

UNION INITIALIZATION

Similar to a structure variable, a union member can be initialized when it is declared but with a value of the same type as the first union member.

For example, the declaration

```
union item abc = {100};
```

is valid but the declaration is invalid.

```
union item abc = {10.75};
```

Accessing values of Union

In c we access the values of union members using dot operator[.] similar as structure.

```
union  student{
    char name[100];
    int age;
}s1;
```

```
printf("%s\n",s1.name);
printf("%d\n",s1.age);
```

Difference between Structure & Union

Structure (struct)	Union (union)
Each member has its own memory.	All members share the same memory.
All members can store values simultaneously.	Normally, only one member's value is used at a time.
Size is approximately the sum of member sizes, including padding.	Size is enough for the largest member, including alignment padding.
Changing one member does not affect the others.	Writing to one member can overwrite the representation of another.
Uses more memory in many cases.	Can save memory when members are used alternatively.

By SAHIL

Thank You