Chapter 8.
C Structure
Lecture plan

1. Structures in C
2. Declaring Structure Variables
3. Declaring Nested Structures
4. Structure initialization
5. Accessing structure members
6. Pointers to structures
7. Structures and functions
8. Structures and arrays
9. Examples
A structure is a collection of logically related data items grouped together under a single name, called a *structure*. The data items that make up a structure are called its *members*, components or fields, and can be different type.

The general format for defining a structure is

```c
struct StructureTypeName { 
    StructureMemberDeclarationList;
};
```
Structures in C

- In this syntax the **StructureTypeName** is also called the *structure tag* and it is a programmer-defined identifier.

- The **StructureMemberDeclarationsList** is a list of declaration of its members.

- Suppose we want to declare a structure type called **student_struct**, which has its members the data items **student_idno**, **student_name**, **age** and **birth_year**.
Structures in C

struct student_struct
{
    long student_idno;
    char student_name[20];
    int age;
    int birth_year;
};
All the members of a structure can be of the same type, as in the following definition of the structure date:

```
struct date
{
   int day, month, year;
};
```

Two member variables in different structures can have the same name.
Structures in C

Example:

```c
int age;
/* no conflict with age in the structure
student_struct */
struct currentday{
    int day;  /* no conflict with the day in the
    int year;  structure date */
};
```

- Each occurrence of a structure definition introduces a new structure type that is neither the same nor equivalent to any other type.
Declaring Structure Variables

After declaring a structure type, we may declare variables that are of that type. A structure variable declaration requires,

- The keyword `struct`
- The structure type name
- A list of variable names separated by commas
- A concluding semicolon

To declare a variable `student_record` as a variable of type `student_struct` can be written as
Declaring Structure Variables

```c
struct student_struct student_record;
```

It is also possible to combine the declaration of a structure type and structure variable by including the name of the variable at the end of the structure type declaration.

```c
struct student_struct
{
    long student_idno;
    char student_name[20];
    int age;
    int birth_year;
} student_record;
```
Thus, given that

\[
\begin{align*}
\text{struct } s1 & \{ \text{ char } c ; \text{ int } i ; \} \ u ; \\
\text{struct } s2 & \{ \text{ char } c ; \text{ int } i ; \} \ v ; \\
\text{struct } s3 & \{ \text{ char } c ; \text{ int } i ; \} \ w ; \\
\text{struct } s4 & \{ \text{ char } c ; \text{ int } i ; \} \ x ; \\
\text{struct } s4 & \ y ; \\
\end{align*}
\]

The types of \( u \), \( v \), \( w \), and \( x \) are all different, but the types of \( x \) and \( y \) are the same.
Declaring Nested Structures

Members of a structure declaration can be of any type, including another structure variable. Such a structure type is called nested.

Suppose we have the following structure declaration.

```c
struct date
{
    int day, month, year;
}
```
Declaring Nested Structures

We can use this structure type and rewrite the structure declaration

```c
struct student_struct {
    long student_idno;
    char student_name[20];
    struct date birth_date;
    int age;
    float weight;
};
```
Declaring Nested Structures

In declaring a nested structure type, we should be careful to place the declaration of the inner structure type before the nested structure type. If the inner structure type will be used only in a nested structure type, the two declaration can be combined.

```c
struct student_struct{
    long student_idno;
    char student_name[20];
    struct {int day, month, year; } birth_date;
    int age;
    float weight;
};
```
Structure initialization

A variable of a particular structure type can be initialized by the following definition

```c
struct date
{
    int day, month, year;
}
birth_date = {31, 12, 1988};
```
Initializes the member variables `day`, `month`, and `year` of the structure variable `birthdate` to 31, 12, 1988 respectively, and the declaration

```
struct date republic = {29, 10, 1922};
```

Initializes the member variables `day`, `month`, and `year` of the structure variable `republic` to 29, 10, and 1922 respectively. Both of the structure variables `birthdate` and `republic` allocates different memory locations of the storage.
Structure initialization

If there are fewer initializers than there are member variables in the structure, the remaining member variables are initialized to zero.

```
struct date newyear = {1, 1};
```

is the same as

```
struct date newyear = {1, 1, 0};
```
Accessing Structure Members

C provides a special operator, the **structure member or dot operator**, to access the individual members of a structure variable by the construction of the form

\[
\text{structure-variable.member-name}
\]

Thus, the statements

```c
struct date man_on_moon;
man_on_moon.day = 20;
man_on_moon.month = 7 ;
man_on_moon.year = 1969 ;
```
Accessing Structure Members

Set the values of the member variables `day`, `month`, and `year` 20, 7, and 1969 respectively.

And the statement can be accessed by

```c
struct date today;

if (today.day == 4 && today.month == 10 && today.year == 1988)
    printf("This is My birth Day ");
```
Accessing Structure Members

```c
struct date American, revolution = {4, 7, 1776};
```

the assignment

```c
American = revolution;
```

Assigns 4 to American.day, 7 to American.month, and 1776 to American.year.
Accessing Structure Members

The size of structure can be determined using `sizeof` operator.

For example, given the declaration:

```c
struct two { int x;
    char c[5];
    long m; };
```

the statement `i = sizeof(two);`

Assigns to `i` the size of the structure `two` in bytes. If you print the value of `i` it will be 17. Size of each variables `x`, `c` and, `m` will be 4, 5 and, 8 respectively.
A pointer to a structure identifies its address in memory, and is created in the same way that a pointer to a simple data type such as `int` or `char` is created. Consider the following declaration,

```c
struct student_struct {
    long  idno;
    char  name[20];
    int   age;
    float weight;
};
struct student_struct  student *pstudent;
```
Pointers to structures

The address operator & is applied to a structure variable to obtain its address. Thus, the assignment

\[
pstudent = \&\text{student} ;
\]

makes \text{pstudent} point to \text{student\_struct}.

The pointer variable \text{pstudent} can now be used to access the member variables of \text{student\_struct} using the dot operator as:

\[
(*\text{pstudent}).\text{idno}
\]
\[
(*\text{pstudent}).\text{name}
\]
\[
(*\text{pstudent}).\text{age}
\]
\[
(*\text{pstudent}).\text{weight}
\]

The parentheses are necessary because the precedence of the dot operator (\.) is higher than that of the dereferencing operator(*).
Pointers to structures

Pointers are so commonly used with structures. C provides a special operator, ->, called the *structure pointer or arrow operator*, for accessing members of a structure variable pointed by a pointer. The general form for the use of the operator ->

```
  pointer-name -> member-name
```

Thus, the preceding expression can be written as

```
pstudent -> idno
ptudent -> name
pstudent -> student_age
pstudent -> weight
```
Structures and Functions

1. A structure type definition may be local to a function or it may be external to any function. Structures may be passed as function arguments and functions may return structures.

2. C provides three methods of passing structures to a function.

The first method involves supplying structure members as the arguments in a function call.

To illustrate this method, we write a function for the following problem.

Computing the distance of point from the center of the circle and comparing this distance with radius of the circle.
Structures and Functions

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

struct circle{
    float r;       /* radius */
    struct point o ; /* center */
};

Recalling that the distance between two points (xc , yc) and (xp , yp) is given by:

\[
((xc – xp)^2 + ((yc – yp)^2)^(1/2)
\]

int contains (float cr , float cx , float cy, float px , float py)
{
    return pow(cx – px,2) + pow(cy – py,2 ) > pow(cr,2) ? 0 : 1 ;
}
```
Structures and Functions

Thus given that,

```c
struct circle c = { 2 , {1 , 1}};
struct point p ={ 2 , 2};
```

The function contains can be called as

```c
contains(c.r , c.o.x , c.o.y , p.x, p.y);
```

and it will return 1(true), since the distance of the point (2 , 2) from center of circle (1 , 1) is less than 2, the radius of the circle.
The second method involves passing the complete structure to a function by simply providing the name of the structure variables as the argument in the function call. The corresponding parameter in the called function must be of the same structure type. Thus using this method same function contains can be written as

```c
int contains(struct circle c , struct point p) {
    return pow(c.o.x – p.x,2) + pow(c.o.y – p.y,2) > pow(c.r,2) ? 0 : 1 ;
}
```

and called simply as

```c
contains(c , p);
```

Structure names are not pointers and are passed by value. Thus member variables of the formal parameters are not reflected in the corresponding structures variable in the calling function.
The third method involves passing pointers to the structure variables as the function arguments. Using this method, the function contains can be written as

```c
int contains(struct circe *c , struct point *p){
    return pow(c->o.x – p->x,2) + pow(c->o.y – p->y,2) > pow(c->r) ? 0 : 1 ;
}
```

and called as

```c
contains ( &c , & p)
```

The arrow operator has been used inside `contains` to access member variables, since `c` and `p` are now pointers to structures.
Structures and arrays

Arrays of structures are commonly used when a large number of similar records are required to be processed together. For example, if there were 500 students in a department, this can be organized in an array of structures as

```c
struct student_struct {
    long idno;
    char name[20];
    int grade;
};

struct student_struct allstudent[500];
```

This statement declares `allstudent` to be an array containing 500 elements of the type `student_struct`. 
Structures and arrays

A particular member variable inside an array of structures can be accessed using the array subscript and dot operator. Thus, the statement

\[
\text{allstudent}[k].\text{name} = \text{"ali"}; \\
\text{allstudent}[2].\text{grade} = 88;
\]

assigns “ali” to the \(k+1\)th structure elements \text{allstudent} and member variable \text{name} and, assigns 88 to the third structures elements \text{allstudent} and member variable \text{grade}. 
Example 1

```c
#include <stdio.h>
#define NUMRECS 5
struct PayRecord /* construct a global structure type */
{
    int id;
    char name[20];
    double rate;
};

int main()
{
    int i;

    for (i = 0; i < NUMRECS; i++)
        printf("%d %-20s %4.2f\n",
                employee[i].id,employee[i].name,employee[i].rate);

    return 0;
}
```
Example 2

```c
#include <stdio.h>

struct Employee {/* declare a global structure type */
    int idNum;
    double payRate;
    double hours;
};

double calcNet(struct Employee *); /* function prototype */

int main() {
    struct Employee emp = {6787, 8.93, 40.5};
    double netPay;
    netPay = calcNet(&emp); /* pass an address*/
    printf("The net pay for employee %d is $%6.2f\n", emp.idNum, netPay);
    return 0;
}

double calcNet(struct Employee *pt) {
    return(pt->payRate * pt->hours);
}
```