9.1 INTRODUCTION

A *structure* in C is a heterogenous user defined data type, similar to *records* in PASCAL. A structure may contain different data types. It groups variables into a single entity. An example of a structure declaration is shown below.

```
struct Student
{
   char name[25], address[80];
   int  age;
   float  height, weight;
};
```

The above declaration *groups* the `name`, `address`, `age`, `height` and `weight` of a person into an entity called `Student`. The declaration starts with the reserved word `struct`, followed by the name of the structure `Student`. Following this, enclosed within a pair of curly braces are a series of declarations. These declarations specify the members of the *structure*. In the above declaration, the structure `Student` consists of the following attributes:

- Two *strings*, `name` and `address`
- An *integer* variable `age`
- Two *floating point* numbers called `height` and `weight`

The identifier `Student` names the structure, and is called the *structure tag* or simply *tag*. The identifier `Student` is considered to be a new type defined by the user. No memory has been allocated to the structure so far. To use this structure in a program, we must define a structure variable as:

```
struct Student pers;
```

Note that the keyword `struct` must always precede the structure name (in this example `Student`) in variable declarations. `pers` is the name of a variable of type `Student`. It has memory allocated to it. After declaring `pers` to be of type `Student`, we can access the identifiers inside this structure. A variable of type `Student` contains the identifiers `name`, `address`, `age`, `height` and `weight`. These identifiers are accessed using the dot(.) operator. For example, to assign 65 to `age`, the statement,

```
pers.age = 65;
```

can be used. The identifier `pers.age` can be used just as an integer variable. For example, to input the age, the `scanf` statement,

```
scanf( "%i", &pers.age );
```

can be used. To input the member `name`, we use

```
scanf( "%s", pers.name );
```

As usual, the name of the string gives the address of its first character. This illustrates the fact that the identifiers inside a structure can be used as other variables, but must be preceded by the structure variable name and a dot(.).

To maintain information about 10 students, declare an array of `Student` structures. This is done with the statement,

```
struct Student persarray[ 10 ];
```

This indicates that `persarray` is an array of 10 `Student` structures. To set the `age` of the first person, use the statement,

```
persarray[ 0 ].age = 65;
```

`persarray[0]` is the first among the 10 `Student` structures. The expression `persasrray[0].age` accesses the `age` identifier of the first `Student` structure. Recall that arrays in C are numbered beginning with 0. Note that instead of maintaining information about 10 persons by declaring an array of 10 `Student` structures, we could have achieved the same thing using an array of 10 strings for the `name`, another array of 10 strings for the `address`, and so on. But using structures seems more natural and readable.

Consider another simpler version of the `Student` structure having the following members:

```
struct Student
{
   int Rollno;
   int subject;
   float  marks;
};
```

The syntax for the `Student` structure specifier is illustrated in Figure 9.1.

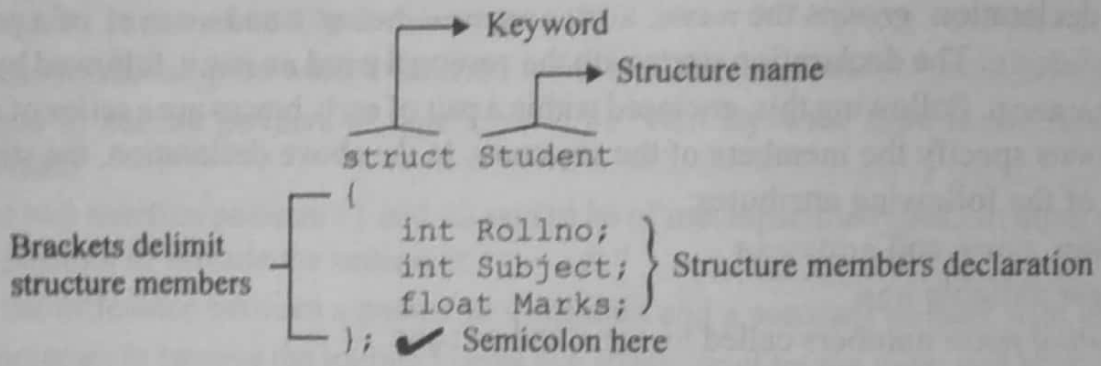

Fig. 9.1 Syntax of the structure Student

Rules for Declaration of Individual Members

The individual members of a structure may be any of the common data types (such as `int`, `float`, etc.), pointers, arrays or even other structures.

All member names within a particular structure must be different. However, member names may be the same as those of the variables declared outside the structure. Individual members cannot be initialized inside the structure declaration. For example, the following declaration is *invalid*.

```
struct Student
{
   int Rollno = 0; /* initialization not allowed in structure declaration, gives compiler error */
   int Subject;
   float marks;
};
```

Declaring Structure Variables

Structure variables may be declared in the following ways:

(a) In the structure declaration: The structure variables can be specified after the closing brace. An example follows.

```
struct Student
{
   int Rollno ;
   int Subject;
   float Marks;
}student1, student2, student3;
```

`Student` is the *structure tag*, while `student1`, `student2` and `student3` are variables of type `Student`. If other variables of the structure are not required, the *tag* name `Student` can be omitted, as shown below:

```
struct
{
   int Rollno;
   int Subject;
   float Marks;
}student1, student2, student3;
```

(b) Using the structure tag: The variables of a structure may also be declared separately by using the *structure tag* as illustrated below.

```
struct Student student1,student2,student3;
```

`student1`, `student2` and `student3` are structure variables of the type indicated by the structure tag (here, `Student`). Figure 9.2 illustrates variables of type `Student` whose structure has been detailed in Figure 9.1.

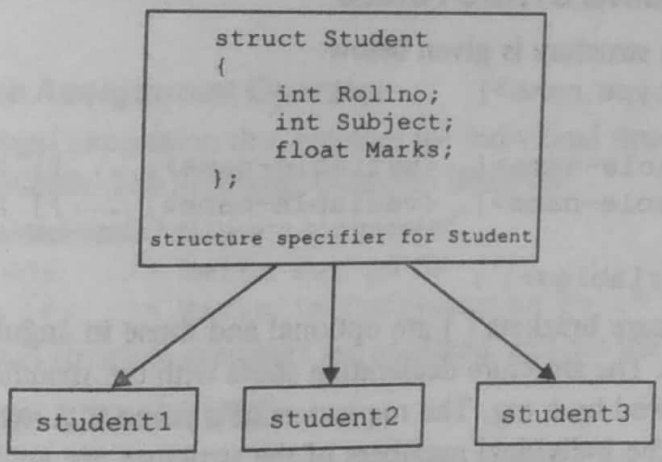

Fig. 9.2 Variables of type Student

Note: It is better to include the *tag* in the structure specifier. This is due to the fact that most programming situations call for the use of both local and global structure variables at different places in the program. The local structure is frequently passed to different functions. The *tag* helps us to avoid specifying the whole structure again. In such situations, we can use the structure *tag* to declare individual structure variables as shown before.

Figure 9.3 shows the storage of the members of a variable of type `Student`.

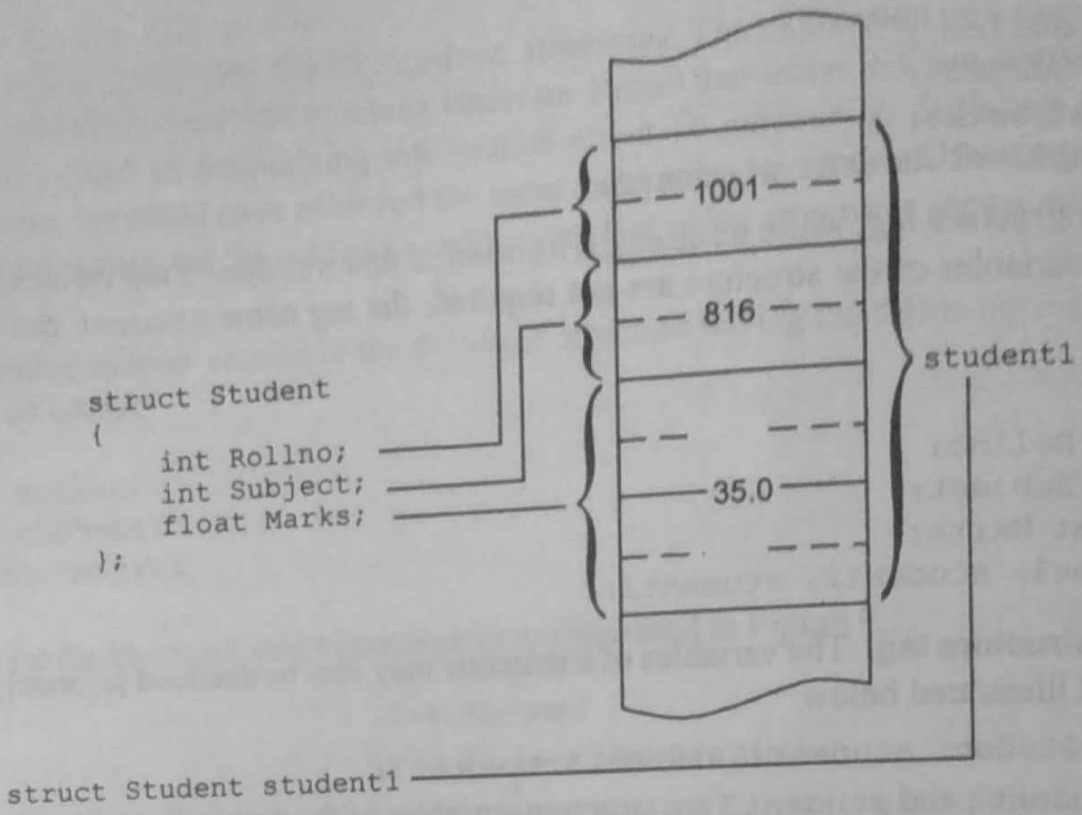

Fig. 9.3 Structure members in memory of type Student

9.2 DECLARING AND USING STRUCTURES

The syntax used to declare a structure is given below:

```
struct [<struct type name>]
{
   [<type> <variable-name>[, <variable-name>, ...]] ;
   [<type> <variable-name>[, <variable-name>, ...]] ;
   ...
} [<structure variables>] ;
```

Entities enclosed in square brackets [ ] are optional and those in angular brackets < > are to be substituted by actual names. The structure declaration starts with the *structure header* that consists of the keyword `struct` followed by a *tag*. The *tag* serves as a name that may be used for a particular template of the structure. The individual members of the structure are enclosed within curly braces. Observe that the individual members of the structure may be of different types. The data type of each variable is specified in individual member declarations. The closing brace is followed by a *semicolon*.

`<struct type name>` is an optional tag name that refers to the structure type

`<structure variables>` refers to the data definitions, also optional.

Though, both `<struct type name>` and `<structure variables>` are optional, at least one of the two must appear. Elements in the structure are defined by a type, followed by one or more identifiers (separated by commas). Declarations involving different variable types must be separated by semicolons.

9.3 STRUCTURE INITIALIZATION

If the structure Student is declared as,

```
struct Student
{
    char name[25];
    int Rollno;
    int Subject;
    float Marks;
};
```

A variable of this structure can be initialized during its declaration as shown below:

```
struct Student student1 = {"Tejaswi", 1024, 11, 98.5};
```

The initial values for the components of the structure are placed within curly braces and separated by commas. In the above example, student1 is initialized with name as Tejaswi, Rollno as 1024, Subject as 11, and marks as 98.5.

The old standard allows the initialization of global and static structures only. So, if the above statement gives a compilation error, it indicates that the compiler follows the old standard. It becomes necessary to change the declaration to:

```
static struct Student student1 = {"Tejaswi", 1024, 11, 98.5};
```

The individual values are separated by commas and enclosed between braces. The closing brace is followed by a semicolon.

Note: The original standard allows initialization of only global and static structures. But the new ANSI standard allows initialization of ordinary structure variables also. The initialization of a structure is illustrated below.

Operations Involving the Assignment Operator

As mentioned before, any legal expression that accesses the individual structure member can be used just as any other ordinary variable. The following examples illustrate.

The implications of the statements below are commented.

```
Student.Marks = 80;     /* Marks set to 80 */
Student.Marks += 10;    /* Marks is incremented by 10 */
```

Notice that only individual structure members are accessed, and not the entire structure.

9.4 STRUCTURE WITHIN STRUCTURE

The individual members of a structure can be other structures as well. This is illustrated by expanding the person structure already introduced. We now include a new member called date which itself is a structure. There are two ways of declaring such a structure. They are illustrated below.

```
struct date
{
    int day;
    int month;
    int year;
};
struct person
{
    char name[25];
    struct date birthday;
    float salary;
};
```

The embedded structure type (date) must be declared before its use within the containing structure. This is because, only legal data types are allowed in the structure declarations, and only those declared till that point are considered legal data types. There is a second way of achieving the same result, as illustrated below.

```
struct person
{
    char name[25];
    struct date
    {
        int day;
        int month;
        int year;
    }birthday;
    float salary;
};
```

In this method, we combine the two structure declarations. The embedded structure date is defined within the enclosing structure declaration. In the former case, where the date structure is declared outside the person structure, it can be used directly in other places, as an ordinary structure. This is not possible when date is declared inside the person structure.

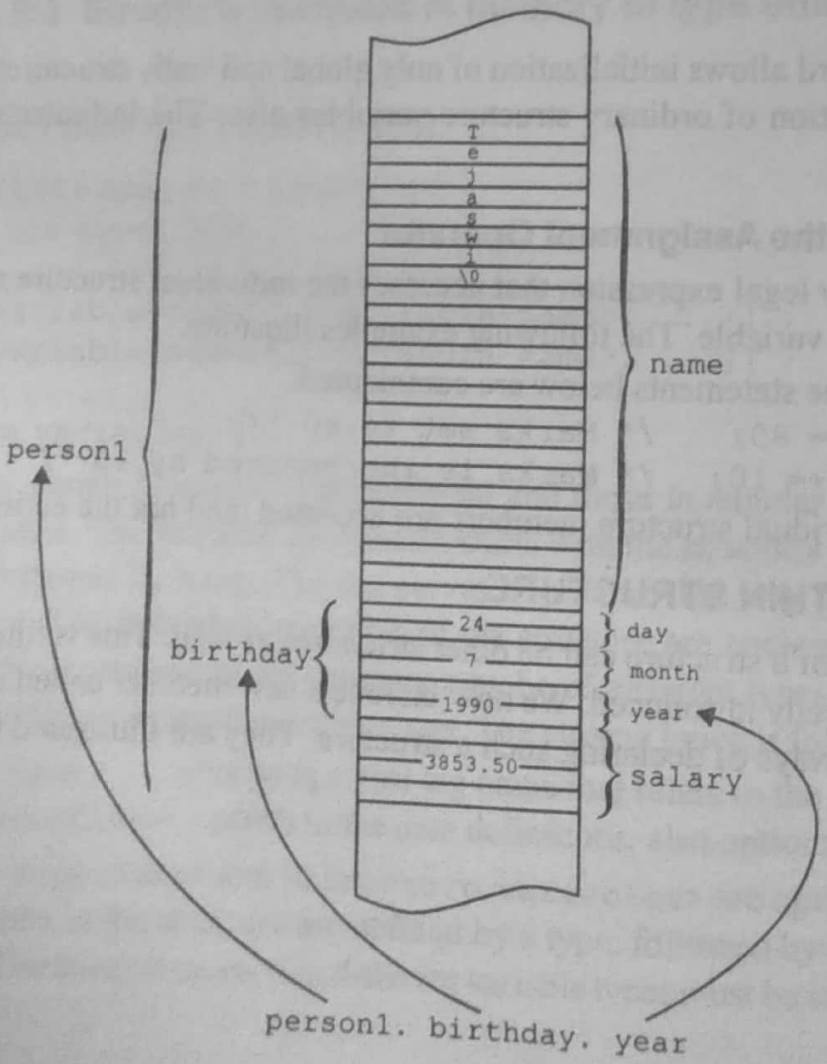

Fig. 9.4 Structure within a structure

In both the above declarations, a variable of type person can be declared using the statement:

```
struct person person1;
```

The year in which the person is born is given by person1.birthday.year. To print it out, the following statement is used.

```
printf( "%i\n", person1.birthday.year );
```

To set the birthday of a person to 24th July 1990, use the statements:

```
person1.birthday.day = 24;
person1.birthday.month = 7;
person1.birthday.year = 1990;
```

The structure person1 is stored in memory as shown in Figure 9.4.

Precedence of DOT Operator

The *dot* operator is a member of the highest precedence group and associates from *left to right*. Since it is an operator of the highest precedence, the *dot* operator will take precedence over various arithmetic, relational, logical, assignment and unary operators. Thus, `++person1.age` is equivalent to `++(person1.age)`, implying that the unary operator ++ will act only on a particular member of the structure and not the entire structure.

9.5 OPERATIONS ON STRUCTURES

C provides the period or *dot* (.) operator to access the members of a structure independently. The *dot* operator connects a member with the structure variable. This can be represented by the following general format:

```
structvar.membervar
```

Here, structvar is a structure variable and membervar is one of its members. Thus, the *dot* operator must have a structure variable on its left and a legal member name on its right. This is illustrated in the examples that follow.

```
struct person
{
   char name[25];
   int age;
   float salary;
}emprec;
```

emprec is a structure variable of type person, and name, age, salary are the members of the structure

emprec.name	will access emprec's name
emprec.age	will access emprec's age
emprec.salary	will access emprec's salary

The following are valid operations on the structure variable emprec

```
emprec.age = 50;
strcpy(emprec.name,"Tejaswi");
emprec.salary=3853.50;
```

Note: The structure *tag* (i.e., person) is not a variable name. It is just a name given to the template of a structure. Thus, the following statement will cause the compiler to generate an error.

```
person.age = 25; /* ERROR: person is not a structure variable */
```

person is a data type like int, and not a variable. Just as int = 10 is invalid, person.age = 25 is also invalid.

Figure 9.5 illustrates the use of dot operator to access the members of a structure.

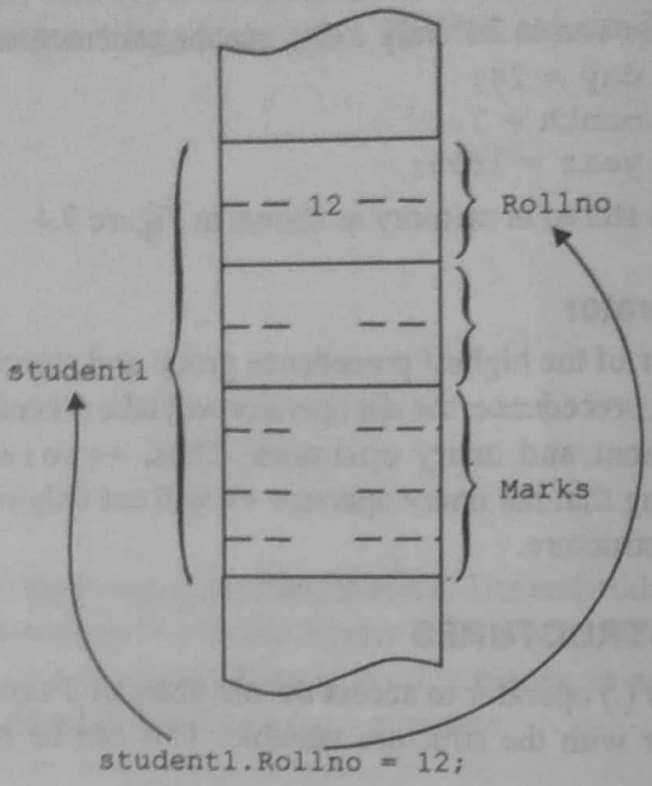

Fig. 9.5 Dot operator to access structure members

9.6 ARRAY OF STRUCTURES

It is possible to declare an array of structures. The array will have individual structures as its elements. Similar to declaring structure variables, an array of structures can be declared in two ways as illustrated below.

```
struct person
{
    char name[25];
    struct date birthday;
    float salary;
}emprec[10];
```

Here `emprec` is an array of 10 `person` structures. Each element of the array `emprec` will contain the structure of type `person`. The `person` structure consists of three individual members: an array `name`, `salary` and another structure `date`. Thus, it is possible to represent a data structure of any complexity by u 'ng this approach. The second approach to the same problem (which has the same effect as the first) involves the use of the structure tag as illustrated below.

```
struct person
{
    char name[25];
    struct date birthday;
    float salary;
};
struct person emprec[10];
```

Thus, `emprec` is a 10 element array of structures of the type `person`.

Accessing Elements in Array of Structures

```
struct date
{
    int day;
    int month;
    int year;
};
struct person
{
    char name[25];
    struct date birthday;
    float salary;
}emprec[10];
```

In the above example, emprec is a 10 element array of structures of the type person. To access the 5th structure, we can use the statement,

```
emprec[4];                    /* arrays are numbered from 0 to n-1 */
```

According the principles already mentioned, the following statements are legal, and their meaning is indicated as comments.

```
emprec[4].name;          /* access the name of 5th structure */
emprec[0].name[5];       /* access 6th character of 1st structure's name member */
emprec[2].birthday.day;/* access day submember of  the birthday member of  the 3rd  structure  */
emprec[2].salary;        /* access the 3rd structure's salary member */
emprec[2].name[4];       /* access the 5th character of the name member of the 3rd structure */
emprec[2].birthday;      /* access the birthday structure of the 3rd structure */
emprec[2].birthday.day    /* access day submember of birthday structure of 3rd emprec structure */
```

Initializing Array of Structures

We can initialize an array of structures in the same way as a single structure. The discussion regarding the initialization of a single structure is still pertinent in this case. This is illustrated by the following example.

```
struct person
{
    char name[25];
    struct date birthday;
    float salary;
};
struct person emprec[5] =
{
    "Tejaswi",24,7,70,3000.00,
    "Vinod",21,9,66,3850.00,
    "Narayan",7,6,64,4000.00,
    "Anil",19,8,68,2550.00,
    "Rakesh",29,1,63,3200.00
};
```

emprec is an array of 5 elements of type person. Thus, emprec[0] will be assigned the first set of values, emprec[1] the second set of values, and so on. Note that there are 5 sets of values in the initialization which are placed in different rows for clarity. The values are separated by commas and

enclosed within braces, with the closing brace being followed by a semicolon. To further improve the readability of the code, enclose individual sets of values within braces as shown below:

```
struct person emprec[5] =
{
    {"Tejaswi",24,7,70,3000.00},
    {"Vinod",21,9,66,3850.00},
    {"Narayan",7,6,64,4000.00},
    {"Anil",19,8,68,2550.00},
    {"Rakesh",29,1,63,3200.00}
};
```

9.7 ARRAY WITHIN STRUCTURE

It is illustrated by the example given below:

```
struct person
{
    char name[25];
    int age;
    float salary;
    int acct_nos[10];
}emprec;
```

In the above structure declaration, `emprec` is a structure variable of the type `person`. The statement given below initializes the `name` member of the structure variable `emprec` to `Anil`.

```
strcpy(emprec.name,"Anil");
```

The third character of the array `name` can be accessed as shown below:

```
emprec.name[2];
```

9.8 CREATING USER DEFINED DATA TYPES

C provides a facility called *type definition* by which new type names can be created. This is accomplished by using the `typedef` keyword along with the definition of the type as shown below:

```
typedef ExistingTypeName NewTypeName;
```

`ExistingTypeName` is the name of an existing data type and `NewTypeName` is the new user defined data type. Notice that a new user defined data type is created only from the existing data types like arrays, structures, other user defined data types or even common data types like `char`, `int` and so on. The following examples illustrate the above concepts.

```
typedef int Length;
```

`Length` now becomes a synonym for `int` and variables can be declared by using the new type name that has been created. Note that `Length` denotes a type name and is not a variable.

```
Length len1,len2;
```

The above declaration shows how variables of the new type may be declared. Note that operations possible on the variables `len1` and `len2` are precisely the same as the operations that are possible on integer variables defined using the keyword `int`. Consider the following set of statements:

```
typedef int emprec[10];
emprec person1,person2;
```

`emprec` is now a new data type, which is a 10 element integer array. `person1` and `person2` are two variables of this new type and are therefore, 10 element integer arrays. Thus, the following expressions

are legal.

```
person1[3]      /* access the 4th element of person1 */
person1         /* access the starting address of person1 */
&person1[0]     /* access the starting address of person1 */
```

Uses of the `typedef` statement

There are several important uses of the `typedef` statement as illustrated below:

- It helps in effective documentation of the program thus increasing its clarity. It thereby provides greater ease of maintenance of the program, which is an important part of software management
- The `typedef` statement is often used for defining new data types involving structures. A new data type representing the structure is declared using the `typedef` keyword. The structure can be referenced using the type name alone. This increases the clarity of the program code. The usage of the `typedef` statement is illustrated below:

```
typedef struct StudentStruct
{
   int Rollno;
   int Subject;
   int Marks;
} Student;
```

Here, `StudentStruct` is the tag while `Student` is the name of the new type. We are familiar with declaring variables by using the tag. For example, the statement,

```
struct StudentStruct stud1;
```

The type `Student`, however, does not require the `struct` keyword. For example,

```
Student stud2;
```

Since variables of the new structure can be created using the type name `Student`, the tag `StudentStruct` may be eliminated from the structure declaration as shown below:

```
typedef struct
{
   int Rollno;
   int Subject;
   int Marks;
} Student;
```

Given below are different ways of using the `typedef` statement, illustrated using the `person` and `date` structures. However, access procedures to the individual structure members are same as those already given before.

Version 1

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

struct person
{
   char name[25];
   date birthday;     /* note that date is a new data type */
   float salary;
}emprec[10];
```

Version 2

```
typedef struct
{
   int day;
   int month;
   int year;
}date;
typedef struct pers
{  /* The tag, pers is included for clarity */
   char name[25];
   date birthday;       /* note that date is a new  data type   */
   float salary;
}person[10];
person emprec;          /* person is a new data type; It is a 10 element
                           array of structures of type pers */
```

Version 3

```
typedef struct
{
   int day;
   int month;
   int year;
}date;
typedef struct pers
{
   /* tag is included for clarity */
   char name[25];
   date birthday;                  /* note that date is a new data type */
   float salary;
}person;
person emprec[10];                 /* person is now a new data type that is
                                      a structure of type pers */
```

9.9 POINTERS TO STRUCTURES

The address of a given structure variable can be obtained by using the `&` operator. This is illustrated in the following expression.

```
&stvar
```

Here `stvar` is a structure variable. Thus, `&stvar` will give the address of this structure variable. There are different ways to declare a pointer to a structure.

Method 1

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

```
struct person
{
   char name[25];
   struct date birthday;
   float salary;
}emprec,*ptr = &emprec;
```

Here, `emprec` is a structure variable of type `person` and `ptr` is a pointer to a structure of type `person`. Notice that `ptr` is declared along with the declaration of the structure variable. The structure *tag* (which in this case is `person`) may be avoided in this method, in case no other variables of type `person` are needed. Any number of structure variables and structure pointers may be declared in this manner. The pointer `ptr` is initialized to the address of `emprec` in the above declaration.

Method 2

Frequently, it is necessary to declare pointers to structures in functions, where they are needed only for temporary usage. In such cases, it is impractical to adopt the first method for two reasons. First, it is unnecessary to declare global pointers when these are needed only on a temporary basis. Second, when pointers are to be declared locally, the entire structure must be declared in the function again, leading to lack of clarity in the program. This may be avoided by using the structure header as illustrated below.

```
struct date
{
   int day;
   int month;
   int year;
};
struct person
{
   char name[25];
   struct date birthday;
   float salary;
}emprec;
```

After declaring the above structure, a pointer to the `person` structure can be declared anywhere in the program using the statement,

```
struct person *ptr;
```

Here, the structure tag (`person`) is used in the declaration of the pointer variable. Thus, it is possible to declare pointers to structures anywhere in the program code without having to provide the structure template again.

Operations Using Pointers to Structures

Pointers to structures, like all other pointer variables may be assigned addresses. The following statements illustrate this concept.

```
struct person person1, *ptr;
ptr = &person1;    /* ptr is a pointer to the person structure */
```

Here, `ptr`, which is a pointer to a structure is assigned the address of the structure variable `person1`.

Accessing Structure Members Using Pointers

Structure members may be accessed using pointers to structures. This is possible through the use of a special access operator ->. This is explained in the following example.

Here, emprec is a structure variable of type person and ptr is a pointer variable to a structure of the type person (i.e., the declaration struct person emprec, *ptr; is used). Access to members of the structure is shown below.

```
ptr = &emprec;          /* assign the address of emprec to ptr */
ptr->salary;            /* access emprec's salary */
ptr->name;              /* get address of name array of emprec */
ptr->name[5];           /* get address of 6th element of emprec's name */
ptr->birthday;          /* access the emprec's birthday */
ptr->birthday.day;      /* access the day submember of birthday */
```

Note: The arrow operator (->), a hyphen followed by a > symbol, also belongs to the highest precedence group, like the dot operator.

9.10 POINTERS WITHIN STRUCTURES

The following example incorporates the concepts discussed till now regarding the access of structure members, using both the *dot* and the *arrow* operators. Note that the person structure has two pointers, lastnam and salary associated with it.

Example 9.1

```
/* struct1.c: Program to implement a structure */
#include <stdio.h>
void main()
{
   float x;
   typedef struct
   {
      int day;
      int month;
      int year;
   }date;
   typedef struct
   {
     char name[20];
     char *lastnam;
     date birthday;
     float *salary;
   }person;
   person emprec;
   person *ptr=&emprec;
   date *birth=&emprec.birthday;
   strcpy(emprec.name,"Tejaswi.V.");
   emprec.lastnam="Rajuk";
   ptr->birthday.day=24;
   emprec.birthday.month=7;
   birth->year=90;
```

```
    x=6500.00;
    ptr->salary=&x;
    printf("\nEmployee Details\n");
    printf("\nName: %s %s",ptr->name,ptr->lastnam);
    printf("\nBirthdate: %3d:%3d:%3d ",(*ptr).birthday.day, \
       birth->month,emprec.birthday.year);
    printf("\nSalary: %6.2f",*emprec.salary);
}
```

In the above program, the structure of type `person` is declared locally within the function `main`. Hence, the type `person` is visible only in the function `main`. As before, `emprec` is a variable of the type `person`, and `ptr` is a pointer to a structure of type `person`. Notice that there are two pointers in the structure `person`. The first pointer serves to provide a way to include the last name of an employee in the records. The second is a pointer to the salary member of a given structure variable of type `person`. The assignments are now done in the function `main`, and not by initializing the variable along with its declaration.

Another new feature in the above example is the access of a structure member by using a combination of the *dot* operator and the indirection operator. This is accomplished by the following statement.

```
(*ptr).birthday.day;
```

Notice that `*ptr` will be a structure variable, as `ptr` is a pointer to a structure variable and `*` is the indirection operator. The parentheses in the above statement are essential because of the precedence rules in C. The usage of the expression

```
*ptr.birthday.day;
```

will generate an error message from the compiler as the `*` operator has a lower precedence compared to the dot operator. The expression has the effect of:

```
*(ptr.birthday.day)
```

Since `ptr` is a pointer variable which is not compatible with the dot operator, an error would be generated.

Another feature that is illustrated in the above program is the access of a submember through the use of pointers. Notice that `birth` is a pointer variable to a structure of the type `date`. This pointer after initialization, may be used to access the submember directly without accessing the container structure. This is illustrated by the statement below:

```
birth->month = 7;
```

The output of the above program is:

```
Employee Details
Name: Tejaswi.V. Rajuk
Birthdate:   24: 7: 90
Salary: 6500.00
```

9.11 STRUCTURES AND FUNCTIONS

Passing Structure Members to functions

A structure member can be treated just as any other variable of that type. For example, an integer structure variable can be treated just as an integer. Thus, structure members can be passed to functions, like ordinary variables. The program segment given below illustrates the communication between functions, through individual structure members.

```
float increment(float x,int y)
{
   x+=500;
   ....
   return x;
}
void main()
{
   typedef struct
   {
      int day, month, year;
   } date;
   typedef struct person
   {
      char name[20];
      date birthday;
      float salary;
   }emprec;
   emprec per;
   per.salary=increment(per.salary,per.birthday.year);
   .....
   printf("Salary: %6.2f\n",per.salary);
}
```

In the above program segment, two structure members are passed to the function `increment`, namely, `per.salary` and `per.birthday.year`. Only one value is returned, namely `x`, which is assigned to the structure member `per.salary` in the function `main`. Like ordinary parameters, structures are always passed by value. Hence, modifications to `x` inside the function `increment` do not affect the structure member `per.salary`. However, the function returns the incremented value of `x`, which is assigned to `per.salary` in `main`. The scope rules which apply for structure member variables are the same as those that apply for ordinary variables.

Example 9.2

Let us now consider a complete program which illustrates the above concepts.

```
/* struct2.c: program to implement a structure for employees */
#include <stdio.h>
#define CURRENT_YEAR   96
float increment(float sal,int year,int inc)
{
   /* give increments to employees if age is greater than 30  */
   if (CURRENT_YEAR - year > 30)
   {
      sal+=inc;
      return(sal);
   }
}
void main()
{
   typedef struct
   {
      int day,month,year;
   } date;
```

```
typedef struct person
{
   char name[20];
   date birthday;
   float salary;
}emprec;
int n=500;
emprec per = { "Arun R.",10,8,64,4000.00 };
printf("\nEmployee Details\n");
printf("Name: %s\n",per.name);
printf("Birthdate: %3d:%3d:%3d\n",
   per.birthday.day,per.birthday.month,per.birthday.year);
printf("Salary: %6.2f\n",per.salary);
per.salary=increment(per.salary,per.birthday.year,n);
printf("\nEmployee Details\n");
printf("Name: %s\n",per.name);
printf("Birthdate: %3d:%3d:%3d\n",
   per.birthday.day,per.birthday.month,per.birthday.year);
printf("Salary: %6.2f\n",per.salary);
}
```

In the above program, `per.salary`, `per.birthday.year` and `n` are passed to the function `increment`. The parameter `sal` is manipulated and returned from the function `increment`. The function `increment` checks the `age` of the employee and gives an increment of 500 if his age is above 30. Note that the amount to be incremented (which is `n`, initialized with 500) is also passed to the function `increment`. The contents of the structure `emprec` are displayed before and after the call to `increment`. It is seen that the `per.salary` member has the incremented salary, since the return value from the function `increment` is assigned to it. Also, note that the output of the program is generated by passing the structure members as arguments to the `printf` function. The output of the program is given below.

```
Employee Details
Name: Arun R.
Birthdate:  10:  8: 64
Salary: 4000.00

Employee Details
Name: Arun R.
Birthdate:  10:  8: 64
Salary: 4500.00
```

Passing Structure Pointers to Functions

It is often found that more than one member variable is computed in the function. In such a situation, pointers to structures may be used. If the pointer to a structure is passed as an argument to a function, then any change that are made in the function are visible in the caller. This is analogous to the passing of ordinary pointers. The following program is similar to the previous program, except that, now, a pointer to the structure is passed to the function `increment`. Note that the address of the `per` structure variable is passed to `increment`. Since the function does not need any return value now, its return type is `void`.

Example 9.3

```
/* struct3.c: implement employee record using pointers to structures */
#include <stdio.h>
#define CURRENT_YEAR  96
typedef struct
{
   int day;
   int month;
   int year;
} date;
typedef struct person
{
   char name[20];
   date birthday;
   float salary;
}emprec;
void increment(emprec *x)
{
   if (CURRENT_YEAR - x->birthday.year > 30)
      x->salary+=500;
}
main()
{
   emprec per = { "Arun R.",10,8,64,4000.00 };
   printf("\nEmployee Details\n");
   printf("Name: %s\n",per.name);
   printf("Birthdate: %3d:%3d:%3d\n",per.birthday.day,\
      per.birthday.month, per.birthday.year);
   printf("Salary: %6.2f\n",per.salary);
   increment(&per);
   printf("\nEmployee Details\n");
   printf("Name: %s\n",per.name);
   printf("Birthdate: %3d:%3d:%3d\n",per.birthday.day,\
      per.birthday.month, per.birthday.year);
   printf("Salary: %6.2f\n",per.salary);
}
```

The output of the program is as follows:

```
Employee Details
Name: Arun R.
Birthdate:  10:  8: 64
Salary: 4000.00

Employee Details
Name: Arun R.
Birthdate:  10:  8: 64
Salary: 4500.00
```

Passing Structures to Functions

In all compilers based on ANSI C, it is possible to send entire structures to functions as arguments in the function call. The structure variable is treated as any ordinary variable. The following program illustrates

he passing of entire structures to functions. Note that person here is the type and not the tag name.

Example 9.4

```
/* struct4.c: demonstrates passing structures to functions */
#include <stdio.h>
typedef struct
{
   int day,month,year;
} date;
typedef struct
{
   char name[20];
   date birthday;
   float salary;
}person ;
void main()
{
   void printout(person );
   void readin(person );
   emprec highest(person ,int);
   emprec test= { "Arun R.",10,8,75,4000.00 };
   emprec temp;
   readin(test);
   printout(test);
}
/* Function to print out the values stored in the structure */
void printout(person  per)
{
   printf("\nEmployee Details\n");
   printf("\nName: %s\n",per.name);
   printf("Birthdate: %3d:%3d:%3d\n",per.birthday.day,
          per.birthday. month,  per.birthday.year);
   printf("Salary: %6.2f\n",per.salary);
}
/* Function to take input values for the structure */
void readin(person record)
{
   printf("\n\nEnter the name: ");
   scanf("%s",record.name);
   printf("\nEnter the birthdate: ");
   printf("\nEnter the day: ");
   scanf("%d",&record.birthday.day);
   printf("\nEnter the month: ");
   scanf("%d",&record.birthday.month);
   printf("\nEnter the year: ");
   scanf("%d",&record.birthday.year);
   printf("\nEnter the salary: ");
   scanf("%f",&record.salary);
}
```

The above program has two functions, `readin` and `printout` to which the structure is pas directly. The function `readin` is used to input values for the different structure members. The funct `printout` prints out the values contained in the structure. When a structure is passed directly as argument to the function, it is *passed by value* like an ordinary variable. Thus, any change that is ma in the function are not visible in the caller. This can be observed by the output of the program sho below. Observe that even after calling `readin`, the structure members have the same values as befo

```
Enter the name: Balaji
Enter the birthdate:
Enter the day: 24
Enter the month: 2
Enter the year: 74
Enter the salary: 6500.50

Employee Details

Name: Arun R.
Birthdate:  10:  8: 75
Salary: 4000.00
```

If the structure `person` is to be modified by the function `readin`, it has to accept a pointer to the `person` structure. All members of `person` in `readin` must be accessed with the `->` operator instead of the dot operator. The argument passed to `readin` in `main` will be `&test`.

Passing the structure by value to `printout`, however, is not a problem, since the `printout` function does not modify the structure passed. However, in case of large structures, this can be very inefficient, since the whole structure must be copied on every call. A better alternative in such situations is to pass the structure by a pointer. But in that case, any change accidentally made to the structure in `printout` will have repercussions in the rest of the program. The solution is to accept the structure as a `const` pointer, i.e., change the prototype of `printout` to

```
void printout(const person *per)
```

and use `->` instead of the dot operator in the function. In `main`, call the function using `printout(&test);`. Now, any modification done to the structure passed, in function `printout` will cause a compiler error.

Returning Structures from Functions

Structures can be returned from functions just as variables of any other type. For example, consider the function `readin`. Instead of accepting a pointer to a structure, it can construct a structure by itself and return this structure variable. This is illustrated in the following program.

Example 9.5

```
/* struct5.c: demonstration of returning structures from functions */
#include <stdio.h>
typedef struct
{
   int day,month,year;
} date;
```

```
typedef struct person
{
   char name[20];
   date birthday;
   float salary;
}emprec;
void main()
{
   void printout(emprec);
   emprec readin(emprec);
   emprec highest(emprec,int);
   emprec test= {  "Arun R.",10,8,75,4000.0  };
   test=readin();
   printout(test);
}
void printout(emprec per)
{
   printf("\nEmployee Details\n");
   printf("\nName: %s\n",per.name);
   printf("Birthdate: %3d:%3d:%3d\n",per.birthday.day,\
          per.birthday.month, per.birthday.year);
   printf("Salary: %6.2f\n",per.salary);
}
emprec readin()
{
   emprec record;
   printf("\n\nEnter the name: ");
   scanf("%s",record.name);
   printf("\nEnter the birthdate: ");
   printf("\nEnter the day: ");
   scanf("%d",&record.birthday.day);
   printf("\nEnter the month: ");
   scanf("%d",&record.birthday.month);
   printf("\nEnter the year: ");
   scanf("%d",&record.birthday.year);
   printf("\nEnter the salary: ");
   scanf("%f",&record.salary);
   return(record);
}
```

he structure returned by the function `readin` is copied to the structure variable test in `main`. Since it volves copying structures, it is less efficient than passing by pointers. The output is as follows.

```
Enter the name: Balaji
Enter the birthdate:
Enter the day: 24
Enter the month: 2
Enter the year: 74
Enter the salary: 6500.00
```

```
Employee Details

Name: Balaji
Birthdate:  24:  2: 74
Salary: 6500.00
```

Passing Array of Structures to Functions

Passing an *array of structures* to functions involves the same syntax and properties as passing of any array to a function. The passing is done using a pointer and consequently, any change made in the function to the structures is visible in the caller.

Example 9.6

The following program illustrates the passing of an array of structures to a function.

```
/* struct6.c: passing an array of structures to function */
#include <stdio.h>
typedef struct
{
   int day,month,year;
}date;
typedef struct person
{
   char name[20];
   date birthday;
   float salary;
}emprec;
void main()
{
   int j;
   void printout(emprec);
   int highest(emprec rec[],int);
   emprec record[] = {
                        {"Arun R.",10,8,75,4000.00},
                        {"Vinod S.",3,10,74,3000.00},
                        {"Tanuj M.",21,1,73,5500.00}
                      };
   j=highest(record,3);
   printf("\nThe Highest Salary being paid is %6.2f\n",
        record[j].salary);
   printout(record[j]);
}

/* Find the employee with the highest salary */
int highest(emprec record[],int n)
{
   float l;
   int i,j=0;
   l=record[0].salary;
   /* find index of employee with highest salary */
```

```
    for (i=1;i<n;i++)
    { if (l<record[i].salary)
      {
         l=record[i].salary;
         j=i;
      } /* if loop */
    } /* for loop */
    /* return index of employee with highest salary */
    return(j);
}
/* printout the values stored in the structure passed as argument */
void printout(emprec per)
{
    printf("\nEmployee Details\n");
    printf("\nName: %s\n",per.name);
    printf("Birthdate: %3d:%3d:%3d\n",per.birthday.day,
            per.birthday.month, per.birthday.year);
    printf("Salary: %6.2f\n",per.salary);
}
```

The objective of the above program is to find the employee with the highest salary. The function `highest` accepts two parameters. The first is an array of structures, called `record`. The second argument is an integer which denotes the number of employees stored in `record`.

The index of the structure in `record` with the highest salary is found and returned to the function `main`. This structure is then passed to the function `printout` to generate the output as shown below.

```
The Highest Salary being paid is 5500.00
Employee Details
Name: Tanuj M.
Birthdate:  21:  1: 73
Salary: 5500.00
```

Returning Pointers to Structures

Returning pointers to structures has the same syntax as returning pointers to ordinary variables. This is illustrated in the program below.

Example 9.7

```
/* struct7.c: Returning pointers to structures */
#include <stdio.h>
typedef struct
{
   int day,month,year;
} date;
typedef struct person
{
   char name[20];
   date birthday;
   float salary;
}emprec;
```

```
void main()
{
   int i,j,n;
   float x,y;
   void printout(emprec*);
   emprec* highest(emprec rec[],int);
   emprec *temp;
   emprec record[] = {   {"Tejaswi", 24,7,70,7000.00},
                         {"Vinod S.",3,10,74,3000.00},
                         {"Tanuj M.",21,1,73,5500.00}
                     };
   temp=highest(record,3);
   printf("\nThe Highest Salary being paid is %6.2f\n",
          temp->salary);
   printout(temp);
}
emprec* highest(emprec record[],int n)
{
   float l;
   int i,j=0;
   l=record[0].salary;
   /* Find index of employee with highest salary */
   for (i=1;i<n;i++)
   {
      if (l<record[i].salary)
      {
         l=record[i].salary;
         j=i;
      } /* if loop */
   } /* for loop */
   /* return address of employee with highest salary */
   return(&record[j]);
}
/* Function takes pointer to structure as arguments and uses  arrow as
the structure member access operator */
void printout(const emprec *per)
{
   printf("\nEmployee Details\n");
   printf("\nName: %s\n",per->name);
   printf("Birthdate: %3d:%3d:%3d\n",
          per->birthday.day, per->birthday.month, per->birthday.year);
   printf("Salary: %6.2f\n",per->salary);
}
```

The objective of the above program is similar to that of the previous program. The only difference is that the function `highest` now has a return value of type pointer to a structure. Thus, instead of returning an index from the function, a pointer to the structure is returned. The `printout` function now takes a pointer to a structure as the sole argument. An *arrow operator* is used in `printout` to access structure members. The output is shown below.

```
The Highest Salary being paid is 7000.00
Employee Details
Name: Tejaswi
Birthdate:  24:  7: 70
Salary: 7000.00
```

Note that the last statement in the function `highest`, which is,

```
return (&record[j]);
```

can be replaced by,

```
return (record + j);
```

as in the case of any array.

As already stated, when an array of structures is passed to a function, the changes made within the function to the array of structures is visible in the caller. The following program explains this concept.

Example 9.8

```
/* struct8.c: Passing array of structures */
#include <stdio.h>
typedef struct
{
   int day,month,year;
} date;
typedef struct person
{
   char name[20];
   date birthday;
   float salary;
}emprec;
void main()
{
   int i,j,n=0;
   float x,y;
   char instr[2];
   void printout(emprec);
   void readin(emprec rec[],int*);
   emprec highest(emprec rec[],int);
   emprec record[] = {
                        {"Tejaswi",24,7,70,4000.00},
                        {"Vinod S.",3,10,74,3000.00},
                        {"Tanuj M.",21,1,73,5500.00}
                      };
   readin(record);
   for (i=0;i<3;i++)
   {
      printf("\nHit Enter to continue...\n");
      gets(instr);
      printout(record[i]);
   }
}
void printout(emprec per)
{
```

```
    printf("\nEmployee Details\n");
    printf("\nName: %s\n",per.name);
    printf("Birthdate:%3d:%3d:%3d\n",per.birthday.day,
            per.birthday.month,\ per.birthday.year);
    printf("Salary: %6.2f\n",per.salary);
}
void readin(emprec record[])
{
    int i;
    /* prompt the user for the required number of records */
    for (i=0;i<3;i++)
    {
       printf("\nEmployee Number %2d",i+1);
       printf("\n\nEnter the name: ");
       scanf("%s",record[i].name);
       printf("\nEnter the birthdate: ");
       printf("\nEnter the day: ");
       scanf("%d",&record[i].birthday.day);
       printf("\nEnter the month: ");
       scanf("%d",&record[i].birthday.month);
       printf("\nEnter the year: ");
       scanf("%d",&record[i].birthday.year);
       printf("\nEnter the salary: ");
       scanf("%f",&record[i].salary);
    }
}
```

The program consists of two functions `readin` and `printout`. The `readin` function accepts an array of structures and has no return value. It reads in three structures into the array passed. If changes are made inside the function to the array of structures, they are visible in the caller as shown by the output below.

```
Employee Number  1
Enter the name: Balaji
Enter the birthdate:
Enter the day: 24
Enter the month: 2
Enter the year: 74
Enter the salary: 6500.50
Employee Number  2
Enter the name: Tarun
Enter the birthdate:
Enter the day: 30
Enter the month: 9
Enter the year: 69
Enter the salary: 6890.00
Employee Number  3
Enter the name: Vinod
Enter the birthdate:
Enter the day: 7
```

```
Enter the month: 6
Enter the year: 70
Enter the salary: 6700.00

Hit Enter to continue...
Employee Details

Name: Balaji
Birthdate:  24:  2: 74
Salary: 6500.50

Hit Enter to continue...

Employee Details

Name: Tarun
Birthdate:  30:  9: 69
Salary: 6890.00

Hit Enter to continue...

Employee Details

Name: Vinod
Birthdate:   7:  6: 70
Salary: 6700.00
```

A general example

A general example which integrates all the concepts discussed so far is illustrated. The program is capable of reading upto 10 structures. Input is done in the `readin` function, similar to the previous example. However, the number of structures read is returned. The array of structures is global, so that all functions can access it. The objective of the program is to sort these structures according to `name`, `age` or `salary`. The program is interactive with the user being provided with a choice of three keys based on which an appropriate sorting function will be called. The three sorting functions are `sortalpha`, `sortage` and `sortsal`. The *bubble sort* technique is used to arrange the records. Though bubble sort is not the most efficient sorting procedure, it is one of the simplest to comprehend. The function `printout` does not take any argument and displays the entire array of structures, pausing after each employee. All functions now access the array of structures directly, since it is declared as a global variable. The program is given below.

Example 9.9

```
/* struct9.c: program to use structures */
#include <stdio.h>
#include <string.h>
typedef struct
{
   int day;
   int month;
   int year;
} date;
typedef struct person
{
   char name[20];
   date birthday;
```

```
    float salary;
}emprec;
emprec record[10];
int n;
void main()
{
    int i,flag=1;
    void printout();
    int readin();
    void sortalpha();
    void sortage();
    void sortsal();
    n=readin();
    while (flag)
    {
       printf("\n1: SORT BY NAME");
       printf("\n2: SORT BY AGE");
       printf("\n3: SORT BY SALARY");
       printf("\n4: QUIT");
       printf("\nEnter your choice: ");
       scanf("%d",&i);
       switch(i)
       {
          case 1: sortalpha();
                  printout();
                  break;
          case 2: sortage();
                  printout();
                  break;
          case 3: sortsal();
                  printout();
                  break;
          case 4: flag=0;
                  break;
                  default: flag=0;
                  break;
       }; /* end switch */
    } /* end while */
} /* end main */
/* print the entire array of structures pausing after each structure */
void printout()
{
    int i;
    char buf[ 2 ];
    for (i=0;i<n;i++)
    {
       printf("\nEmployee Number %d\n",i);
       printf("\nName: %s\n",record[i].name);
       printf("Birthdate: %3d:%3d:%3d\n",record[i].birthday.day,\
```

```
                record[i].birthday.month,record[i].birthday.year);
        printf("Salary: %6.2f\n",record[i].salary);
    }
    printf("\nHit Enter to continue...\n");
    gets( buf );
}
/* read in the given number of structures */
int readin()
{
    int i,n;
    printf("\nEnter the number of records ");
    scanf("%d",&n);
    for (i=0;i<n;i++)
    {
        printf("\nEmployee Number %2d",i+1);
        printf("\n\nEnter the name: ");
        scanf("%s",record[i].name);
        printf("\nEnter the birthdate: ");
        printf("\nEnter the day: ");
        scanf("%d",&record[i].birthday.day);
        printf("\nEnter the month: ");
        scanf("%d",&record[i].birthday.month);
        printf("\nEnter the year: ");
        scanf("%d",&record[i].birthday.year);
        printf("\nEnter the salary: ");
        scanf("%f",&record[i].salary);
    }
    return(n);
}
/* sort the array of structures with the employee name as the key */
void sortalpha()
{
    int i,j;
    emprec temp;
    for (i=0;i<n-1;i++)
    {
        for (j=0;j<n-i-1;j++)
        {
            if (strcmp(record[j].name,record[j+1].name)>0)
            {
                temp=record[j];
                record[j]=record[j+1];
                record[j+1]=temp;
            }
        }
    }
}
/* sort the array of structures with the employee age as the key */
void sortage()
```

```
{
   int i,j;
   emprec temp;
   for (i=0;i<n-1;i++)
   {
      for (j=0;j<n-i-1;j++)
      {
         if ((record[j].birthday.year>record[j+1].birthday.year)||
            ((record[j].birthday.year==record[j+1].birthday.year)&&
            (record[j].birthday.month>record[j+1].birthday.month))||
            ((record[j].birthday.year==record[j+1].birthday.year)&&
            (record[j].birthday.month==record[j+1].birthday.month)&&
            (record[j].birthday.day>record[j+1].birthday.day)))
         {
            temp=record[j];
            record[j]=record[j+1];
            record[j+1]=temp;
         } /* end if */
      } /* end inner for */
   } /* end outer for */
} /* end sortage */

/* sort the array of structures with employee salary as the key */
void sortsal()
{
   int i,j;
   emprec temp;
   for (i=0;i<n-1;i++)
   {
      for (j=0;j<n-i-1;j++)
      {
         if (record[j].salary>record[j+1].salary)
         {
            temp=record[j];
            record[j]=record[j+1];
            record[j+1]=temp;
         }
      }
   }
}
```

All the sorting functions in the program arrange the employees in ascending order. The `sortalpha` function sorts the array with employee `name` as the *key* element. The `sortage` function sorts the array with employee `age` as the key element. The `sortsal` function sorts the array with `salary` as the key element. Note that each of these functions have a temporary structure `temp`, used for exchanging structures. The structures are exchanged if necessary, by the use of the assignment operator. Thus, this method will work only on ANSI C compilers or those which allow structures to be assigned to one another. A complete interactive session is given below.

```
Enter the number of records 5
Employee Number  1
Enter the name: Vinod
```

```
Enter the birthdate:
Enter the day: 23
Enter the month: 8
Enter the year:74
Enter the salary: 6500

Employee Number  2
Enter the name: Arun
Enter the birthdate:
Enter the day: 10
Enter the month: 5
Enter the year: 70
Enter the salary: 7000.00

Employee Number  3
Enter the name: Puneet
Enter the birthdate:
Enter the day: 30
Enter the month: 6
Enter the year: 74
Enter the salary: 6800

Employee Number  4
Enter the name: Tanuj
Enter the birthdate:
Enter the day: 7
Enter the month: 1
Enter the year: 70
Enter the salary: 7700.00

Employee Number  5
Enter the name: Rohit
Enter the birthdate:
Enter the day: 17
Enter the month: 12
Enter the year: 73
Enter the salary: 6900.00

1: SORT BY NAME
2: SORT BY AGE
3: SORT BY SALARY
4: QUIT

Enter your choice: 1

Employee Number 0
Name: Arun
Birthdate:  10:  5: 70
Salary: 7000.00

Employee Number 1
Name: Puneet
Birthdate:  30:  6: 74
Salary: 6800.00

Employee Number 2
```

```
Name: Rohit
Birthdate:  17: 12: 73
Salary: 6900.00

Employee Number 3
Name: Tanuj
Birthdate:   7:  1: 70
Salary: 7700.00

Employee Number 4
Name: Vinod
Birthdate:  23:  8: 74
Salary: 6500.00

Hit Enter to continue...

1: SORT BY NAME
2: SORT BY AGE
3: SORT BY SALARY
4: QUIT
Enter your choice: 2

Employee Number 0
Name: Tanuj
Birthdate:   7:  1: 70
Salary: 7700.00

Employee Number 1
Name: Arun
Birthdate:  10:  5: 70
Salary: 7000.00

Employee Number 2
Name: Rohit
Birthdate:  17: 12: 73
Salary: 6900.00

Employee Number 3
Name: Puneet
Birthdate:  30:  6: 74
Salary: 6800.00
Employee Number 4

Name: Vinod
Birthdate:  23:  8: 74
Salary: 6500.00

Hit Enter to continue...

1: SORT BY NAME
2: SORT BY AGE
3: SORT BY SALARY
4: QUIT

Enter your choice: 3

Employee Number 0
Name: Vinod
Birthdate:  23:  8: 74
```

```
Salary: 6500.00

Employee Number 1
Name: Puneet
Birthdate:  30:  6: 74
Salary: 6800.00

Employee Number 2
Name: Rohit
Birthdate:  17: 12: 73
Salary: 6900.00

Employee Number 3
Name: Arun
Birthdate:  10:  5: 70
Salary: 7000.00
Employee Number 4
Name: Tanuj
Birthdate:   7:  1: 70
Salary: 7700.00

Hit Enter to continue...

1: SORT BY NAME
2: SORT BY AGE
3: SORT BY SALARY
4: QUIT
Enter your choice: 4
```

9.12 UNIONS

A `union` is a data type in C which allows the overlay of more than one variable in the same memory area. Normally, every variable is stored in a separate location and as a result, each of these variables have their own addresses. Often, it is found that some variables used in the program are used only in a small portion of the source code. For example, if a string of 200 bytes called `filename` is required by the first 500 lines of the code only, and another string called `output` of 400 bytes is required by the rest of the code (i.e., both strings are not needed simultaneously), it would be a waste of memory to declare separate arrays of 200 and 400 bytes. The `union` construct provides a means by which the memory space can be shared, and only 400 bytes of memory are used.

Declaring Unions

In terms of declaration syntax, `union` is similar to a *structure*. The only change in the declaration is the substitution of the keyword `union` for the keyword `struct`.

Syntax :

```
union [<union type name>]
{
    <type> <variable names> ;
    ...
} [<union variables>] ;
```

All variables inside a `union` share storage space. The compiler will allocate sufficient storage for the `union` variables to accommodate the largest element in the `union`. Other elements of the `union` use the

same space. This is how a union differs from a structure. Individual variables in a union occupy the same location in memory. Thus, writing into one will overwrite the other. Elements of a union are accessed in the same manner as the elements of a structure.

Declaring Variables and Pointers to Unions

A union and pointer to a union are declared in the same way as they are declared for structures. The following example declares a union called name_or_id along with a variable identifier1 of that type and a pointer ptr.

```
union name_or_id
{
   char name [40];
   char id [20];
   type member n;
}identifier1 *ptr;
```

identifier1 and ptr are declared with the union itself. To declare separately, use:

```
union name_or_id name1, id1, *ptr2;
```

Unions and pointer variables can be declared along with the union declaration, or declared separately using the tag name, as shown above.

Member Access

Members can be accessed using either the *dot* or -> operator. The syntax is similar to structures, as illustrated by the example below.

```
union desc
{
   char name[25];
   int idno;
   float salary;
};
union desc var1,*var2;
var2=&var1;
```

The individual members may be accessed by statements such as :

```
var1.name              /* access the name */
var1.idno;             /* access the idno */
var2->salary;          /* access the salary */
```

Assignments may be made in the same way and with the same restrictions as structures.

9.13 DIFFERENCES BETWEEN STRUCTURES AND UNIONS

There are important differences between structures and unions though the syntax used for declaring them is very similar.

Memory Allocation

The amount of memory required to store a structure variable is the sum of sizes of all the members in addition to the padding bytes that may be provided by the compiler. On the other hand, in case of a union, the amount of memory required is the same as that required by its largest member. This is illustrated in the example below.

```
/* mem.c: program to allocate memory */
#include <stdio.h>
struct
{
   char name[25];
   int idno;
   float salary;
}emp;
union
{
   char name[25];
   int idno;
   float salary;
}desc;
void main()
{  printf("The size of the structure is %d",sizeof(emp));
   printf("The size of the union is %d",sizeof(desc));
}
```

The output of the above program is:

```
The size of the structure is 31
The size of the union is 25
```

Operations on Members

While all the structure members can be accessed at any point of time, only one member of a `union` may be accessed at any given time. This is because, at any instant only one of the `union` variables will have a meaningful value. Only that member which is last written, can be read. At this point, other variables will contain garbage. It is the responsibility of the programmer to keep track of the *active* variable (i.e., variable which was last accessed). This is illustrated by the following program.

```
#include <stdio.h>
#include <string.h>
union
{
   char name[25];
   int idno;
   float salary;
}desc;
void main()
{
   strcpy(desc.name,"Vinod");
   printf("\nEmployee Details\n\n");
   printf("The name is %s\n",desc.name);
   printf("The idno is %d\n",desc.idno);
   printf("The salary is %6.2f\n",desc.salary);
   desc.idno=10;
   printf("\nEmployee Details\n\n");
   printf("The name is %s",desc.name);
   printf("The idno is %d\n",desc.idno);
   printf("The salary is %6.2f\n",desc.salary);
```

```
    desc.salary=6500.00;
    printf("\nEmployee Details\n\n");
    printf("The name is %s\n",desc.name);
    printf("The idno is %d\n",desc.idno);
    printf("The salary is %6.2f\n",desc.salary);
}
```

Run:

```
Employee Details
The name is Vinod
The idno is 26966
The salary is 737847756720571892000000000000.00

Employee Details
The name is
The idno is 10
The salary is 736574795611449551000000000000.00

Employee Details
The name is
The idno is 8192
The salary is 6500.00
```

As seen above, access of *non active* members will lead to meaningless values.

Identifying Active Members

In the above example, there is no way to find which of the members is active, at any given point. The program must keep track of active members explicitly. For example, this can be done by keeping an integer with the value 0 for `name`, 1 for `idno` and 2 for `salary`. Care must be taken to allocate a different integer variable for each variable of the `union`.

9.14 OPERATIONS ON A `union`

In addition to the features discussed above, `union` has all the features provided to a structure except for minor changes which are a consequence of the memory sharing properties of a `union`. This is made evident by the following operations on unions which are legal.

- An `union` variable can be assigned to another `union` variable
- A `union` variable can be passed to a function as a parameter
- The address of the `union` variable can be extracted by using the *address-of* operator (&)
- A function can accept and return a `union` or a pointer to a `union`.

Note: Performing operations on unions *as a whole*, for example, arithmetic or logical operations, are illegal, as in structures.

9.15 SCOPE OF A `union`

The members of a `union` have the same scope as the `union` itself, as in the following example:

```
/* union.c: program to demonstrate the usage of a union */
#include <stdio.h>
void main()
{
```

```
    union
    {
       int i;
       char c;
       float f;
    };

    i = 10;
    c = 9;
    f = 4.5;
    printf( "The value of i is %d\n", i );
}
```

The scope of the `union` is the function `main`. The scope of its members, `i`, `c` and `f`, are the same as the `union` itself. In function `main`, they can be accessed as other local variables. The only difference is that the variables share the same memory. The output of this program is:

```
The value of i is 0
```

But on your computer, the message might be different, depending on how integers and floating point numbers are stored.

9.16 BIT FIELDS IN STRUCTURES

The following example declares a structure with bit fields.

```
struct with_bits
{
   unsigned first : 5;
   unsigned second : 9;
};
```

The identifier `with_bits` is a structure with two members, `first` and `second`. The member `first` is an integer with 5 bits and `second` is an integer with 9 bits. Both the numbers can be stored in a 16-bit entity now (even though they add up to 14 bits, a 14-bit entity cannot exist in memory) rather than two separate integers. Combined with unions, such structures can be used with great effectiveness, as in the following example:

```
void main()
{
   union
   {
      with_bits b;
      int i;
   };
   i = 0;              /* Both first and second are cleared to 0 */
   b.first = 9;        /* first is set to 9; second remains 0 */
}
```

The members `first` and `second` can be used as if they are members of the structure `with_bits`. Bit fields are treated as *unsigned integers*.