10

Structures and Unions

10.1 INTRODUCTION

int array[ ]

We have seen that arrays can be used to represent a group of data items that belong to the same type, such as **int** or **float**. However, we cannot use an array if we want to represent a collection of data items of different types using a single name. Fortunately, C supports a constructed data type known as *structures*, a mechanism for packing data of different types. A structure is a convenient tool for handling a group of logically related data items. For example, it can be used to represent a set of attributes, such as student_name, roll_number and marks. The concept of a structure is analogous to that of a 'record' in many other languages. More examples of such structures are:

time	:	seconds, minutes, hours
date	:	day, month, year
book	:	author, title, price, year
city	:	name, country, population
address	:	name, door-number, street, city
inventory	:	item, stock, value
customer	:	name, telephone, city, category

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design. This chapter is devoted to the study of structures and their applications in program development. Another related concept known as *unions* is also discussed.

10.2 DEFINING A STRUCTURE

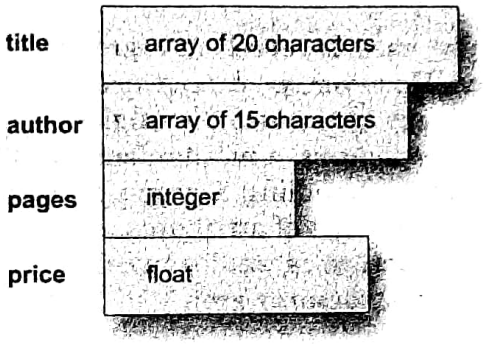
Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of

book name, author, number of pages, and price. We can define a structure to hold this information as follows:

```
struct book_bank
{
    char    title[20];
    char    author[15];
    int     pages;
    float   price;
};
```

The keyword **struct** declares a structure to hold the details of four data fields, namely **title, author, pages,** and **price**. These fields are called *structure elements* or *members*. Each member may belong to a different type of data. **book_bank** is the name of the structure and is called the *structure tag*. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above definition has not declared any variables. It simply describes a format called *template* to represent information as shown below:



The general format of a structure definition is as follows:

```
struct        tag_name
{
    data_type              member1;
    data_type              member2;
      ----                 ----
      ----                 ----
};
```

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as **book_bank** can be used to declare structure variables of its type, later in the program.

Arrays Vs Structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways.

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas a structure is a programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

10.3 DECLARING STRUCTURE VARIABLES

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types. It includes the following elements:

1. The keyword **struct.**
2. The structure tag name.
3. List of variable names separated by commas.
4. A terminating semicolon.

For example, the statement

```
struct book_bank, book1, book2, book3;
```

declares **book1, book2**, and **book3** as variables of type **struct book_bank.**

Each one of these variables has four members as specified by the template. The complete declaration might look like this:

```
struct book_bank
{
   char  title[20];
   char  author[15];
   int   pages;
   float price;
};
struct book_bank book1, book2, book3;
```

Remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as **book1.** When the compiler comes across a declaration statement, it reserves memory space for the structure variables. It is also allowed to combine both the structure definition and variables declaration in one statement.

The declaration

```
struct book_bank
{
   char title[20];
   char author[15];
   int pages;
   flat price;
} book1, book2, book3;
```

is valid. The use of tag name is optional here. For example:

```
struct
{ ........
  ........
  ........
} book1, book2, book3;
```

declares **book1, book2,** and **book3** as structure variables representing three books, but does not include a tag name. However, this approach is not recommended for two reasons.

1. Without a tag name, we cannot use it for future declarations:
2. Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the **main**, along with macro definitions, such as **#define.** In such cases, the definition is *global* and can be used by other functions as well.

Type-Defined Structures

We can use the keyword **typedef** to define a structure as follows:

```
typedef struct
{ . . . . .
  type member1;
  type member2;
  . . . . .
  . . . . .
} type_name;
```

The type_name represents structure definition associated with it and therefore can be used to declare structure variables as shown below:

type_name variable1, variable2, ... ;

Remember that (1) the name *type_name* is the type definition name, not a variable and (2) we cannot define a variable with *typedef* declaration.

10.4 ACCESSING STRUCTURE MEMBERS

We can access and assign values to the members of a structure in a number of ways. As mentioned earlier, the members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word **title**, has no meaning whereas the phrase 'title of book3' has a meaning. The link between a member and a variable is established using the *member operator* '.' which is also known as 'dot operator' or 'period operator'. For example,

```
book1.price
```

is the variable representing the price of **book1** and can be treated like any other ordinary variable. Here is how we would assign values to the members of **book1**:

```
strcpy(book1.title, "BASIC");
strcpy(book1.author, "Balagurusamy");
book1.pages = 250;
book1.price = 120.50;
```

We can also use **scanf** to give the values through the keyboard.

```
scanf("%s\n", book1.title);
scanf("%d\n", &book1.pages);
```

are valid input statements.

Example 10.1 Define a structure type, **struct personal** that would contain person name, date of joining and salary. Using this structure, write a program to read this information for one person from the keyboard and print the same on the screen.

Structure definition along with the program is shown in Fig. 10.1. The **scanf** and **printf** functions illustrate how the member operator '.' is used to link the structure members to the structure variables. The variable name with a period and the member name is used like an ordinary variable.

```
Program
   struct  personal
   {
        char  name[20];
        int   day;
        char  month[10];
        int   year;
        float salary;
   };
   main()
   {
        struct personal person;
      printf("Input Values\n");
```

```
        scanf("%s %d %s %d %f",
                person.name,
               &person.day,
                person.month,
               &person.year,
               &person.salary);
        printf("%s %d %s %d %f\n",
                person.name,
                person.day,
                person.month,
                person.year,
                person.salary);
}
```

Output

```
Input Values
M.L.Goel 10 January 1945 4500
M.L.Goel 10 January 1945 4500.00
```

Fig. 10.1 *Defining and accessing structure members*

10.5 STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```
main()
{
    struct
    {
      int weight;
      float height;
    }
    student = {60, 180.75};
    .....
    .....
}
```

This assigns the value 60 to **student. weight** and 180.75 to **student. height.** There is a one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main()
{
  struct st_record
  {
```

```
    int weight;
    float height;
  };
  struct st_record student1 = { 60, 180.75 };
  struct st_record student2 = { 53, 170.60 };
  .....
  .....
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
struct st_record
{
  int weight;
  float height;
} student1 = {60, 180.75};
main()
{
  struct st_record student2 = {53, 170.60};
  .....
  .....
}
```

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables.

Note that the compile-time initialization of a structure variable must have the following elements:

1. The keyword **struct**.
2. The structure tag name.
3. The name of the variable to be declared.
4. The assignment operator =.
5. A set of values for the members of the structure variable, separated by commas and enclosed in braces.
6. A terminating semicolon.

Rules for Initializing Structures

There are a few rules to keep in mind while initializing structure variables at compile-time.

1. We cannot initialize individual members inside the structure template.
2. The order of values enclosed in braces must match the order of members in the structure definition.
3. It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values as follows:

- Zero for integer and floating point numbers.
- '\0' for characters and strings.

10.6 COPYING AND COMPARING STRUCTURE VARIABLES

Two variables of the same structure type can be copied the same way as ordinary variables. If **person1** and **person2** belong to the same structure, then the following statements are valid:

```
person1 = person2;
person2 = person1;
```

However, the statements such as

```
person1 == person2
person1 != person2
```

are not permitted. C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

Example 10.2 Write a program to illustrate the comparison of structure variables.

The program shown in Fig. 10.2 illustrates how a structure variable can be copied into another of the same type. It also performs member-wise comparison to decide whether two structure variables are identical.

```
program
   struct class
   {
       int  number;
       char name[20];
       float marks;
   };

   main()
   {
       int  x;
       struct class student1 = {111,"Rao",72.50};
       struct class student2 = {222,"Reddy", 67.00};
       struct class student3;

       student3 = student2;

       x = ((student3.number ==  student2.number) &&
            (student3.marks  ==  student2.marks)) ? 1 : 0;

      if(x == 1)
      {
         printf("\nstudent2 and student3 are same\n\n");
```

```
        printf("%d %s %f\n", student3.number,
                             student3.name,
                             student3.marks);
    }
    else
        printf("\nstudent2 and student3 are different\n\n");

}
```

Output

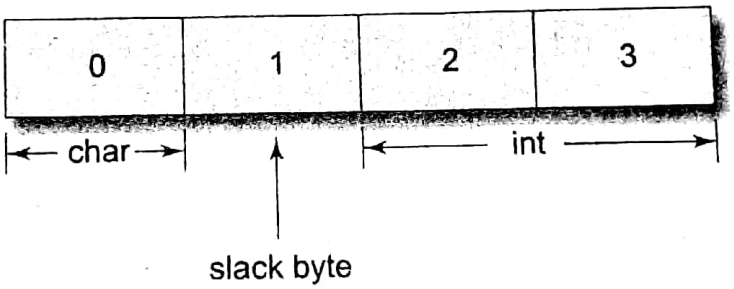
```
student2 and student3 are same

222 Reddy 67.000000
```

Fig. 10.2 *Comparing and copying structure variables*

Word Boundaries and Slack Bytes

Computer stores structures using the concept of "word boundary". The size of a word boundary is machine dependent. In a computer with two bytes word boundary, the members of a structure are stored left_aligned on the word boundary, as shown below. A character data takes one byte and an integer takes two bytes. One byte between them is left unoccupied. This unoccupied byte is known as the *slack byte*.



When we declare structure variables, each one of them may contain slack bytes and the values stored in such slack bytes are undefined. Due to this, even if the members of two variables are equal, their structures do not necessarily compare equal. C, therefore, does not permit comparison of structures. However, we can design our own function that could compare individual members to decide whether the structures are equal or not.

10.7 OPERATIONS ON INDIVIDUAL MEMBERS

As pointed out earlier, the individual members are identified using the member operator, the *dot*. A member with the *dot operator* along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators. Consider the program in Fig. 10.2. We can perform the following operations:

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

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

We can also apply increment and decrement operators to numeric type members. For example, the following statements are valid:

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

The precedence of the *member* operator is higher than all *arithmetic* and *relational* operators and therefore no parentheses are required.

Three Ways to Access Members

We have used the dot operator to access the members of structure variables. In fact, there are two other ways. Consider the following structure:

```
typedef struct
{
        int x;
        int y;
} VECTOR;

VECTOR v, *ptr;
ptr = & v;
```

The identifier **ptr** is known as **pointer** that has been assigned the address of the structure variable n. Now, the members can be accessed in three ways:

- using dot notation : v.x
- using indirection notation : (*ptr).x
- using selection notation : ptr –> x

The second and third methods will be considered in Chapter 11.

10.8 ARRAYS OF STRUCTURES

We use structures to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example:

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

defines an array called **student**, that consists of 100 elements. Each element is defined to be of the type **struct class.** 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}};
```

This 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;
```

Note that the array is declared just as it would have been with any other array. Since **student** is an array, we use the usual array-accessing methods to access individual elements and then the member operator to access members. Remember, each element of **student** array is a structure variable with three members.

An array of structures is stored inside the memory in the same way as a multi-dimensional array. The array **student** actually looks as shown in Fig. 10.3.

Example 10.3 For the **student** array discussed above, write a program to calculate the subject-wise and student-wise totals and store them as a part of the structure.

The program is shown in Fig. 10.4. We have declared a four-member structure, the fourth one for keeping the student-totals. We have also declared an **array** total to keep the subject-totals and the grand-total. The grand-total is given by **total.total.** Note that a member name can be any valid C name and can be the same as an existing structure variable name. The linked name **total.total** represents the **total** member of the structure variable **total**.

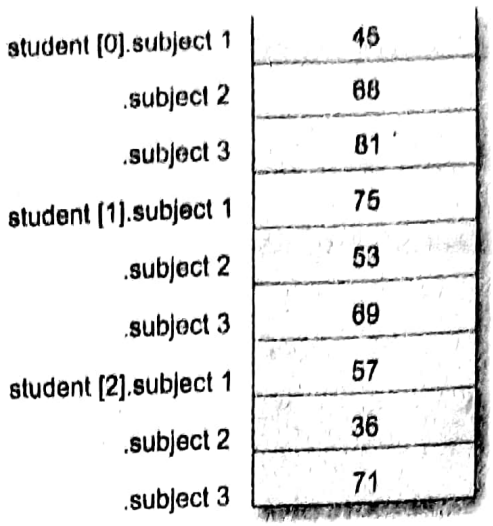


Fig. 10.3 *The array student inside memory*

```
Program
    struct marks
    {
        int  sub1;
        int  sub2;
        int  sub3;
        int  total;
    };
    main()
    {
        int  i;
        struct marks student[3] =  {{45,67,81,0},
                                    {75,53,69,0},
                                    {57,36,71,0}};

        struct marks total;
        for(i = 0; i <= 2; i++)
        {
            student[i].total = student[i].sub1 +
                               student[i].sub2 +
                               student[i].sub3;
            total.sub1 = total.sub1 + student[i].sub1;
            total.sub2 = total.sub2 + student[i].sub2;
            total.sub3 = total.sub3 + student[i].sub3;
            total.total = total.total + student[i].total;
        }
        printf(" STUDENT          TOTAL\n\n");
        for(i = 0; i <= 2; i++)
           printf("Student[%d]      %d\n", i+1,student[i].total);
        printf("\n SUBJECT          TOTAL\n\n");
        printf("%s       %d\n%s       %d\n",
```

```
            "Subject 1   ", total.sub1,
            "Subject 2   ", total.sub2,
            "Subject 3   ", total.sub3);

      printf("\nGrand Total = %d\n", total.total);
}
```

Output

```
        STUDENT          TOTAL
        Student[1]        193
        Student[2]        197

        Student[3]        164

        SUBJECT          TOTAL
        Subject 1         177
        Subject 2         156
        Subject 3         221

        Grand Total  = 554
```

Fig. 10.4 *Arrays of structures: Illustration of subscripted structure variables*

10.9 ARRAYS WITHIN STRUCTURES

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single-dimensional or multi-dimensional arrays of type **int** or **float.** For example, the following structure declaration is valid:

```
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, the name

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

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

Example 10.4 Rewrite the program of Example 10.3 using an array member to represent the three subjects.

The modified program is shown in Fig. 10.5. You may notice that the use of array name for subjects has simplified in code.

```
Program
   main()
   {
       struct  marks
       {
           int  sub[3];
           int  total;
       };
       struct marks student[3] =
       {45,67,81,0,75,53,69,0,57,36,71,0};

       struct marks total;
       int  i,j;

       for(i = 0; i <= 2; i++)
       {
          for(j = 0; j <= 2; j++)
          {
             student[i].total += student[i].sub[j];
             total.sub[j] += student[i].sub[j];
          }
          total.total += student[i].total;
       }
       printf("STUDENT          TOTAL\n\n");
       for(i = 0; i <= 2; i++)
          printf("Student[%d]      %d\n", i+1, student[i].total);

       printf("\nSUBJECT          TOTAL\n\n");
       for(j = 0; j <= 2; j++)
          printf("Subject-%d       %d\n", j+1, total.sub[j]);

       printf("\nGrand Total  =   %d\n", total.total);

   }
```

Output

```
STUDENT          TOTAL
Student[1]        193
Student[2]        197
Student[3]        164

STUDENT          TOTAL
Student-1         177
Student-2         156
Student-3         221

Grand Total  =    554
```

Fig. 10.5 *Use of subscripted members arrays in structures*

10.10 STRUCTURES WITHIN STRUCTURES

Structures within a structure means *nesting* of structures. Nesting of structures is permitted in C. Let us consider the following structure defined to store information about the salary of employees.

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

This structure defines name, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a substructure as shown below:

```
struct salary
{
   char name;
   char department;
   struct
   {
      int dearness;
      int house_rent;
      int city;
   }
   allowance;
}
employee;
```

The salary structure contains a member named **allowance**, which itself is a structure with three members. The members contained in the inner structure namely **dearness, house_rent**, and **city** can be referred to as:

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

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator. The following are invalid:

employee.allowance (actual member is missing)
employee.house_rent (inner structure variable is missing)

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct salary
{
  .....
  struct
  {
    int dearness;
    .....
  }
  allowance,
  arrears;
}
employee[100];
```

The inner structure has two variables, **allowance** and **arrears**. This implies that both of them have the same structure template. Note the comma after the name **allowance**. A base member can be accessed as follows:

employee[1].allowance.dearness
employee[1].arrears.dearness

We can also use tag names to define inner structures. Example:

```
struct pay
  {
    int dearness;
    int house_rent;
    int city;
  };
  struct salary
  {
    char name;
    char department;
    struct pay allowance;
    struct pay arrears;
  };
  struct salary employee[100];
```

pay template is defined outside the **salary** template and is used to define the structure of **allowance** and **arrears** inside the **salary** structure.

It is also permissible to nest more than one type of structures.

```
struct personal_record
{
  struct name_part name;
  struct addr_part address;
  struct date date_of_birth;
  .....
  .....
};
struct personal_record person1;
```

The first member of this structure is **name**, which is of the type **struct name_part**. Similarly, other members have their structure types.

NOTE: C permits nesting upto 15 levels. However, C99 allows 63 levels of nesting.

10.11 STRUCTURES AND FUNCTIONS

We know that the main philosophy of C language is the use of functions. And therefore, it is natural that C supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from one function to another.

1. The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.
2. The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.
3. The third approach employs a concept called *pointers* to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to function. This method is more efficient as compared to the second one.

In this section, we discuss in detail the second method, while the third approach using pointers is discussed in the next chapter, where pointers are dealt in detail.

The general format of sending a copy of a structure to the called function is:

```
function_name (structure_variable_name);
```

The called function takes the following form:

```
data_type function_name(struct_type st_name)
{
   ......
   ......
   return(expression);
}
```

The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as **struct** with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
3. The **return** statement is necessary only when the function is returning some data back to the calling function. The *expression* may be any simple variable or structure variable or an expression using simple variables.

4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called functions must be declared in the calling function appropriately.

Example 10.5 Write a simple program to illustrate the method of sending an entire structure as a parameter to a function.

program to update an item is shown in Fig. 10.6. The function **update** receives a copy of ne structure variable **item** as one of its parameters. Note that both the function **update** nd the formal parameter **product** are declared as type **struct stores.** It is done so because the function uses the parameter **product** to receive the structure variable **item** nd also to return the updated values of **item**.

The function **mul** is of type **float** because it returns the product of **price** and **quantity.** However, the parameter **stock,** which receives the structure variable **item** is declared as type **struct stores.**

The entire structure returned by **update** can be copied into a structure of identical type. The statement

```
item = update(item,p_increment,q_increment);
```

replaces the old values of **item** by the new ones.

Program

```
/*        Passing a copy of the entire structure .        */
  struct stores
  {
      char  name[20];
      float price;
      int   quantity;
  };
  struct stores update (struct stores product, float p, int q);
  float mul (struct stores stock);
  main()
  {
      float    p_increment, value;
      int      q_increment;

      struct stores item = {"XYZ", 25.75, 12};

      printf("\nInput increment values:");
      printf("   price increment and quantity increment\n");
      scanf("%f %d", &p_increment, &q_increment);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
      item  = update(item, p_increment, q_increment);
  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
      printf("Updated values of item\n\n");
```

```
    printf("Name       : %s\n",item.name);
    printf("Price      : %f\n",item.price);
    printf("Quantity   : %d\n",item.quantity);
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
    value  = mul(item);
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
    printf("\nValue of the item  =  %f\n", value);
}
struct stores update(struct stores product, float p, int q)
{

    product.price += p;
    product.quantity += q;
    return(product);
}
float mul(struct stores stock)
{
    return(stock.price * stock.quantity);
}
```

Output

```
Input increment values:   price increment and quantity increment
10 12
Updated values of item
Name       : XYZ
Price      : 35.750000
Quantity   : 24
Value of the item  =  858.000000
```

Fig. 10.6 *Using structure as a function parameter*

You may notice that the template of **stores** is defined before **main()**. This has made the data type **struct stores** as *global* and has enabled the functions **update** and **mul** to make use of this definition.

10.12 UNIONS

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword union as follows:

```
union item
{
   int m;
   float x;
   char c;
} code;
```

This declares a variable **code** of type **union item.** The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

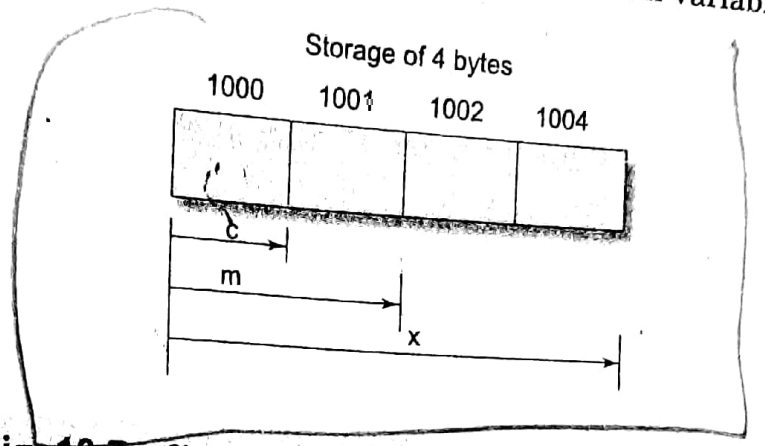


Fig. 10.7 *Sharing of a storage locating by union members*

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member x requires 4 bytes which is the largest among the members. Figure 10.7 shows how all the three variables share the same address. This assumes that a float variable requires 4 bytes of storage.

To access a union member, we can use the same syntax that we use for structure members. That is,

```
code.m
code.x
code.c
```

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statements such as

```
code.m = 379;
code.x = 7859.36;
printf("%d", code.m);
```

would produce erroneous output (which is machine dependent).

In effect, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous member's value.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

Unions may be initialized when the variable is declared. But, unlike structures, it can be initialized only with a value of the same type as the first union member. For example, with the preceding, the declaration

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

is valid but the declaration

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

is invalid. This is because the type of the first member is **int**. Other members can be initialized by either assigning values or reading from the keyboard.

10.13 SIZE OF STRUCTURES

We normally use structures, unions, and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the unary operator **sizeof** to tell us the size of a structure (or any variable). The expression

```
sizeof(struct x)
```

will evaluate the number of bytes required to hold all the members of the structure **x**. If **y** is a simple structure variable of type **struct x**, then the expression

```
sizeof(y)
```

would also give the same answer. However, if **y** is an array variable of type **struct x**, then

```
sizeof(y)
```

would give the total number of bytes the array **y** requires.

This kind of information would be useful to determine the number of records in a database. For example, the expression

```
sizeof(y)/sizeof(x)
```

would give the number of elements in the array **y**.

10.14 BIT FIELDS

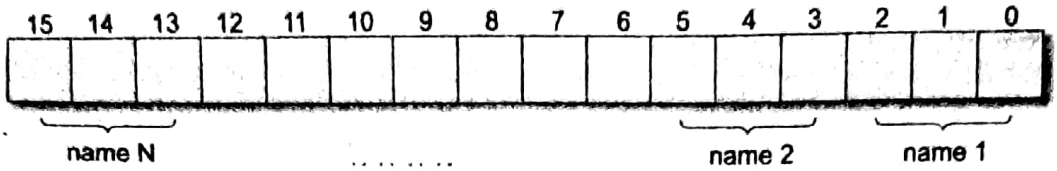
So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small *bit fields* to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of string of a string of preselected bits as if it represented an integral quantity.

A *bit field* is a set of adjacent bits whose size can be from 1 to 16 bits in length. A word can therefore be divided into a number of bit fields. The name and size of bit fields are defined using a structure. The general form of bit field definition is:

```
struct tag-name
{
    data-type name1: bit-length;
    data-type name2: bit-length;
    . . . . . .
    . . . . . .
    data-type nameN: bit-length;
}
```

The *data-type* is either **int** or **unsigned int** or **signed int** and the *bit-length* is the number of bits used for the specified name. Remember that a **signed** bit field should have at least 2 bits (one bit for sign). Note that the field name is followed by a colon. The *bit-length* is decided by the range of value to be stored. The largest value that can be stored is 2^{n-1}, where **n** is bit-length.

The internal representation of bit fields is machine dependent. That is, it depends on the size of **int** and the ordering of bits. Some machines store bits from left to right and others from right to left. The sketch below illustrates the layout of bit fields, assuming a 16-bit word that is ordered from right to left.



There are several specific points to observe:

1. The first field always starts with the first bit of the word.
2. A bit field cannot overlap integer boundaries. That is, the sum of lengths of all the fields in a structure should not be more than the size of a word. In case, it is more, the overlapping field is automatically forced to the beginning of the next word.
3. There can be unnamed fields declared with size. Example:

   Unsigned : *bit-length*

   Such fields provide padding within the word.
4. There can be unused bits in a word.
5. We cannot take the address of a bit field variable. This means we cannot use **scanf** to read values into bit fields. We can neither use pointer to access the bit fields.
6. Bit fields cannot be arrayed.
7. Bit fields should be assigned values that are within the range of their size. If we try to assign larger values, behaviour would be unpredicted.

Suppose, we want to store and use personal information of employees in compressed form, this can be done as follows:

```
struct personal
{
unsigned sex          :    1
unsigned age          :    7
unsigned m_status     :    1
unsigned children     :    3
unsigned              :    4
} emp;
```

This defines a variable name **emp** with four bit fields. The range of values each field could have is follows:

Bit field	*Bit length*	*Range of value*
sex	1	0 or 1
age	7	0 or 127 ($2^7 - 1$)
m_status	1	0 or 1
children	3	0 to 7 (2^3–1)

Once bit fields are defined, they can be referenced just as any other structure-type data item would be referenced. The following assignment statements are valid.

```
emp.sex = 1;
emp.age = 50;
```

Remember, we cannot use **scanf** to read values into a bit field. We may have to read into a temporary variable and then assign its value to the bit field. For example:

```
scanf(%d %d", &AGE,&CHILDREN);
emp.age = AGE;
emp.children = CHILDREN;
```

One restriction in accessing bit fields is that a pointer cannot be used. However, they can be used in normal expressions like any other variable. For example:

```
sum = sum + emp.age;
if(emp.m_status). . . . .;
printf("%d\n", emp.age);
```

are valid statements.

It is possible to combine normal structure elements with bit field elements. For example:

```
struct personal
{
        char        name[20];   /* normal variable */
        struct addr address;    /* structure variable */
        unsigned    sex : 1;
        unsigned    age : 7;
        . . . . .
        . . . . .
}
emp[100];
```

This declares **emp** as a 100 element array of type **struct personal.** This combines normal variable name and structure type variable **address** with bit fields.

Bit fields are packed into words as they appear in the definition. Consider the following definition.

```
struct pack
{
        unsigned a:2;
        int count;
        unsigned b : 3;
};
```

Here, the bit field **a** will be in one word, the variable **count** will be in the second word and the bit field **b** will be in the third word. The fields **a** and **b** would not get packed into the same word.

NOTE: Other related topics such as 'Structures with Pointers' and 'Structures and Linked Lists' are discussed in Chapter 11 and Chapter 12, respectively.