Chapter 10

Structures and Unions

LEARNING OBJECTIVES

- LO 10.1 Explain how structures are used in a program
- LO 10.2 Describe how structure variables and members are manipulated
- LO 10.3 Discuss structures and arrays
- LO 10.4 Illustrate nested structures and 'structures and functions'
- LO 10.5 Determine how structures and unions differ in terms of their storage technique

INTRODUCTION

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.

DEFINING A STRUCTURE

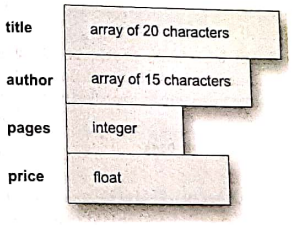
LO 10.1 Explain how structures are used in a program

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.

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.

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*

STRUCTURE INITIALIZATION

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

```
main()
{
      struct
      {
         int weight;
         float height;
```

Rules for Initializing Structures

There are a few rules to keep in mind while initializing structure variables at compile-time which are as follows:

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.

COPYING AND COMPARING STRUCTURE VARIABLES

LO 10.2
Describe how structure variables and members are manipulated

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.

WORKED-OUT PROBLEM 10.3

M

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.

```
        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

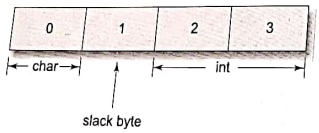
```
student2 and student3 are same

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.

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 the following 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.

ARRAYS OF STRUCTURES

LO 10.3
Discuss structures and arrays

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  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%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*

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.

WORKED-OUT PROBLEM 10.5

E

Rewrite the program of Program 10.4 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*

STRUCTURES WITHIN STRUCTURES

LO 10.4 Illustrate nested structures and 'structures and functions'

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;
```

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.*

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,

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.

UNIONS

LO 10.5
Determine how structures and unions differ in terms of their storage technique

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.

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.

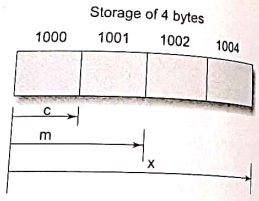


Fig. 10.7 *Sharing of a storage locating by union members*

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.

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**.

BIT FIELDS

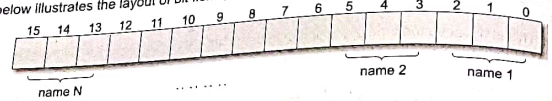
So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small *bit fields* to hold data items and thereby to pack several data items in a 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.*

Scan the QR code to access the online course on Structures and Unions
OR visit the link
http://qrcode.flipick.com/index.php/369

KEY CONCEPTS

- **ARRAY:** It is a fixed-size sequenced collection of elements of the same data type. **[LO 10.1]**
- **DOT OPERATOR:** This links a structure variable with a structure member. It is used to read/write member values. **[LO 10.1]**
- **STRUCTURE:** This is a user-defined data type that allows different data types to be combined together to represent a data record. **[LO 10.1]**
- **UNION:** It is similar to a structure in syntax but differs in storage technique. Unlike structures, union members use the same memory location for storing all member values. **[LO 10.5]**
- **BIT FIELD:** This refers to a set of adjacent bits with size ranging from 1 to 16 bits. **[LO 10.5]**

ALWAYS REMEMBER

- Remember to place a semicolon at the end of definition of structures and unions. **[LO 10.1]**
- We can declare a structure variable at the time of definition of a structure by placing it after the closing brace but before the semicolon. **[LO 10.1]**
- Do not place the structure tag name after the closing brace in the definition. That will be treated as a structure variable. The tag name must be placed before the opening brace but after the keyword **struct**. **[LO 10.1]**
- When we use **typedef** definition, the *type_name* comes after the closing brace but before the semicolon. **[LO 10.1]**
- We cannot declare a variable at the time of creating a **typedef** definition. We must use the *type_name* to declare a variable in an independent statement. **[LO 10.1]**
- It is an error to use a structure variable as a member of its own **struct** type structure. **[LO 10.1]**
- Declaring a variable using the tag name only (without the keyword struct) is an error. **[LO 10.1]**
- It is illegal to refer to a structure member using only the member name. **[LO 10.1]**
- When using **scanf** for reading values for members, we must use address operator & with non-string members. **[LO 10.1]**
- Always provide a structure tag name when creating a structure. It is convenient to use tag name to declare new structure variables later in the program. **[LO 10.1]**
- Use short and meaningful structure tag names. **[LO 10.1]**
- Avoid using same names for members of different structures (although it is not illegal). **[LO 10.1]**
- It is an error to compare two structure variables. **[LO 10.2]**
- Assigning a structure of one type to a structure of another type is an error. **[LO 10.2]**

- When accessing a member with a pointer and dot notation, parentheses are required around the pointer, like (*ptr).number. **[LO 10.2]**
- The selection operator (–>) is a single token. Any space between the symbols – and > is an error. **[LO 10.2]**
- Forgetting to include the array subscript when referring to individual structures of an array of structures is an error. **[LO 10.3]**
- When structures are nested, a member must be qualified with all levels of structures nesting it. **[LO 10.4]**
- Passing structures to functions by pointers is more efficient than passing by value. (Passing by pointers are discussed in Chapter 11.) **[LO 10.4]**
- A union can store only one of its members at a time. We must exercise care in accessing the correct member. Accessing a wrong data is a logic error. **[LO 10.5]**
- It is an error to initialize a union with data that does not match the type of the first member. **[LO 10.5]**
- We cannot take the address of a bit field. Therefore, we cannot use **scanf** to read values in bit fields. We can neither use pointer to access the bit fields. **[LO 10.5]**
- Bit fields cannot be arrayed. **[LO 10.5]**

BRIEF CASES

1. Book Shop Inventory [LO 10.1, 10.2, 10.3, 10.4, 10.5 M]

A book shop uses a personal computer to maintain the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher, stock position, etc. Whenever a customer wants a book, the shopkeeper inputs the title and author of the book and the system replies whether it is in the list or not. If it is not, an appropriate message is displayed. If book is in the list, then the system displays the book details and asks for number of copies. If the requested copies are available, the total cost of the books is displayed; otherwise the message "Required copies not in stock" is displayed.

A program to accomplish this is shown in Fig. 10.8. The program uses a template to define the structure of the book. Note that the date of publication, a member of **record** structure, is also defined as a structure.

When the title and author of a book are specified, the program searches for the book in the list using the function

look_up(table, s1, s2, m)

The parameter **table** which receives the structure variable **book** is declared as type **struct record.** The parameters **s1** and **s2** receive the string values of **title** and **author** while **m** receives the total number of books in the list. Total number of books is given by the expression

sizeof(book)/sizeof(struct record)

The search ends when the book is found in the list and the function returns the serial number of the book. The function returns –1 when the book is not found. Remember that the serial number of the first book in the list is zero. The program terminates when we respond "NO" to the question

Do you want any other book?

Note that we use the function

get(string)

to get title, author, etc. from the terminal. This enables us to input strings with spaces such as "C Language". We cannot use **scanf** to read this string since it contains two words.

LEARNING OBJECTIVES

LO 11.1 Know the concept of pointers
LO 11.2 Determine how pointer variables are used in a program
LO 11.3 Describe chain of pointers
LO 11.4 Illustrate pointer expressions
LO 11.5 Discuss pointers and arrays
LO 11.6 Explain how pointers are used with functions and structures

INTRODUCTION

A pointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory addresses as their values. Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Pointers are undoubtedly one of the most distinct and exciting features of C language. It has added power and flexibility to the language. Although they appear little confusing and difficult to understand for a beginner, they are a powerful tool and handy to use once they are mastered.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They include:

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.

7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Of course, the real power of C lies in the proper use of pointers. In this chapter, we will examine the pointers in detail and illustrate how to use them in program development. Chapter 13 examines the use of pointers for creating and managing linked lists.

UNDERSTANDING POINTERS

The computer's memory is a sequential collection of *storage cells* as shown in Fig. 11.1. Each cell, commonly known as a *byte*, has a number called *address* associated with it. Typically, the addresses are numbered consecutively, starting from zero. The last address depends on the memory size. A computer system having 64 K memory will have its last address as 65,535.

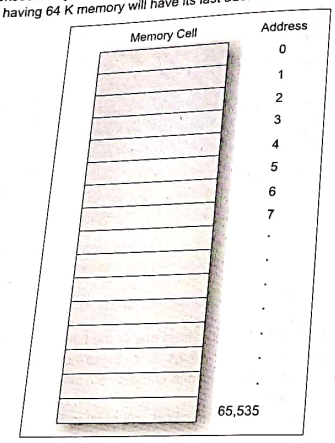


Fig. 11.1 *Memory organisation*

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number. Consider the following statement

```
int quantity = 179;
```

This statement instructs the system to find a location for the integer variable **quantity** and puts the value 179 in that location. Let us assume that the system has chosen the address location 5000 for **quantity**. We may represent this as shown in Fig. 11.2. (Note that the address of a variable is the address of the first bye occupied by that variable.)

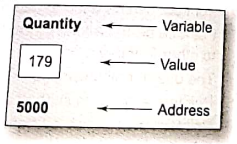


Fig. 11.2 *Representation of a variable*

During execution of the program, the system always associates the name **quantity** with the address 5000. (This is something similar to having a house number as well as a house name.) We may have access to the value 179 by using either the name **quantity** or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables, that can be stored in memory, like any other variable. Such variables that hold memory addresses are called *pointer variables*. A pointer variable is, therefore, nothing but a variable that contains an address, which is a location of another variable in memory.

Remember, since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of **quantity** to a variable **p**. The link between the variables **p** and **quantity** can be visualized as shown in Fig.11.3. The address of **p** is 5048.

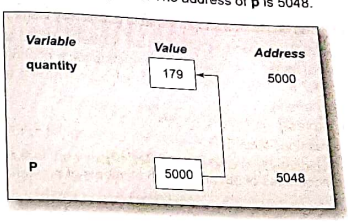
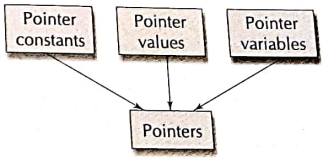


Fig. 11.3 *Pointer variable*

Since the value of the variable **p** is the address of the variable **quantity**, we may access the value of **quantity** by using the value of **p** and therefore, we say that the variable **p** 'points' to the variable **quantity**. Thus, **p** gets the name 'pointer'. (We are not really concerned about the actual values of pointer variables. They may be different everytime we run the program. What we are concerned about is the relationship between the variables **p** and **quantity**.)

Underlying Concepts of Pointers

Pointers are built on the three underlying concepts as illustrated below:



Memory addresses within a computer are referred to as *pointer constants*. We cannot change them; we can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as *pointer value*. The pointer value (i.e. the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a *pointer variable*.

Fig. 11.4 *Accessing the address of a variable*

DECLARING POINTER VARIABLES

LO 11.2
Determine how pointer variables are used in a program

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

***data_type* *pt_name;**

This tells the compiler three things about the variable **pt_name**.

1. The asterisk (*) tells that the variable **pt_name** is a pointer variable.
2. **pt_name** needs a memory location.
3. **pt_name** points to a variable of type *data_type*.

For example,

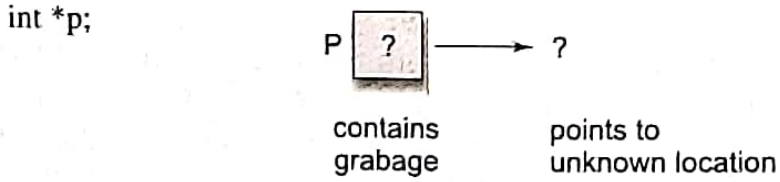
```
int *p; /* integer pointer */
```

declares the variable **p** as a pointer variable that points to an integer data type. Remember that the type **int** refers to the data type of the variable being pointed to by **p** and not the type of the value of the pointer. Similarly, the statement

```
float *x;   / * float pointer */
```

declares **x** as a pointer to a floating-point variable.

The declarations cause the compiler to allocate memory locations for the pointer variables **p** and **x**. Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:



2. This style matches with the format used for accessing the target values. Example:

```
int x, *p, y;
x = 10;
p = & x;
y = *p;     /* accessing x through p */
*p = 20;    /* assigning 20 to x */
```

We use in this book the style 2, namely,

```
int *p;
```

INITIALIZATION OF POINTER VARIABLES

The process of assigning the address of a variable to a pointer variable is known as *initialization*. As pointed out earlier, all uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable. Example:

```
int quantity;
int *p;                 /* declaration */
p = &quantity;          /* initialization */
```

We can also combine the initialization with the declaration. That is,

```
int *p = &quantity;
```

is allowed. The only requirement here is that the variable **quantity** must be declared before the initialization takes place. Remember, this is an initialization of **p** and not ***p**.

We must ensure that the pointer variables always point to the corresponding type of data. For example,

```
float a, b;
int x, *p;
p = &a;          /* wrong */
b = *p;
```

will result in erroneous output because we are trying to assign the address of a **float** variable to an ***integer pointer***. When we declare a pointer to be of int type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step. For example,

```
int x, *p = &x;          /* three in one */
```

is perfectly valid. It declares **x** as an integer variable and **p** as a pointer variable and then initializes **p** to the address of **x**. And also remember that the target variable **x** is declared first. The statement

```
int *p = &x, x;
```

is not valid.

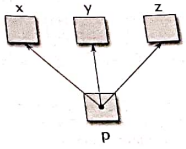
We could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statements are valued

```
int *p = NULL;
int *p = 0;
```


Pointer Flexibility

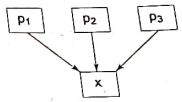
Pointers are flexible. We can make the same pointer to point to different data variables in different statements. Example;

```
int x, y, z, *p;
. . . . .
p = &x;
. . . . .
p = &y;
. . . . .
p = &z;
. . . . .
```



We can also use different pointers to point to the same data variable. Example;

```
int x;
int *p1 = &x;
int *p2 = &x;
int *p3 = &x;
. . . . .
. . . . .
```



With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

```
int *p = 5360;          / *absolute address */
```

ACCESSING A VARIABLE THROUGH ITS POINTER

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer? This is done by using another unary operator * (asterisk), usually known as the *indirection operator*. Another name for the indirection operator is the *dereferencing operator*. Consider the following statements:

```
int quantity, *p, n;
quantity = 179;
p = &quantity;
n = *p;
```

The first line declares **quantity** and **n** as integer variables and **p** as a pointer variable pointing to an integer. The second line assigns the value 179 to **quantity** and the third line assigns the address of **quantity** to the pointer variable **p**. The fourth line contains the indirection operator *. When the operator* is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address. In this case, ***p** returns the value of the variable **quantity**, because **p** is the address of **quantity**. The * can be remembered as 'value at address'. Thus, the value of **n** would be 179. The two statements

```
p = &quantity;
n = *p;
```

are equivalent to

```
n = *&quantity;
```

which in turn is equivalent to

```
n = quantity;
```

In C, the assignment of pointers and addresses is always done symbolically, by means of symbolic names. You cannot access the value stored at the address 5368 by writing *5368. It will not work. Program 11.2 illustrates the distinction between pointer value and the value it points to.

WORKED-OUT PROBLEM 11.2

E

Write a program to illustrate the use of indirection operator '' to access the value pointed to by a pointer.*

The program and output are shown in Fig.11.5. The program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer **ptr** is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

```
x = *(&x) = *ptr = y
&x = &*ptr
```

Program

```
main()
{
    int  x, y;
    int  *ptr;
    x = 10;
    ptr = &x;
    y = *ptr;
    printf("Value of x is %d\n\n",x);
    printf("%d is stored at addr %u\n", x, &x);
    printf("%d is stored at addr %u\n", *&x, &x);
    printf("%d is stored at addr %u\n", *ptr, ptr);
    printf("%d is stored at addr %u\n", ptr, &ptr);
    printf("%d is stored at addr %u\n", y, &y);
    *ptr = 25;
    printf("\nNow x = %d\n",x);

}
```

Output

```
Value of x is 10

10   is stored at addr 4104
10   is stored at addr 4104
10   is stored at addr 4104
4104 is stored at addr 4106
10   is stored at addr 4108

Now x = 25
```

Fig. 11.5 *Accessing a variable through its pointer*

The actions performed by the program are illustrated in Fig. 11.6. The statement **ptr = &x** assigns the address of **x** to **ptr** and **y = *ptr** assigns the value pointed to by the pointer **ptr** to **y**. Note the use of the assignment statement

```
*ptr = 25;
```

This statement puts the value of 25 at the memory location whose address is the value of **ptr**. We know that the value of **ptr** is the address of **x** and therefore, the old value of **x** is replaced by 25. This, in effect, is equivalent to assigning 25 to **x**. This shows how we can change the value of a variable *indirectly* using a pointer and the *indirection operator*.

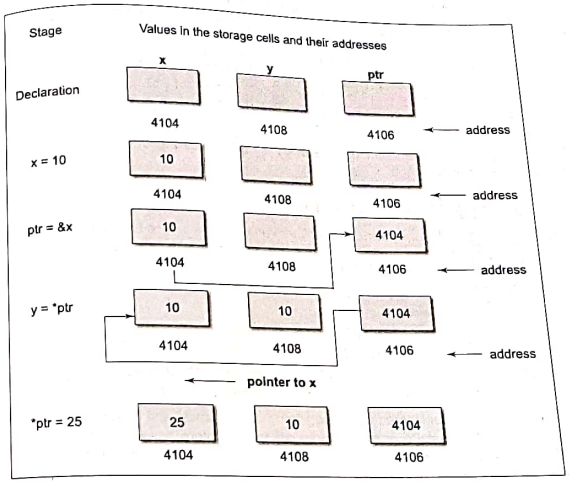
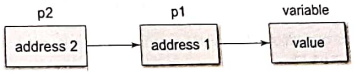


Fig. 11.6 *Illustration of pointer assignments*

CHAIN OF POINTERS

LO 11.3 Describe chain of pointers

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown.



Here, the pointer variable **p2** contains the address of the pointer variable **p1**, which points to the location that contains the desired value. This is known as *multiple indirections*.

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. Example:

```
int **p2;
```

This declaration tells the compiler that **p2** is a pointer to a pointer of **int** type. Remember, the pointer **p2** is not a pointer to an integer, but rather a pointer to an integer pointer.

We can access the target value indirectly pointed to by pointer to a pointer by applying the indirection operator twice. Consider the following code:

```
main ( )
{
    int x, *p1, **p2;
    x = 100;
    p1 = &x;          /* address of  x */
    p2 = &p1          /* address of  p1 */
    printf ("%d",     **p2);
}
```

This code will display the value 100. Here, **p1** is declared as a pointer to an integer and **p2** as a pointer to a pointer to an integer.

POINTER EXPRESSIONS

LO 11.4 Illustrate pointer expressions

Like other variables, pointer variables can be used in expressions. For example, if **p1** and **p2** are properly declared and initialized pointers, then the following statements are valid:

```
y = *p1 * *p2;          same as (*p1) * (*p2)
sum = sum + *p1;
z = 5* – *p2/ *p1;      same as (5 * (– (*p2)))/(*p1)
*p2 = *p2 + 10;
```

Note that there is a blank space between / and * in the item3 above. The following is wrong:

```
z = 5* – *p2 /*p1;
```

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. p1 + 4, p2–2, and p1 – p2 are all allowed. If p1 and p2 are both pointers to the same array, then **p2 – p1** gives the number of elements between **p1** and **p2**.

We may also use short-hand operators with the pointers.

```
p1++;
–p2;
sum += *p2;
```

In addition to arithmetic operations discussed above, pointers can also be compared using the relational operators. The expressions such as **p1 > p2, p1 == p2**, and **p1 != p2** are allowed. However, any comparison of pointers that refer to separate and unrelated variables makes no sense. Comparisons can be used meaningfully in handling arrays and strings.

We may not use pointers in division or multiplication. For example, expressions such as

```
p1 / p2 or p1 * p2 or p1 / 3
```

are not allowed. Similarly, two pointers cannot be added. That is, p1 + p2 is illegal.

WORKED-OUT PROBLEM 11.3

M

Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig.11.7 shows how the pointer variables can be directly used in expressions. It also illustrates the order of evaluation of expressions. For example, the expression

```
4* – *p2 / *p1 + 10
```

is evaluated as follows:

```
((4 * (–(*p2))) / (*p1)) + 10
```

When *p1 = 12 and *p2 = 4, this expression evaluates to 9. Remember, since all the variables are of type int, the entire evaluation is carried out using the integer arithmetic.

Program

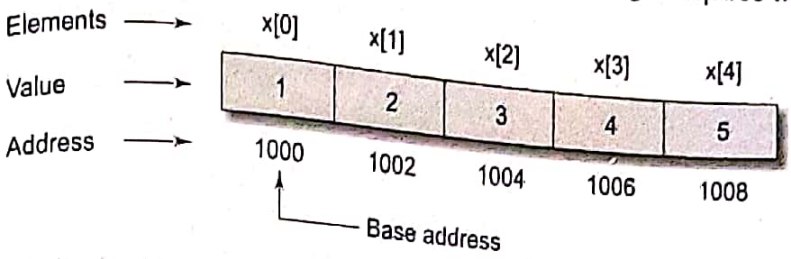
```
main()
{
    int  a, b, *p1, *p2, x, y, z;
    a  = 12;
    b  =  4;
    p1 = &a;
    p2 = &b;
    x  =  *p1 * *p2 – 6;
    y  =  4*  – *p2 / *p1 + 10;
    printf("Address of a = %u\n", p1);
    printf("Address of b = %u\n", p2);
    printf("\n");
    printf("a = %d, b = %d\n", a, b);
    printf("x = %d, y = %d\n", x, y);
    *p2  = *p2 + 3;
    *p1  = *p2 – 5;
    z    = *p1 * *p2 – 6;
    printf("\na = %d, b = %d,", a, b);
    printf(" z = %d\n", z);
}
```

Output

```
Address of a = 4020
Address of b = 4016
a = 12, b = 4
x = 42, y = 9
a = 2, b = 7, z = 8
```

Fig. 11.7 *Evaluation of pointer expressions*

Suppose the base address of x is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows:



The name **x** is defined as a constant pointer pointing to the first element, **x[0]** and therefore the value of **x** is 1000, the location where **x[0]** is stored. That is,

```
x = &x[0] = 1000
```

If we declare **p** as an integer pointer, then we can make the pointer **p** to point to the array **x** by the following assignment:

```
p = x;
```

This is equivalent to

```
p = &x[0];
```

Now, we can access every value of **x** using p++ to move from one element to another. The relationship between **p** and **x** is shown as:

```
p = &x[0] (= 1000)
p+1 = &x[1] (= 1002)
p+2 = &x[2] (= 1004)
p+3 = &x[3] (= 1006)
p+4 = &x[4] (= 1008)
```

You may notice that the address of an element is calculated using its index and the scale factor of the data type. For instance,

address of **x[3]** = base address + (3 x scale factor of **int**)

= 1000 + (3 x 2) = 1006

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that ***(p+3)** gives the value of **x[3]**. The pointer accessing method is much faster than array indexing.

The Worked-Out Problem 11.4 illustrates the use of pointer accessing method.

WORKED-OUT PROBLEM 11.4

M

Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in Fig. 11.8 illustrates how a pointer can be used to traverse an array element. Since incrementing an array pointer causes it to point to the next element, we need only to add one to p each time we go through the loop.

Program

```
main()
{
    int *p, sum, i;
    int x[5] = {5,9,6,3,7};
```

```
    i  = 0;
    p  = x;      /* initializing with base address of x */
    printf("Element   Value   Address\n\n");
    while(i < 5)
{
    printf(" x[%d] %d %u\n", i, *p, p);
    sum = sum + *p;   /* accessing array element  */
    i++, p++;         /* incrementing pointer     */
}
printf("\n  Sum    =  %d\n", sum);
printf("\n  &x[0]  =  %u\n", &x[0]);
printf("\n  p      =  %u\n", p);
}
```

Output

```
Element      Value      Address
   x[0]        5          166
   x[1]        9          168
   x[2]        6          170
   x[3]        3          172
   x[4]        7          174
   Sum    =  55
   &x[0]  =  166
   p      =  176
```

Program

```
main()
{
      char  *name;
      int   length;
      char  *cptr = name;
      name  = "DELHI";
      printf ("%s\n", name);
      while(*cptr != '\0')
      {
           printf("%c is stored at address %u\n", *cptr, cptr);
           cptr++;
      }
      length = cptr - name;
      printf("\nLength of the string = %d\n", length);
}
```

Output

```
DELHI
D is stored at address 54
E is stored at address 55
L is stored at address 56
H is stored at address 57
I is stored at address 58

Length of the string = 5
```

Remember the difference between the notations ***p[3]** and **(*p)[3]**. Since * has a lower precedence than [ ], *p[3] declares p as an array of 3 pointers while **(*p)[3]** declares **p** as a pointer to an array of three elements.

POINTERS AS FUNCTION ARGUMENTS

LO 11.6
Explain how pointers are used with functions and structures

We have seen earlier that when an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. If **x** is an array, when we call **sort(x)**, the address of **x[0]** is passed to the function **sort**. The function uses this address for manipulating the array elements. Similarly, we can pass the address of a variable as an argument to a function in the normal fashion. We used this method when discussing functions that return multiple values (see Chapter 9).

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variables is known as '*call by reference*'. (You know, the process of passing the actual value of variables is known as "call by value".) The function which is called by 'reference' can change the value of the variable used in the call.

Consider the following code:

```
main()
{
        int x;
        x = 20;
        change(&x);  /* call by reference or address */
        printf("%d\n",x);
}
change(int *p)
{
        *p = *p + 10;
}
```

When the function **change( )** is called, the address of the variable **x**, not its value, is passed into the function **change()**. Inside **change( )**, the variable **p** is declared as a pointer and therefore **p** is the address of the variable **x**. The statement,

```
*p = *p + 10;
```

means 'add 10 to the value stored at the address **p**'. Since **p** represents the address of **x**, the value of **x** is changed from 20 to 30. Therefore, the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function. Note that this mechanism is also known as "*call by address*" or "*pass by pointers*".

Note *C99 adds a new qualifier* ***restrict*** *to the pointers passed as function parameters. See the Appendix "C99 Features".*

The program in Fig. 11.11 shows how the contents of two locations can be exchanged using their address locations. The function **exchange()** receives the addresses of the variables **x** and **y** and exchanges their contents.

Program

```
void exchange (int *, int *);     /* prototype */
main()
{
     int  x, y;
     x = 100;
     y = 200;
     printf("Before exchange  : x = %d   y = %d\n\n", x, y);
     exchange(&x,&y);      /* call */
     printf("After exchange   : x = %d   y = %d\n\n", x, y);
}
exchange (int *a, int *b)
{
     int t;
     t = *a;    /* Assign the value at address a to t */
     *a = *b;   /* put b into a */
     *b = t;    /* put t into b */
}
```

Output

```
Before exchange  : x = 100   y = 200
After exchange   : x = 200   y = 100
```

Fig. 11.11 *Passing of pointers as function parameters*

You may note the following points:

1. *The function parameters are declared as pointers.*
2. *The dereferenced pointers are used in the function body.*
3. *When the function is called, the addresses are passed as actual arguments.*

The use of pointers to access array elements is very common in C. We have used a pointer to traverse array elements in Program 11.4. We can also use this technique in designing user-defined functions discussed in Chapter 9. Let us consider the problem sorting an array of integers discussed in Program 9.6.

The function **sort** *may be written using pointers (instead of array indexing) as shown:*

```
void sort (int m, int *x)
{    int i, j, temp;
     for (i=1; i<= m-1; i++)
          for (j=1; j<= m-1; j++)
          if (*(x+j-1) >= *(x+j))
          {
               temp = *(x+j-1);
               *(x+j-1) = *(x+j);
               *(x+j) = temp;
          }
}
```


Note that we have used the pointer x (instead of array x[ ]) to receive the address of array passed and therefore the pointer x can be used to access the array elements (as pointed out in Section 11.10). This function can be used to sort an array of integers as follows:

```
     .....
int score[4] = {45, 90, 71, 83};
     .....
sort(4, score); /* Function call */
     .....
```

The calling function must use the following prototype declaration.

```
void sort (int, int *);
```

This tells the compiler that the formal argument that receives the array is a pointer, not array variable.

Pointer parameters are commonly employed in string functions. Consider the function copy which copies one string to another.

```
copy(char *s1, char *s2)
{
     while( (*s1++ = *s2++) != '\0')
     ;
}
```

This copies the contents of **s2** into the string **s1.** Parameters **s1** and **s2** are the pointers to character strings, whose initial values are passed from the calling function. For example, the calling statement

```
copy(name1, name2);
```

will assign the address of the first element of **name1** to **s1** and the address of the first element of **name2** to **s2.**

Note that the value of ***s2++** is the character that **s2** pointed to before **s2** was incremented. Due to the postfix ++, **s2** is incremented only after the current value has been fetched. Similarly, **s1** is incremented only after the assignment has been completed.

Each character, after it has been copied, is compared with '\0' and therefore, copying is terminated as soon as the '\0' is copied.

WORKED-OUT PROBLEM 11.7 M

The program of Fig. 11.12 shows how to calculate the sum of two numbers which are passed as arguments using the call by reference method.

Program

```
#include<stdio.h>
#include<conio.h>
void swap (int *p, *q);
main()
{
```

The function **larger** receives the addresses of the variables **a** and **b**, decides which one is larger using the pointers **x** and **y** and then returns the address of its location. The returned value is then assigned to the pointer variable **p** in the calling function. In this case, the address of **b** is returned and assigned to **p** and therefore the output will be the value of **b**, namely, 20.

Note that the address returned must be the address of a variable in the calling function. It is an error to return a pointer to a local variable in the called function.

POINTERS TO FUNCTIONS

A function, like a variable, has a type and an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

```
type (*fptr) ();
```

This tells the compiler that **fptr** is a pointer to a function, which returns *type* value. The parentheses around ***fptr** are necessary. Remember that a statement like

```
type *gptr();
```

would declare **gptr** as a function returning a pointer to *type*.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer. For example, the statements

```
double mul(int, int);
double (*p1)();
p1 = mul;
```

declare **p1** as a pointer to a function and **mul** as a function and then make **p1** to point to the function **mul**. To call the function **mul**, we may now use the pointer **p1** with the list of parameters. That is,

```
(*p1)(x,y) /* Function call */
```

is equivalent to

```
mul(x,y)
```

Note the parentheses around ***p1**.

WORKED-OUT PROBLEM 11.8

H

Write a program that uses a function pointer as a function argument.

A program to print the function values over a given range of values is shown in Fig. 11.13. The printing is done by the function **table** by evaluating the function passed to it by the **main**.

With **table**, we declare the parameter **f** as a pointer to a function as follows:

```
double (*f)();
```

The value returned by the function is of type **double**. When **table** is called in the statement

```
table (y, 0.0, 2, 0.5);
```

we pass a pointer to the function **y** as the first parameter of **table**. Note that **y** is not followed by a parameter list.

During the execution of **table**, the statement

```
value = (*f)(a);
```

calls the function **y** which is pointed to by **f**, passing it the parameter **a**. Thus the function **y** is evaluated over the range 0.0 to 2.0 at the intervals of 0.5.

Similarly, the call

```
table (cos, 0.0, PI, 0.5);
```

passes a pointer to **cos** as its first parameter and therefore, the function **table** evaluates the value of **cos** over the range 0.0 to PI at the intervals of 0.5.

Program

```
#include  <math.h>
#define  PI  3.1415926
double y(double);
double cos(double);
double table (double(*f)(), double, double, double);

main()
{  printf("Table of y(x) = 2*x*x-x+1\n\n");
   table(y, 0.0, 2.0, 0.5);
   printf("\nTable of cos(x)\n\n");
   table(cos, 0.0, PI, 0.5);
}
double table(double(*f)(),double min, double max, double step)
{    double a, value;
     for(a = min; a <= max; a += step)
     {
       value = (*f)(a);
       printf("%5.2f  %10.4f\n", a, value);
     }
}
double y(double x)
{
return(2*x*x-x+1);
}
```

Output

```
Table of y(x) = 2*x*x-x+1
  0.00       1.0000
  0.50       1.0000
  1.00       2.0000
  1.50       4.0000
  2.00       7.0000
Table of cos(x)
  0.00       1.0000
  0.50       0.8776
  1.00       0.5403
  1.50       0.0707
  2.00      -0.4161
  2.50      -0.8011
  3.00      -0.9900
```

Fig. 11.13 Use of pointers to functions

Compatibility and Casting

A variable declared as a pointer is not just a *pointer type* variable. It is also a pointer to a *specific* fundamental data type, such as a character. A pointer therefore always has a type associated with it. We cannot assign a pointer of one type to a pointer of another type, although both of them have memory addresses as their values. This is known as *incompatibility* of pointers.

All the pointer variables store memory addresses, which are compatible, but what is not compatible is the underlying data type to which they point to. We cannot use the assignment operator with the pointers of different types. We can however make explicit assignment between incompatible pointer types by using **cast** operator, as we do with the fundamental types. Example:

```
int x;
char *p;
p = (char *) & x;
```

In such cases, we must ensure that all operations that use the pointer **p** must apply casting properly.

We have an exception. The exception is the void pointer (void *). The void pointer is a ***generic pointer*** that can represent any pointer type. All pointer types can be assigned to a void pointer and a void pointer can be assigned to any pointer without casting. A void pointer is created as follows:

```
void *vp;
```

Remember that since a void pointer has no object type, it cannot be de-referenced.

POINTERS AND STRUCTURES

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose **product** is an array variable of **struct** type. The name **product** represents the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
     char    name[30];
     int     number;
     float   price;
}    product[2], *ptr;
```

This statement declares **product** as an array of two elements, each of the type **struct inventory** and **ptr** as a pointer to data objects of the type **struct inventory.** The assignment

```
ptr = product;
```

would assign the address of the zeroth element of **product** to **ptr**. That is, the pointer **ptr** will now point to **product[0].** Its members can be accessed using the following notation.

```
ptr -> name
ptr -> number
ptr -> price
```

The symbol –> is called the *arrow operator* (also known as *member selection operator*) and is made up of a minus sign and a greater than sign. Note that **ptr–>** is simply another way of writing **product[0].**

When the pointer **ptr** is incremented by one, it is made to point to the next record, i.e., product[1]. The following **for** statement will print the values of members of all the elements of **product** array.

```
for(ptr = product; ptr < product+2; ptr++)
        printf ("%s %d %f\n", ptr->name, ptr->number, ptr->price);
```

We could also use the notation

```
(*ptr).number
```

to access the member **number**. The parentheses around ***ptr** are necessary because the member operator '.' has a higher precedence than the operator *.

WORKED-OUT PROBLEM 11.9

H

Write a program to illustrate the use of structure pointers.

A program to illustrate the use of a structure pointer to manipulate the elements of an array of structures is shown in Fig. 11.14. The program highlights all the features discussed above. Note that the pointer **ptr** (of type **struct invent**) is also used as the loop control index in **for** loops.

Program

```
struct invent
{
   char    *name[20];
   int     number;
   float price;
};
main()
{
  struct invent product[3], *ptr;
  printf("INPUT\n\n");
  for(ptr = product; ptr < product+3; ptr++)
    scanf("%s %d %f", ptr->name, &ptr->number, &ptr->price);
  printf("\nOUTPUT\n\n");
   ptr = product;
   while(ptr < product + 3)
   {
        printf("%-20s %5d %10.2f\n",
                ptr->name,
                ptr->number,
                ptr->price);
       ptr++;
   }
}
```

Output

```
INPUT
Washing_machine    5     7500
Electric_iron      12    350
Two_in_one         7     1250

OUTPUT
Washing machine    5     7500.00
Electric_iron      12    350.00
Two_in_one         7     1250.00
```

Fig. 11.14 *Pointer to structure variables*

While using structure pointers, we should take care of the precedence of operators.

The operators '->' and '.', and () and [] enjoy the highest priority among the operators. They bind very tightly with their operands. For example, given the definition

```
struct
{
   int count;
   float *p;      /* pointer inside the struct */
} ptr;            /* struct type pointer */
```

then the statement

```
++ptr->count;
```

increments **count**, not **ptr**. However,

```
(++ptr)->count;
```

increments **ptr** first, and then links **count**. The statement

```
ptr++ -> count;
```

is legal and increments **ptr** after accessing **count**.

The following statements also behave in the similar fashion.

***ptr->p**	Fetches whatever **p** points to.
***ptr->p++**	Increments **p** after accessing whatever it points to.
(*ptr->p)++	Increments whatever **p** points to.
***ptr++->p**	Increments **ptr** after accessing whatever it points to.

In the previous chapter, we discussed about passing of a structure as an argument to a function. We also saw an example where a function receives a copy of an entire structure and returns it after working on it. As we mentioned earlier, this method is inefficient in terms of both, the execution speed and memory. We can overcome this drawback by passing a pointer to the structure and then using this pointer to work on the structure members. Consider the following function:

```
print_invent(struct invent *item)
{
   printf("Name: %s\n", item->name);
   printf("Price: %f\n", item->price);
}
```

This function can be called by

```
print_invent(&product);
```

The formal argument **item** receives the address of the structure **product** and therefore it must be declared as a pointer of type **struct invent**, which represents the structure of **product.**

ALWAYS REMEMBER

- Only an address of a variable can be stored in a pointer variable. [LO 11.1]
- Do not store the address of a variable of one type into a pointer variable of another type. [LO 11.1]
- The value of a variable cannot be assigned to a pointer variable. [LO 11.1]
- A very common error is to use (or not to use) the address operator (&) and the indirection operator (*) in certain places. Be careful. The compiler may not warn such mistakes. [LO 11.1]
- Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing. [LO 11.1]
- A pointer variable contains garbage until it is initialized. Therefore, we must not use a pointer variable before it is assigned, the address of a variable. [LO 11.2]
- It is an error to assign a numeric constant to a pointer variable. [LO 11.2]
- It is an error to assign the address of a variable to a variable of any basic data types. [LO 11.2]
- A proper understanding of a precedence and associativity rules is very important in pointer applications. For example, expressions like *p++, *p[ ], (*p)[ ], (p).member should be carefully used. [LO 11.4]
- Be careful while using indirection operator with pointer variables. A simple pointer uses single indirection operator (*ptr) while a pointer to a pointer uses additional indirection operator symbol (**ptr). [LO 11.4]
- When an array is passed as an argument to a function, a pointer is actually passed. In the header function, we must declare such arrays with proper size, except the first, which is optional. [LO 11.5]
- If we want a called function to change the value of a variable in the calling function, we must pass the address of that variable to the called function. [LO 11.6]
- When we pass a parameter by address, the corresponding formal parameter must be a pointer variable. [LO 11.6]
- It is an error to assign a pointer of one type to a pointer of another type without a cast (with an exception of void pointer). [LO 11.6]

BRIEF CASES

1. Processing of Examination Marks [LO 11.2, 11.5, 11.6 H]

Marks obtained by a batch of students in the Annual Examination are tabulated as follows:

Student name	Marks obtained
S. Laxmi	45 67 38 55
V.S. Rao	77 89 56 69
-	- - - -

It is required to compute the total marks obtained by each student and print the rank list based on the total marks.

The program in Fig. 11.15 stores the student names in the array **name** and the marks in the array **marks**. After computing the total marks obtained by all the students, the program prepares and prints the rank list. The declaration

```
int marks[STUDENTS][SUBJECTS+1];
```

defines **marks** as a pointer to the array's first row. We use **rowptr** as the pointer to the row of **marks**. The **rowptr** is initialized as follows: