10

Structures and Unions

10.1 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.

10.2 DEFINING A STRUCTURE

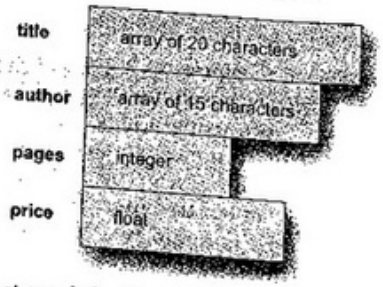
Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of

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.

```
        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");

    }
```

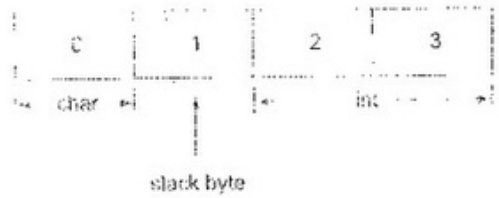
Output

```
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 (slack byte) in between 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.**

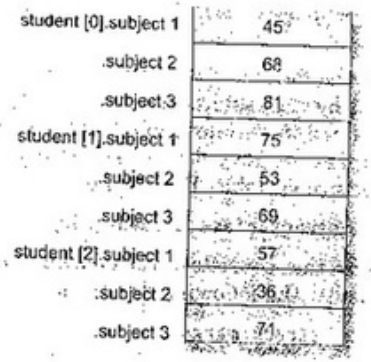


Fig. 10.3 *The array student inside memory*

```
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%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.

1. 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.

A program to update an item is shown in Fig. 10.6. The function **update** receives a copy of the structure variable **item** as one of its parameters. Note that both the function **update** and 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** and 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.

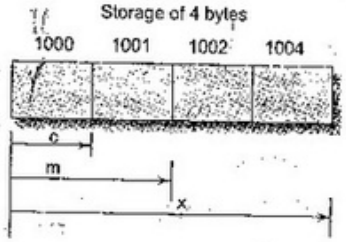


Fig. 10.7 *Sharing of a storage locating by union members*

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;
}
```

Just Remember

- Remember to place a semicolon at the end of definition of structures and unions.
- 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.
- 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**.
- When we use **typedef** definition, the *type_name* comes after the closing brace but before the semicolon.
- 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.
- It is an error to use a structure variable as a member of its own **struct** type structure.
- Assigning a structure of one type to a structure of another type is an error.
- Declaring a variable using the tag name only (without the keyword struct) is an error.
- It is an error to compare two structure variables.
- It is illegal to refer to a structure member using only the member name.
- When structures are nested, a member must be qualified with all levels of structures nesting it.
- When accessing a member with a pointer and dot notation, parentheses are required around the pointer, like (*ptr).number.
- The selection operator (->) is a single token. Any space between the symbols – and > is an error.
- When using **scanf** for reading values for members, we must use address operator & with non-string members.
- Forgetting to include the array subscript when referring to individual structures of an array of structures is an error.
- 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.
- It is an error to initialize a union with data that does not match the type of the first member.
- 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.
- Use short and meaningful structure tag names.
- Avoid using same names for members of different structures (although it is not illegal).
- Passing structures to functions by pointers is more efficient than passing by value. (Passing by pointers are discussed in Chapter 11.)

- 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.
- Bit fields cannot be arrayed.

Case Studies

Book Shop Inventory

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.

Since we are reading the quantity as a string using the **get(string)** function, we have to convert it to an integer before using it in any expressions. This is done using the **atoi()** function.

Programs

```
#include   <stdio.h>
#include   <string.h>
struct  record
{
    char    author[20];
    char    title[30];
    float   price;
    struct
    {
        char   month[10];
        int    year;
    }
  date;
    char   publisher[10];
    int    quantity;
};
int look_up(struct record table[],char s1[],char s2[],int m);
void get (char string [ ] );
main()
{
      char title[30], author[20];
      int  index, no_of_records;
      char response[10], quantity[10];
      struct record book[] = {
      {"Ritche","C Language",45.00,"May",1977,"PHI",10},
      {"Kochan","Programming in C",75.50,"July",1983,"Hayden",5},
      {"Balagurusamy","BASIC",30.00,"January",1984,"TMH",0},
      {"Balagurusamy","COBOL",60.00,"December",1988,"Macmillan",25}
                                  };

 no_of_records = sizeof(book)/ sizeof(struct record);
  do
  {
    printf("Enter title and author name as per the list\n");
    printf("\nTitle:    ");
    get(title);
    printf("Author:   ");
    get(author);
    index = look_up(book, title, author, no_of_records);
    if(index != -1)     /*  Book found  */
    {
        printf("\n%s %s %.2f %s %d %s\n\n",
                book[index].author,
                book[index].title,
                book[index].price,
                book[index].date.month,
                book[index].date.year,
                book[index].publisher);

        printf("Enter number of copies:");
        get(quantity);
        if(atoi(quantity) < book[index].quantity)

          printf("Cost of %d copies = %.2f\n",atoi(quantity),
              book[index].price * atoi(quantity));
        else
          printf("\nRequired copies not in stock\n\n");
      }
      else
          printf("\nBook not in list\n\n");

      printf("\nDo you want any other book? (YES / NO):");
      get(response);
  }
  while(response[0] == 'Y' || response[0] == 'y');
  printf("\n\nThank you.  Good bye!\n");
}
void get(char string [] )
{
   char  c;
   int  i = 0;
   do
   {
      c = getchar();
      string[i++] = c;
   }
   while(c != '\n');
   string[i-1] = '\0';
}

int look_up(struct record table[],char s1[],char s2[],int m)
{
   int  i;
   for(i = 0; i < m;  i++)
      if(strcmp(s1, table[i].title) == 0  &&
         strcmp(s2, table[i].author) == 0)
         return(i);           /* book found      */
   return(-1);                /* book not found  */
}
```

Output

```
Enter title and author name as per the list
Title:    BASIC
Author:   Balagurusamy
Balagurusamy BASIC 30.00 January 1984 TMH

Enter number of copies:5
Required copies not in stock

Do you want any other book? (YES / NO):y
Enter title and author name as per the list
Title:    COBOL
Author:   Balagurusamy
Balagurusamy COBOL 60.00 December 1988 Macmillan
Enter number of copies:7
Cost of 7 copies = 420.00

Do you want any other book? (YES / NO):y
Enter title and author name as per the list
Title:    C Programming
Author:   Ritche

Book not in list


Do you want any other book? (YES / NO):n


Thank you.  Good bye!
```

Fig. 10.8 *Program of bookshop inventory*

Review Questions

10.1 State whether the following statements are *true* or *false*.

(a) A **struct** type in C is a built-in data type.
(b) The tag name of a structure is optional.
(c) Structures may contain members of only one data type.
(d) A structure variable is used to declare a data type containing multiple fields.
(e) It is legal to copy a content of a structure variable to another structure variable of the same type.
(f) Structures are always passed to functions by printers.
(g) Pointers can be used to access the members of structure variables.
(h) We can perform mathematical operations on structure variables that contain only numeric type members.

(i) The keyword **typedef** is used to define a new data type.
(j) In accessing a member of a structure using a pointer p, the following two are equivalent:
(*p).member_name and p -> member_name
(k) A union may be initialized in the same way a structure is initialized.
(l) A union can have another union as one of the members.
(m) A structure cannot have a union as one of its members.
(n) An array cannot be used as a member of a structure.
(o) A member in a structure can itself be a structure.

10.2 Fill in the blanks in the following statements:

(a) The _____ _____ can be used to create a synonym for a previously defined data type.
(b) A_____ is a collection of data items under one name in which the items share the same storage.
(c) The name of a structure is referred to as _____.
(d) The selection operator -> requires the use of a _____ to access the members of a structure.
(e) The variables declared in a structure definition are called its _____.

10.3 A structure tag name **abc** is used to declare and initialize the structure variables of type **struct abc** in the following statements. Which of them are incorrect? Why? Assume that the structure **abc** has three members, **int, float** and **char** in that order.

(a) `struct a,b,c;`
(b) `struct abc a,b,c`
(c) `abc x,y,z;`
(d) `struct abc a[ ];`
(e) `struct abc a = { };`
(f) `struct abc = b, { 1+2, 3.0, "xyz"}`
(g) `struct abc c = {4,5,6};`
(h) `struct abc a = 4, 5.0, "xyz";`

10.4 Given the declaration

```
struct abc a,b,c;
```

which of the following statements are legal?

(a) `scanf ("%d, &a);`
(b) `printf ("%d", b);`
(c) `a = b;`
(d) `a = b + c;`
(e) `if (a>b)`
. . . . .

10.5 Given the declaration

```
struct item_bank
{
        int number;
        double cost;
};
```

which of the following are correct statements for declaring one dimensional array of type **struct item_bank?**

```
(a) bank items[10];
(b) items[10] item_bank;
(c) item_bank items (10);
(d) struct item_bank items [10];
(e) struct items item_bank [10];
```

10.6 Given the following declaration

```
typedef struct abc
{
    char x;
    int y;
    float z[10];
} ABC;
```

State which of the following declarations are invalid? Why?

```
(a) struct abc v1;
(b) struct abc v2[10];
(c) struct ABC v3;
(d) ABC a,b,c;
(e) ABC a[10];
```

10.7 How does a structure differ from an array?

10.8 Explain the meaning and purpose of the following:

(a) Template
(b) **struct** keyword
(c) **typedef** keyword
(d) **sizeof** operator
(e) Tag name

10.9 Explain what is wrong in the following structure declaration:

```
struct
{
        int number;
        float price;
}
main( )
{
        . . . . .
        . . . . .
}
```

10.10 When do we use the following?

(a) Unions
(b) Bit fields
(c) The **sizeof** operator

10.11 What is meant by the following terms?

(a) Nested structures
(b) Array of structures

Give a typical example of use of each of them.

10.12 Given the structure definitions and declarations

```
struct abc
{
    int a;
    float b;
};
struct xyz
{
    int x;
    float y;
};
abc a1, a2;
xyz x1, x2;
```

find errors, if any, in the following statements:

```
(a) a1 = x1;
(b) abc.a1 = 10.75;
(c) int m = a + x;
(d) int n = x1.x + 10;
(e) a1 = a2;
(f) if (a.a1 > x.x1) . . .
(g) if (a1.a < x1.x) . . .
(h) if (x1 != x2) . . .
```

10.13 Describe with examples, the different ways of assigning values to structure members.

10.14 State the rules for initializing structures.

10.15 What is a 'slack byte'? How does it affect the implementation of structures?

10.16 Describe three different approaches that can be used to pass structures as function arguments.

10.17 What are the important points to be considered when implementing bit-fields in structures?

10.18 Define a structure called **complex** consisting of two floating-point numbers **x** and **y** and declare a variable **p** of type **complex**. Assign initial values 0.0 and 1.1 to the members.

10.19 What is the error in the following program?

```
typedef struct product
{
    char name [ 10 ];
    float price ;
} PRODUCT products [ 10 ];
```

10.20 What will be the output of the following program?

```
main ( )
{
    union x
    {
        int a;
        float b;
        double c ;
    };
```

```
    printf("%d\n", sizeof(x));
      a.x = 10;
    printf("%d%f%f\n", a.x, b.x, c.x);
      c.x = 1.23;
    printf("%d%f%f\n", a.x, b.x, c.x);
}
```

Programming Exercises

10.1 Define a structure data type called **time_struct** containing three members integer **hour,** integer **minute** and integer **second.** Develop a program that would assign values to the individual members and display the time in the following form:

16:40:51

10.2 Modify the above program such that a function is used to input values to the members and another function to display the time.

10.3 Design a function **update** that would accept the data structure designed in Exercise 10.1 and increments time by one second and returns the new time. (If the increment results in 60 seconds, then the second member is set to zero and the minute member is incremented by one. Then, if the result is 60 minutes, the minute member is set to zero and the hour member is incremented by one. Finally when the hour becomes 24, it is set to zero.)

10.4 Define a structure data type named **date** containing three integer members **day, month** and **year.** Develop an interactive modular program to perform the following tasks;

- To read data into structure members by a function
- To validate the date entered by another function
- To print the date in the format

April 29, 2002

by a third function.

The input data should be three integers like 29, 4, and 2002 corresponding to day, month and year. Examples of invalid data:

31, 4, 2002 – April has only 30 days

29, 2, 2002 – 2002 is not a leap year

10.5 Design a function **update** that accepts the **date** structure designed in Exercise 10.4 to increment the date by one day and return the new date. The following rules are applicable:

- If the date is the last day in a month, month should be incremented
- If it is the last day in December, the year should be incremented
- There are 29 days in February of a leap year

10.6 Modify the input function used in Exercise 10.4 such that it reads a value that represents the date in the form of a long integer, like 19450815 for the date 15-8-1945 (August 15, 1945) and assigns suitable values to the members **day, month** and **year.**

Use suitable algorithm to convert the long integer 19450815 into year, month and day.

10.7 Add a function called **nextdate** to the program designed in Exercise 10.4 to perform the following task;

- Accepts two arguments, one of the structure **data** containing the present date and the second an integer that represents the number of days to be added to the present date.
- Adds the days to the present date and returns the structure containing the next date correctly.

Note that the next date may be in the next month or even the next year.

10.8 Use the **date** structure defined in Exercise 10.4 to store two dates. Develop a function that will take these two dates as input and compares them.

- It returns 1, if the **date1** is earlier than **date2**
- It returns 0, if **date1** is later date

10.9 Define a structure to represent a vector (a series of integer values) and write a modular program to perform the following tasks:

- To create a vector
- To modify the value of a given element
- To multiply by a scalar value
- To display the vector in the form
  (10, 20, 30, ...)

10.10 Add a function to the program of Exercise 10.9 that accepts two vectors as input parameters and return the addition of two vectors.

10.11 Create two structures named **metric** and **British** which store the values of distances. The **metric** structure stores the values in metres and centimetres and the British structure stores the values in feet and inches. Write a program that reads values for the structure variables and adds values contained in one variable of **metric** to the contents of another variable of **British.** The program should display the result in the format of feet and inches or metres and centimetres as required.

10.12 Define a structure named **census** with the following three members:

- A character array city [ ] to store names
- A long integer to store population of the city
- A float member to store the literacy level

Write a program to do the following:

- To read details for 5 cities randomly using an array variable
- To sort the list alphabetically
- To sort the list based on literacy level
- To sort the list based on population
- To display sorted lists

10.13 Define a structure that can describe an hotel. It should have members that include the name, address, grade, average room charge, and number of rooms.

Write functions to perform the following operations:

- To print out hotels of a given grade in order of charges
- To print out hotels with room charges less than a given value

10.14 Define a structure called **cricket** that will describe the following information:

player name
team name
batting average

Using **cricket**, declare an array **player** with 50 elements and write a program to read the information about all the 50 players and print a team-wise list containing names with their batting average.

10.15 Declare a structure **student_record** to contain name, date of birth and total marks obtained. Use the **date** structure designed in Exercise 10.4 to represent the date of birth.

Develop a program to read data for 10 students in a class and list them rank-wise.

11 Pointers

11.1 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.

11.2 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.

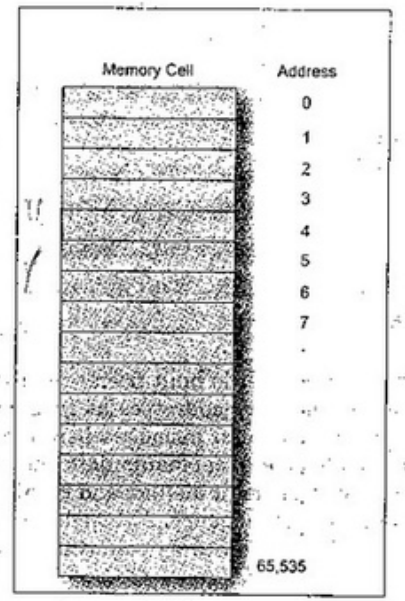


Fig. 11.1 *Memory organisation*

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)

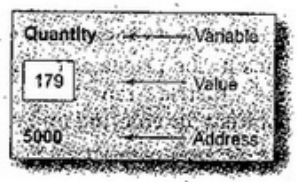


Fig. 11.2 *Representation of a variable*

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.

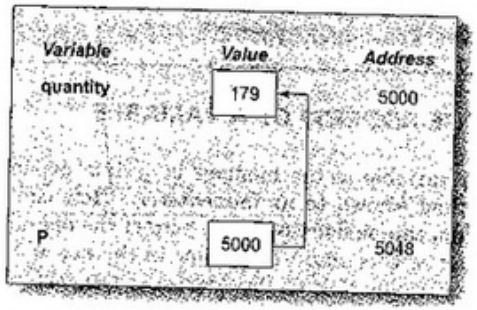
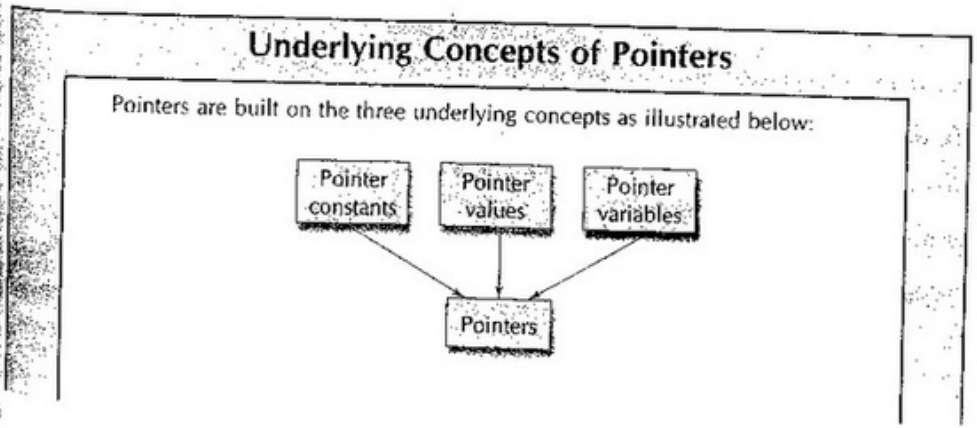


Fig. 11.3 *Pointer variable*

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**.)



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 stored 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*.

11.3 ACCESSING THE ADDRESS OF A VARIABLE

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. How can we then determine the address of a variable? This can be done with the help of the operator & available in C. We have already seen the use of this *address operator* in the **scanf** function. The operator & immediately preceding a variable returns the address of the variable associated with it. For example, the statement

```
p = &quantity;
```

would assign the address 5000 (the location of **quantity**) to the variable **p**. The & operator can be remembered as 'address of'.

The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

1. &125 (pointing at constants).
2. int x[10];
   &x (pointing at array names).
3. &(x+y) (pointing at expressions).

If **x** is an array, then expressions such as

&x[0] and &x[i+3]

are valid and represent the addresses of 0th and (i+3)th elements of **x**.

Example 11.1 Write a program to print the address of a variable along with its value.

The program shown in Fig. 11.4, declares and initializes four variables and then prints out these values with their respective storage locations. Note that we have used %u format for printing address values. Memory addresses are unsigned integers.

```
Program
 main()
 {
     char   a;
     int    x;
     float  p, q;

     a  = 'A';
     x  = 125;
     p  = 10.25, q = 18.76;
     printf("%c is stored at addr %u.\n", a, &a);
     printf("%d is stored at addr %u.\n", x, &x);
     printf("%f is stored at addr %u.\n", p, &p);
     printf("%f is stored at addr %u.\n", q, &q);

 }
```

Output

```
A is stored at addr 4436.
125 is stored at addr 4434.
10.250000 is stored at addr 4442.
18.760000 is stored at addr 4438.
```

Fig. 11.4 *Accessing the address of a variable*

11.4 DECLARING POINTER VARIABLES

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 */
```

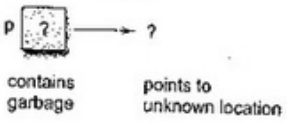
declares the variable **p** as a pointer variable that points to an integer data type. Remember that the type **int** refers to the data type of the variable being pointed to by **p** and not the type of the value of the pointer. Similarly, the statement

```
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:

```
int *p;
```



Pointer Declaration Style

Pointer variables are declared similarly as normal variables except for the addition of the unary * operator. This symbol can appear anywhere between the type name and the printer variable name. Programmers use the following styles:

```
int*   p;    /* style 1 */
int    *p;   /* style 2 */
int    *  p;  /* style 3 */
```

However, the style2 is becoming increasingly popular due to the following reasons:

1. This style is convenient to have multiple declarations in the same statement. Example:

   ```
   int *p, x, *q;
   ```

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;
   ```

11.5 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.

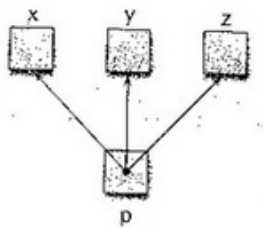
We could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statements are valued

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

Pointer Flexibility

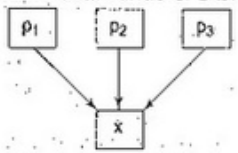
Pointers are flexible. We can make the same pointer to point to different data variables in different statements. Example;

```
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*/
```

11.6 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. Example 11.2 illustrates the distinction between pointer value and the value it points to.

Example 11.2 Write a program to illustrate the use of indirection operator '*' to access the value pointed to by a printer.

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*.

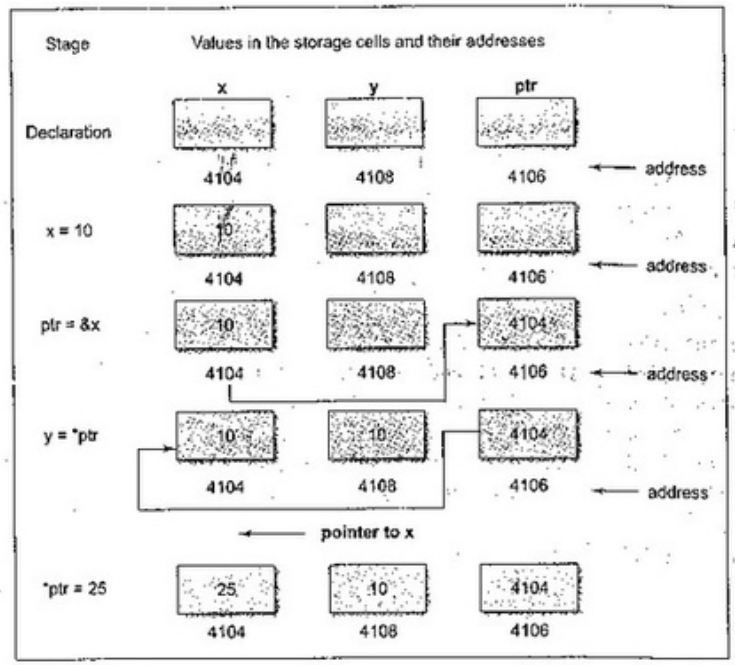
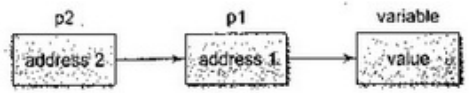


Fig. 11.6 *Illustration of pointer assignments*

11.7 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.

11.8 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.

Example 11.3 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*

11.9 POINTER INCREMENTS AND SCALE FACTOR

We have seen that the pointers can be incremented like

```
p1 = p2 + 2;
p1 = p1 + 1;
```


and so on. Remember, however, an expression like

```
p1++;
```

will cause the pointer **p1** to point to the next value of its type. For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1 = p1 + 1**, the value of **p1** will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the 'length' of the data type that it points to. This length called the *scale factor*.

For an IBM PC, the length of various data types are as follows:

characters	1 byte
integers	2 bytes
floats	4 bytes
long integers	4 bytes
doubles	8 bytes

The number of bytes used to store various data types depends on the system and can be found by making use of the **sizeof** operator. For example, if **x** is a variable, then **sizeof(x)** returns the number of bytes needed for the variable. (Systems like Pentium use 4 bytes for storing integers and 2 bytes for short integers.)

Rules of Pointer Operations

The following rules apply when performing operations on pointer variables.

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from another.
7. When two pointers point to the objects of the same data types, they can be compared using relational operators.
8. A pointer variable cannot be multiplied by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address (i.e &x = 10; is illegal).

```
        sum += *p;
        p++;
        }
        .....
```

Here, we compare the pointer **p** with the address of the last element to determine when the array has been traversed.

Pointers can be used to manipulate two-dimensional arrays as well. We know that in a one-dimensional array **x**, the expression

```
*(x+i) or *(p+i)
```

represents the element **x[i]**. Similarly, an element in a two-dimensional array can be represented by the pointer expression as follows:

```
*(*(a+i)+j) or *(*(p+i)+j)
```

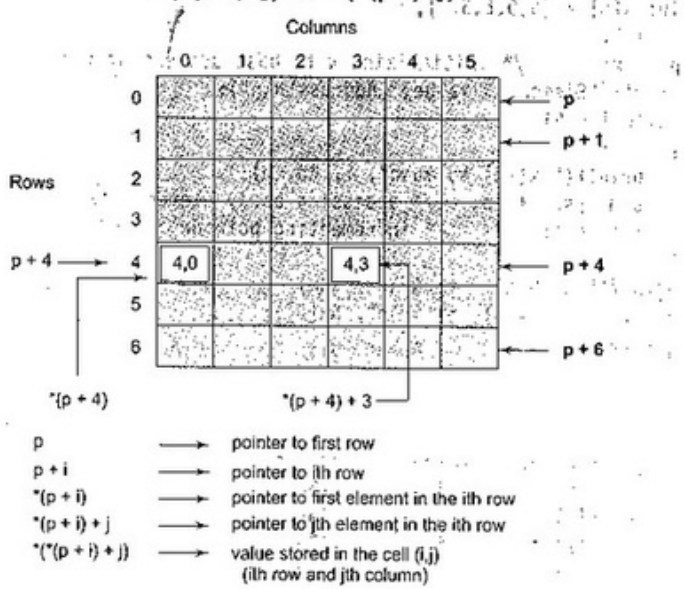
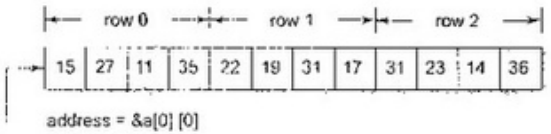


Fig. 11.9 *Pointers to two-dimensional arrays*

Figure 11.9 illustrates how this expression represents the element **a[i][j]**. The base address of the array **a** is &a[0][0] and starting at this address, the compiler allocates contiguous space for all the elements *row-wise*. That is, the first element of the second row is placed immediately after the last element of the first row, and so on. Suppose we declare an array **a** as follows:

```
int a[3][4] = {  {15,27,11,35},
                 {22,19,31,17},
                 {31,23,14,36}
                 };
```

The elements of **a** will be stored as:



If we declare **p** as an **int** pointer with the initial address of &a[0][0], then

a[i][j] is equivalent to *(p+4 × i+j)

You may notice that, if we increment **i** by 1, the **p** is incremented by 4, the size of each row. Then the element **a[2][3]** is given by ***(p+2 × 4+3) = *(p+11)**.

This is the reason why, when a two-dimensional array is declared, we must specify the size of each row so that the compiler can determine the correct storage mapping.

11.11 POINTERS AND CHARACTER STRINGS

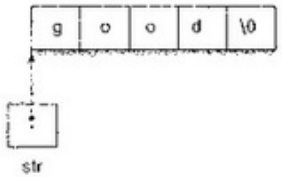
We have seen in Chapter 8 that strings are treated like character arrays and therefore, they are declared and initialized as follows:

```
char str [5] = "good";
```

The compiler automatically inserts the null character '\0' at the end of the string. C supports an alternative method to create strings using pointer variables of type **char**. Example:

```
char *str = "good";
```

This creates a string for the literal and then stores its address in the pointer variable **str**. The pointer **str** now points to the first character of the string "good" as:



We can also use the run-time assignment for giving values to a string pointer. Example

```
char * string1;
string1 = "good";
```

Note that the assignment

```
string1 = "good";
```

is not a string copy, because the variable **string1** is a pointer, not a string.

(As pointed out in Chapter 8, C does not support copying one string to another through the assignment operation.)

We can print the content of the string **string1** using either **printf** or **puts** functions as follows:

```
printf("%s", string1);
puts (string1);
```

Remember, although **string1** is a pointer to the string, it is also the name of the string. Therefore, we do not need to use indirection operator * here.

Like in one-dimensional arrays, we can use a pointer to access the individual characters in a string. This is illustrated by the example 11.5.

Ex. ... C program using pointers to determine the length of a character string.

A program to count the length of a string is shown in Fig.11.10. The statement

```
char *cptr = name;
```

declares cptr as a pointer to a character and assigns the address of the first character of name as the initial value. Since a string is always terminated by the null character, the statement

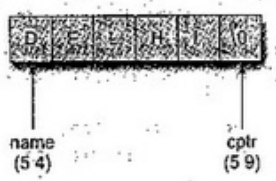
```
while(*cptr != '\0')
```

is true until the end of the string is reached.

When the **while** loop is terminated, the pointer **cptr** holds the address of the null character. Therefore, the statement

```
length = cptr - name;
```

gives the length of the string **name**.



The output also shows the address location of each character. Note that each character occupies one memory cell (byte).

```
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
```

Fig. 11.10 *String handling by pointers*

In C, a constant character string always represents a pointer to that string. And therefore the following statements are valid:

```
char *name;
name = "Delhi";
```

These statements will declare **name** as a pointer to character and assign to **name** the constant character string "Delhi". You might remember that this type of assignment does not apply to character arrays. The statements like

```
char name[20];
name = "Delhi";
```

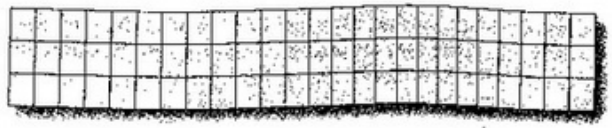
do not work.

11.12 ARRAY OF POINTERS

One important use of pointers is in handling of a table of strings. Consider the following array of strings:

```
char name [3][25];
```

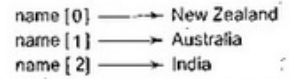
This says that the **name** is a table containing three names, each with a maximum length of 25 characters (including null character). The total storage requirements for the **name** table are 75 bytes.



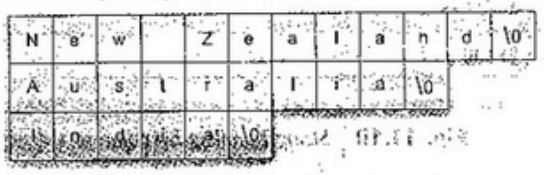
We know that rarely the individual strings will be of equal lengths. Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length. For example,

```
char *name[3] = {
                    "New Zealand",
                    Australia",
                    "India"
                };
```

declares **name** to be an *array of three pointers* to characters, each pointer pointing to a particular name as:



This declaration allocates only 28 bytes, sufficient to hold all the characters as shown



The following statement would print out all the three names:

```
for(i = 0; i <= 2; i++)
  printf("%s\n", name[i]);
```

To access the jth character in the ith name, we may write as

```
*(name[i]+j)
```

The character arrays with the rows of varying length are called 'ragged arrays' and are better handled by pointers.

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.

11.13 POINTERS AS FUNCTION ARGUMENTS

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".

Example 11.6 Write a function using pointers to exchange the values stored in two locations in the memory.

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. [illegible] 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 Example 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 Example 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.

11.14 FUNCTIONS RETURNING POINTERS

We have seen so far that a function can return a single value by its name or return multiple values through pointer parameters. Since pointers are a data type in C, we can also force a function to return a pointer to the calling function. Consider the following code:

```
int *larger (int *, int *);        /* prototype */
main ( )
{
     int a = 10;
     int b = 20;
     int *p;
     p = larger(&a, &b);    /Function call */
     printf ("%d", *p);
}
int *larger (int *x, int *y)
{
     if (*x>*y)
          return (x);       / *address of a */
     else
          return (y);       /* address of b */
}
```

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.

11.15 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**.

Example 11.7 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. 11.12. 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. 11.12 *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.

11.16 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 *.

Example 11.8 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. 11.13. 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. 11.13 *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**.

11.17 TROUBLES WITH POINTERS

Pointers give us tremendous power and flexibility. However, they could become a nightmare when they are not used correctly. The major problem with wrong use of pointers is that the compiler may not detect the error in most cases and therefore the program is likely to produce unexpected results. The output may not given us any clue regarding the use of a bad pointer. Debugging therefore becomes a difficult task.

We list here some pointer errors that are more commonly committed by the programmers.

- Assigning values to uninitialized pointers

```
int * p, m = 100 ;
*p = m ;                /* Error */
```

- Assigning value to a pointer variable

```
int *p, m = 100 ;
p = m;                  /* Error */
```

- Not dereferencing a pointer when required

```
int *p, x = 100;
p = &x;
printf("%d",p);         /* Error */
```

- Assigning the address of an uninitialized variable

```
int m, *p
p = &m;                 /* Error */
```

- Comparing pointers that point to different objects

```
char name1 [ 20 ], name2 [ 30 ];
    char *p1 = name1;
    char *p2 = name2;
    if(p1 > p2).....    /* Error */
```

We must be careful in declaring and assigning values to pointers correctly before using them. We must also make sure that we apply the address operator & and referencing operator * correctly to the pointers. That will save us from sleepless nights.

Just Remember

- Only an address of a variable can be stored in a pointer variable.
- Do not store the address of a variable of one type into a pointer variable of another type.
- The value of a variable cannot be assigned to a pointer variable.
- 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.

```
/*    Prepare rank list based on total marks      */

get_rank_list(char *string [ ],
              int array [ ] [SUBJECTS + 1]
              int m,
              int n)
{
  int i, j, k, (*rowptr)[SUBJECTS+1] = array;
  char *temp;

  for(i = 1; i <= m-1; i++)
     for(j = 1; j <= m-i; j++)
        if( (*(rowptr + j-1))[n-1] < (*(rowptr + j))[n-1])
        {
         swap_string(string[j-1], string[j]);

         for(k = 0; k < n; k++)
         swap_int(&(*(rowptr + j-1))[k],&(*(rowptr+j))[k]);
         }
}
/*      Print out the ranked list              */
print_list(char *string[ ],
           int array [] [SUBJECTS + 1],
           int m,
           int n)
{
    int  i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++)
    {
       printf("%-20s", string[i]);
       for(j = 0; j < n; j++)
          printf("%5d", (*(rowptr + i))[j]);
          printf("\n");
    }
}
/*     Exchange of integer values              */
swap_int(int *p, int *q)
{
    int  temp;
    temp = *p;
    *p   = *q;
    *q   = temp;
}
```



```
/*     Exchange of strings          */
swap_string(char s1[ ], char s2[ ])
{
    char  swaparea[256];
    int   i;
    for(i = 0; i < 256; i++)
       swaparea[i] = '\0';
    i = 0;
    while(s1[i] != '\0' && i < 256)
    {
       swaparea[i] = s1[i];
       i++;
    }
    i = 0;
    while(s2[i] != '\0' && i < 256)
    {
       s1[i] = s2[i];
       s1[++i] = '\0';
    }
    i = 0;
    while(swaparea[i] != '\0')
    {
       s2[i] = swaparea[i];
       s2[++i] = '\0';
    }
}
```

Output

```
Input students names & their marks in four subjects
S.Laxmi 45 67 38 55
V.S.Rao 77 89 56 69
A.Gupta 66 78 98 45
S.Mani 86 72 0 25
R.Daniel 44 55 66 77

S.Laxmi              45   67   38   55  205
V.S.Rao              77   89   56   69  291
A.Gupta              66   78   98   45  287
S.Mani               86   72    0   25  183
R.Daniel             44   55   66   77  242

Ranked List
V.S.Rao              77   89   56   69  291
A.Gupta              66   78   98   45  287
```

```
R.Daniel        44   55   66   77  242
S.Laxmi         45   67   38   55  205
S.Mani          86   72    0   25  183
```

Fig. 11.14 *Preparation of the rank list of a class of students*

2. Inventory Updating

The price and quantity of items stocked in a store changes every day. They may either increase or decrease. The program in Fig. 11.15 reads the incremental values of price and quantity and computes the total value of the items in stock.

The program illustrates the use of structure pointers as function parameters. **&item,** the address of the structure **item,** is passed to the functions **update()** and **mul().** The formal arguments **product** and **stock,** which receive the value of **&item,** are declared as pointers of type **struct stores.**

Program

```
struct stores
{
     char  name[20];
     float price;
     int   quantity;

};
main()
{
     void update(struct stores *, float, int);
     float         p_increment, value;
     int           q_increment;

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

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

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
     update(&item, p_increment, q_increment);
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
     printf("Updated values of item\n\n");
     printf("Name      : %s\n",ptr->name);
     printf("Price     : %f\n",ptr->price);
     printf("Quantity  : %d\n",ptr->quantity);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
     value  = mul(&item);
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
     printf("\nValue of the item  =  %f\n", value);
}


void update(struct stores *product, float p, int q)
{
     product->price += p;
     product->quantity += q;
}
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. 11.15 *Use of structure pointers as function parameters*

Review Questions

11.1 State whether the following statements are *true* or *false*.
(a) Pointer constants are the addresses of memory locations.
(b) Pointer variables are declared using the address operator.
(c) The underlying type of a pointer variable is void.
(d) Pointers to pointers is a term used to describe pointers whose contents are the address of another pointer.
(e) It is possible to cast a pointer to float as a pointer to integer.
(f) An integer can be added to a pointer.
(g) A pointer can never be subtracted from another pointer.
(h) When an array is passed as an argument to a function, a pointer is passed.
(i) Pointers cannot be used as formal parameters in headers to function definitions.
(j) Value of a local variable in a function can be changed by another function.

11.2 Fill in the blanks in the following statements:
(a) A pointer variable contains as its value the ____ of another variable.
(b) The _____operator is used with a pointer to de-reference the address contained in the pointer.

(c) The _____operator returns the value of the variable to which its operand points.
(d) The only integer that can be assigned to a pointer variable is _____.
(e) The pointer that is declared as _____cannot be de-referenced.

11.3 What is a pointer?

11.4 How is a pointer initialized?

11.5 Explain the effects of the following statements:

```
(a) int a, *b = &a;
(b) int p, *p;
(c) char *s;
(d) a = (float *) &x;
(e) double(*f)();
```

11.6 If **m** and **n** have been declared as integers and **p1** and **p2** as pointers to integers, the state errors, if any, in the following statements.

```
(a) p1 = &m;
(b) p2 = n;
(c) *p1 = &n;
(d) p2 = &*&m;
(e) m = p2-p1;
(f) p1 = &p2;
(g) m = *p1 + *p2++;
```

11.7 Distinguish between (*m)[5] and *m[5].

11.8 Find the error, if any, in each of the following statements:

```
(a) int x = 10;
(b) int *y = 10;
(c) int a, *b = &a;
(d) int m;
    int **x = &m;
```

11.9 Given the following declarations:

```
int x = 10, y = 10;
int *p1 = &x, *p2 = &y;
```

What is the value of each of the following expressions?

```
(a) (*p1)++
(b) --(*p2)
(c) *p1 + (*p2)--
(d) ++(*p2) - *p1
```

11.10 Describe typical applications of pointers in developing programs.

11.11 What are the arithmetic operators that are permitted on pointers?

11.12 What is printed by the following program?

```
int m = 100;
int * p1 = &m;
int **p2 = &p1;
printf("%d", **p2);
```


11.13 What is wrong with the following code?

```
int **p1, *p2;
p2 = &p1;
```

11.14 Assuming **name** as an array of 15 character length, what is the difference between the following two expressions?

(a) name + 10; and
(b) *(name + 10).

11.15 What is the output of the following segment?

```
int m[2];
*(m+1) = 100;
*m = *(m+1);
printf("%d", m [0]);
```

11.16 What is the output of the following code?

```
int m [2];
int *p = m;
m [0] = 100 ;
m [1] = 200 ;
printf("%d %d", ++*p, *p);
```

11.17 What is the output of the following program?

```
int f(char *p);
main ( )
{
    char str[ ] = "ANSI";
    printf("%d", f(str) );
}
int f(char *p)
{
    char *q = p;
    while (*++p)
        ;
    return (p-q);
}
```

11.18 Given below are two different definitions of the function **search( )**

a)
```
void search (int* m[ ], int x)
{
}
```

b)
```
void search (int ** m, int x)
{
}
```

Are they equivalent? Explain.

11.19 Do the declarations

```
char s [ 5 ] ;
char *s;
```

represent the same? Explain.

11.20 Which one of the following is the correct way of declaring a pointer to a function? Why?

(a) int (*p) (void) ;
(b) int *p (void);

Programming Exercises

11.1 Write a program using pointers to read in an array of integers and print its elements in reverse order.

11.2 We know that the roots of a quadratic equation of the form

$$ax^2 + bx + c = 0$$

are given by the following equations:

$$x_1 = \frac{-b + \text{square-root}\,(b^2 - 4ac)}{2a}$$

$$x_2 = \frac{-b - \text{square-root}\,(b^2 - 4ac)}{2a}$$

Write a function to calculate the roots. The function must use two pointer parameters, one to receive the coefficients a, b, and c, and the other to send the roots to the calling function.

11.3 Write a function that receives a sorted array of integers and an integer value, and inserts the value in its correct place.

11.4 Write a function using pointers to add two matrices and to return the resultant matrix to the calling function.

11.5 Using pointers, write a function that receives a character string and a character as argument and deletes all occurrences of this character in the string. The function should return the corrected string with no holes.

11.6 Write a function **day_name** that receives a number n and returns a pointer to a character string containing the name of the corresponding day. The day names should be kept in a **static** table of character strings local to the function.

11.7 Write a program to read in an array of names and to sort them in alphabetical order. Use **sort** function that receives pointers to the functions **strcmp** and **swap.sort** in turn should call these functions via the pointers.

11.8 Given an array of sorted list of integer numbers, write a function to search for a particular item, using the method of *binary search*. And also show how this function may be used in a program. Use pointers and pointer arithmetic.
(Hint: In binary search, the target value is compared with the array's middle element. Since the table is sorted, if the required value is smaller, we know that all values greater than the middle element can be ignored. That is, in one attempt, we eliminate one half the list. This search can be applied recursively till the target value is found.)

11.9 Write a function (using a pointer parameter) that reverses the elements of a given array.

11.10 Write a function (using pointer parameters) that compares two integer arrays to see whether they are identical. The function returns 1 if they are identical, 0 otherwise.

12

File Management in C

12.1 INTRODUCTION

Until now we have been using the functions such as **scanf** and **printf** to read and write data. These are console oriented I/O functions, which always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small. However, many real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of *files* to store data. A file is a place on the disk where a group of related data is stored. Like most other languages, C supports a number of functions that have the ability to perform basic file operations, which include:

- naming a file,
- opening a file,
- reading data from a file,
- writing data to a file, and
- closing a file.

There are two distinct ways to perform file operations in C. The first one is known as the *low-level* I/O and uses UNIX system calls. The second method is referred to as the *high-level* I/O operation and uses functions in C's standard I/O library. We shall discuss in this chapter, the important file handling functions that are available in the C library. They are listed in Table 12.1.

Table 12.1 *High Level I/O Functions*

Function name	*Operation*
fopen()	* Creates a new file for use. * Opens an existing file for use.
fclose()	* Closes a file which has been opened for use.
getc()	* Reads a character from a file.
putc()	* Writes a character to a file.
fprintf()	* Writes a set of data values to a file.
fscanf()	* Reads a set of data values from a file.
getw()	* Reads an integer from a file.
putw()	* Writes an integer to a file.
fseek()	* Sets the position to a desired point in the file.
ftell()	* Gives the current position in the file (in terms of bytes from the start).
rewind()	* Sets the position to the beginning of the file.

There are many other functions. Not all of them are supported by all compilers. You should check your C library before using a particular I/O function.

12.2 DEFINING AND OPENING A FILE

If we want to store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include:

1. Filename.
2. Data structure.
3. Purpose.

Filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a *primary name* and an *optional period* with the extension. Examples:

Input.data
store
PROG.C
Student.c
Text.out

Data structure of a file is defined as **FILE** in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. **FILE** is a defined data type.

When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data.

Following is the general format for declaring and opening a file:

```
FILE *fp;
fp = fopen("filename", "mode");
```

The first statement declares the variable **fp** as a "pointer to the data type **FILE**". As stated earlier, **FILE** is a structure that is defined in the I/O library. The second statement

opens the file named filename and assigns an identifier to the **FILE** type pointer **fp**. This pointer, which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The mode does this job. Mode can be one of the following:

- **r** open the file for reading only.
- **w** open the file for writing only.
- **a** open the file for appending (or adding) data to it.

Note that both the filename and mode are specified as strings. They should be enclosed in double quotation marks.

When trying to open a file, one of the following things may happen:

1. When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is 'reading', and if it exists, then the file is opened with the current contents safe otherwise an error occurs.

Consider the following statements:

```
FILE *p1, *p2;
p1 = fopen("data", "r");
p2 = fopen("results", "w");
```

The file **data** is opened for reading and **results** is opened for writing. In case, the **results** file already exists, its contents are deleted and the file is opened as a new file. If **data** file does not exist, an error will occur.

Many recent compilers include additional modes of operation. They include:

- **r+** The existing file is opened to the beginning for both reading and writing.
- **w+** Same as **w** except both for reading and writing.
- **a+** Same as **a** except both for reading and writing.

We can open and use a number of files at a time. This number however depends on the system we use.

12.3 CLOSING A FILE

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. In case, there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. Another instance where we have to close a file is when we want to reopen the same file in a different mode. The I/O library supports a function to do this for us. It takes the following form:

```
fclose(file_pointer);
```

This would close the file associated with the FILE pointer **file_pointer**. Look at the following segment of a program.

```
.....
.....
FILE *p1, *p2;
p1 = fopen("INPUT", "w");
p2 = fopen("OUTPUT", "r");
.....
.....
fclose(p1);
fclose(p2);
.....
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file.

As a matter of fact all files are closed automatically whenever a program terminates. However, closing a file as soon as you are done with it is a good programming habit.

12.4 INPUT/OUTPUT OPERATIONS ON FILES

Once a file is opened, reading out of or writing to it is accomplished using the standard I/O routines that are listed in Table 12.1.

The *getc* and *putc* Functions

The simplest file I/O functions are **getc** and **putc**. These are analogous to **getchar** and **putchar** functions and handle one character at a time. Assume that a file is opened with mode **w** and file pointer **fp1**. Then, the statement

```
putc(c, fp1);
```

writes the character contained in the character variable **c** to the file associated with FILE pointer **fp1**. Similarly, **getc** is used to read a character from a file that has been opened in read mode. For example, the statement

```
c = getc(fp2);
```

would read a character from the file whose file pointer is fp2.

The file pointer moves by one character position for every operation of **getc** or **putc**. The **getc** will return an end-of-file marker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

Example 12.1 Write a program to read data from the keyboard, write it to a file called **INPUT**, again read the same data from the **INPUT** file, and display it on the screen.

A program and the related input and output data are shown in Fig.12.1. We enter the input data via the keyboard and the program writes it, character by character, to the file **INPUT**. The end of the data is indicated by entering an **EOF** character, which is *control-Z* in the reference system. (This may be control-D in other systems.) The file INPUT is closed at this signal.


```
Program
   #include <stdio.h>

   main()
   {
       FILE *f1;
       char c;
       printf("Data Input\n\n");
       /* Open the file INPUT */
       f1 = fopen("INPUT", "w");

       /* Get a character from keyboard   */
       while((c=getchar()) != EOF)

           /* Write a character to INPUT  */
           putc(c,f1);

       /* Close the file INPUT   */

       fclose(f1);
       printf("\nData Output\n\n");

       /* Reopen the file INPUT    */
       f1 = fopen("INPUT","r");

      /* Read a character from INPUT*/
       while((c=getc(f1)) != EOF)

           /* Display a character on screen */
           printf("%c",c);

       /* Close the file INPUT       */
       fclose(f1);
   }
Output

   Data Input
   This is a program to test the file handling
   features on this system^Z

   Data Output
   This is a program to test the file handling
   features on this system
```

Fig. 12.1 *Character oriented read/write operations on a file*

The file INPUT is again reopened for reading. The program then reads its content character by character, and displays it on the screen. Reading is terminated when **getc** encounters the end-of-file mark EOF.

Testing for the end-of-file condition is important. Any attempt to read past the end of file might either cause the program to terminate with an error or result in an infinite loop situation.

The *getw* and *putw* Functions

The **getw** and **putw** are integer-oriented functions. They are similar to the **getc** and **putc** functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of **getw** and **putw** are:

```
putw(integer, fp);
getw(fp);
```

Example 12.2 illustrates the use of **putw** and **getw** functions.

Example 12.2 A file named **DATA** contains a series of integer numbers. Code a program to read these numbers and then write all 'odd' numbers to a file to be called **ODD** and all 'even' numbers to a file to be called **EVEN**.

The program is shown in Fig. 12.2. It uses three files simultaneously and therefore, we need to define three-file pointers **f1, f2** and **f3.**

First, the file DATA containing integer values is created. The integer values are read from the terminal and are written to the file **DATA** with the help of the statement

putw(number, f1);

Notice that when we type –1, the reading is terminated and the file is closed. The next step is to open all the three files, **DATA** for reading, **ODD** and **EVEN** for writing. The contents of **DATA** file are read, integer by integer, by the function **getw(f1)** and written to **ODD** or **EVEN** file after an appropriate test. Note that the statement

(number = getw(f1)) != EOF

reads a value, assigns the same to **number**, and then tests for the end-of-file mark.

Finally, the program displays the contents of ODD and EVEN files. It is important to note that the files **ODD** and **EVEN** opened for writing are closed before they are reopened for reading.

Program

```
#include  <stdio.h>
main()
{
    FILE  *f1, *f2, *f3;
    int   number, i;

    printf("Contents of DATA file\n\n");
    f1 = fopen("DATA", "w");       /* Create DATA file    */
    for(i = 1; i <= 30; i++)
    {
       scanf("%d", &number);
       if(number == -1) break;
       putw(number,f1);
    }
    fclose(f1);

    f1 = fopen("DATA", "r");
    f2 = fopen("ODD", "w");
    f3 = fopen("EVEN", "w");

    /* Read from DATA file */
    while((number = getw(f1)) != EOF)
    {
        if(number %2 == 0)
          putw(number, f3);   /*  Write to EVEN file  */
        else
          putw(number, f2);   /*  Write to ODD file   */
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);

    f2 = fopen("ODD","r");
    f3 = fopen("EVEN", "r");

    printf("\n\nContents of ODD file\n\n");
    while((number = getw(f2)) != EOF)
       printf("%4d", number);

    printf("\n\nContents of EVEN file\n\n");
    while((number = getw(f3)) != EOF)
       printf("%4d", number);

    fclose(f2);
    fclose(f3);

}
```

Output

```
Contents of DATA file
111 222 333 444 555 666 777 888 999 000 121 232 343 454 565 -1
```

```
Contents of ODD file
111 333 555 777 999 121 343 565

Contents of EVEN file
222 444 666 888   0 232 454
```

Fig. 12.2 *Operations on integer data*

The *fprintf* and *fscanf* Functions

So far, we have seen functions, that can handle only one character or integer at a time. Most compilers support two other functions, namely **fprintf** and **fscanf**, that can handle a group of mixed data simultaneously.

The functions **fprintf** and **fscanf** perform I/O operations that are identical to the familiar **printf** and **scanf** functions, except of course that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of **fprintf** is

```
fprintf(fp, "control string", list);
```

where *fp* is a file pointer associated with a file that has been opened for writing. The *control string* contains output specifications for the items in the list. The *list* may include variables, constants and strings. Example:

```
fprintf(f1, "%s %d %f", name, age, 7.5);
```

Here, **name** is an array variable of type char and **age** is an **int** variable.
The general format of **fscanf** is

```
fprintf(fp, "control string", list);
```

This statement would cause the reading of the items in the list from the file specified by fp, according to the specifications contained in the *control string*. Example:

```
fscanf(f2, "%s %d", item, &quantity);
```

Like **scanf, fscanf** also returns the number of items that are successfully read. When the end of file is reached, it returns the value **EOF.**

Example 12.3 Write a program to open a file named INVENTORY and store in it the following data:

Item name	Number	Price	Quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item.

The program is given in Fig.12.3. The filename INVENTORY is supplied through the keyboard. Data is read using the function **fscanf** from the file **stdin,** which refers to the terminal and it is then written to the file that is being pointed to by the file pointer **fp.** Remember that the file pointer **fp** points to the file INVENTORY.

After closing the file INVENTORY, it is again reopened for reading. The data from the file, along with the item values are written to the file **stdout,** which refers to the screen. While reading from a file, care should be taken to use the same format specifications with which the contents have been written to the file....é

Program

```
#include  <stdio.h>

main()
{
    FILE  *fp;
    int    number, quantity, i;
    float  price, value;
    char   item[10], filename[10];

    printf("Input file name\n");
    scanf("%s", filename);
    fp = fopen(filename, "w");
    printf("Input inventory data\n\n");
    printf("Item name  Number   Price   Quantity\n");
    for(i = 1; i <= 3; i++)
    {
       fscanf(stdin, "%s %d %f %d",
                     item, &number, &price, &quantity);
       fprintf(fp, "%s %d %.2f %d",
                     item, number, price, quantity);
    }
    fclose(fp);
    fprintf(stdout, "\n\n");

    fp = fopen(filename, "r");

    printf("Item name  Number   Price   Quantity    Value\n");
    for(i = 1; i <= 3; i++)
    {
       fscanf(fp, "%s %d %f d",item,&number,&price,&quantity);
       value = price * quantity;
       fprintf(stdout, "%-8s %7d %8.2f %8d %11.2f\n",
                     item, number, price, quantity, value);
    }
    fclose(fp);
}
```

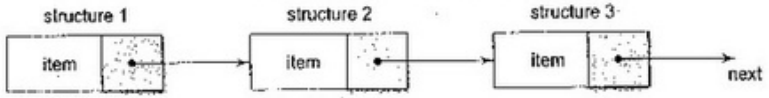
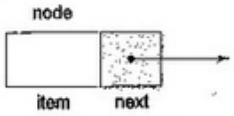


Fig. 13.4 A linked list

Each structure of the list is called a *node* and consists of two fields, one containing the item and the other containing the address of the next item (a pointer to the next item) in the list. A linked list is therefore a collection of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as part of the data in the structure itself. The link is in the form of a pointer to another structure of the same type. Such a structure is represented as follows:

```
struct node
{
  int item;
  struct node *next;
};
```

The first member is an integer item and the second a pointer to the next node in the list as shown below. Remember, the **item** is an integer here only for simplicity, and could be any complex data type.

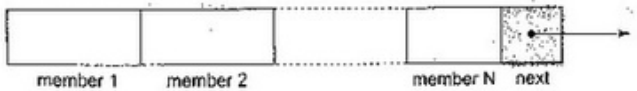


Such structures, which contain a member field that points to the same structure type are called *self-refrential* structure.

A node may be represented in general form as follows:

```
struct tag-name
{
  type member1;
  type member2;
  . . . .
  . . . .
  struct tag-name *next;
};
```

The structure may contain more than one item with different data types. However, one of the items must be a pointer of the type **tag-name**.



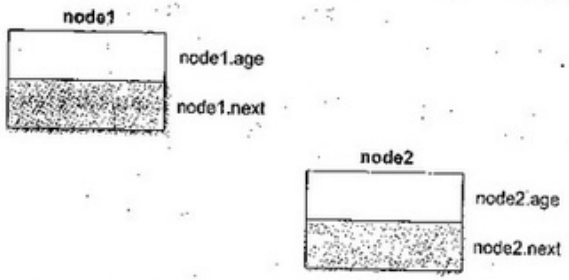
Let use consider a simple example to illustrate the concept of linking. Suppose we define a structure as follows:

```
struct link_list
{
    float age;
    struct link_list *next;
};
```

For simplicity, let as assume that the list contains two nodes **node1** and **node2**. They are of type **struct link_list** and are defined as follows:

```
struct link_list node1, node2;
```

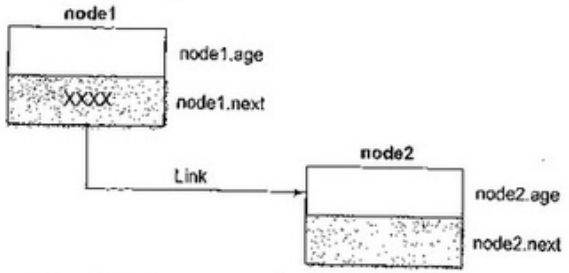
This statement creates space for two nodes each containing two empty fields as shown:



The **next** pointer of **node1** can be made to point to **node2** by the statement

```
node1.next = &node2;
```

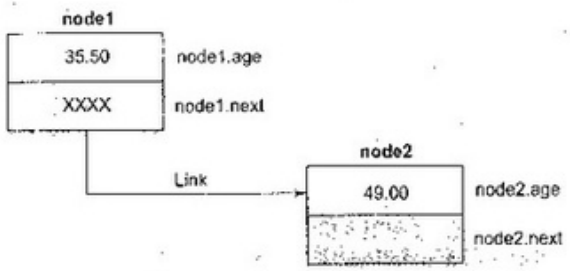
This statement stores the address of **node2** into the field **node1.next** and thus establishes a "link" between **node1** and **node2** as shown:



"xxxx" is the address of **node2** where the value of the variable **node2.age** will be stored. Now let us assign values to the field age.

```
node1.age = 35.50;
node2.age = 49.00;
```

The result is as follows:



We may continue this process to create a liked list of any number of values. For example:

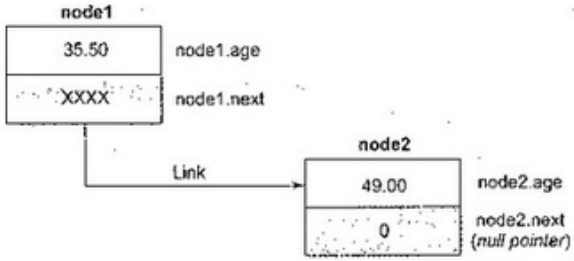
```
node2.next = &node3;
```

would add another link provided **node3** has been declared as a variable of type **struct link list.**

No list goes on forever. Every list must have an end. We must therefore indicate the end of a linked list. This is necessary for processing the list. C has a special pointer value called **null** that can be stored in the **next** field of the last node. In our two-node list, the end of the list is marked as follows:

```
node2.next = 0;
```

The final linked list containing two nodes is as shown:



The value of the age member of **node2** can be accessed using the **next** member of **node1** as follows:

```
printf("%f\n", node1.next->age);
```

13.8 ADVANTAGES OF LINKED LISTS

A linked list is *dynamic data structure*. Therefore, the primary advantage of linked lists over arrays is that linked lists can grow or shrink in size during the execution of a program. A linked list can be made just as long as required.

Another advantage is that a linked list does not waste memory space. It uses the memory that is just needed for the list at any point of time. This is because it is not necessary to specify the number of nodes to be used in the list.

The third, and the most important advantage is that the linked lists provide flexibility is allowing the items to be rearranged efficiently. It is easier to insert or delete items by rearranging the links. This is shown in Fig. 13.5.

The major limitation of linked lists is that the access to any arbitrary item is little cumbersome and time consuming. Whenever we deal with a fixed length list, it would be better to use an array rather than a linked list. We must also note that a linked list will use more storage than an array with the same number of items. This is because each item has an additional link field.

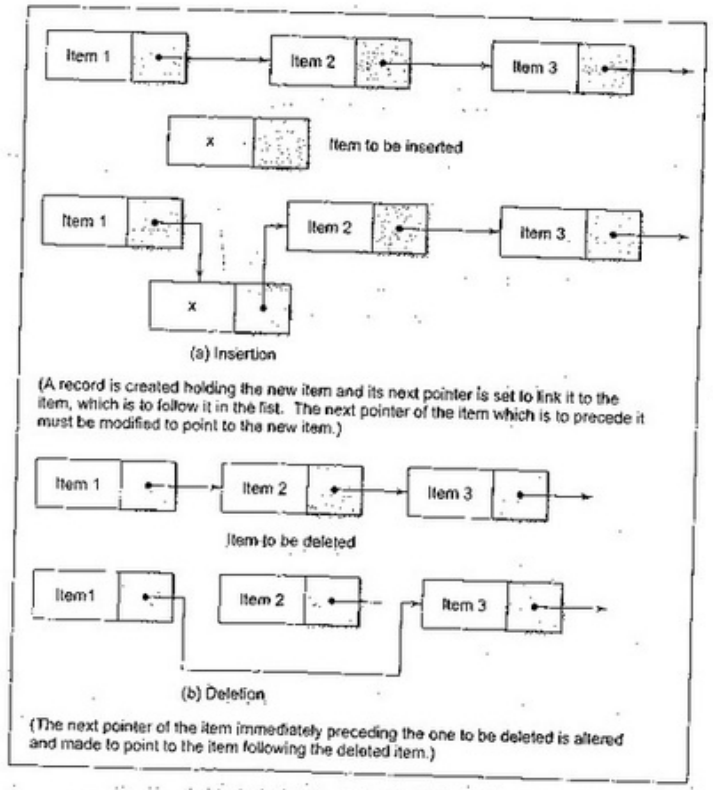


Fig. 13.5 *Insertion into and deletion from a linked list*

13.9 TYPES OF LINKED LISTS

There are different types of lined lists. The one we discussed so far is known as *linear singly* linked list. The other *linked* lists are:

- Circular linked lists.
- Two-way or doubly linked lists.
- Circular doubly linked lists.

The circular linked lists have no beginning and no end. The last item points back to the first item. The doubly linked list uses double set of pointers, one pointing to the next item and other pointing to the preceding item. This allows us to traverse the list in either direction. Circular doubly linked lists employs both the forward pointer and backward pointer in circular form. Figure 13.6 illustrates various kinds of linked lists.

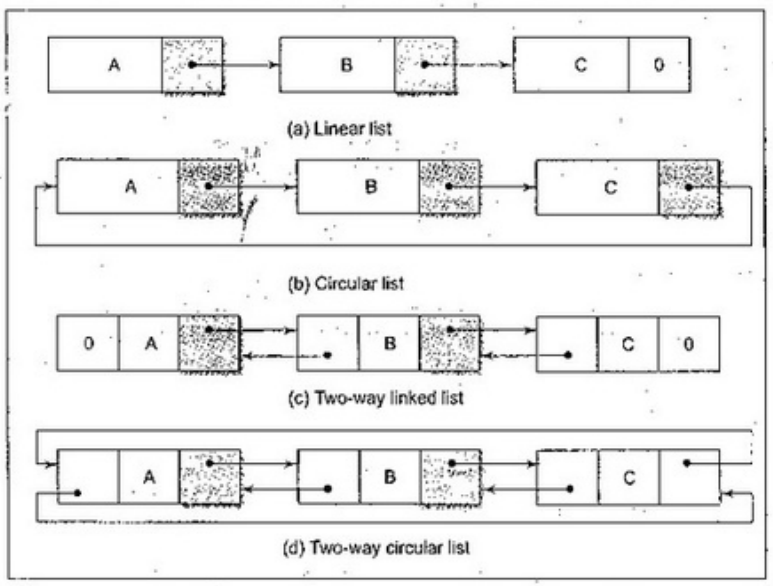


Fig. 13.6 *Different types of linked lists*

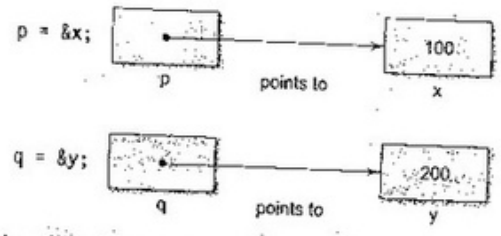
13.10 POINTERS REVISITED

The concept of pointers was discussed in Chapter 11. Since pointers are used extensively in processing of the linked lists, we shall briefly review some of their properties that are directly relevant to the processing of lists.

We know that variables can be declared as pointers, specifying the type of data item they can point to. In effect, the pointer will hold the address of the data item and can be used to access its value. In processing linked lists, we mostly use pointers of type structures.

It is most important to remember the distinction between the pointer variable **ptr**, which contain the address of a variable, and the referenced variable ***ptr**, which denotes the valu of variable to which **ptr**'s value points. The following examples illustrate this distinction. I these illustrations, we assume that the pointers **p** and **q** and the variables **x** and **y** ar declared to be of same type.

(a) Initialization

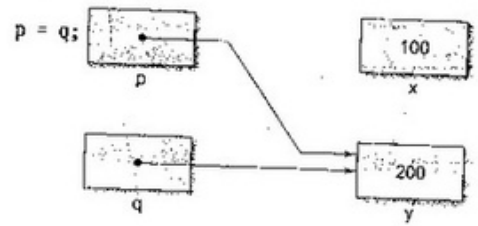


The pointer **p** contains the address of **x** and **q** contains the address of **y**.

***p =100 and *q = 200 and p< >q**

(b) Assignment p = q

The assignment **p = q** assigns the address of the variable **y** to the pointer variable **p** and therefore **p** now points to the variable **y**.

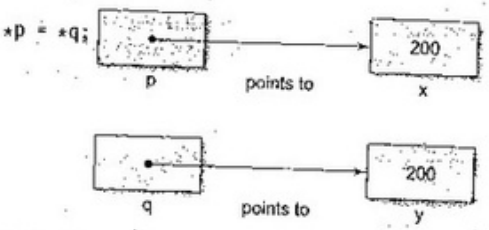


Both the pointer variables point to the same variable.

***p = *q = 200 but x <> y**

(c) Assignment *p = *q

This assignment statement puts the value of the variable pointed to by **q** in the location of the variable pointed to by **p**.



The pointer **p** still points to the same variable **x** but the old value of **x** is replaced by 200 (which is pointed to by **q**).

x = y = 200 but p <> q

(d) NULL pointers

A special constant known as NULL pointer (0) is available in C to initialize pointers that point to nothing. That is the statements

p = 0; (or p = NULL;) p → 0

q = 0; (q = NULL;) q → 0

make the pointers **p** and **q** point to nothing. They can be later used to point any values.

We know that a pointer must be initialized by assigning a memory address before using it. There are two ways of assigning memory address to a pointer.

1. Assigning an existing variable address (static assignment)

```
ptr = &count;
```

2. Using a memory allocation function (dynamic assignment)

```
ptr = (int*) malloc(sizeof(int));
```

13.11 CREATING A LINKED LIST

We can treat a linked list as an abstract data type and perform the following basic operations:

1. Creating a list.
2. Traversing the list.
3. Counting the items in the list.
4. Printing the list (or sub list).
5. Looking up an item for editing or printing.
6. Inserting an item.
7. Deleting an item.
8. Concatenating two lists.

In Section 13.7 we created a two-element linked list using the structure variable names **node1** and **node2**. We also used the address operator **&** and member operators . and –> for creating and accessing individual items. The very idea of using a linked list is to avoid any reference to specific number of items in the list so that we can insert or delete items as and when necessary. This can be achieved by using "anonymous" locations to store nodes. Such locations are accessed not by name, but by means of pointers, which refer to them. (For example, we must avoid using references like **node1.age** and **node1.next –> age.**)

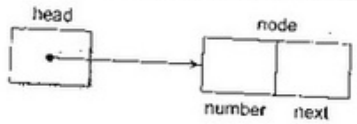
Anonymous locations are created using pointers and dynamic memory allocation functions such as **malloc**. We use a pointer **head** to create and access anonymous nodes. Consider the following:

```
struct linked_list
{
    int number;
    struct linked_list *next;
};
typedef struct linked_list node;
node *head;
head = (node *) malloc(sizeof(node));
```

The **struct** declaration merely describes the format of the nodes and does not allocate storage. Storage space for a node is created only when the function **malloc** is called in the statement

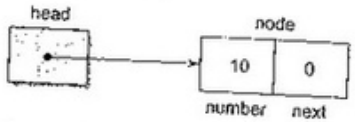

```
head = (node *) malloc(sizeof(node));
```

This statement obtains a piece of memory that is sufficient to store a node and assigns its address to the pointer variable **head**. This pointer indicates the beginning of the linked list.



The following statements store values in the member fields:

```
head -> number = 10;
head -> next = NULL;
```



The second node can be added as follows:

```
head -> next = (node *)malloc(sizeof(node));
head -> next ->number = 20;
head->next->next = NULL;
```

Although this process can be continued to create any number of nodes, it becomes cumbersome and clumsy if nodes are more than two. The above process may be easily implemented using both recursion and iteration techniques. The pointer can be moved from the current node to the next node by a self-replacement statement such as:

```
head = head -> next;
```

The Example 13.3 shows creation of a complete linked list and printing of its contents using recursion.

Example 13.3 Write a program to create a linear linked list interactively and print out the list and the total number of items in the list.

The program shown in Fig. 13.7 first allocates a block of memory dynamically for the first node using the statement

```
head = (node *)malloc(sizeof(node));
```

which returns a pointer to a structure of type **node** that has been type defined earlier. The linked list is then created by the function **create**. The function requests for the number to be placed in the current node that has been created. If the value assigned to the current node is –999, then null is assigned to the pointer variable **next** and the list ends. Otherwise, memory space is allocated to the next node using again the **malloc** function and the next value is placed into it. Not that the function **create** calls itself recursively and the process will continue until we enter the number –999.

The items stored in the linked list are printed using the function **print**, which accept a pointer to the current node as an argument. It is a recursive function and stops when it receives a NULL pointer. Printing algorithm is as follows;

1. Start with the first node.
2. While there are valid nodes left to print
   (a) print the current item; and
   (b) advance to next node.

Similarly, the function **count** counts the number of items in the list recursively and return the total number of items to the **main** function. Note that the counting does not include the item –999 (contained in the dummy node).

Program

```
#include    <stdio.h>
#include    <stdlib.h>
#define NULL 0

struct linked_list
{
        int number;
        struct linked_list *next;
};
typedef struct linked_list node;  /* node type defined */

main()
{
        node *head;
        void create(node *p);
        int count(node *p);
        void print(node *p);
        head = (node *)malloc(sizeof(node));
        create(head);
        printf("\n");
        printf(head);
        printf("\n");
        printf("\nNumber of items = %d \n", count(head));
}
void create(node *list)
{
        printf("Input a number\n");
        printf("(type -999 at end): ");
        scanf("%d", &list -> number); /* create current node */

        if(list->number == -999)
        {
          list->next = NULL;
        }

        else /*create next node */
        {
          list->next = (node *)malloc(sizeof(node));
          create(list->next);  */ Recursion occurs */
        }
        return;
}
void print(node *list)
{
        if(list->next != NULL)
        {
          printf("%d-->",list ->number);  /* print current item */

           if(list->next->next == NULL)
             printf("%d", list->next->number);

          print(list->next);   /* move to next item */
        }
        return;
}

int count(node *list)
{
        if(list->next == NULL)
                return (0);
        else
                return(1+ count(list->next));
}
```

Output

```
Input a number
(type -999 to end); 60
Input a number
(type -999 to end); 20
Input a number
(type -999 to end); 10
Input a number
(type -999 to end); 40
Input a number
(type -999 to end); 30
Input a number
(type -999 to end); 50
Input a number
(type -999 to end); -999

60 -->20 -->10 -->40 -->30 -->50 --> -999

Number of items = 6
```

Fig. 13.7 *Creating a linear linked list*

13.12 INSERTING AN ITEM

One of the advantages of linked lists is the comparative case with which new nodes can be inserted. It requires merely resetting of two pointers (rather than having to move around a list of data as would be the case with arrays).

Inserting a new item, say X, into the list has three situations:

1. Insertion at the front of the list.
2. Insertion in the middle of the list.
3. Insertion at the end of the list.

The process of insertion precedes a search for the place of insertion. The search involves in locating a node after which the new item is to be inserted.

A general algorithm for insertion is as follows:

Begin
  if the list is empty or
  the new node comes before the head node *then*,
  insert the new node as the head node,
  else
  if the new node comes after the last node, *then*,
  insert the new node as the end node,
  else
  insert the new node in the body of the list.
End

Algorithm for placing the new item at the beginning of a linked list:

1. Obtain space for new node.
2. Assign data to the item field of new node.
3. Set the *next* field of the new node to point to the start of the list.
4. Change the head pointer to point to the new node.

Algorithm for inserting the new node X between two existing nodes, say, N1 and N2;

1. Set space for new node X.
2. Assign value to the item field of X.
3. Set the *next* field of X to point to node N2.
4. Set the *next* field of N1 to point to X.

Algorithm for inserting an item at the end of the list is similar to the one for inserting in the middle, except the *next* field of the new node is set to NULL (or set to point to a dummy or sentinel node, if it exists).

Example 13.4 Write a function to insert a given item *before* a specified node known as key node.

The function **insert** shown in Fig. 13.8 requests for the item to be inserted as well as the "key node". If the insertion happens to be at the beginning, then memory space is created for the new node, the value of new item is assigned to it and the pointer **head** is assigned to the next member. The pointer **new,** which indicates the beginning of the new node is assigned to **head.** Note the following statements:


```
new->number = x;
new->next = head;
head = new;
```

```
node *insert(node *head)
{
        node *find(node *p, int a);
        node *new;    /* pointer to new node */
        node *n1;  /* pointer to node preceding key node */
        int key;
        int x;      /* new item (number) to be inserted */

        printf("Value of new item?");
        scanf("%d", &x);
        printf("Value of key item ? (type -999 if last) ");
        scanf("%d", &key);

        if(head->number == key)  /* new node is first */
        {
            new = (node *)malloc(size of(node));
            new->number = x;
            new->next = head;
            head = new;
        }
        else   /* find key node and insert new node */
        {      /* before the key node */
          n1 = find(head, key);    /* find key node */

          if(n1 == NULL)
             printf("\n key is not found \n");
          else   /* insert new node */
          {
             new = (node *)malloc(sizeof(node));
             new->number = x;
             new->next = n1->next;
             n1->next = new;
          }
        }
        return(head);
}
node *find(node *lists, int key)
{
          if(list->next->number == key) /* key found */
            return(list);
        else
```

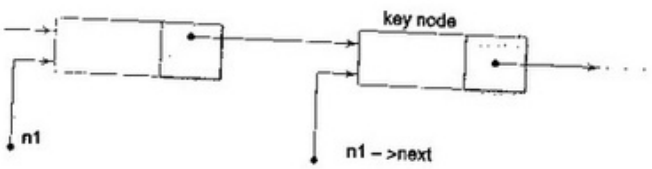
```
        if(list->next->next == NULL)  /* end */
          return(NULL);
      else
        find(list->next, key);
}
```

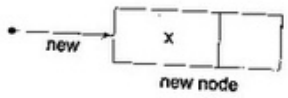
Fig. 13.8 *A function for inserting an item into a linked list*

lowever, if the new item is to be inserted after an existing node, then we use the function l recursively to locate the 'key node'. The new item is inserted before the key node using algorithm discussed above. This is illustrated as:

ore insertion

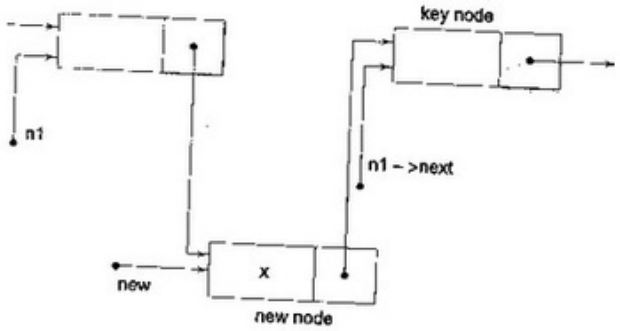
```
new = (node *)malloc(sizeof(node));
new->number = x;
```





r insertion

```
new->next = n1->next;
n1->next = new;
```



13.13 DELETING AN ITEM

Deleting a node from the list is even easier than insertion, as only one pointer value needs to be changed. Here again we have three situations.

1. Deleting the first item.
2. Deleting the last item.
3. Deleting between two nodes in the middle of the list.

In the first case, the head pointer is altered to point to the second item in the list. In the other two cases, the pointer of the item immediately preceding the one to be deleted is altered to point to the item following the deleted item. The general algorithm for deletion is as follows:

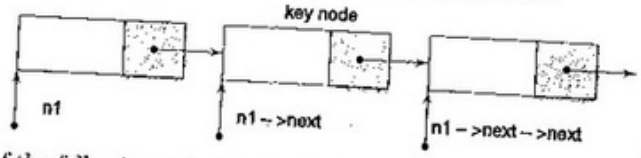
Begin
  if the list is empty, *then*,
  node cannot be deleted
  else
  if node to be deleted is the first node, *then*,
  make the head to point to the second node,
  else
  delete the node from the body of the list.
End

The memory space of deleted node may be released for re-use. As in the case of insertion, the process of deletion also involves search for the item to be deleted.

Example 13.5 Write a function to delete a specified node.

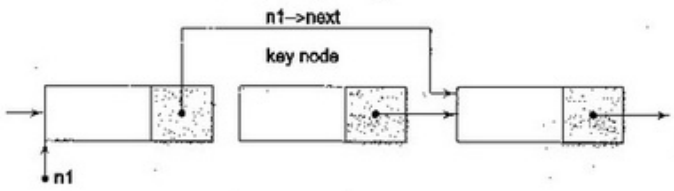
A function to delete a specified node is given in Fig. 13.9. The function first checks whether the specified item belongs to the first node. If yes, then the pointer to the second node is temporarily assigned the pointer variable **p**, the memory space occupied by the first node is freed and the location of the second node is assigned to **head**. Thus, the previous second node becomes the first node of the new list.

If the item to be deleted is not the first one, then we use the **find** function to locate the position of 'key node' containing the item to be deleted. The pointers are interchanged with the help of a temporary pointer variable making the pointer in the preceding node to point to the node following the key node. The memory space of key node that has been deleted if freed. The figure below shows the relative position of the key node.



The execution of the following code deletes the key node.

```
p = n1->next->next;
free (n1->next);
n1->next = p;
```




```
node *delete(node *head)
{
        node *find(node *p, int a);
        int  key;   /* item to be deleted */
        node *n1;   /* pointer to node preceding key node */
        node *p;    /* temporary pointer */
        printf("\n What is the item (number) to be deleted?");
        scanf("%d", &key);
        if(head->number == key)/* first node to be deleted) */
        {
          p = head->next;    /* pointer to 2nd node  in list */
          free(head);        /* release space of key node */
          head = p;          /* make head to point to 1st node */
        }
        else
        {
          n1 = find(head, key);
          if(n1 == NULL)
             printf("\n key not found \n");
          else                    /* delete key node */
          {
            p = n1->next->next;    /*  pointer to the node
                                       following the keynode */

            free(n1->next);        /* free key node */
            n1->next = p;          /* establish link */
          }
        }
        return(head);
}
        /* USE FUNCTION find() HERE */
```

Fig. 13.9 *A function for deleting an item from linked list*

13.14 APPLICATION OF LINKED LISTS

Linked list concepts are useful to model many different abstract data types such as queues, stacks and trees.

If we restrict the process of insertion to one end of the list and deletions to the other end, then we have a model of a *queue*. That is, we can insert an item at the rear and remove an item at the front (see Fig. 13.10a). This obeys the discipline of "first in, first out" (FIFO). There are many examples of queues in real-life applications.

If we restrict insertions and deletions to occur only at the beginning of list, then we model another data structure known as *stack*. Stacks are also referred to as *push-down* lists. An example of a stack is the "in" tray of a busy executive. The files pile up in the tray, and whenever the executive has time to clear the files, he takes it off from the top. That is, files are added at the top and removed from the top (see Fig. 13.10b). Stacks are sometimes referred to as "last in, first out" (LIFO) structure.

Lists, queues and stacks are all inherently one-dimensional. A *tree* represents a two-dimensional linked list. Trees are frequently encountered in everyday life. One example is the organizational chart of a large company. Another example is the chart of sports tournaments.

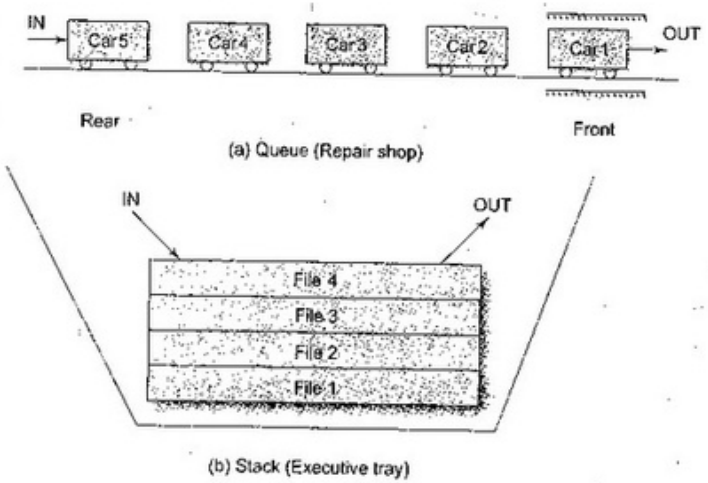


Fig. 13.10 *Application of linked lists*

Just Remember

- Use the **sizeof** operator to determine the size of a linked list.
- When using memory allocation functions **malloc** and **calloc**, test for a NULL pointer return value. Print appropriate message if the memory allocation fails.
- Never call memory allocation functions with a zero size.
- Release the dynamically allocated memory when it is no longer required to avoid any possible "memory leak".
- Using **free** function to release the memory not allocated dynamically with **malloc** or **calloc** is an error.
- Use of a invalid pointer with **free** may cause problems and, sometimes, system crash.
- Using a pointer after its memory has been released is an error.
- It is an error to assign the return value from **malloc** or **calloc** to anything other than a pointer.
- It is a logic error to set a pointer to NULL before the node has been released. The node is irretrievably lost.
- It is an error to declare a self-referential structure without a structure tag.
- It is an error to release individually the elements of an array created with **calloc**.
- It is a logic error to fail to set the link filed in the last node to null.

Case Studies

1. Insertion in a Sorted List

The task of inserting a value into the current location in a sorted linked list involves two operations:

1. Finding the node before which the new node has to be inserted. We call this node as 'Key node'.
2. Creating a new node with the value to be inserted and inserting the new node by manipulating pointers appropriately.

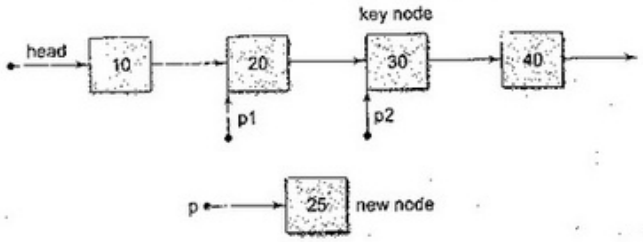
In order to illustrate the process of insertion, we use a sorted linked list created by the create function discussed in Example 13.3. Figure 13.11 shows a complete program that creates a list (using sorted input data) and then inserts a given value into the correct place using function insert.

Program

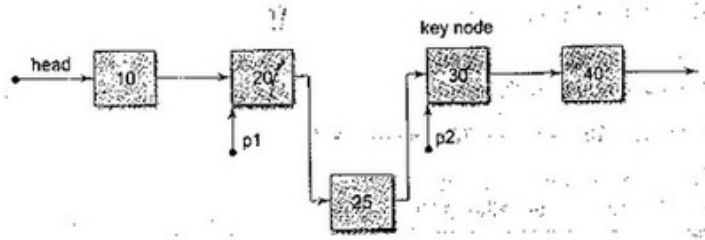
```
#include <stdio.h>
#include <stdio.h>
```



```
#define NULL 0

struct linked_list
{
        int number;
        struct linked-list *next;
};
typedef struct linked_lit node;

main()
{
        int n;
        node *head;
        void create(node *p);
        node *insert(node *p, int n);
        void print(node *p);
        head = (node *)malloc(sizeof(node));
        create(head);
        printf("\n");
        printf("Original list: ");
        print(head);
        printf("\n\n");
        printf("Input number to be inserted: ");
        scanf("%d", &n);
        head = inert(head,n);
        printf("\n");
        printf("New list:  ");
        print(head);
}
void create(node *list)
{
        printf("Input a number \n");
        printf("(type -999 at end): ");
        scanf("%d", &list->number);

        if(list->number == -999)
        {
          list->next = NULL;
        }
        else /* create next node */
        {
          list->next = (node *)malloc(sizeof(node));
          create(list->next);
        }
        return;
}
```



When new node is created



When new node is inserted

2. Building a Sorted List

The program in Fig. 13.11 can be used to create a sorted list. This is possible by creating 'one item' list using the create function and then inserting the remaining items one after another using insert function.

A new program that would build a sorted list from a given list of numbers is shown in Fig. 13.12. The **main** function creates a 'base node' using the first number in the list and then calls the function **insert_sort** repeatedly to build the entire sorted list. It uses the same sorting algorithm discussed above but does not use any dummy node. Note that the last item points to NULL.

Program

```
#include <stdio.h>
#include <stdlib.h>
#define NULL 0

struct linked_list
{
        int number;
        struct linked_list *next;
};
typedef struct linked_list node;

main ()
{
        int n;
        node *head = NULL;
        void print(node *p);
        node *insert_Sort(node *p, int n);

        printf("Input the list of numbers.\n");
        printf("At end, type -999.\n");
        scanf("%d",&n);

        while(n != -999)
        {
          if(head == NULL)       /* create 'base' node */
          {
            head = (node *)malloc(sizeof(node));
            head ->number = n;
            head->next = NULL;

          }

          else       /* insert next item */
          {
            head = insert_sort(head,n);
          }
            scanf("%d", &n);
        }
        printf("\n");
        print(head);
        print("\n");
}
node *insert_sort(node *list, int x)
{
        node *p1, *p2, *p;
        p1 = NULL;
        p2 = list; /* p2 points to first node */

        for( ; p2->number < x ; p2 = p2->next)
        {
          p1 = p2;

          if(p2->next == NULL)
          {
            p2 = p2->next;          /* p2 set to NULL */
            break;                  /* insert new node at end */
          }
        }
```

```
    /* key node found */
    p = (node *)malloc(sizeof(node)); /* space for new node */
    p->number = x;      /* place value in the new node */
    p->next = p2;       /* link new node to key node */
    if (p1 == NULL)
            list = p;       /* new node becomes the first node */
    else
            p1->next = p;  /* new node inserted after 1st node */
    return (list);
}
void print(node *list)
{
        if (list == NULL)
          printf("NULL");
        else
        {
          printf("%d-->",list->number);
          print(list->next);
        }
        return;
}
```

Output

```
Input the list of number.
At end, type -999.
80 70 50 40 60 -999
40-->50-->60-->70-->80 -->NULL
Input the list of number.
At end, type -999.
40 70 50 60 80 -999
40-->50-->60-->70-->80-->NULL
```

Fig. 13.12 *Creation of sorted list from a given list of numbers*

Review Questions

13.1 State whether the following statements are *true* or *false*
- (a) Dynamically allocated memory can only be accessed using pointers.
- (b) **calloc** is used to change the memory allocation previously allocated with **malloc**.
- (c) Only one call to free is necessary to release an entire array allocated with **calloc**.
- (d) Memory should be freed when it is no longer required.
- (e) To ensure that it is released, allocated memory should be freed before the program ends.

- (f) The link field in a linked list always points to successor.
- (g) The first step in a adding a node to a linked list is to allocate memory for the next node.

13.2 Fill in the blanks in the following statements
- (a) Function _____ is used to dynamically allocate memory to arrays.
- (b) A_____ is an ordered collection of data in which each element contains the location of the next element.
- (c) Data structures which contain a member field that points to the same structure type are called _____ structures.
- (d) A_____identifies the last logical node in a linked list.
- (e) Stacks are referred to as _____

13.3 What is a linked list? How is it represented?

13.4 What is dynamic memory allocation? How does it help in building complex programs?

13.5 What is the principal difference between the functions **malloc** and **calloc**

13.6 Find errors, if any, in the following memory management statements:

```
a. *ptr = (int *)malloc(m, sizeof(int));
b. table = (float *)calloc(100);
c. node = free(ptr);
```

13.7 Why a linked list is called a dynamic data structure? What are the advantages of using linked lists over arrays?

13.8 Describe different types of linked lists.

13.9 Identify errors, if any, in the following structure definition statements:

```
struct
{
  char name[30]
  struct *next;
};
typedef struct node;
```

13.10 The following code is defined in a header file *list.h*

```
typedef struct
{
    char name[15];
    int age;
    float weight;
}DATA;

struct linked_list
{
    DATA person;
    Struct linked_list *next;
};
typedef struct linked_list NODE;
typedef NODE *NDPTR;
```

Explain how could we use this header file for writing programs.

13.11 What does the following code achieve?

```
int * p ;
p = malloc (sizeof (int) ) ;
```

13.12 What does the following code do?

```
float *p;
p = calloc (10,sizeof(float) ) ;
```

13.13 What is the output of the following code?

```
int i, *ip ;
ip = calloc ( 4, sizeof(int) );
for (i = 0 ; i < 4 ; i++)
        * ip++ = i * i;
for (i = 0 ; i < 4 ; i++)
        printf("%d\n", *--ip );
```

13.14 What is printed by the following code?

```
int *p;
 p = malloc (sizeof (int) );
*p = 100 ;
 p = malloc (sizeof (int) );
*p = 111;
 printf("%d", *p);
```

13.15 What is the output of the following segment?

```
struct node
{
    int m ;
    struct node *next;
} x, y, z, *p;
  x.m = 10 ;
  y.m = 20 ;
  z.m = 30 ;
  x.next = &y;
  y.next = &z;
  z.next = NULL;
  p = x.next;
  while (p != NULL)
{
    printf("%d\n", p -> m);
    p = p -> next;
}
```

Programming Exercises

13.1 In Example 13.3, we have used print() in recursive mode. Rewrite this function using iterative technique in for loop.

13.2 Write a menu driven program to create a linked list of a class of students and perform the following operations:

a. Write out the contents of the list.
b. Edit the details of a specified student.
c. Count the number of students above a specified age and weight.

Make use of the header file defined in Review Question 13.10.

13.3 Write recursive and non-recursive functions for reversing the elements in a linear list Compare the relative efficiencies of them.

13.4 Write an interactive program to create linear linked lists of customer names and their telephone numbers. The program should be menu driven and include features for adding a new customer and deleting an existing customer.

13.5 Modify the above program so that the list is always maintained in the alphabetical order of customer names.

13.6 Develop a program to combine two sorted lists to produce a third sorted lists which contains one occurrence of each of the elements in the original lists.

13.7 Write a program to create a circular linked list so that the input order of data item is maintained. Add function to carry out the following operations on circular linked list.

a. Count the number of nodes
b. Write out contents
c. Locate and write the contents of a given node

13.8 Write a program to construct an ordered doubly linked list and write out the contents of a specified node.

13.9 Write a function that would traverse a linear singly linked list in reverse and write out the contents in reverse order.

13.10 Given two ordered singly linked lists, write a function that will merge them into a third ordered list.

13.11 Write a function that takes a pointer to the first node in a linked list as a parameter and returns a pointer to the last node. NULL should be returned if the list is empty.

13.12 Write a function that counts and returns the total number of nodes in a linked list.

13.13 Write a function that takes a specified node of a linked list and makes it as its last node.

13.14 Write a function that computers and returns the length of a circular list.

13.15 Write functions to implement the following tasks for a doubly linked list.

(a) To insert a node.
(b) To delete a node.
(c) To find a specified node.

14 The Preprocessor

14.1 INTRODUCTION

C is a unique language in many respects. We have already seen features such as structures and pointers. Yet another unique feature of the C language is the *preprocessor*. The C preprocessor provides several tools that are unavailable in other high-level languages. The programmer can use these tools to make his program easy to read, easy to modify, portable, and more efficient.

The preprocessor, as its name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as *preprocessor command lines or directives*. Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.

Preprocessor directives follow special syntax rules that are different from the normal C syntax. They all begin with the symbol # in column one and do not require a semicolon at the end. We have already used the directives **#define** and **#include** to a limited extent. A set of commonly used preprocessor directives and their functions is given in Table 14.1.

Table 14.1 *Preprocessor Directives*

Directive	*Function*
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included
#ifdef	Test for a macro definition
#endif	Specifies the end of #if.
#ifndef	Tests whether a macro is not defined.
#if	Test a compile-time condition
#else	Specifies alternatives when #if test fails.

These directives can be divided into three categories:

1. Macro substitution directives.
2. File inclusion directives.
3. Compiler control directives.

14.2 MACRO SUBSTITUTION

Macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. The preprocessor accomplishes this task under the direction of **#define** statement. This statement, usually known as a *macro definition* (or simply a macro) takes the following general form:

#define *identifier string*

If this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the **identifier** in the source code by the string. The keyword **#define** is written just as shown (starting from the first column) followed by the *identifier* and a *string*, with at least one blank space between them. Note that the definition is not terminated by a semicolon. The *string* may be any text, while the *identifier* must be a valid C name.

There are different forms of macro substitution. The most common forms are:

1. Simple macro substitution.
2. Argumented macro substitution.
3. Nested macro substitution.

Simple Macro Substitution

Simple string replacement is commonly used to define constants. Examples of definition of constants are:

```
#define   COUNT      100
#define   FALSE      0
#define   SUBJECTS   6
#define   PI         3.1415926
#define   CAPITAL    "DELHI"
```

Notice that we have written all macros (identifiers) in capitals. It is a convention to write all macros in capitals to identify them as symbolic constants. A definition, such as

```
#define  M  5
```

will replace all occurrences of M with 5, starting from the line of definition to the end of the program. However, a macro inside a string does not get replaced. Consider the following two lines:

```
total = M * value;
printf("M = %d\n", M);
```

These two lines would be changed during preprocessing as follows:

```
total = 5 * value;
printf("M = %d\n", 5);
```

Notice that the string **"M=%d\n"** is left unchanged.

A macro definition can include more than a simple constant value. It can include expressions as well. Following are valid definitions:

```
#define     AREA        5 * 12.46
#define     SIZE        sizeof(int) * 4
#define     TWO-PI      2.0 * 3.1415926
```

Whenever we use expressions for replacement, care should be taken to prevent an unexpected order of evaluation. Consider the evaluation of the equation

```
ratio = D/A;
```

where D and A are macros defined as follows:

```
#define D   45 - 22
#define A   78 + 32
```

The result of the preprocessor's substitution for D and A is:

```
ratio = 45-22/78+32;
```

This is certainly different from the expected expression

$$(45 - 22)/(78+32)$$

Correct results can be obtained by using parentheses around the strings as:

```
#define D   (45 - 22)
#define A   (78 + 32)
```

It is a wise practice to use parentheses for expressions used in macro definitions.

As mentioned earlier, the preprocessor performs a literal text substitution, whenever the defined name occurs. This explains why we cannot use a semicolon to terminate the #define statement. This also suggests that we can use a macro to define almost anything. For example, we can use the definitions

```
#define     TEST        if (x > y)
#define     AND
#define     PRINT       printf("Very Good. \n");
```

to build a statement as follows:

```
TEST AND PRINT
```

The preprocessor would translate this line to

```
if(x>y) printf("Very Good.\n");
```

Some tokens of C syntax are confusing or are error-prone. For example, a common programming mistake is to use the token = in place of the token == in logical expressions. Similar is the case with the token &&.

Following are a few definitions that might be useful in building error free and more readable programs:

```
#define     EQUALS      ==
#define     AND         &&
#define     OR          ||
#define     NOT_EQUAL   !=
#define     START       main() {
#define     END         }
#define     MOD         %
```



```
#define     BLANK_LINE      printf("\n");
#define     INCREMENT       ++
```

An example of the use of syntactic replacement is:

```
START
... ...
... ...
if(total EQUALS 240 AND average EQUALS 60)
INCREMENT count;
... ...
... ...
END
```

Macros with Arguments

The preprocessor permits us to define more complex and more useful form of replacements. It takes the form:

```
#define     identifier(f1, f2, ..... fn)        string
```

Notice that there is no space between the macro *identifier* and the left parentheses. The identifiers f1, f2, ...,fn are the formal macro arguments that are analogous to the formal arguments in a function definition.

There is a basic difference between the simple replacement discussed above and the replacement of macros with arguments. Subsequent occurrence of a macro with arguments is known as a *macro call* (similar to a function call). When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters. Hence, the string behaves like a template.

A simple example of a macro with arguments is

```
#define     CUBE(x)     (x*x*x)
```

If the following statement appears later in the program

```
volume = CUBE(side);
```

Then the preprocessor would expand this statement to:

```
volume = (side * side * side );
```

Consider the following statement:

```
volume = CUBE(a+b);
```

This would expand to:

```
volume = (a+b * a+b * a+b);
```

which would obviously not produce the correct results. This is because the preprocessor performs a blind test substitution of the argument a+b in place of x. This shortcoming can be corrected by using parentheses for each occurrence of a formal argument in the *string*. Example:

```
#define     CUBE(x)     ( (x) * (x) *(x) )
```

This would result in correct expansion of **CUBE**(a+b) as:

```
volume = ( (a+b) * (a+b) * (a+b));
```

Remember to use parentheses for each occurrence of a formal argument, as well as the whole *string*.

Some commonly used definitions are:

```
#define    MAX(a,b)      (((a) > (b)) ? (a) : (b))
#define    MIN(a,b)      (((a) < (b)) ? (a) : (b))
#define    ABS(x)        (((x) > 0) ? (x) : (-(x)))
#define    STREQ(s1,s2)  (strcmp((s1,) (s2)) == 0)
#define    STRGT(s1,s2)  (strcmp((s1,) (s2)) > 0)
```

The argument supplied to a macro can be any series of characters. For example, the definition

```
#define PRINT(variable, format)   printf("variable = %format \n", variable)
```

can be called-in by

```
PRINT(price x quantity, f);
```

The preprocessor will expand this as

```
printf( "price x quantity = %f\n", price x quantity);
```

Note that the actual parameters are substituted for formal parameters in a macro call, although they are within a string. This definition can be used for printing integers and character strings as well.

Nesting of Macros

We can also use one macro in the definition of another macro. That is, macro definitions may be nested. For instance, consider the following macro definitions.

```
#define    M          5
#define    N          M+1
#define    SQUARE(x)  ( (x) * (x) )
#define    CUBE(x)    (SQUARE (x) * (x))
#define    SIXTH(x)   (CUBE(x) * CUBE(x))
```

The preprocessor expands each **#define** macro, until no more macros appear in the text. For example, the last definition is first expanded into

```
((SQUARE(x) * (x)) * (SQUARE(x) * (x)))
```

Since **SQUARE (x)** is still a macro, it is further expanded into

```
( ( ((x)*(x)) * (x) ) * ( ((x) * (x)) * (x)) )
```

which is finally evaluated as x^6.

Macros can also be used as parameters of other macros. For example, given the definitions of M and N, we can define the following macro to give the maximum of these two:

```
#define MAX(M,N) (( (M) > (N) ) ? (M) : (N))
```

Macro calls can be nested in much the same fashion as function calls. Example:

```
#define    HALF(x)    ( (x)/2.0)
#define    Y          HALF(HALF(x))
```


Similarly, given the definition of MAX(a,b) we can use the following nested call to give the maximum of the three values x,y, and z:

```
MAX (x, MAX(y,z))
```

Undefining a Macro

A defined macro can be undefined, using the statement

#undef *identifier*

This is useful when we want to restrict the definition only to a particular part of the program.

14.3 FILE INCLUSION

An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive

#include "filename"

where *filename* is the name of the file containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of *filename* into the source code of the program. When the *filename* is included within the double quotation marks, the search for the file is made first in the current directory and then in the standard directories.

Alternatively this directive can take the form

#include <filename>

without double quotation marks. In this case, the file is searched only in the standard directories.

Nesting of included files is allowed. That is, an included file can include other files. However, a file cannot include itself.

If an included file is not found, an error is reported and compilation is terminated.

Let use assume that we have created the following three files:

SYNTAX.C	contains syntax definitions.
STAT.C	contains statistical functions.
TEST.C	contains test functions.

We can make use of a definition or function contained in any of these files by including them in the program as:

```
#include   <stdio.h>
#include   "SYNTAX.C"
#include   "STAT.C"
#include   "TEST.C"
#define    M       100
main ()
{
     ----------
     ----------
     ----------
}
```

14.4 COMPILER CONTROL DIRECTIVES

While developing large programs, you may face one or more of the following situations:

1. You have included a file containing some macro definitions. It is not known whether a particular macro (say, TEST) has been defined in that header file. However, you want to be certain that Test is defined (or not defined).
2. Suppose a customer has two different types of computers and you are required to write a program that will run on both the systems. You want to use the same program, although certain lines of code must be different for each system.
3. You are developing a program (say, for sales analysis) for selling in the open market. Some customers may insist on having certain additional features. However, you would like to have a single program that would satisfy both types of customers.
4. Suppose you are in the process of testing your program, which is rather a large one. You would like to have print calls inserted in certain places to display intermediate results and messages in order to trace the flow of execution and errors, if any. Such statements are called 'debugging' statements. You want these statements to be a part of the program and to become 'active' only when you decide so.

One solution to these problems is to develop different programs to suit the needs of different situations. Another method is to develop a single, comprehensive program that includes all optional codes and then directs the compiler to skip over certain parts of source code when they are not required. Fortunately, the C preprocessor offers a feature known as *conditional compilation*, which can be used to 'switch' on or off a particular line or group of lines in a program.

Situation 1

This situation refers to the conditional definition of a macro. We want to ensure that the macro TEST is always defined, irrespective of whether it has been defined in the header file or not. This can be achieved as follows:

```
#include   "DEFINE.H"
#ifndef    TEST
#define    TEST 1
#endif
... ...
```

DEFINE.H is the header file that is supposed to contain the definition of **TEST** macro. The directive.

#ifndef TEST

searches for the definition of **TEST** in the header file and *if not defined*, then all the lines between the **#ifndef** and the corresponding **#endif** directive are left 'active' in the program. That is, the preprocessor directive

 # define TEST is processed.

In case, the TEST has been defined in the header file, the **#ifndef** condition becomes false, therefore the directive **#define TEST** is ignored. Remember, you cannot simply write

 # define TEST 1

because if **TEST** is already defined, an error will occur.

Similar is the case when we want the macro **TEST** never to be defined. Looking at the following code:

```
... ... ...
#ifdef    TEST
#undef     TEST
#endif
... ...
... ...
```

This ensures that even if **TEST** is defined in the header file, its definition is removed. Here again we cannot simply say

#undef TEST

because, if TEST is not defined, the directive is erroneous.

Situation 2

The main concern here is to make the program portable. This can be achieved as follows:

```
... ...
... ...
main()
{
   ... ...
   ... ...
#ifdef IBM_PC
{
   ... ...
   ... ...              code for IBM_PC
   ... ...
}
#else
{
   ... ...
   ... ...              code for HP machine
   ... ...
}
#endif
   ... ..
   ... ..
}
```

If we want the program to run on IBM PC, we include the directive

#define IBM_PC

in the program; otherwise we don't. Note that the compiler control directives are inside the function. Care must be taken to put the # character at column one.

The compiler complies the code for IBM PC if **IBM-PC** is defined, or the code for the HP machine if it is not.

Situation 3

This is similar to the above situation and therefore the control directives take the following form:

```
#ifdef   ABC
     group-A lines
#else
     group-B lines
#endif
```

Group-A lines are included if the customer **ABC** is defined. Otherwise, group-B lines are included.

Situation 4

Debugging and testing are done to detect errors in the program. While the Compiler can detect syntactic and semantic errors, it cannot detect a faulty algorithm where the program executes, but produces wrong results.

The process of error detection and isolation begins with the testing of the program with a known set of test data. The program is divided down and **printf** statements are placed in different parts to see intermediate results. Such statements are called debugging statements and are not required once the errors are isolated and corrected. We can either delete all of them or, alternately, make them inactive using control directives as:

```
...
...
#ifdef TEST
{
      printf("Array elements\n");
      for (i = 0; i< m; i++)
           printf("x[%d] = %d\n", i, x[i]);
}
#endif
...
...
#ifdef TEST
      printf(...);
#endif
...
```

The statements between the directives **#ifdef** and **#endif** are included only if the macro **TEST** is defined. Once everything is OK, delete or undefine the **TEST**. This makes the **#ifdef TEST** conditions false and therefore all the debugging statements are left out.

The C preprocessor also supports a more general form of test condition - **#if** directive. This takes the following form:


```
#if constant expression
{
        statement-1;
        statement-2;
        ...
        ...
}
#endif
```

The *constant-expression* may be any logical expression such as:

```
TEST <= 3
(LEVEL == 1 || LEVEL == 2)
MACHINE == 'A'
```

If the result of the constant-expression is nonzero (true), then all the statements between the **#if** and **#endif** are included for processing; otherwise they are skipped. The names **TEST**, **LEVEL**, etc. may be defined as macros.

14.5 ANSI ADDITIONS

ANSI committee has added some more preprocessor directives to the existing list given in Table 14.1. They are:

#elif	Provides alternative test facility
#pragma	Specifies certain instructions
#error	Stops compilation when an error occurs

The ANSI standard also includes two new preprocessor operations:

#	Stringizing operator
##	Token-pasting operator

elif Directive

The #elif enables us to establish an "if..else..if.." sequence for testing multiple conditions. The general form of use of **#elif** is:

```
#if   expression 1
      statement sequence 1
#elif expression 2
      statement sequence 2
      .....
      .....
        #elif  expression N
               statement sequence N
#endif
```

For example:

```
#if  MACHINE == HCL
     #define FILE "hcl.h"
```

```
#elif   MACHINE == WIPRO
    #define FILE "wipro.h"

#elif   MACHINE == DCM
    #define FILE "dcm.h"
#endif
#include FILE
```

#pragma Directive

The **#pragma** is an implementation oriented directive that allows us to specify various instructions to be given to the compiler. It takes the following form:

```
#pragma name
```

where, *name* is the name of the **pragma** we want. For example, under Microsoft C,

```
#pragma loop_opt(on)
```

causes loop optimization to be performed. It is ignored, if the compiler does not recognize it.

#error Directive

The **#error** directive is used to produce diagnostic messages during debugging. The general form is

```
#error error message
```

When the **#error** directive is encountered, it displays the error message and terminates processing. Example

```
#if !defined(FILE_G)
#error NO GRAPHICS FACILITY
#endif
```

Note that we have used a special processor operator **defined** along with **#if**. **defined** is a new addition and takes a *name* surrounded by parentheses. If a compiler does not support this, we can replace it as follows:

#if !defined	by	#ifndef
#if defined	by	#ifdef

Stringizing Operator

ANSI C provides an operator # called *stringizing operator* to be used in the definition of macro functions. This operator allows a formal argument within a macro definition to be converted to a string. Consider the example below:

```
#define sum(xy) printf(#xy " = %f\n", xy)
main()
```



```
{
    ...
    ...
    sum(a+b);
    ...
}
```

The preprocessor will convert the line

```
sum(a+b);
```

into

```
printf("a+b" "=%f\n", a+b);
```

which is equivalent to

```
printf("a+b =%f\n", a+b);
```

Note that the ANSI standard also stipulates that adjacent strings will be concatenated.

Token Pasting Operator

The token pasting operator ## defined by ANSI standard enables us to combine two tokens within a macro definition to form a single token. For example:

```
#define combine(s1,s2) s1 ## s2
main()
{
    ... ..
    ... ..
    printf("%f", combine(total, sales));
    ... ..
    ... ..
}
```

The preprocessor transforms the statement

```
printf("%f", combine(total, sales));
```

into the statement

```
printf("%f", totalsales);
```

Consider another macro definition:

```
#define print(i) printf("a" #i "=%f", a##i)
```

This macro will convert the statement

```
print(5);
```

into the statement

```
printf("a5 = %f", a5)
```

Review Questions

14.1 Explain the role of the C preprocessor.
14.2 What is a macro and how is it different from a C variable name?
14.3 What precautions one should take when using macros with argument?
14.4 What are the advantages of using macro definitions in a program?
14.5 When does a programmer use **#include** directive?
14.6 The value of a macro name cannot be changed during the running of a program. Comment?
14.7 What is conditional compilation? How does it help a programmer?
14.8 Distinguish between **#ifdef** and **#if** directives.
14.9 Comment on the following code fragment:

```
#if 0
{
        line-1;
        line-2;
        ...
        ...
        line-n;
}
#endif
```

14.10 Identify errors, if any, in the following macro definitions:
(a) #define until(x) while(!x)
(b) #define ABS(x) (x > 0) ? (x) : (–x)
(c) #ifdef(FLAG)
  #undef FLAG
  #endif
(d) #if n == 1 update(item)
  #else print-out(item)
  #endif

14.11 State whether the following statements are true or false.
(a) The keyword **#define** must be written starting from the first column.
(b) Like other statements, a processor directive must end with a semicolon.
(c) All preprocessor directives begin with #.
(d) We cannot use a macro in the definition of another macro.

14.12 Fill in the blanks in the following statements.
(a) The _____ directive discords a macro.
(b) The operator _____ is used to concatenate two arguments.
(c) The operator _____ converts its operand.
(d) The _____ directive causes an implementation-oriented action.

14.13 Enumerate the differences between functions and parameterized macros.
14.14 In **#include** directives, some file names are enclosed in angle brackets while others are enclosed in double quotation marks. Why?
14.15 Why do we recommend the use of parentheses for formal arguments used in a macro definition? Give an example.

Programming Exercises

14.1 Define a macro PRINT_VALUE that can be used to print two values of arbitrary type.
14.2 Write a nested macro that gives the minimum of three values.
14.3 Define a macro with one parameter to compute the volume of a sphere. Write a program using this macro to compute the volume for spheres of radius 5, 10 and 15 metres.
14.4 Define a macro that receives an array and the number of elements in the array as arguments. Write a program using this macro to print out the elements of an array.
14.5 Using the macro defined in the exercise 14.4, write a program to compute the sum of all elements in an array.
14.6 Write symbolic constants for the binary arithmetic operators +, –, * and /. Write a short program to illustrate the use of these symbolic constants.
14.7 Define symbolic constants for { and } and printing a blank line. Write a small program using these constants.
14.8 Write a program to illustrate the use of stringizing operator.

15

Developing a C Program: Some Guidelines

15.1 INTRODUCTION

We have discussed so far various features of C language and are ready to write and execute programs of modest complexity. However, before attempting to develop complex programs, it is worthwhile to consider some programming techniques that would help design efficient and error-free programs.

The program development process includes three important stages, namely, program design, program coding and program testing. All the three stages contribute to the production of high-quality programs. In this chapter we shall discuss some of the techniques used for program design, coding and testing.

15.2 PROGRAM DESIGN

Program design is the foundation for a good program and is therefore an important part of the program development cycle. Before coding a program, the program should be well conceived and all aspects of the program design should be considered in detail.

Program design is basically concerned with the development of a strategy to be used in writing the program, in order to achieve the solution of a problem. This includes mapping out a solution procedure and the form the program would take. The program design involves the following four stages:

1. Problem analysis.
2. Outlining the program structure.
3. Algorithm development.
4. Selection of control structures.

Problem Analysis

Before we think of a solution procedure to the problem, we must fully understand the nature of the problem and what we want the program to do. Without the comprehension and

definition of the problem at hand, program design might turn into a hit-or-miss approach. We must carefully decide the following at this stage;

What kind of data will go in?;
What kind of outputs are needed?; and .
What are the constraints and conditions under which the program has to operate?

Outlining the Program Structure

Once we have decided what we want and what we have, then the next step is to decide how to do it. C as a structured language lends itself to a *top-down* approach. Top-down means decomposing of the solution procedure into tasks that form a hierarchical structure, as shown in Fig. 15.1. The essence of the top-down design is to cut the whole problem into a number of independent constituent tasks, and then to cut the tasks into smaller subtasks, and so on, until they are small enough to be grasped mentally and to be coded easily. These tasks and subtasks can form the basis of functions in the program.

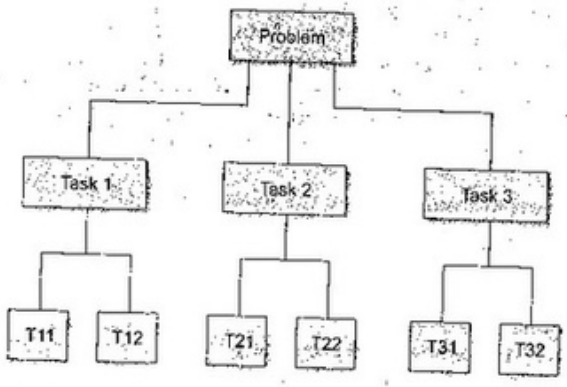


Fig. 15.1 *Hierarchical structure*

An important feature of this approach is that at each level, the details of the design of lower levels are hidden. The higher-level functions are designed first, assuming certain broad tasks of the immediately lower-level functions. The actual details of the lower-level functions are not considered until that level is reached. Thus the design of functions proceeds from top to bottom, introducing progressively more and more refinements.

This approach will produce a readable and modular code that can be easily understood and maintained. It also helps us classify the overall functioning of the program in terms of lower-level functions.

Algorithm Development

After we have decided a solution procedure and an overall outline of the program, the next step is to work out a detailed definite, step-by-step procedure, known as *algorithm* for each function. The most common method of describing an algorithm is through the use of *flowcharts*. The other method is to write what is known as *pseudocode*. The flow chart presents

the algorithm pictorially, while the pseudocode describe the solution steps in a logical order. Either method involves concepts of logic and creativity.

Since algorithm is the key factor for developing an efficient program, we should devote enough attention to this step. A problem might have many different approaches to its solution. For example, there are many sorting techniques available to sort a list. Similarly, there are many methods of finding the area under a curve. We must consider all possible approaches and select the one, which is simple to follow, takes less execution time, and produces results with the required accuracy.

Control Structures

A complex solution procedure may involve a large number of control statements to direct the flow of execution. In such situations, indiscriminate use of control statements such as **goto** may lead to unreadable and uncomprehensible programs. It has been demonstrated that any algorithm can be structured, using the three basic control structure, namely, sequence structure, selection structure, and looping structure.

Sequence structure denotes the execution of statements sequentially one after another. Selection structure involves a decision, based on a condition and may have two or more branches, which usually join again at a later point. **if . . . .else** and **switch** statements in C can be used to implement a selection structure. Looping structure is used when a set of instructions is evaluated repeatedly. This structure can be implemented using **do**, **while**, or **for** statements.

A well-designed program would provide the following benefits:

1. Coding is easy and error-free.
2. Testing is simple.
3. Maintenance is easy.
4. Good documentation is possible.
5. Cost estimates can be made more accurately.
6. Progress of coding may be controlled more precisely.

15.3 PROGRAM CODING

The algorithm developed in the previous section must be translated into a set of instructions that a computer can understand. The major emphasis in coding should be simplicity and clarity. A program written by one may have to be read by others later. Therefore, it should be readable and simple to understand. Complex logic and tricky coding should be avoided. The elements of coding style include:

- Internal documentation.
- Construction of statements.
- Generality of the program.
- Input/output formats.

Internal Documentation

Documentation refers to the details that describe a program. Some details may be built-in as an integral part of the program. These are known as *internal documentation.*

Two important aspects of internal documentation are, selection of meaningful variable names and the use of comments. Selection of meaningful names is crucial for understanding the program. For example,

area = breadth * length

is more meaningful than

```
a = b * l;
```

Names that are likely to be confused must be avoided. The use of meaningful function names also aids in understanding and maintenance of programs.

Descriptive comments should be embedded within the body of source code to describe processing steps.

The following guidelines might help the use of comments judiciously:

1. Describe blocks of statements, rather than commenting on every line.
2. Use blank lines or indentation, so that comments are easily readable.
3. Use appropriate comments; an incorrect comment is worse than no comment at all.

Statement Construction

Although the flow of logic is decided during design, the construction of individual statements is done at the coding stage. Each statement should be simple and direct. While multiple statements per line are allowed, try to use only one statement per line with necessary indentation. Consider the following code:

```
if(quantity>0){code = 0; quantity = rate;}
else { code = 1; sales = 0;}
```

Although it is perfectly valid, it could be reorganized as follows:

```
if(quantity>0)
{
    code = 0;
    quantity = rate;
}
else
{
    code = 1;
    sales = 0;
}
```

The general guidelines for construction of statements are:

1. Use one statement per line.
2. Use proper indentation when selection and looping structures are implemented.
3. Avoid heavy nesting of loops, preferably not more than three levels.
4. Use simple conditional tests; if necessary break complicated conditions into simple conditions.
5. Use parentheses to clarify logical and arithmetic expressions.
6. Use spaces, wherever possible, to improve readability.

Input/Output Formats

Input/output formats should be simple and acceptable to users. A number of guidelines should be considered during coding.

1. Keep formats simple.
2. Use end-of-file indicators, rather than the user requiring to specify the number of items.
3. Label all interactive input requests.
4. Label all output reports.
5. Use output messages when the output contains some peculiar results.

Generality of Programs

Care should be taken to minimize the dependence of a program on a particular set of data, or on a particular value of a parameter. Example:

```
for(sum = 0, i=1; i <= 10; i++)
    sum = sum + i;
```

This loop adds numbers 1,2, ...10. This can be made more general as follows;

```
sum =0;
for(i =m; i <=n; i = i+ step);
    sum = sum + i;
```

The initial value **m**, the final value **n**, and the increment size **step** can be specified interactively during program execution. When **m**=2, **n**=100, and **step** =2, the loop adds all even numbers up to, and including 100.

15.4 COMMON PROGRAMMING ERRORS

By now you must be aware that C has certain features that are easily amenable to bugs. Added to this, it does not check and report all kinds of run-time errors. It is therefore, advisable to keep track of such errors and to see that these known errors are not present in the program. This section examines some of the more common mistakes that a less experienced C programmer could make.

Missing Semicolons

Every C statement must end with a semicolon. A missing semicolon may cause considerable confusion to the compiler and result in 'misleading' error messages. Consider the following statements:

```
a = x+y
b = m/n;
```

The compiler will treat the second line as a part of the first one and treat b as a variable name. You may therefore get an "undefined name" error message in the second line. Note that both the message and location are incorrect. In such situations where there are no errors in a reported line, we should check the preceding line for a missing semicolon.

There may be an instance when a missing semicolon might cause the compiler to go 'crazy' and to produce a series of error messages. If they are found to be dubious errors, check for a missing semicolon in the beginning of the error list.

Misuse of Semicolon

Another common mistake is to put a semicolon in a wrong place. Consider the following code:

```
for(i = 1; i<=10; i++);
    sum = sum + i;
```

This code is supposed to sum all the integers from 1 to 10. But what actually happens is that only the 'exit' value of i is added to the sum. Other examples of such mistake are:

1.
```
while (x < Max);
{
}
```
2.
```
if(T>= 200);
grade = 'A';
```

A simple semicolon represents a null statement and therefore it is syntactically valid. The compiler does not produce any error message. Remember, these kinds of errors are worse than syntax errors.

Use of = Instead of = =

It is quite possible to forget the use of double equal sings when we perform a relational test. Example:

```
if(code = 1)
  count ++;
```

It is a syntactically valid statement. The variable code is assigned 1 and then, because code = 1 is true, the count is incremented. In fact, the above statement does not perform any relational test on code. Irrespective of the previous value of code, **count ++;** is always executed.

Similar mistakes can occur in other control statements, such as **for** and **while**. Such a mistake in the loop control statements might cause infinite loops.

Missing Braces

It is common to forget a closing brace when coding a deeply nested loop. It will be usually detected by the compiler because the number of opening braces should match with the closing ones. However, if we put a matching brace in a wrong place, the compiler won't notice the mistake and the program will produce unexpected results.

Another serious problem with the braces is, not using them when multiple statements are to be grouped together. For instance, consider the following statements:

```
for(i=1; i <= 10; i++)
    sum1 = sum 1 +i;
    sum2 = sum2 + i*i;
printf("%d %d\n", sum1,sum2);
```

This code is intended to compute **sum1**, **sum2** for i varying from 1 to 10, in steps of 1 and then to print their values. But, actually the **for** loop treats only the first statement, namely,

```
sum = sum1 + i;
```

as its body and therefore the statement

```
sum2 = sum2 + i*i;
```

is evaluated only once when the loop is exited. The correct way to code this segment is to place braces as follows:

```
for(i=1; i<=10; i++)
{
    sum1 = sum1 + i;
    sum2 = sum2 +i*i;
}
printf("%d %d\n", sum1 sum2);
```

In case, only one brace is supplied, the behaviour of the compiler becomes unpredictable.

Missing Quotes

Every string must be enclosed in double quotes, while a single character constant in single quotes. If we miss them out, the string (or the character) will be interpreted as a variable name. Examples:

```
if(response !=YES)   /* YES is a string */
Grade = A;           /* A is a character constant */
```

Here YES and A are treated as variables and therefore, a message "undefined names" may occur.

Misusing Quotes

It is likely that we use single quotes whenever we handle single characters. Care should be exercised to see that the associated variables are declared properly. For example, the statement

```
city = 'M';
```

would be invalid if **city** has been declared as a **char** variable with dimension (i.e., pointer to **char**).

Improper Comment Characters

Every comment should start with a /* and end with a */. Anything between them is ignored by the compiler. If we miss out the closing */, then the compiler searches for a closing */ further down in the program, treating all the lines as comments. In case, it fails to find a closing */, we may get an error message. Consider the following lines:

```
. . . . .
/* comment line 1
statement1;
statement2;
/* comment line 2 */
statement 3;
. . . . .
```

Since the closing */ is missing in the comment line 1, all the statements that follow, until the closing comment */ in comment line 2 are ignored.

We should remember that C does not support nested comments. Assume that we want to comment out the following segment:

```
. . . . .
x = a-b;
Y = c-d;
/* compute ratio */
ratio = x/y;
. . . . .
. . . . .
```

we may be tempted to add comment characters as follows:

```
/* x = a-b;
y = c-d;
/* Compute ratio */
ratio = x/y; */
```

This is incorrect. The first opening comment matches with the first closing comment and therefore the lines between these two are ignored. The statement

```
ratio = x/y;
```

is not commented out. The correct way to comment out this segment is shown as:

```
/* x = a-b;
y = c-d; */
/* compute ratio */
/* ratio = x/y; */
```

Undeclared Variables

C requires every variable to be declared for its type, before it is used. During the development of a large program, it is quite possible to use a variable to hold intermediate results and to forget to declare it.

Forgetting the Precedence of Operators

Expressions are evaluated according to the precedence of operators. It is common among beginners to forget this. Consider the statement

```
if (value = product ( ) >= 100)
    tax = 0.05 * value;
```

The call **product ( )** returns the product of two numbers, which is compared to 100. If it is equal to or greater than 100, the relational test is true, and a 1 is assigned to **value**, otherwise a 0 is assigned. In either case, the only values **value** can take on are 1 or 0. This certainly is not what the programmer wanted.

The statement was actually expected to assign the value returned by **product( )** to **value** and then compare **value** with 100. If **value** was equal to or greater than 100, tax should have been computed, using the statement

```
tax = 0.05 * value;
```

The error is due to the higher precedence of the relational operator compared to the assignment operator. We can force the assignment to occur first by using parentheses as follows:

```
if(value = product()) >=100)
tax = 0.05 * value;
```

Similarly, the logical operators **&&** and **||** have lower precedence than arithmetic and relational operators and among these two, **&&** has higher precedence than **||**. Try, if there is any difference between the following statements:

1. if(p > 50 || c > 50 && m > 60 && T > 180)
   x = 1;
2. if((p > 50 || c > 50) && m > 60 && T > 180)
   x = 1;
3. if(p > 50 || c > 50 && m > 60) && T > 180)
   x = 1;

Ignoring the Order of Evaluation of Increment/Decrement Operators

We often use increment or decrement operators in loops. Example

```
... ...
i = 0;
while ((c = getchar()) != '\n';
{
    string[i++] = c;
}
string[i-1] = '\n';
```

The statement **string[i++]** = c; is equivalent to :

```
string[i] = c;
i = i+1;
```

This is not the same as the statement **string[++i]** = c; which is equivalent to

```
i =i+1;
string[i] = c;
```

Forgetting to Declare Function Parameters

Remember to declare all function parameters in the function header.

Mismatching of Actual and Formal Parameter Types in Function Calls

When a function with parameters is called, we should ensure that the type of values passed match with the type expected by the called function. Otherwise, erroneous results may occur. If necessary, we may use the *type* cast operator to change the type locally. Example

```
y = cos((double)x);
```


Nondeclaration of Functions

Every function that is called should be declared in the calling function for the types of value it returns. Consider the following program:

```
main()
{
    float a =12.75;
    float b = 7.36;
    printf("%f\n", division(a,b));
}
double division(float x, float y)
{
    return(x/y);
}
```

The function returns a **double** type value but this fact is not known to the calling function and therefore it expects to receive an **int** type value. The program produces either meaningless results or error message such as "redefinition".

The function **division** is like any other variable for the **main** and therefore it should be declared as **double** in the main.

Now, let us assume that the function division is coded as follows:

```
division(float x, float y)
{
  return(x/y);
}
```

Although the values x and y are floats and the result of x/y is also float, the function returns only integer value because no type specifier is given in the function definition. This is wrong too. The function header should include the type specifier to force the function to return a particular type of value.

Missing & Operator in scanf Parameters

All non-pointer variables in a scanf call should be preceded by an & operator. If the variable code is declared as an integer, then the statement

```
scanf("%d", code);
```

is wrong. The correct one is scanf("%d", &code);

Remember, the compiler will not detect this error and you may get a crazy output.

Crossing the Bounds of an Array

All C indices start from zero. A common mistake is to start the index from 1. For example, the segment

```
int x[10], sum i;
Sum = 0;
for (i = 1; i < = 10; i++)
    sum = sum + x[i];
```

would not find the correct sum of the elements of array x. The for loop expressions should be corrected as follows:

```
for(i=0;i<10;i++)
```

Forgetting a Space for Null character in a String

All character arrays are terminated with a null character and therefore their size should be declared to hold one character more than the actual string size.

Using Uninitialized Pointers

An uninitialized pointer points to garbage. The following program is wrong:

```
main()
{
    int a, *ptr;
    a = 25;
    *ptr = a+5;
}
```

The pointer **ptr** has not been initialized.

Missing Indirection and Address Operators

Another common error is to forget to use the operators * and & in certain places. Consider the following program:

```
main()
{
    int m, *p1;
    m = 5;
    p1 = m;
    printf("%d\n", *p1);
}
```

This will print some unknown value because the pointer assignment

```
p1 =m;
```

is wrong. It should be:

```
p1 = &m;
```

Consider the following expression:

```
y = p1 + 10;
```

Perhaps, **y** was expected to be assigned the value at location **p1** plus 10. But it does not happen. **y** will contain some unknown address value. The above expression should be rewritten as:

```
y = *p1 + 10;
```


Missing Parentheses in Pointer Expressions

The following two statements are not the same:

```
x = *p1 + 1;
x = *(p1 + 1);
```

The first statement would assign the value at location **p1** plus 1 to **x**, while the second would assign the value at location **p1 + 1**.

Omitting Parentheses around Arguments in Macro Definitions

This would cause incorrect evaluation of expression when the macro definition is substituted.

Example:

The call `# define f(x) x * x + 1`
`y = f(a+b);`
will be evaluated as `y = a+b * a+b+1;` which is wrong.

Some other mistakes that we commonly make are:

- Wrong indexing of loops.
- Wrong termination of loops.
- Unending loops.
- Use of incorrect relational test.
- Failure to consider all possible conditions of a variable.
- Trying to divide by zero.
- Mismatching of data specifications and variables in **scanf** and **printf** statements.
- Forgetting truncation and rounding off errors.

15.5 PROGRAM TESTING AND DEBUGGING

Testing and debugging refer to the tasks of detecting and removing errors in a program, so that the program produces the desired results on all occasions. Every programmer should be aware of the fact that rarely does a program run perfectly the first time. No matter how thoroughly the design is carried out, and no matter how much care is taken in coding, one can never say that the program would be 100 per cent error-free. It is therefore necessary to make efforts to detect, isolate and correct any errors that are likely to be present in the program.

Types of Errors

We have discussed a number of common errors. There might be many other errors, some obvious and others not so obvious. All these errors can be classified under four types, namely, syntax errors, run-time errors, logical errors, and latent errors.

Syntax errors: Any violation of rules of the language results in syntax errors. The compiler can detect and isolate such errors. When syntax errors are present, the compilation fails and is terminated after listing the errors and the line numbers in the source program, where the errors have occurred. Remember, in some cases, the line number may not exactly indicate

the place of the error. In other cases, one syntax error may result in a long list of errors. Correction of one or two errors at the beginning of the program may eliminate the entire list.

Run-time errors: Errors such as a mismatch of data types or referencing an out-of-range array element go undetected by the compiler. A program with these mistakes will run, but produce erroneous results and therefore, the name run-time errors is given to such errors. Isolating a run-time error is usually a difficult task.

Logical errors: As the name implies, these errors are related to the logic of the program execution. Such actions as taking a wrong path, failure to consider a particular condition, and incorrect order of evaluation of statements belong to this category. Logical errors do not show up as compiler-generated error messages. Rather, they cause incorrect results. These errors are primarily due to a poor understanding of the problem, incorrect translation of the algorithm into the program and a lack of clarity of hierarchy of operators. Consider the following statement:

```
if(x ==y)
  printf("They are equal\n");
```

when x and y are float types values, they rarely become equal, due to truncation errors. The printf call may not be executed at all. A test like **while(x != y)** might create an infinite loop.

Latent errors: It is a 'hidden' error that shows up only when a particular set of data is used. For example, consider the following statement:

```
ratio = (x+y)/(p-q);
```

An error occurs only when **p** and **q** are equal. An error of this kind can be detected only by using all possible combinations of test data.

Program Testing

Testing is the process of reviewing and executing a program with the intent of detecting errors, which may belong to any of the four kinds discussed above. We know that while the compiler can detect syntactic and semantic errors, it cannot detect run-time and logical errors that show up during the execution of the program. Testing, therefore, should include necessary steps to detect all possible errors in the program. It is, however, important to remember that it is impractical to find all errors. Testing process may include the following two stages:

1. Human testing.
2. Computer-based testing.

Human testing is an effective error-detection process and is done before the computer-based testing begins. Human testing methods include code inspection by the programmer, code inspection by a test group, and a review by a peer group. The test is carried out statement by statement and is analyzed with respect to a checklist of common programming errors. In addition to finding the errors, the programming style and choice of algorithm are also reviewed.

Computer-based testing involves two stages, namely *compiler testing* and *run-time testing*. Compiler testing is the simplest of the two and detects yet undiscovered syntax errors. The program executes when the compiler detects no more errors. Should it mean that the

program is correct? Will it produce the expected results? The answer is negative. The program may still contain run-time and logic errors.

Run-time errors may produce run-time error messages such as "null pointer assignment" and "stack overflow". When the program is free from all such errors, it produces output ,which might or might not be correct. Now comes the crucial test, the test for the *expected output*. The goal is to ensure that the program produces expected results under all conditions of input data.

Test for correct output is done using *test data* with known results for the purpose of comparison. The most important consideration here is the design or invention of effective test data. A useful criteria for test data is that all the various conditions and paths that the processing may take during execution must be tested.

Program testing can be done either at module (function) level or at program level. Module level test, often known as *unit test*, is conducted on each of the modules to uncover errors within the boundary of the module. Unit testing becomes simple when a module is designed to perform only one function.

Once all modules are unit tested, they should be *integrated together* to perform the desired function(s). There are likely to be interfacing problems, such as data mismatch between the modules. An *integration test* is performed to discover errors associated with interfacing.

Program Debugging

Debugging is the process of isolating and correcting the errors. One simple method of debugging is to place print statements throughout the program to display the values of variables. It displays the dynamics of a program and allows us to examine and compare the information at various points. Once the location of an error is identified and the error corrected, the debugging statements may be removed. We can use the conditional compilation statements, discussed in Chapter 14, to switch on or off the debugging statements.

Another approach is to use the process of deduction. The location of an error is arrived at using the process of elimination and refinement. This is done using a list of possible causes of the error.

The third error-locating method is to *backtrack* the incorrect results through the logic of the program until the mistake is located. That is, beginning at the place where the symptom has been uncovered, the program is traced backward until the error is located.

15.6 PROGRAM EFFICIENCY

Two critical resources of a computer system are execution time and memory. The efficiency of a program is measured in terms of these two resources. Efficiency can be improved with good design and coding practices.

Execution Time

The execution time is directly tied to the efficiency of the algorithm selected. However, certain coding techniques can considerably improve the execution efficiency. The following are some of the techniques, which could be applied while coding the program.

1. Select the fastest algorithm possible.
2. Simplify arithmetic and logical expressions.
3. Use fast arithmetic operations, whenever possible.
4. Carefully evaluate loops to avoid any unnecessary calculations within the loops.
5. If possible, avoid the use of multi-dimensional arrays.
6. Use pointers for handling arrays and strings.

However, remember the following, while attempting to improve efficiency.

1. Analyse the algorithm and various parts of the program before attempting any efficiency changes.
2. Make it work before making it faster.
3. Keep it right while trying to make it faster.
4. Do not sacrifice clarity for efficiency.

Memory Requirement

Memory restrictions in the micro-computer environment is a real concern to the programmer. It is therefore, desirable to take all necessary steps to compress memory requirements.

1. Keep the program simple. This is the key to memory efficiency.
2. Use an algorithm that is simple and requires less steps.
3. Declare arrays and strings with correct sizes.
4. When possible, limit the use of multi-dimensional arrays.
5. Try to evaluate and incorporate memory compression features available with the language.

Review Questions

15.1 Discuss the various aspects of program design.

15.2 How does program design relate to program efficiency?

15.3 Readability is more important than efficiency. Comment.

15.4 Distinguish between the following:
   a. Syntactic errors and semantic errors.
   b. Run-time errors and logical errors.
   c. Run-time errors and latent errors.
   d. Debugging and testing.
   e. Compiler testing and run-time testing.

15.5 A program has been compiled and linked successfully. When you run this program you face one or more of the following situations.
   a. Program is executed but no output.
   b. It produces incorrect answers.
   c. It does not stop running.

15.6 List five common programming mistakes. Write a small program containing these errors and try to locate them with the help of computer.

15.7 In a program, two values are compared for convergence, using the statement

```
if( (x-y) < 0.00001) ...
```

Does the statement contain any error? If yes, explain the error.

15.8 A program contains the following if statements:

```
... ..
... ..
if(x>1&&y == 0)p = p/x;
if(x == 5|| p > 2) p = p+2;
... ..
... ..
```

Draw a flow chart to illustrate various logic paths for this segment of the program and list test data cases that could be used to test the execution of every path shown.

15.9 Given below is a function to compute the yth power of an integer x.

```
power(int x, int y)
{
  int p;
  p = y;
  while(y > 0)
        x *= y --;
  return(x);
}
```

This function contains some bugs. Write a test procedure to locate the errors with the help of a computer.

15.10 A program reads three values from the terminal, representing the lengths of three sides of a box namely length, width and height and prints a message stating whether the box is a cube, rectangle, or semi-rectangle. Prepare sets of data that you feel would adequately test this program.

I

Bit-Level Programming

1 INTRODUCTION

One of the unique features of C language as compared to other high-level languages is that it allows direct manipulation of individual bits within a word. Bit-level manipulations are used in setting a particular bit or group of bits to 1 or 0. They are also used to perform certain numerical computations faster. As pointed out in Chapter 3, C supports the following operators:

1. Bitwise logical operators.
2. Bitwise shift operators.
3. One's complement operator.

All these operators work only on integer type operands.

2 BITWISE LOGICAL OPERATORS

There are three logical bitwise operators. They are:

- *Bitwise* AND (&)
- *Bitwise* OR (|)
- *Bitwise exclusive* OR (^)

These are binary operators and require two integer-type operands. These operators work on their operands bit by bit starting from the least significant (i.e. the rightmost) bit, setting each bit in the result as shown in Table 1.

Table 1 *Result of Logical Bitwise Operations*

op1	*op2*	*op1 & op2*	*op1 | op2*	*op1 ^ op2*
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Bitwise AND

The bitwise AND operator is represented by a single ampersand (&) and is surrounded on both sides by integer expressions. The result of ANDing operation is 1 if both the bits have a value of 1; otherwise it is 0. Let us consider two variables **x** and **y** whose values are 13 and 25. The binary representation of these two variables are

```
x - - -> 0000 0000 0000 1101
y - - -> 0000 0000 0001 1001
```

If we execute statement

```
z = x & y ;
```

then the result would be:

```
z - - -> 0000 0000 0000 1001
```

Although the resulting bit pattern represents the decimal number 9, there is no apparent connection between the decimal values of these three variables.

Bitwise ANDing is often used to test whether a particular bit is 1 or 0. For example, the following program tests whether the fourth bit of the variable **flag** is 1 or 0.

```
#define TEST 8 /* represents 00.....01000 */
main()
{
        int flag;
        ....
        ....
        if((flag & TEST) != 0) /* test 4th bit */
        {
          printf(" Fourth bit is set \n");
        }
        ....
        ....
```

Note that the bitwise logical operators have lower precedence than the relational operators and therefore additional parentheses are necessary as shown above.

The following program tests whether a given number is odd or even.

```
main()
{
        int test = 1;
        int number;

        printf("Input a number \n");
        scanf("%d", &number);

        while (number != -1)
        {
            if(number & test)
                print("Number is odd\n\n");
            else
```

```
            printf("Number is even\n\n");
        else
            printf("Number is odd\n\n");
        printf("Input a number \n");
        scanf("%d", &number);
    }
}
```

Output

```
Input a number
20
Number is even

Input a number
9
Number is odd

Input a number
-1
```

Bitwise OR

The bitwise OR is represented by the symbol | (vertical bar) and is surrounded by two integer operands. The result of OR operation is 1 if *at least* one of the bits has a value of 1; otherwise it is zero. Consider the variables **x** and **y** discussed above.

```
x - - ->      0000 0000 0000 1101
y - - ->      0000 0000 0001 1001

x|y - - ->    0000 0000 0001 1101
```

The bitwise inclusion OR operation is often used to set a particular bit to 1 in a flag. Example:

```
#define SET 8
main()
{
    int flag;
    ....
    ....
    flag = flag | SET;

    if (( flag & SET) != 0)
    {
      printf("flag is set \n");
    }
    ....
    ....
}
```

The statement

```
flag = flag | SET;
```

causes the fourth bit of flag to set 1 if it is 0 and does not change it if it is already 1.

Bitwise Exclusive OR

The bitwise *exclusive* OR is represented by the symbol ^. The result of exclusive OR is 1 if *only one* of the bits is 1; otherwise it is 0. Consider again the same variable **x** and **y** discussed above.

```
x - - ->      0000 0000 0000 1101
y - - ->      0000 0000 0001 1001

x^y - - ->    0000 0000 0001 0100
```

3 BITWISE SHIFT OPERATORS

The shift operators are used to move bit patterns either to the left or to the right. The shift operators are represented by the symbols << and >> and are used in the following form:

```
Left shift:   op << n
Right shift:  op >> n
```

op is the integer expression that is to be shifted and *n* is the number of bit positions to be shifted.

The left-shift operation causes all the bits in the operand *op* to be shifted to the left by *n* positions. The leftmost *n* bits in the original bit pattern will be lost and the rightmost *n* bit positions that are vacated will be filled with 0s.

Similarly, the right-shift operation causes all the bits in the operand *op* to be shifted to the right by *n* positions. The rightmost *n* bits will be lost. The leftmost *n* bit positions that are vacated will be filled with zero, if the *op* is an *unsigned integer*. If the variable to be shifted is **signed**, then the operation is machine dependent.

Both the operands *op* and *n* can be constants or variables. There are two restrictions on the value of *n*. It may not be negative and it may not exceed the number of bits used to represent the left operand *op*.

Let us suppose **x** is an unsigned integer whose bit pattern is

```
0100 1001 1100 1011
```

then,

```
                          vacated
                           positions
x << 3 = 0100 1110 0101 1000
x >> 3 = 0000 1001 0011 1001
         vacated
         positions
```

Shift operators are often used for multiplication and division by powers of two. Consider the following statement:

```
x = y << 1;
```

This statement shifts one bit to the left in **y** and then the result is assigned to **x**. The decimal value of **x** will be the value of **y** multiplied by 2. Similarly, the statement

```
x = y >> 1;
```

shifts **y** one bit to the right and assigns the result to **x**. In this case, the value of **x** will be the value of **y** divided by 2.

The shift operators, when combined with the logical bitwise operators, are useful for extracting data from an integer field that holds multiple pieces of information. This process is known as *masking*. Masking is discussed in Section 5.

4 BITWISE COMPLEMENT OPERATORS

The complement operator ~ (also called the one's complement operator) is an unary operator and inverts all the bits represented by its operand. That is, 0s become 1s and 1s become zero. Example:

```
x = 1001 0110 1100 1011
~x = 0110 1001 0011 0100
```

This operator is often combined with the bitwise AND operator to turn off a particular bit. For example, the statement

```
x = 8;          /* 0000 0000 0000 1000 */
flag = flag & ~x;
```

would turn off the fourth bit in the variable **flag**.

5 MASKING

Masking refers to the process of extracting desired bits from (or transforming desired bits in) a variable by using logical bitwise operation. The operand (a constant or variable) that is used to perform masking is called the *mask*. Examples:

```
y = x & mask;
y = x | mask;
```

Masking is used in many different ways.

- To decide bit pattern of an integer variable.
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 0s (using bitwise AND).
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 1s (using bitwise OR).
- To copy a portion of a given bit pattern to a new variable, while the remainder of the original bit pattern is inverted within the new variable (using bitwise *exclusive* OR).

The following function uses a mask to display the bit pattern of a variable.

```
void bit_pattern(int u)
{
    int i, x, word;
    unsigned mask;

    mask = 1;
    word = 8 * sizeof(int);
    mask = mask << (word - 1);
           /* shift 1 to the leftmost position */
```



```
for(i = 1; i<= word; i++)
{
  x = (u & mask) ? 1 : 0; /* identify the bit */
  printf("%d", x); /* print bit value */
  mask >>= 1; /* shift mask by 11 position to right */
```

II

ASCII Values of Characters

ASCII Value	Character	ASCII Value	Character	ASCII Value	Character	ASCII Value	Character
000	NUL	032	blank	064	@	096	←
001	SOH	033	!	065	A	097	a
002	STX	034	"	066	B	098	b
003	ETX	035	#	067	C	099	c
004	EOT	036	$	068	D	100	d
005	ENQ	037	%	069	E	101	e
006	ACK	038	&	070	F	102	f
007	BEL	039	'	071	G	103	g
008	BS	040	(	072	H	104	h
009	HT	041	)	073	I	105	i
010	LF	042	*	074	J	106	j
011	VT	043	+	075	K	107	k
012	FF	044	,	076	L	108	l
013	CR	045	–	077	M	109	m
014	SO	046	.	078	N	110	n
015	SI	047	/	079	O	111	o
016	DLE	048	0	080	P	112	p
017	DC1	049	1	081	Q	113	q
018	DC2	050	2	082	R	114	r
019	DC3	051	3	083	S	115	s
020	DC4	052	4	084	T	116	t
021	NAK	053	5	085	U	117	u
022	SYN	054	6	086	V	118	v
023	ETB	055	7	087	W	119	w
024	CAN	056	8	088	X	120	x
025	EM	057	9	089	Y	121	y
026	SUB	058	:	090	Z	122	z

(Contd.)

ASCII Value	Character	ASCII Value	Character	ASCII Value	Character	ASCII Value	Character
027	ESC	059	;	091	[	123	{
028	FS	060	<	092	\	124	|
029	GS	061	=	093	]	125	}
030	RS	062	>	094	↑	126	~
031	US	063	?	095	-	127	DEL

NOTE: The first 32 characters and the last character are control characters; they cannot be printed.

III

ANSI C Library Functions

The C language is accompanied by a number of library functions that perform various tasks. The ANSI committee has standardized header files which contain these functions. What follows is a slit of commonly used functions and the header files where they are defined. For a more complete list, the reader should refer to the manual of the version of C that is being used.

The header files that are included in this Appendix are:

<ctype.h>	Character testing and conversion functions
<math.h>	Mathematical functions
<stdio.h>	Standard I/O library functions
<stdlib.h>	Utility functions such as string conversion routines, memory allocation routines, random number generator, etc.
<string.h>	String manipulation functions
<time.h>	Time manipulation functions

Note: The following function parameters are used:

- c - character type argument
- d - double precision argument
- f - file argument
- i - integer argument
- l - long integer argument
- p - pointer argument
- s - string argument
- u - unsigned integer argument

An asterisk (*) denotes a pointer

Function	*Data type returned*	*Task*
<ctype.h>		
isalnum(c)	int	Determine if argument is alphanumeric. Return nonzero value if true; 0 otherwise.
isalpha(c)	int	Determine if argument is alphabetic. Return nonzero value if true; 0 otherwise.
isascii(c)	int	Determine if argument is an ASCII character. Return nonzero value if true; 0 otherwise.
iscntrl(c)	int	Determine if argument is an ASCII control character. Return nonzero value if true; 0 otherwise.
isdigit(c)	int	Determine if argument is a decimal digit. Return nonzero value if true; 0 otherwise.
isgraph(c)	int	Determine if argument is a graphic printing ASCII character. Return nonzero value if true; 0 otherwise.
islower(c)	int	Determine if argument is lowercase. Return nonzero value if true; 0 otherwise.
isodigit(c)	int	Determine if argument is an octal digit. Return nonzero value if true; 0 otherwise.
isprint(c)	int	Determine if argument is a printing ASCII character. Return nonzero value if true; 0 otherwise.
ispunct(c)	int	Determine if argument is a punctuation character. Return non-zero value if true; 0 otherwise.
isspace(c)	int	Determine if argument is a whitespace character. Return non-zero value if true; 0 otherwise.
isupper(c)	int	Determine if argument is uppercase. Return nonzero value if true; 0 otherwise.
isxdigit(c)	int	determine if argument is a hexadecimal digit. Return nonzero value if true; 0 otherwise.
toascii(c)	int	Convert value of argument to ASCII.
tolower(c)	int	Convert letter to lowercase.
toupper(c)	int	Convert letter to uppercase.
<math.h>		
acos(d)	double	Return the arc cosine of d.
asin(d)	double	Return the arc sine of d.
atan(d)	double	Return the arc tangent of d.
atan2(d1,d2)	double	Return the arc tangent of d1/d2.
ceil(d)	double	Return a value rounded up to the next higher integer.
cos(d)	double	Return the cosine of d.
cosh(d)	double	Return the hyperbolic cosine of d.
exp(d)	double	Raise e to the power d.
fabs(d)	double	Return the absolute value of d.
floor(d)	double	Return a value rounded down to the next lower integer.
fmod(d1, d2)	double	Return the remainder of d1/d2 (with same sign as d1).
labs(l)	long int	Return the absolute value of l.
log(d)	double	Return the natural logarithm of d.
log10(d)	double	Return the logarithm (base 10) of d.
pow(d1,d2)	double	Return d1 raised to the d2 power.
sin(d)	double	Return the sine of d.

Function	Data type returned	Task
sinh(d)	double	Return the hyperbolic sine of d.
sqrt(d)	double	Return the square root of d.
tan(d)	double	Return the tangent of d.
tanh(d)	double	Return the hyperbolic tangent of d.
<stdio.h>		
fclose(f)	int	Close file f. Return 0 if file is successfully closed.
feof(f)	int	Determine if an end-of-file condition has been reached. If so, return a nonzero value; otherwise, return 0.
fgetc(f)	int	Enter a single character form file f.
fgets(s, i, f)	char*	Enter string s, containing i characters, from file f.
fopen(s1,s2)	file*	Open a file named s1 of type s2. Return a pointer to the file.
fprint(f,...)	int	Send data items to file f.
fputc(c,f)	int	Send a single character to file f.
fputs(s,f)	int	Send string s to file f.
fread(s,i1,i2,f)	int	Enter i2 data items, each of size i1 bytes, from file f to string s.
fscanf(f,...)	int	Enter data items from file f
fseek(f,l,i)	int	Move the pointer for file f a distance l bytes from location i.
ftell(f)	long int	Return the current pointer position within file f.
fwrite(s,i1,i2,f)	int	Send i2 data items, each of size i1 bytes from string s to file f.
getc(f)	int	Enter a single character from file f.
getchar(void)	int	Enter a single character from the standard input device.
gets(s)	char*	Enter string s from the standard input device.
printf(...)	int	Send data items to the standard output device.
putc(c,f)	int	Send a single character to file f.
putchar(c)	int	Send a single character to the standard output device.
puts(s)	int	Send string s to the standard output device.
rewind(f)	void	Move the pointer to the beginning of file f.
scanf(...)	int	Enter data items from the standard input device.
<stdlib.h>		
abs(i)	int	Return the absolute value of i.
atof(s)	double	Convert string s to a double-precision quantity.
atoi(s)	int	Convert string s to an integer.
atol(s)	long	Convert string s to a long integer.
calloc(u1,u2)	void*	Allocate memory for an array having u1 elements, each of length u2 bytes. Return a pointer to the beginning of the allocated space.
exit(u)	void	Close all files and buffers, and terminate the program. (Value of u is assigned by the function, to indicate termination status).
free(p)	void	Free a block of allocated memory whose beginning is indicated by p.
malloc(u)	void*	Allocate u bytes of memory. Return a pointer to the beginning of the allocated space.
rand(void)	int	Return a random positive integer.
realloc(p, u)	void*	Allocate u bytes of new memory to the pointer variable p. Return a pointer to the beginning of the new memory space.
srand(u)	void	Initialize the random number generator.
system(s)	int	Pass command string s to the operating system. Return 0 if the command is successfully executed, otherwise, return a nonzero value typically –1.

Function	Data type returned	Task
<string.h>		
strcmp(s1, s2)	int	Compare two strings lexicographically. Return a negative value if s1<s2; 0 if s1 and s2 are identical; and a positive value if s1>s2.
strcmpi(s1, s2)	int	Compare two strings lexicographically, without regard to case. Return a negative value if s1<s2; 0 if s1 and s2 are identical; and a value of s1 > s2.
strcpy(s1, s2)	char*	Copy string s2 to string s1.
strlen(s)	int	Return the number of characters in string s.
strset(s, c)	char*	Set all characters within s to c(excluding the terminating null character \0).
<time.h>		
difftime(l1,l2)	double	Return the time difference l1 ~ l2, where l1 and l2 represent elapsed time beyond a designated base time (see the time function).
time(p)	long int	Return the number of seconds elapsed beyond a designated base time.

NOTE: C99 adds many more header files and adds many new functions to the existing header files. For more details, refer to the manual of C99.