```
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";
```

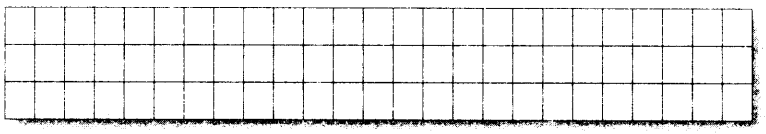
do not work.

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];
```

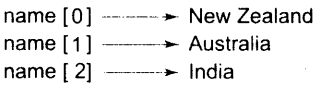
This says that the **name** is a table containing three names, each with a maximum length of 25 characters (including null character). The total storage requirements for the **name** table are 75 bytes.



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

N	e	w		Z	e	a	l	a	n	d	\0
A	u	s	t	r	a	l	i	a	\0		
I	n	d	i	a	\0						

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 noted 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*"

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. When the function is called, the addresses are passed as actual arguments.

The use of pointers to access array elements is very common in C. We have used a pointer to traverse array elements in 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.**

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.
- Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing.
- If we want a called function to change the value of a variable in the calling function, we must pass the address of that variable to the called function.
- When we pass a parameter by address, the corresponding formal parameter must be a pointer variable.
- It is an error to assign a numeric constant to a pointer variable.
- It is an error to assign the address of a variable to a variable of any basic data types.
- It is an error to assign a pointer of one type to a pointer of another type without a cast (with an exception of void pointer).
- A proper understanding of a precedence and associativity rules is very important in pointer applications. For example, expressions like *p++, *p[ ], (*p)[ ], (p).member should be carefully used.
- When an array is passed as an argument to a function, a pointer is actually passed. In the header function, we must declare such arrays with proper size, except the first, which is optional.
- A very common error is to use (or not to use) the address operator (&) and the indirection operator (*) in certain places. Be careful. The compiler may not warn such mistakes.

CASE STUDIES

1. Processing of Examination Marks

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

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

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

The program in Fig. 11.14 stores the student names in the array **name** and the marks in the array **marks.** After computing the total marks obtained by all the students, the program prepares and prints the rank list. The declaration

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

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

```
int (*rowptr)[SUBJECTS+1] = array;
```

Note that **array** is the formal argument whose values are replaced by the values of the actual argument **marks.** The parentheses around ***rowptr** makes the **rowptr** as a pointer to an array of **SUBJECTS+1** integers. Remember, the statement

```
int *rowptr[SUBJECTS+1];
```

would declare **rowptr** as an array of **SUBJECTS+1** elements.

When we increment the **rowptr** (by **rowptr+1**), the incrementing is done in units of the size of each row of **array,** making **rowptr** point to the next row. Since **rowptr** points to a particular row, **(*rowptr)[x]** points to the xth element in the row.

```
Program
  #define  STUDENTS  5
  #define  SUBJECTS  4
  #include <string.h>

  main()
  {
    char name[STUDENTS][20];
    int  marks[STUDENTS][SUBJECTS+1];

    printf("Input students names & their marks in four subjects\n");
    get_list(name, marks, STUDENTS, SUBJECTS);
    get_sum(marks, STUDENTS, SUBJECTS+1);
    printf("\n");
    print_list(name,marks,STUDENTS,SUBJECTS+1);
    get_rank_list(name, marks, STUDENTS, SUBJECTS+1);
```

```
 printf("\nRanked List\n\n");
 print_list(name,marks,STUDENTS,SUBJECTS+1);
}
   /*   Input student name and marks        */
get_list(char *string[ ],
         int array [ ] [SUBJECTS +1], int m, int n)
{
    int   i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++)
    {
       scanf("%s", string[i]);
       for(j = 0; j < SUBJECTS; j++)
          scanf("%d", &(*(rowptr + i))[j]);
    }
}
/*    Compute total marks obtained by each student   */
get_sum(int array [ ] [SUBJECTS +1], int m, int n)
{
    int   i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++)
    {
       (*(rowptr + i))[n-1] = 0;
       for(j =0; j < n-1; j++)
          (*(rowptr + i))[n-1] += (*(rowptr + i))[j];
    }
}


/*    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, then 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.

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.

Chapter

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.

12.3 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, which 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 familar **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);
}
```

Output

```
Input file name
```

```
INVENTORY
Input inventory data

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

Item name  Number   Price   Quantity    Value
AAA-1        111    17.50     115     2012.50
BBB-2        125    36.00      75     2700.00
C-3          247    31.75     104     3302.00
```

Fig. 12.3 *Operations on mixed data types*

12.8 ERROR HANDLING DURING I/O

It is possible that an error may occur during I/O operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Attempting to write to a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output. Fortunately, we have two status-inquiry library functions; **feof** and **ferror** that can help us detect I/O errors in the files.

The **feof** function can be used to test for an end of file condition. It takes a **FILE** pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise. If **fp** is a pointer to file that has just been opened for reading, then the statement

```
if(feof(fp))
  printf("End of data.\n");
```

would display the message "End of data." on reaching the end of file condition.

The **ferror** function reports the status of the file indicated. It also takes a **FILE** pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. The statement

```
if(ferror(fp) != 0)
  printf("An error has occurred.\n");
```

would print the error message, if the reading is not successful.

We know that whenever a file is opened using **fopen** function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not. Example:

```
if(fp == NULL)
  printf("File could not be opened.\n");
```

Example 12.4 Write a program to illustrate error handling in file operations.

The program shown in Fig. 12.4 illustrates the use of the **NULL** pointer test and **feof** function. When we input filename as TETS, the function call

```
fopen("TETS", "r");
```

returns a **NULL** pointer because the file TETS does not exist and therefore the message "Cannot open the file" is printed out.

Similarly, the call **feof(fp2)** returns a non-zero integer when the entire data has been read, and hence the program prints the message "Ran out of data" and terminates further reading.

Program

```
#include  <stdio.h>

main()
{
   char  *filename;
    FILE  *fp1, *fp2;
    int   i, number;

    fp1 = fopen("TEST", "w");
    for(i = 10; i <= 100; i += 10)
       putw(i, fp1);

    fclose(fp1);

    printf("\nInput filename\n");

open_file:
    scanf("%s", filename);

    if((fp2 = fopen(filename,"r")) == NULL)
    {
       printf("Cannot open the file.\n");
       printf("Type filename again.\n\n");
       goto open_file;
    }
    else

    for(i = 1; i <= 20; i++)
```

```
        {  number = getw(fp2);
           if(feof(fp2))
           {
              printf("\nRan out of data.\n");
              break;
           }
           else
              printf("%d\n", number);
        }

        fclose(fp2);
    }
```

Output

```
Input filename
TETS
Cannot open the file.
Type filename again.

TEST
10
20
30
40
50
60
70
80
90
100

Ran out of data.
```

Fig. 12.4 *Illustration of error handling in file operations*

12.6 RANDOM ACCESS TO FILES

So far we have discussed file functions that are useful for reading and writing data sequentially. There are occasions, however, when we are interested in accessing only a particular part of a file and not in reading the other parts. This can be achieved with the help of the functions **fseek, ftell,** and **rewind** available in the I/O library.

ftell takes a file pointer and return a number of type **long,** that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form:

```
n = ftell(fp);
```

n would give the relative offset (in bytes) of the current position. This means that n bytes have already been read (or written).

rewind takes a file pointer and resets the position to the start of the file. For example, the statement

```
rewind(fp);
n = ftell(fp);
```

would assign 0 to n because the file position has been set to the start of the file by **rewind.** Remember, the first byte in the file is numbered as 0, second as 1, and so on. This function helps us in reading a file more than once, without having to close and open the file. Remember that whenever a file is opened for reading or writing, a **rewind** is done implicitly.

fseek function is used to move the file position to a desired location within the file. It takes the following form:

```
fseek(file_ptr, offset, position);
```

file_ptr is a pointer to the file concerned, *offset* is a number or variable of type long, and *position* is an integer number. The *offset* specifies the number of positions (bytes) to be moved from the location specified by *position.* The *position* can take one of the following three values:

Value	*Meaning*
0	Beginning of file
1	Current position
2	End of file

The offset may be positive, meaning move forwards, or negative, meaning move backwards. Examples in Table 12.2 illustrate the operations of the **fseek** function:

Table 12.2 *Operations of fseek Function*

Statement	*Meaning*
fseek(fp,0L,0);	Go to the beginning. (Similar to rewind)
fseek(fp,0L,1);	Stay at the current position. (Rarely used)
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end. (Positions the file to the mth character from the end.)

When the operation is successful, **fseek** returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and **fseek** returns –1 (minus one). It is good practice to check whether an error has occurred or not, before proceeding further.

Example 12.5 Write a program that uses the functions **ftell** and **fseek.**

A program employing **ftell** and **fseek** functions is shown in Fig. 12.5. We have created a file **RANDOM** with the following contents:

```
Position ----> 0   1   2   ... 25
Character
stored ----->  A   B   C   ... Z
```

We are reading the file twice. First, we are reading the content of every fifth position and printing its value along with its position on the screen. The second time, we are reading the contents of the file from the end and printing the same on the screen.

During the first reading, the file pointer crosses the end-of-file mark when the parameter **n** of **fsee(fp,n,0)** becomes 30. Therefore, after printing the content of position 30, the loop is terminated.

For reading the file from the end, we use the statement

```
fseek(fp,-1L,2);
```

to position the file pointer to the last character. Since every read causes the position to move forward by one position, we have to move it back by two positions to read the next character. This is achieved by the function

```
fseek(fp, -2L, 1);
```

in the while statement. This statement also tests whether the file pointer has crossed the file boundary or not. The loop is terminated as soon as it crosses it.

Program

```
#include <stdio.h>
main()
{
    FILE  *fp;
    long  n;
    char c;
    fp = fopen("RANDOM", "w");
    while((c = getchar()) != EOF)
        putc(c,fp);

    printf("No. of characters entered = %ld\n", ftell(fp));
    fclose(fp);
    fp = fopen("RANDOM","r");
    n = 0L;

    while(feof(fp) == 0)
    {
        fseek(fp, n, 0);  /*  Position to (n+1)th character */
        printf("Position of %c is %ld\n", getc(fp),ftell(fp));
        n = n+5L;
    }
    putchar('\n');
```

```
        fseek(fp,-1L,2);      /*  Position to the last character */
          do
          {
              putchar(getc(fp));
          }
          while(!fseek(fp,-2L,1));
          fclose(fp);
    }
Output

    ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
    No. of characters entered = 26
    Position of A is 0
    Position of F is 5
    Position of K is 10
    Position of P is 15
    Position of U is 20
    Position of Z is 25
    Position of   is 30

    ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Fig. 12.5 *Illustration of* ***fseek*** *and* ***ftell*** *functions*

Example 12.6 Write a program to append additional items to the file INVENTORY created in Example 12.3 and print the total contents of the file.

The program is shown in Fig. 12.6. It uses a structure definition to describe each item and a function **append()** to add an item to the file.

On execution, the program requests for the filename to which data is to be appended. After appending the items, the position of the last character in the file is assigned to **n** and then the file is closed.

The file is reopened for reading and its contents are displayed. Note that reading and displaying are done under the control of a **while** loop. The loop tests the current file position against n and is terminated when they become equal.

```
Program
    #include  <stdio.h>

    struct invent_record
    {
        char   name[10];
        int    number;
        float  price;
        int    quantity;
    };
```

```
main()
{
    struct invent_record item;
    char  filename[10];
    int   response;
    FILE  *fp;
    long  n;
    void append (struct invent_record *x, file *y);
    printf("Type filename:");
    scanf("%s", filename);

    fp = fopen(filename, "a+");
    do
    {
       append(&item, fp);
       printf("\nItem %s appended.\n",item.name);
       printf("\nDo you want to add another item\
           (1 for YES /0 for NO)?");
       scanf("%d", &response);
    }  while (response == 1);

    n = ftell(fp);       /* Position of last character  */
    fclose(fp);

    fp = fopen(filename, "r");

    while(ftell(fp) < n)
    {
       fscanf(fp,"%s %d %f %d",
       item.name, &item.number, &item.price, &item.quantity);
       fprintf(stdout,"%-8s %7d %8.2f %8d\n",
       item.name, item.number, item.price, item.quantity);
    }
    fclose(fp);
}
void append(struct invent_record *product, File *ptr)
{
    printf("Item name:");
    scanf("%s", product->name);
    printf("Item number:");
    scanf("%d", &product->number);
    printf("Item price:");
    scanf("%f", &product->price);
    printf("Quantity:");
    scanf("%d", &product->quantity);
    fprintf(ptr, "%s %d %.2f %d",
```

```
                    product->name,
                    product->number,
                    product->price,
                    product->quantity);
    }

Output
    Type filename:INVENTORY
    Item name:XXX
    Item number:444
    Item price:40.50
    Quantity:34

    Item XXX appended.

    Do you want to add another item(1 for YES /0 for NO)?1

    Item name:YYY
    Item number:555
    Item price:50.50
    Quantity:45

    Item YYY appended.

    Do you want to add another item(1 for YES /0 for NO)?0
    AAA-1        111    17.50      115
    BBB-2        125    36.00       75
    C-3          247    31.75      104
    XXX          444    40.50       34
    YYY          555    50.50       45
```

Fig. 12.6 *Adding items to an existing file*

12.7 COMMAND LINE ARGUMENTS

What is a command line argument? It is a parameter supplied to a program when the program is invoked. This parameter may represent a filename the program should process. For example, if we want to execute a program to copy the contents of a file named **X_FILE** to another one named Y_FILE, then we may use a command line like

C > PROGRAM X_FILE Y_FILE

where **PROGRAM** is the filename where the executable code of the program is stored. This eliminates the need for the program to request the user to enter the filenames during execution. How do these parameters get into the program?

We know that every C program should have one **main** function and that it marks the beginning of the program. But what we have not mentioned so far is that it can also take arguments like other functions. In fact **main** can take two arguments called **argc** and **argv** and the information contained in the command line is passed on to the program through these arguments, when **main** is called up by the system.

The variable **argc** is an argument counter that counts the number of arguments on the command line. The **argv** is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of **argc**. For instance, for the command line given above, **argc** is three and **argv** is an array of three pointers to strings as shown below:

argv[0] -> PROGRAM
argv[1] -> X_FILE
argv[2] -> Y_FILE

In order to access the command line arguments, we must declare the main function and its parameters as follows:

```
main(int argc, char *argv[])
{
   .....
   .....
}
```

The first parameter in the command line is always the program name and therefore **argv[0]** always represents the program name.

Example 12.7 Write a program that will receive a filename and a line of text as command line arguments and write the text to the file.

Figure 12.7 shows the use of command line arguments. The command line is

F12_7 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG

Each word in the command line is an argument to the **main** and therefore the total number of arguments is 9.

The argument vector **argv[1]** points to the string TEXT and therefore the statement

```
fp = fopen(argv[1], "w");
```

opens a file with the name TEXT. The **for** loop that follows immediately writes the remaining 7 arguments to the file TEXT.

Program

```
#include  <stdio.h>

main(int argc, char *argv[])
{
    FILE   *fp;
    int  i;
    char word[15];
```

```
        fp = fopen(argv[1], "w"); /* open file with name argv[1] */
        printf("\nNo. of arguments in Command line = %d\n\n",argc);
        for(i = 2; i < argc; i++)
           fprintf(fp,"%s ", argv[i]); /* write to file argv[1]  */
        fclose(fp);

    /*  Writing content of the file to screen                    */

        printf("Contents of %s file\n\n", argv[1]);
        fp = fopen(argv[1], "r");
        for(i = 2; i < argc; i++)
        {
           fscanf(fp,"%s", word);
           printf("%s ", word);
        }

        fclose(fp);
        printf("\n\n");

    /*  Writing the arguments from memory */

        for(i = 0; i < argc; i++)
           printf("%*s \n", i*5,argv[i]);
    }
```

Output

```
C>F12_7 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGG

No. of arguments in Command line = 9

Contents of TEXT file

AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG

C:\C\F12_7.EXE
 TEXT
    AAAAAA
         BBBBBB
              CCCCCC
                   DDDDDD
                        EEEEEE
                             FFFFFF
                                  GGGGGG
```

Fig. 12.7 *Use of command line arguments*

Just Remember

- Do not try to use a file before opening it.
- Remember, when an existing file is open using 'w' mode, the contents of file are deleted.
- When a file is used for both reading and writing, we must open it in 'w+' mode.
- EOF is integer type with a value –1. Therefore, we must use an integer variable to test EOF.
- It is an error to omit the file pointer when using a file function.
- It is an error to open a file for reading when it does not exist.
- It is an error to try to read from a file that is in write mode and vice versa.
- It is an error to attempt to place the file marker before the first byte of a file.
- It is an error to access a file with its name rather than its file pointer.
- It is a good practice to close all files before terminating a program.

REVIEW QUESTIONS

12.1 State whether the following statements are *true* or *false*.
  (a) A file must be opened before it can be used.
  (b) All files must be explicitly closed.
  (c) Files are always referred to by name in C programs.
  (d) Using **fseek** to position a file beyond the end of the file is an error.
  (e) Function **fseek** may be used to seek from the beginning of the file only.

12.2 Fill in the blanks in the following statements.
  (a) The mode _____ is used for opening a file for updating.
  (b) The function ___ may be used to position a file at the beginning.
  (c) The function ____gives the current position in the file.
  (d) The function _____ is used to write data to randomly accessed file.

12.3 Describe the use and limitations of the functions **getc** and **putc.**

12.4 What is the significance of EOF?

12.5 When a program is terminated, all the files used by it are automatically closed. Why is it then necessary to close a file during execution of the program?

12.6 Distinguish between the following functions:
  (a) getc and getchar
  (b) printf and fprintf
  (c) feof and ferror

12.7 How does an append mode differ from a write mode?

12.8 What are the common uses of **rewind** and **ftell** functions?

12.9 Explain the general format of **fseek** function?

12.10 What is the difference between the statements **rewind(fp);** and **fseek(fp,0L,0);?**

PROGRAMMING EXERCISES

12.1 Write a program to copy the contents of one file into another.

12.2 Two files DATA1 and DATA2 contain sorted lists of integers. Write a program to produce a third file DATA which holds a single sorted, merged list of these two lists. Use command line arguments to specify the file names.

12.3 Write a program that compares two files and returns 0 if they are equal and 1 is they are not.

12.4 Write a program that appends one file at the end of another.

12.5 Write a program that reads a file containing integers and appends at its end the sum of all the integers.

Chapter

13

Dynamic Memory Allocation and Linked Lists

13.1 INTRODUCTION

Most often we face situations in programming where the data is dynamic in nature. That is, the number of data items keep changing during execution of the program. For example, consider a program for processing the list of customers of a corporation. The list grows when names are added and shrinks when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items. Such situations can be handled more easily and effectively by using what is known as *dynamic data structures* in conjunction with *dynamic memory management* techniques.

Dynamic data structures provide flexibility in adding, deleting or rearranging data items at run time. Dynamic memory management techniques permit us to allocate additional memory space or to release unwanted space at run time, thus, optimizing the use of storage space. This chapter discusses the concept of *linked lists*, one of the basic types of dynamic data structures. Before we take up linked lists, we shall briefly introduce the dynamic storage management functions that are available in C. These functions would be extensively used in processing linked lists.

13.2 DYNAMIC MEMORY ALLOCATION

C language requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgment of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is known as *dynamic memory allocation*. Although C does not inherently have this facility, there are four library routines known as "memory

management functions" that can be used for allocating and freeing memory during program execution. They are listed in Table 13.1. These functions help us build complex application programs that use the available memory intelligently.

Table 13.1 *Memory Allocation Functions*

Function	*Task*
malloc	Allocates request size of bytes and returns a pointer to the first byte of the allocated space
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
free	Frees previously allocated space
realloc	Modifies the size of previously allocated space

Memory Allocation Process

Before we discuss these functions, let us look at the memory allocation process associated with a C program. Figure 13.1 shows the conceptual view of storage of a C program in memory.

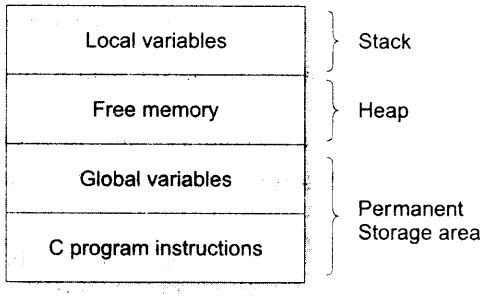


Fig. 13.1 *Storage of a C program*

The program instructions and global and static variables are stored in a region known as *permanent storage area* and the local variables are stored in another area called *stack*. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. This free memory region is called the *heap*. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory "overflow" during dynamic allocation process. In such situations, the memory allocation functions mentioned above return a NULL pointer (when they fail to locate enough memory requested).

13.3 ALLOCATING A BLOCK OF MEMORY: MALLOC

A block of memory may be allocated using the function **malloc**. The **malloc** function reserves a block of memory of specified size and returns a pointer of type **void**. This means that we can assign it to any type of pointer. It takes the following form:

```
ptr = (cast-type *) malloc(byte-size);
```

ptr is a pointer of type *cast-type*. The **malloc** returns a pointer (of *cast type*) to an area of memory with size *byte-size*.
Example:

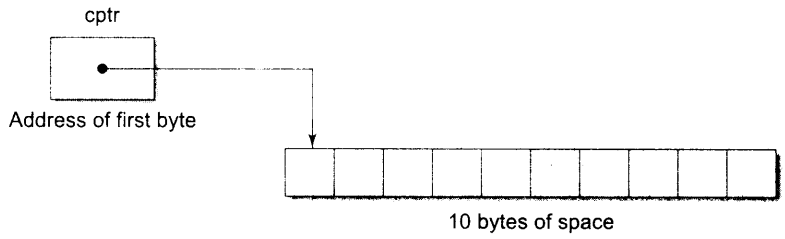
```
x = (int *) malloc (100 *sizeof(int));
```

On successful execution of this statement, a memory space equivalent to "100 times the size of an **int**" bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer **x** of type of **int**.

Similarly, the statement

```
cptr = (char*) malloc(10);
```

allocates 10 bytes of space for the pointer **cptr** of type **char**. This is illustrated as:



Note that the storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer.

We may also use **malloc** to allocate space for complex data types such as structures. Example:

```
st_var = (struct store *)malloc(sizeof(struct store));
```

where **st_var** is a pointer of type **struct store**

Remember, the **malloc** allocates a block of contiguous bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it fails, it returns a NULL. We should therefore check whether the allocation is successful before using the memory pointer. This is illustrated in the program in Fig.13.2.

Example 13.1 Write a program that uses a table of integers whose size will be specified interactively at run time.

The program is given in Fig.13.2. It tests for availability of memory space of required size. If it is available, then the required space is allocated and the address of the first byte of the space allocated is displayed. The program also illustrates the use of pointer variable for storing and accessing the table values.

Program

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

```
main()
{
            int *p, *table;
            int size;
            printf("\nWhat is the size of table?");
            scanf("%d",size);
            printf("\n")
              /*------------Memory allocation --------------*/
            if((table = (int*)malloc(size *sizeof(int))) == NULL)
              {
                printf("No space available \n");
                exit(1);
              }
            printf("\n Address of the first byte is  %u\n", table);
            /* Reading table values*/
            printf("\nInput table values\n");

            for (p=table; p<table + size; p++)
                scanf("%d",p);
            /* Printing table values in reverse order*/
            for (p = table + size -1; p >= table; p --)
              printf("%d is stored at address %u \n",*p,p);

}
```

Output

```
What is the size of the table? 5
Address of the first byte is 2262
Input table values
11 12 13 14 15
15 is stored at address 2270
14 is stored at address 2268
13 is stored at address 2266
12 is stored at address 2264
11 is stored at address 2262
```

Fig. 13.2 *Memory allocation with* ***malloc***

13.4 ALLOCATING MULTIPLE BLOCKS OF MEMORY: CALLOC

calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While **malloc** allocates a single block of storage space, **calloc** allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of **calloc** is:

```
ptr = (cast-type *) calloc (n, elem-size);
```

The above statement allocates contiguous space for *n* blocks, each of size *elem-size* bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

The following segment of a program allocates space for a structure variable:

```
. . . .
. . . .
struct student
{
  char name[25];
  float age;
  long int id_num;
};
typedef struct student record;
record *st_ptr;
int class_size = 30;
st_ptr=(record *)calloc(class_size, sizeof(record));
. . . .
. . . .
```

record is of type **struct student** having three members: **name, age** and **id_num**. The **calloc** allocates memory to hold data for 30 such records. We must be sure that the requested memory has been allocated successfully before using the **st_ptr**. This may be done as follows:

```
if(st_ptr == NULL)
{
  printf("Available memory not sufficient");
  exit(1);
}
```

13.5 RELEASING THE USED SPACE: FREE

Compile-time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited.

When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the **free** function:

```
free (ptr);
```

ptr is a pointer to a memory block which has already been created by **malloc** or **calloc**. Use of an invalid pointer in the call may create problems and cause system crash. We should remember two things here:

1. It is not the pointer that is being released but rather what it points to.
2. To release an array of memory that was allocated by **calloc** we need only to release the pointer once. It is an error to attempt to release elements individually.

The use of **free** function has been illustrated in Example 13.2.

13.6 ALTERING THE SIZE OF A BLOCK: REALLOC

It is likely that we discover later, the previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much larger than necessary and we want to reduce it. In both the cases, we can change the memory size already allocated with the help of the function **realloc**. This process is called the *reallocation* of memory. For example, if the original allocation is done by the statement

```
ptr = malloc(size);
```

then reallocation of space may be done by the statement

```
ptr = realloc(ptr, newsize);
```

This function allocates a new memory space of size *newsize* to the pointer variable **ptr** and returns a pointer to the first byte of the new memory block. The *newsize* may be larger or smaller than the *size*. Remember, the new memory block may or may not begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and move the contents of the old block into the new block. The function guarantees that the old data will remain intact.

If the function is unsuccessful in locating additional space, it returns a NULL pointer and the original block is freed (lost). This implies that it is necessary to test the success of operation before proceeding further. This is illustrated in the program of Example 13.2.

Example 13.2 Write a program to store a character string in a block of memory space created by **malloc** and then modify the same to store a larger string.

The program is shown in Fig. 13.3. The output illustrates that the original buffer size obtained is modified to contain a larger string. Note that the original contents of the buffer remains same even after modification of the original size.

Program

```
#include <stdio.h>
#include<stdlib.h>
#define NULL 0
main()
{
        char *buffer;
        /* Allocating memory */
        if((buffer = (char *)malloc(10)) == NULL)
        {
```

```
            printf("malloc failed.\n");
            exit(1);
          }
          printf("Buffer of size %d created \n",_msize(buffer));
          strcpy(buffer, "HYDERABAD");
          printf("\nBuffer contains: %s \n ", buffer);
          /* Reallocation */
          if((buffer = (char *)realloc(buffer, 15)) == NULL)
          {
            printf("Reallocation failed. \n");
            exit(1);
          }
          printf("\nBuffer size modified. \n");
          printf("\nBuffer still contains: %s \n",buffer);
          strcpy(buffer, "SECUNDERABAD");
          printf("\nBuffer now contains: %s \n",buffer);
  /* Freeing memory */
  free(buffer);
   }
```

Output

```
Buffer of size 10 created
Buffer contains: HYDERABAD
Buffer size modified
Buffer still contains: HYDERABAD
Buffer now contains: SECUNDERABAD
```

Fig. 13.3 *Reallocation and release of memory space*

13.7 CONCEPTS OF LINKED LISTS

We know that a list refers to a set of items organized sequentially. An array is an example of list. In an array, the sequential organization is provided implicitly by its index. We use the index for accessing and manipulation of array elements. One major problem with the arrays is that the size of an array must be specified precisely at the beginning. As pointed out earlier, this may be a difficult task in many real-life applications.

A completely different way to represent a list is to make each item in the list part of a structure that also contains a "link" to the structure containing the next item, as shown in Fig. 13.4. This type of list is called a *linked list* because it is a list whose order is given by links from one item to the next.

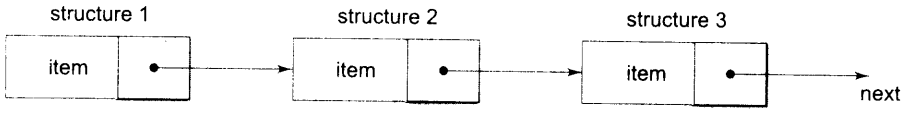
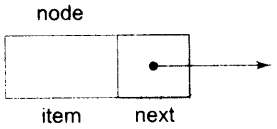


Fig. 13.4 *A linked list*

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**.

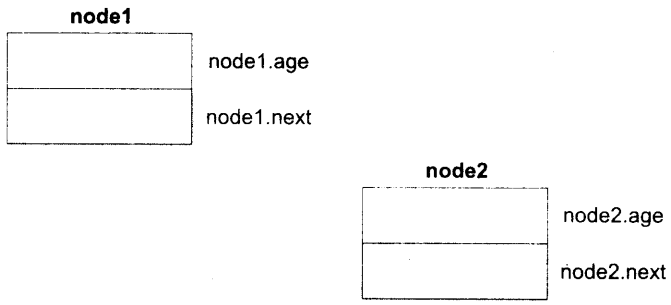
Let use consider a simple example to illustrate the concept of linking. Suppose we define a structure as follows:

```
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;
```

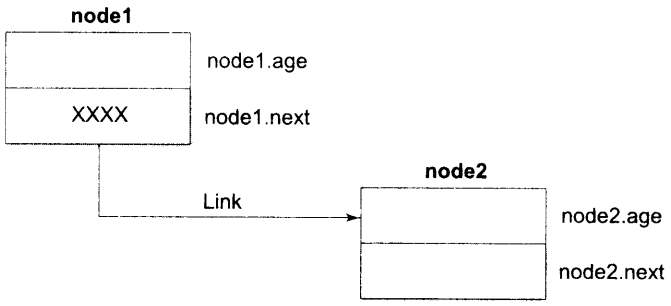
This statement creates space for two nodes each containing two empty fields as shown:



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

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

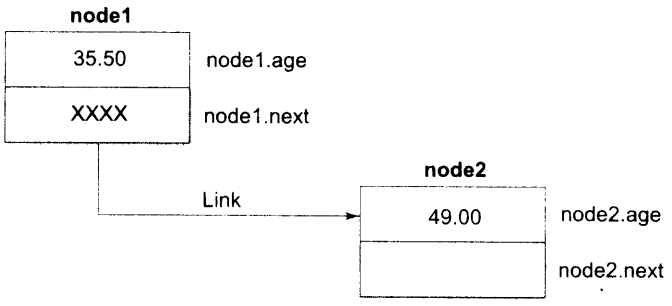
This statement stores the address of **node2** into the field **node1.next** and thus establishes a "link" between **node1** and **node2** as shown:



"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:

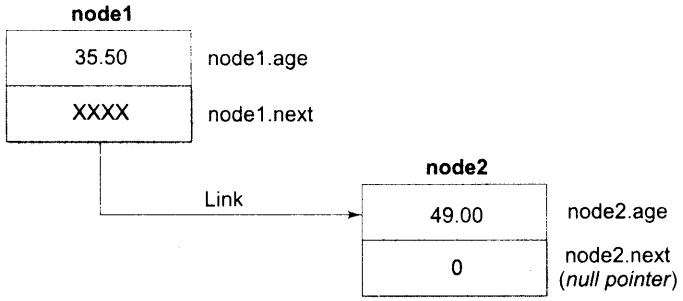
```
node2.next = &node3;
```

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 more 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.