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

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

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

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

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

Rules for Initializing Structures

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

1. We cannot initialize individual members inside the structure template.
2. The order of values enclosed in braces must match the order of members in the structure definition.
3. It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values as follows:
   - Zero for integer and floating point numbers.
   - '\0' for characters and strings.

10.6 COPYING AND COMPARING STRUCTURE VARIABLES

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

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

However, the statements such as

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

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

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

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

Program

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

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

    student3 = student2;

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

    if(x == 1)
    {
        printf("\nstudent2 and student3 are same\n\n");
        printf("%d %s %f\n", student3.number,
                             student3.name,
                             student3.marks);
    }
    else
        printf("\nstudent2 and student3 are different\n\n");

}
```

Output

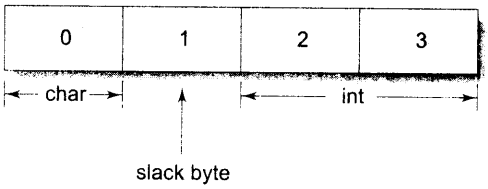
```
student2 and student3 are same

222 Reddy 67.000000
```

Fig. 10.2 *Comparing and copying structure variables*

Word Boundaries and Slack Bytes

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



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

10.7 OPERATIONS ON INDIVIDUAL MEMBERS

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

```
if (student1.number == 111)
        student1.marks += 10.00;
float sum = student1.marks + student2.marks;
student2.marks * = 0.5;
```

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

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

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

Three Ways to Access Members

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

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

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

The identifier **ptr** is known as **pointer** that has been assigned the address of the structure variable v. 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.

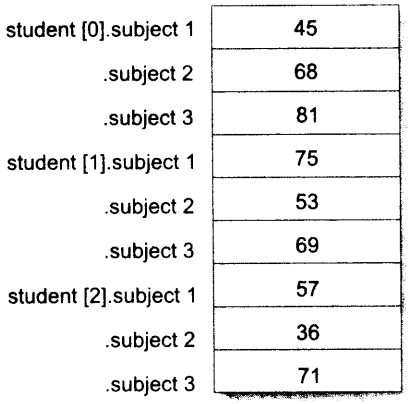


Fig. 10.3 *The array student inside memory*

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**.

```
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- 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

    SUBJECT          TOTAL
    Subject-1         177
    Subject-2         156
    Subject-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.

10.11 STRUCTURES AND FUNCTIONS

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

1. The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.
2. The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function). It is,

therefore, necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.

3. The third approach employs a concept called *pointers* to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to function. This method is more efficient as compared to the second one.

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

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

```
function_name (structure_variable_name);
```

The called function takes the following form:

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

The following points are important to note:

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

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

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.

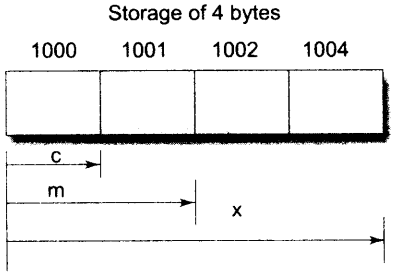


Fig. 10.7 *Sharing of a storage locating by union members*

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

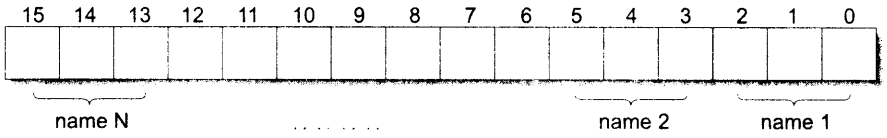
So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases. we waste memory space. Fortunately, C permits us to use small bit fields to hold data items and thereby to pack several data items in a word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

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

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



There are several specific points to observe:

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

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

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

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

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

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

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

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

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

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

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

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

are valid statements.

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

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

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

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

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

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

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 STUDY

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 structures of type **struct item_bank?**

(a) `int item_bank items[10];`
(b) `struct items[10] item_bank;`
(c) `struct 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) . . .`

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 of players with their batting average.

10.15 Design 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.

Chapter 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.

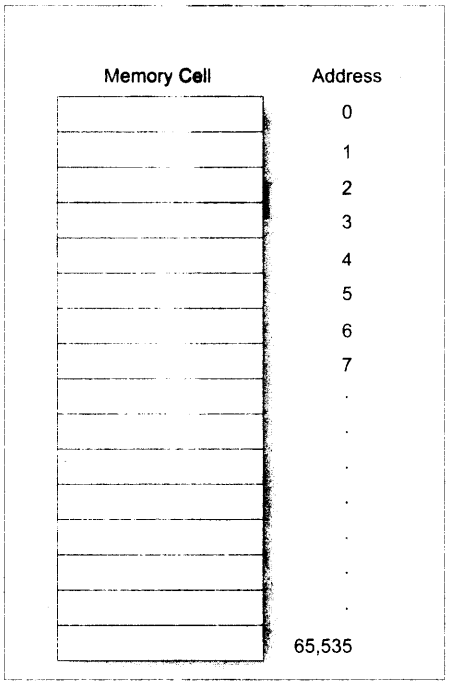


Fig. 11.1 *Memory organisation*

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.)

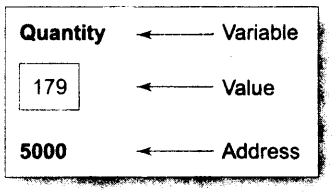


Fig. 11.2 *Representation of a variable*

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, which 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.

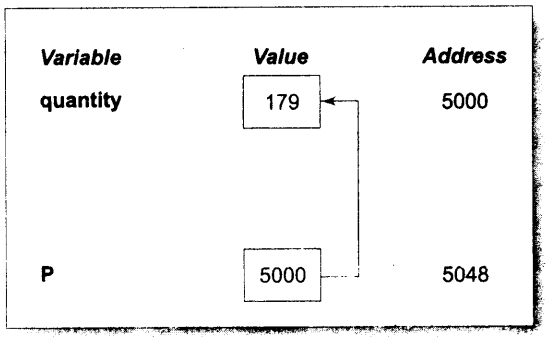
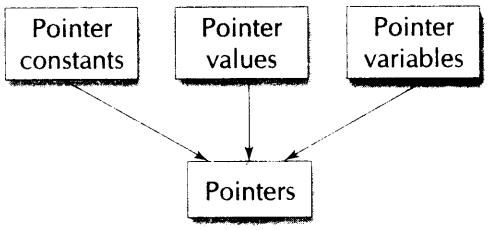


Fig. 11.3 *Pointer variable*

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 every-time we run the program. What we are concerned about is the relationship between the variables **p** and **quantity**.)

Underlying Concepts of Pointers

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



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

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

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

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. Notice 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 */
```

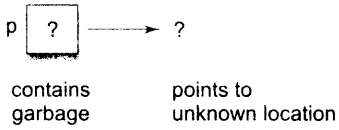
declares the variable **p** as a pointer variable that points to an integer data type. Remember that the type **int** refers to the data type of the variable being pointed to by **p** and not the type of the value of the pointer. Similarly, the statement

```
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 similar to 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;
```

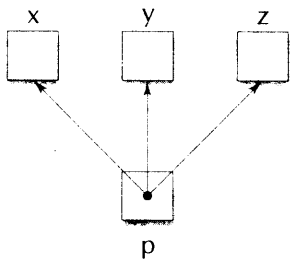
With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

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

Pointer Flexibility

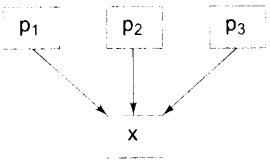
Pointers are flexible. We can make the same pointer to point to different data variables in different statements. Example;

```
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;
.....
.....
```



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*.

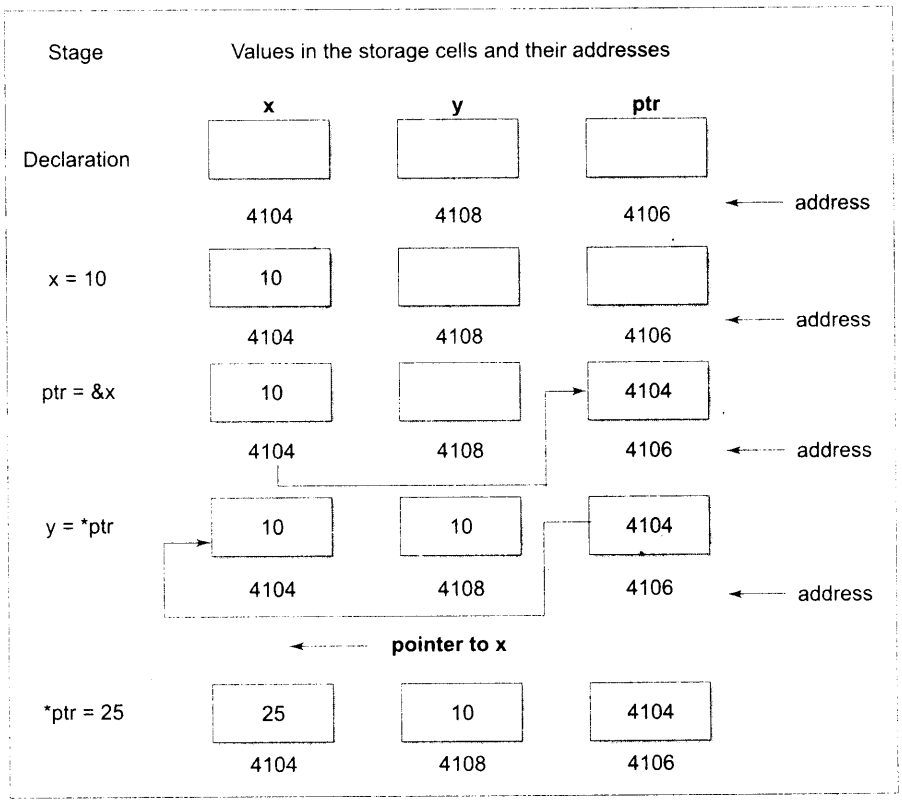
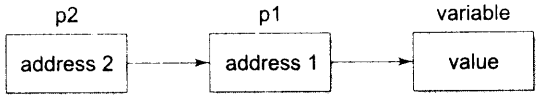


Fig. 11.6 *Illustration of pointer assignments*

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.

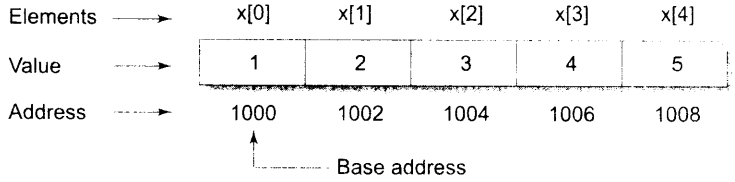
1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
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).

11.10 POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array **x** as follows:

```
int x[5] = {1, 2, 3, 4, 5};
```

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



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

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

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

```
p = x;
```

This is equivalent to

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

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

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

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

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

= 1000 + (3 x 2) = 1006

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

The example 11.4 illustrates the use of pointer accessing method.

Example 11.4 Write a program using pointers to compute the sum of all elements stored in an array.

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

Program

```
main()
{
    int *p, sum, i;
    int x[5] = {5,9,6,3,7};
    i  = 0;
    p  = x;       /* initializing with base address of x */
    printf("Element   Value   Address\n\n");
    while(i < 5)
    {
       printf(" x[%d] %d %u\n", i, *p, p);
       sum = sum + *p;   /* accessing array element  */
       i++, p++;         /* incrementing pointer     */
    }
    printf("\n  Sum    =  %d\n", sum);
    printf("\n  &x[0]  =  %u\n", &x[0]);
    printf("\n  p      =  %u\n", p);
}
```

Output

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

Fig. 11.8 *Accessing one-dimensional array elements using the pointer*

It is possible to avoid the loop control variable **i** as shown:

```
.....
p = x;
while(p <= &x[4])
{
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)
```

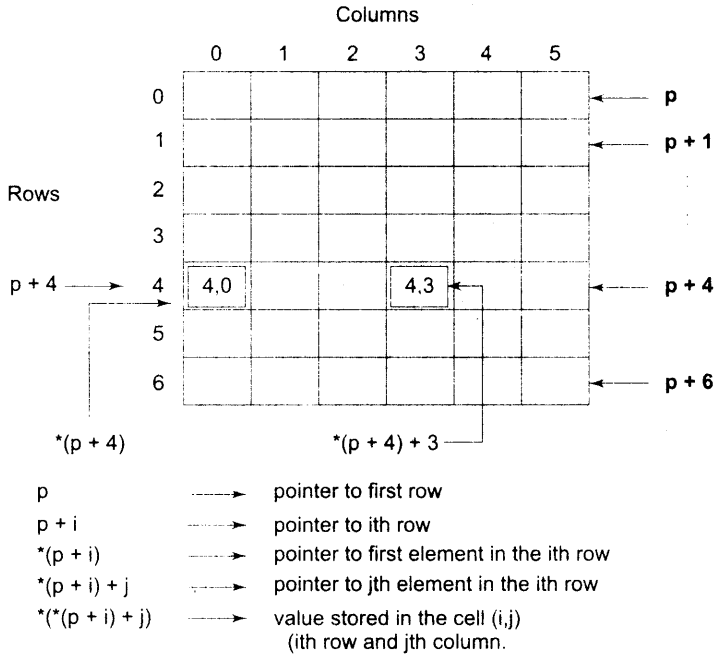
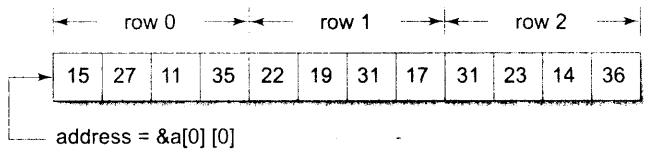


Fig. 11.9 *Pointers to two-dimensional arrays*

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:

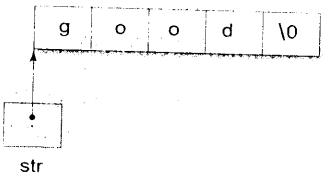
```
char str [5] = "good";
```

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.

Example 11.5 Write a 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

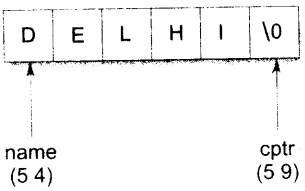
```
while(*cptr != '\0')
```

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
```