Merge Sort

Merging is the process of combining two sorted lists into one sorted list.

⇒ "Merge-Sort is based on "Divide and Conquer" philosophy. Split the unsorted array A[lower, upper] around its middle element, "mid" where mid = (lower+upper)/2. The two unsorted arrays are, A[lower : mid] and A[mid+1 : upper]. These two sorted sub-arrays are merged. The process of dividing the original array continues till there is only one element in the array in which case, the array is sorted.

Eg:→ 24 28 15 13 18 34 19 06

lower = 1, upper = 8, mid = 4

∴, two unsorted sublists are:

(24 28 15 13) (18 34 19 06)

sort (24,28 13,15) (18,34, 06, 19)

merge merge

(13, 15, 24, 28) (06, 18, 19, 34)

merged

06, 13, 15, 18, 19, 24, 28, 34.

Pgm:- Merge Sort implementation using array in ascending order in C programming language.

```
#include<stdio.h>
#define Max 50
void mergesort(int arr[], int low, int
                mid, int high);
void partition(int arr[], int low, int high);
int main()
{
    int merge[MAX], i, n;
printf("Enter the total number of elements:");
scanf("%d", &n);
printf("Enter the elements which is to be
                sort:");
for(i=0; i<n; i++)
{
  scanf("%d", &merge[i]);
}
  Partition(merge, 0, n-1);
printf("After merge sorting elements
                are:");
for(i=0; i<n; i++)
{
  printf("%d", merge[i]);
}
  return 0;
}
```

```
void partition (int arr[], int low, int high)
{
    int mid;
    if (low < high)
    {
        mid = (low+high)/2;
        partition (arr, low, mid);
        partition (arr, mid+1, high);
        mergesort (arr, low, mid, high);
    }
}
void mergesort (int arr[], int low, int
                mid, int high)
{
    int i, m, k, l, temp[MAX];
    l = low;
    i = low;
    m = mid+1;
    while ((l <= mid) && (m <= high))
    {
        if (arr[l] <= arr[m])
        {
            temp[i] = arr[l];
            l++;
        }
        else
        {
            temp[i] = arr[m];
            m++;
        }
        i++;
    }
```

```
    if (l>mid)
    {
      for (k=m; k<=high; k++)
      {
        temp[i] = arr[k];
        i++;
      }
    }
    else
    {
      for (k=l; k<=mid; k++)
      {
        temp[i] = arr[k];
        i++;
      }
    }
      for (k= low; k<=high; k++)
      {
        arr[k] = temp[k];
      }
}
```

Sample Output :-

Enter total no of elements : 5
Enter the elements which to be sort :
2 5 0 9 1
After merge sorting elements are : 0 1 2 5 9

Complexity → The function "merge" runs in $O(n+m)$ time if the first and the second list contains "n" and "m" elements respectively. However, this operation requires a space for temporarily storing all $(n+m)$ elements in one single array. If $T(n)$ denotes the time required to sort "n" elements using the function "mergesort", then the following recursive relation is valid for $T(n)$.

$T(n) = T(n/2) + T(n/2) + O(n)$ where, $n = 2^m$ for some "m".

$T(n) = T(n/2) + T(n/2) + k \cdot n$ for constant k

$= 2T(n/2) + k \cdot n$

$= 2(2T(n/4) + k(n/2)) + k \cdot n$

$= 2^2 T(n/2^2) + nk(1+1)$

$= 2^m T(n/2^m) + nk(1+1+1+\dots+1)$

$= nT(1) + nk \cdot m$

$= n + kn \log n$

$= n + k(n \log n)$

$= O(n \log n)$

Disadvantage → requires a large amount of space for temporarily storing sublists and to store the sorted

list with merged elements.

Bottom-up merge Sorting:- This technique is a non-recursive method of sorting elements in bottom-up manner.

Run:- The term "run" is used to denote a sorted segment within an array. The length of a "run" is the number of elements in the "run".

⇒ The first pass merge pairs of runs to create runs of length "2". For this purpose, first and second runs are taken as a pair to be merged.

⇒ Similarly, third and fourth runs are merged and so — on.

At the end of the first pass, $(n/2)$ runs of length "2" are created where "n" is the number of elements in the given array. The second pass consists of merging pairs of runs created in the first pass. The pairs are selected for merging in the same manner as in the case of the first pass and at the end of the second pass, $(n/4)$ runs each of length "4" are created.

Eg:-

* When there are odd no. of runs to be merged in a pair, merge pairs of runs as usual until the last run is reached.

Original array	After first Pass	After second Pass	After third Pass	After fourth Pass
14 39 16 8 34 46 37 53 49 3 42 35 61 89 29 57	① Run 14 39 / ② Run 8 16 / ③ Run 34 46 / ④ 37 53 / ⑤ 3 49 / ⑥ 35 42 / ⑦ 61 89 / ⑧ 29 57 / 8 Runs	8 14 16 39 / 34 37 46 53 / 3 35 42 49 / 29 57 61 89 / 4 Runs	8 14 16 34 37 39 46 53 / 3 29 35 42 49 57 61 89 / 2 Runs	3 8 14 16 29 34 35 37 39 42 46 49 53 57 61 89 / 1 Run = Sorted list

The

Binary Searching

Binary search method can be used only for sorted lists. In this method, the value of the element in the middle of the list is compared with the value of the element to be searched for.

⇒ If the middle element is large than the ~~Key~~ value "A(mid)"

A(mid) > Key value.

Then, the Key element is to be in the upper half of the list, otherwise, if A(mid) < key value then, it is in first half.

Eg:-

2, 5, 8, 11, 14, 15, 18, 21, 24, 29, 31

(a) 1st case:- If key = 8 then mid = $\frac{1+11}{2}$

⇒ MID = 6

S. No	First	Last	MID	Comparison	Remark
1	1	11	6	15 > 8 i.e 8 < 15	MID-1 = 5
2	1	5	3	A(3) = 8	Element found at Pos = 3

2) Assume that we want to search the element = 29

S.No	First	Last	MID	Comparison	Remark
1.	1	11	6	A[6] < 29	Change the position MID+1 = 7
2.	7	11	9	A[9] < 29	Change first position MID+1 = 10
3.	10	11	10	A[10] = 29	Element found at Position = 10

Complexity:-

Binary search is better than sequential (linear) search because in binary search, average number of comparisons are $O(\log_2 N)$ and average number of comparisons in sequential search is $(n+1)/2$.

Algorithm:-

Binary_Search (A, N, X)

Here A is an array with N elements i.e A[1:N]. Let X is an element (key) to be searched for. If the search is successful, this algorithm finds the location or position of X in the array otherwise the value 0 is returned.

Step 1:- [Initialize variables]
FIRST = 1 and LAST = N

MID = INT((FIRST + LAST)/2)

Step 2:→ Repeat step 3 and 4 while FIRST ≤ LAST and A[MID] <> X.

Step 3:→ If X < A[MID] then
LAST = MID - 1
Else
FIRST = MID + 1
[End of If statement]

Step 4:→ MID = INT((FIRST + LAST)/2)
[End of step 2 loop]

Step 5:→ [Result]
If A[MID] = X then
Return (MID)
Else
Return (0)
[End of If statement]

Step 6:→ Exit.

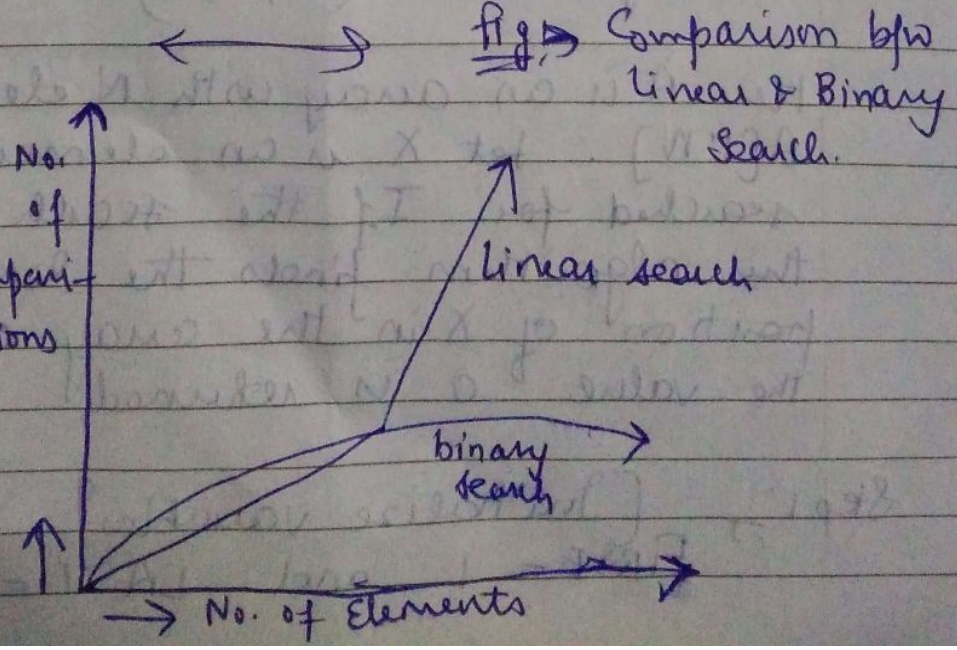


Fig:→ Comparison b/w Linear & Binary Search.

merge sort implementation using array in ascending order in c programming language

```
#include<stdio.h>
#define MAX 50

void mergeSort(int arr[],int low,int mid,int high);
void partition(int arr[],int low,int high);

int main(){

   int merge[MAX],i,n;

   printf("Enter the total number of elements: ");
   scanf("%d",&n);

   printf("Enter the elements which to be sort: ");
   for(i=0;i<n;i++){
      scanf("%d",&merge[i]);
   }

   partition(merge,0,n-1);

   printf("After merge sorting elements are: ");
   for(i=0;i<n;i++){
      printf("%d ",merge[i]);
   }

   return 0;
}

void partition(int arr[],int low,int high){

   int mid;

   if(low<high){
      mid=(low+high)/2;
      partition(arr,low,mid);
      partition(arr,mid+1,high);
      mergeSort(arr,low,mid,high);
   }
}

void mergeSort(int arr[],int low,int mid,int high){

   int i,m,k,l,temp[MAX];

   l=low;
   i=low;
   m=mid+1;
```

Source code of simple bubble sort implementation using array ascending order in c programming language

```
#include<stdio.h>
int main(){

  int s,temp,i,j,a[20];

  printf("Enter total numbers of elements: ");
  scanf("%d",&s);

  printf("Enter %d elements: ",s);
  for(i=0;i<s;i++)
     scanf("%d",&a[i]);

  //Bubble sorting algorithm
  for(i=s-2;i>=0;i--){
     for(j=0;j<=i;j++){
        if(a[j]>a[j+1]){
           temp=a[j];
           a[j]=a[j+1];
           a[j+1]=temp;
        }
     }
  }

  printf("After sorting: ");
  for(i=0;i<s;i++)
     printf(" %d",a[i]);

  return 0;
}
```

Output:
Enter total numbers ~~of elements~~: 5
Enter 5 elements: 6 2 0 11 9
After sorting: 0 2 6 9 11

```
  while((l<=mid)&&(m<=high)){

    if(arr[l]<=arr[m]){
      temp[i]=arr[l];
      l++;
    }
    else{
      temp[i]=arr[m];
      m++;
    }
    i++;
  }

  if(l>mid){
    for(k=m;k<=high;k++){
      temp[i]=arr[k];
      i++;
    }
  }
  else{
    for(k=l;k<=mid;k++){
      temp[i]=arr[k];
      i++;
    }
  }

  for(k=low;k<=high;k++){
    arr[k]=temp[k];
  }
}
```

Sample output:

```
Enter the total number of elements: 5
Enter the elements which to be sort: 2 5 0 9 1
After merge sorting elements are: 0 1 2 5 9
```

Source code of simple insertion sort implementation using array in ascending order in c programming language

```
#include<stdio.h>
int main(){

  int i,j,s,temp,a[20];

  printf("Enter total elements: ");
  scanf("%d",&s);
```

Source code of simple Selection sort implementation using array ascending order in c programming language

```
#include<stdio.h>
int main(){

  int s,i,j,temp,a[20];

  printf("Enter total elements: ");
  scanf("%d",&s);
```

```
printf("Enter %d elements: ",s);
for(i=0;i<s;i++)
  scanf("%d",&a[i]);

for(i=1;i<s;i++){
  temp=a[i];
  j=i-1;
  while((temp<a[j])&&(j>=0)){
    a[j+1]=a[j];
    j=j-1;
  }
  a[j+1]=temp;
}

printf("After sorting: ");
for(i=0;i<s;i++)
  printf(" %d",a[i]);

return 0;
}
```

Output:
Enter total elements: 5
Enter 5 elements: 3 7 9 0 2
After sorting: 0 2 3 7 9

```
printf("Enter %d elements: ",s);
for(i=0;i<s;i++)
  scanf("%d",&a[i]);
for(i=0;i<s;i++){
  for(j=i+1;j<s;j++){
    if(a[i]>a[j]){
      temp=a[i];
      a[i]=a[j];
      a[j]=temp;
    }
  }
}

printf("After sorting is: ");
for(i=0;i<s;i++)
  printf(" %d",a[i]);

return 0;
}
```

Output:
Enter total elements: 5
Enter 5 elements: 4 5 0 21 7
The array after sorting is: 0 4 5 7 21

Source code of simple quick sort implementation using array ascending order in c programming language

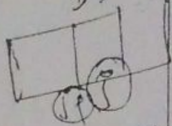
```
#include<stdio.h>

void quicksort(int [10],int,int);

int main(){
 int x[20],size,i;

 printf("Enter size of the array: ");
 scanf("%d",&size);

 printf("Enter %d elements: ",size);
 for(i=0;i<size;i++)
  scanf("%d",&x[i]);

 quicksort(x,0,size-1);

 printf("Sorted elements: ");
 for(i=0;i<size;i++)
  printf(" %d",x[i]);
```

2. **Wap a c program to search an element in an array using binary search**

```
#include<stdio.h>
int main(){

   int a[10],i,n,m,c=0,l,u,mid;

   printf("Enter the size of an array: ");
   scanf("%d",&n);

   printf("Enter the elements in ascending order: ");
   for(i=0;i<n;i++){
      scanf("%d",&a[i]);
   }

   printf("Enter the number to be search: ");
   scanf("%d",&m);

   l=0,u=n-1;
   while(l<=u){
```

```
return 0;
}

void quicksort(int x[10],int first,int last){
  int pivot,j,temp,i;

  if(first<last){
     pivot=first;
     i=first;
     j=last;

     while(i<j){
        while(x[i]<=x[pivot]&&i<last)
           i++;
        while(x[j]>x[pivot])
           j--;
        if(i<j){
           temp=x[i];
           x[i]=x[j];
           x[j]=temp;
        }
     }

     temp=x[pivot];
     x[pivot]=x[j];
     x[j]=temp;
     quicksort(x,first,j-1);
     quicksort(x,j+1,last);

  }
}
```

i → stop greater than pivot
j → stop lesser "
swap (i j)
j → cross i then j & pivot swap
size-1
- left half
- Right Right
www.thinkaloudacademy.com



Output:
Enter size of the array: 5
Enter 5 elements: 3 8 0 1 2
Sorted elements: 0 1 2 3 8

```
     mid=(l+u)/2;
     if(m==a[mid]){
        c=1;
        break;
     }
     else if(m<a[mid]){
        u=mid-1;
     }
     else
        l=mid+1;
  }
  if(c==0)
     printf("The number is not found.");
  else
     printf("The number is found.");

  return 0;
}
```

Sample output:
Enter the size of an array: 5
Enter the elements in ascending order: 4 7 8 11 21
Enter the number to be search: 11
The number is found.

```
if(item == a[mid]){
printf("Binary search scu");
printf("%d found in position %d",
              item, mid+1);
} else{
     printf(search failed
     item, search);
}
```

```
#include <stdio.h>
int main()
{
int n,a[30], item, i, j, mid
  top, bottom;
printf("enter element");
scanf("%d", &n);
printf("enter the %d elements ascending");
for(i=0; i<n; i++){
  scanf("%d", &a[i]);
}
scanf("%d", &item);
bottom=1;
 top=n;
do
{
  mid=(bott+top)/2;
   if(item<a[mid])
     top= mid-1;
   else if(item>a[mid]);
     bottom= mid+1;
} while(item!=a[mid] &&
        bottom<=top);
```

Scanned by CamScanner

preprocessing

preprocessors

↳ extends the powers of c prog.

↳ # → preprocessing directives

include <stdio.h>

use of # define.

identifies

define token_string

↳ optional

define c 2797

include <stdio.h>

define PI 3.14

int main()

{ int r;

float a;

printf("enter the radius");

scanf("%d", &r);

area = PI * r * r;

printf("Area = %f", a);

return 0;

}

macros with argument

define area(r) (3.14 * r * r)

define identifies(identifies1..

identifies n) token string

define area(r)

(3.14 * r * r)

1. Macro

define

constant value

2. Header file inclusion

include <file_name>

The source code of the file "file_name" is included in the main prog. at the specified place

3. conditional compilation

ifdef

endif

if

else

ifndef

included or excluded in SP before compilation w.r.t to cond^n

other directives

1: # undef → used to undefine a

pragma → is used to call a func^n before or after main fun. in a C prog.

STRUCTURES AND UNIONS

10 STRUCTURES AND UNIONS

Key Terms

Array I Structure I Dot operator Union I Bit field

10.1 INTRODUCTION

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

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

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

10.2 DEFINING A STRUCTURE

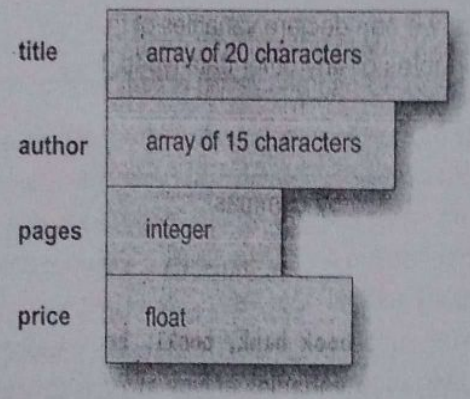
Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of

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

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

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

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



The general format of a structure definition is as follows:

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

In defining a structure you may note the following syntax:

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

Arrays Vs Structures

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

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

10.3 DECLARING STRUCTURE VARIABLES

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

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

For example, the statement

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

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

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

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

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

The declaration

```
struct book_bank
{
```

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

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

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

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

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

Type-Defined Structures

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

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

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

type_name variable1, variable2, ... ;

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

Program 10.1

Explain how complex number can be represented using structures. Write two C functions: one to return the sum of to complex numbers passed as parameters.

A complex number has two parts: real and imaginary. Structures can be used to realize complex numbers in C, as shown below:

```
Struct complex /*Declaring the complex number datatype using structure*/
{
```

```
    double real;/*Real part*/
    double img;/*Imaginary part*/
};
```

Function to return the sum of two complex numbers

```
Struct complex add(struct complex c1, struct complex c1)
{
    struct complex c3;
    c3.real=c1.real+c2.real;
    c3.img=c1.img+c2.img;
    return(c3);
}
```

Function to return the product of two complex numbers

```
Struct complex product(struct complex c1, struct complex c1)
{
    struct complex c3;
    c3.real=c1.real*c2.real-c1.img*c2.img;
    c3.img=c1.real*c2.img+c1.img*c2,real;
    return(c3);
}
```

10.4 ACCESSING STRUCTURE MEMBERS

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

book1.price

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

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

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

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

are valid input statements.

Program 10.2 Define a structure type, **struct personal** that would contain person name, date of joining and salary. Using this structure, write a program to read this information for one person from the keyboard and print the same on the screen.

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

Program

```
struct  personal
{
      char    name[20];
      int     day;
      char    month[10];
      int     year;
      float   salary;
};
main()
{
      struct personal person;

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

Output

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

Fig. 10.1 *Defining and accessing structure members*

10.5 STRUCTURE INITIALIZATION

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

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

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

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

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

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

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

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

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

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

Rules for Initializing Structures

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

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

10.6 COPYING AND COMPARING STRUCTURE VARIABLES

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

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

However, the statements such as

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

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

Program 10.3 Write a program to illustrate the comparison of structure variables.

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

program

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

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

     student3 = student2;

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

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

}
```

Output

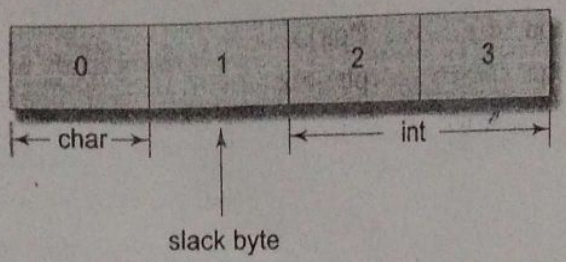
```
student2 and student3 are same

222 Reddy 67.000000
```

Fig. 10.2 *Comparing and copying structure variables*

Word Boundaries and Slack Bytes

Computer stores structures using the concept of "word boundary". The size of a word boundary is machine dependent. In a computer with two bytes word boundary, the members of a structure are

stored left_aligned on the word boundary, as shown below. A character data takes one byte and an integer takes two bytes. One byte between them is left unoccupied. This unoccupied byte is known as the *slack byte*.



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

10.7 OPERATIONS ON INDIVIDUAL MEMBERS

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

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

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

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

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

Three Ways to Access Members

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

```
typedef struct
{
        int x;
        int y;
} VECTOR;
VECTOR v, *ptr;
ptr = & v;
```

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

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

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

10.8 ARRAYS OF STRUCTURES

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

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

defines an array called **student**, that consists of 100 elements. Each element is defined to be of the type **struct class.** Consider the following declaration:

```
struct marks
{
   int  subject1;
   int  subject2;
   int  subject3;
};
main()
{
   struct marks student[3] =
   {{45,68,81}, {75,53,69}, {57,36,71}};
```

This declares the **student** as an array of three elements **student[0]**, **student[1]**, and **student[2]** and initializes their members as follows:

```
student[0].subject1 = 45;
student[0].subject2 = 65;
   ....
   ....
student[2].subject3 = 71;
```

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

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

Program 10.4 For the **student** array discussed above, write a program to calculate the subject-wise and student-wise totals and store them as a part of the structure.

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

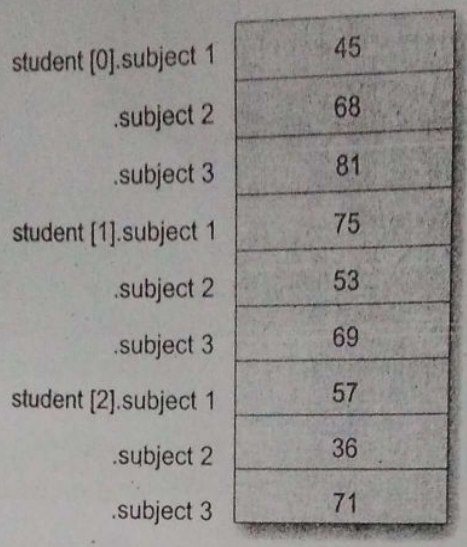


Fig. 10.3 *The array student inside memory*

Program

```
struct marks
 {
     int  sub1;
     int  sub2;
     int  sub3;
     int  total;
 };
 main()
 {
   int  i;
   struct marks student[3] =  {{45,67,81,0},
                              {75,53,69,0},
                              {57,36,71,0}};
   struct marks total;
   for(i = 0; i <= 2; i++)
   {
     student[i].total = student[i].sub1 +
                        student[i].sub2 +
                        student[i].sub3;
    total.sub1 = total.sub1 + student[i].sub1;
    total.sub2 = total.sub2 + student[i].sub2;
```

```
        total.sub3 = total.sub3 + student[i].sub3;
        total.total = total.total + student[i].total;
    }
    printf(" STUDENT                TOTAL\n\n");
    for(i = 0; i <= 2; i++)
        printf("Student[%d]      %d\n", i+1,student[i].total);
    printf("\n SUBJECT       TOTAL\n\n");
    printf("%s        %d\n%s        %d\n%s        %d\n",
                    "Subject 1    ", total.sub1,
                    "Subject 2    ", total.sub2,
                    "Subject 3    ", total.sub3);

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

Output

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

Student[3]       164

SUBJECT          TOTAL
Subject 1        177
Subject 2        156
Subject 3        221

Grand Total  = 554
```

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

10.9 ARRAYS WITHIN STRUCTURES

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

```
struct marks
{
    int number;
    float subject[3];
} student[2];
```

Here, the member **subject** contains three elements, **subject[0], subject[1]** and **subject[2]**. These elements can be accessed using appropriate subscripts. For example, the name

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

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

Program 10.5

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

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

Program

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

    struct marks total;
    int  i,j;

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

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

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

}
```

Output

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

STUDENT          TOTAL
Student-1        177
Student-2        156
Student-3        221
Grand Total  =   554
```

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

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];
```

2 variables

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

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

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

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

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

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

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

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

> **Note** *C permits nesting upto 15 levels. However, C99 allows 63 levels of nesting.*

10.11 STRUCTURES AND FUNCTIONS

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

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

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

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

function_name (structure_variable_name);

The called function takes the following form:

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

The following points are important to note:

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

Program 10.6 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 ize.

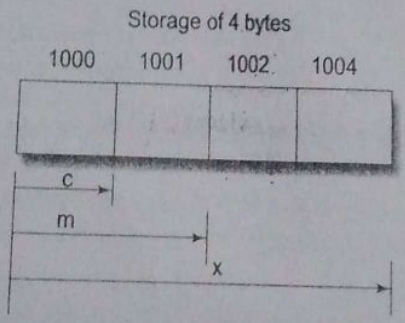


Fig. 10.7 *Sharing of a storage locating by union members*

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

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

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

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

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

would produce erroneous output (which is machine dependent).

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

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

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

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

is valid but the declaration

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

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

10.13 SIZE OF STRUCTURES

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

```
sizeof(struct x)
```

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

```
sizeof(y)
```

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

```
sizeof(y)
```

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

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

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

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

10.14 BIT FIELDS

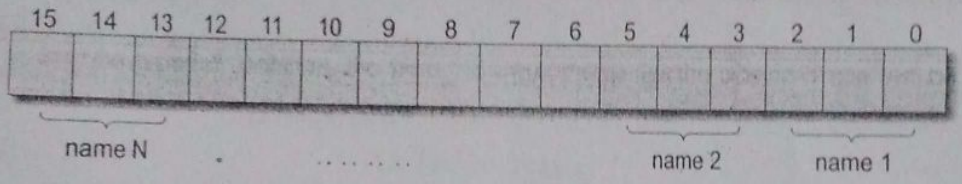
So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small *bit fields* to hold data items and thereby to pack several data items in a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

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

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

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

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



There are several specific points to observe:

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

   Unsigned : *bit-length*

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

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

```
struct personal
{
unsigned sex          :  1
unsigned age          :  7
unsigned m_status     :  1
```

```
        unsigned children      :    3
        unsigned               :    4
        } emp;
```

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

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

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

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

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

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

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

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

are valid statements.

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

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

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

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

```
struct pack
{
        unsigned   a:2;
```

```
        int count;
        unsigned  b : 3;
};
```

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

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

Just Remember

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

- We cannot take the address of a bit field. Therefore, we cannot use **scanf** to read values in bit fields. We can neither use pointer to access the bit fields.
- Bit fields cannot be arrayed.

Case Studies

Book Shop Inventory

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

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

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

look_up(table, s1, s2, m)

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

sizeof(book)/sizeof(struct record)

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

Do you want any other book?

Note that we use the function

get(string)

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

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

Programs

```
#include  <stdio.h>
#include  <string.h>
struct  record
{
    char    author[20];
    char    title[30];
    float   price;
```

```
    struct

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

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

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

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

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

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

Output

```
Enter title and author name as per the list
Title:    BASIC
Author:   Balagurusamy
Balagurusamy BASIC 30.00 January 1984 TMH
Enter number of copies:5
Required copies not in stock
Do you want any other book? (YES / NO):y
Enter title and author name as per the list
Title:    COBOL
Author:   Balagurusamy
Balagurusamy COBOL 60.00 December 1988 Macmillan
Enter number of copies:7
Cost of 7 copies = 420.00

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

Book not in list


Do you want any other book? (YES / NO):n
.
.
Thank you.  Good bye!
```

Fig. 10.8 *Program of bookshop inventory*

Review Questions

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

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

(h) We can perform mathematical operations on structure variables that contain only numeric type members.

(i) The keyword **typedef** is used to define a new data type.

(j) In accessing a member of a structure using a pointer p, the following two are equivalent: (*p).member_name and p –> member_name

(k) A union may be initialized in the same way a structure is initialized.

(l) A union can have another union as one of the members.

(m) A structure cannot have a union as one of its members.

(n) An array cannot be used as a member of a structure.

(o) A member in a structure can itself be a structure.

10.2 Fill in the blanks in the following statements:

(a) The _____ can be used to create a synonym for a previously defined data type.

(b) A_____ is a collection of data items under one name in which the items share the same storage.

(c) The name of a structure is referred to as _____.

(d) The selection operator –> requires the use of a _____ to access the members of a structure.

(e) The variables declared in a structure definition are called its _____.

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

(a) `struct a,b,c;`

(b) `struct abc a,b,c`

(c) `abc x,y,z;`

(d) `struct abc a[ ];`

(e) `struct abc a = { };`

(f) `struct abc = b, { 1+2, 3.0, "xyz"}`

(g) `struct abc c = {4,5,6};`

(h) `struct abc a = 4, 5.0, "xyz";`

10.4 Given the declaration

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

which of the following statements are legal?

(a) `scanf ("%d, &a);`

(b) `printf ("%d", b);`

(c) `a = b;`

(d) `a = b + c;`

(e) `if (a>b)`

`.....`

10.5 Given the declaration

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

which of the following are correct statements for declaring one dimensional array of 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
(c) Unions

Give a typical example of use of each of them.

10.12 Given the structure definitions and declarations

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

find errors, if any, in the following statements:

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

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

10.14 State the rules for initializing structures.

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

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

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

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

10.19 What is the error in the following program?

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

10.20 What will be the output of the following program?

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

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

ramming Exercises

efine a structure data type called **time_struct** containing three members in nute and integer **second**. Develop a program that would assign value mbers and display the time in the following form:

16:40:51

odify the above program such that a function is used to input values to other function to display the time.

esign a function **update** that would accept the data structure designed in crements time by one second and returns the new time. (If the increment res en the second member is set to zero and the minute member is increment result is 60 minutes, the minute member is set to zero and the hour men one. Finally when the hour becomes 24, it is set to zero.)

efine a structure data type named **date** containing three integer member ear. 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

a third function.

e input data should be three integers like 29, 4, and 2002 corresponding ar. Examples of invalid data:

31, 4, 2002 – April has only 30 days
29, 2, 2002 – 2002 is not a leap year

esign a function **update** that accepts the **date** structure designed in Exercis e date by one day and return the new date. The following rules are applica

- 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

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

se suitable algorithm to convert the long integer 19450815 into year, month dd a function called **nextdate** to the program designed in Exercise 10.4 to p sk:

- Accepts two arguments, one of the structure **data** containing the pr second an integer that represents the number of days to be added to th

teger **hour,** integer
s to the individual

the members and

Exercise 10.1 and
sults in 60 seconds,
ed by one. Then, if
ber is incremented

s **day**, **month** and

to day, month and

e 10.4 to increment
ble:

represents the date
, 1945) and assigns

and day.
erform the following

esent date and the
ne 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.