Structures

Introduction

- When we handle real world data, we don’t usually deal with little atoms of information by themselves—things like integers, characters and such
- Instead we deal with entities that are collections of things, each thing having its own attributes, just as the entity we call a ‘book’ is a collection of things such as title, author, call number, publisher, number of pages, date of publication, etc.
- As you can see all this data is dissimilar, for example author is a string, whereas number of pages is an integer.
- For dealing with such collections, C provides a data type called ‘structure’
- A structure gathers together, different atoms of information that comprise a given entity

Why Use Structures

- We have seen earlier how ordinary variables can hold one piece of information and how arrays can hold a number of pieces of information of the same data type
- These two data types can handle a great variety of situations
- But quite often we deal with entities that are collection of dissimilar data types
- For example, suppose you want to store data about a book.
- You might want to store its name (a string), its price (a float) and number of pages in it (an int)

Why Use Structures

- If data about say 3 such books is to be stored, then we can follow two approaches:
- Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages
- Use a structure variable

- Let us examine these two approaches one by one
- For the sake of programming convenience assume that the names of books would be single character long
- Let us begin with a program that uses arrays
- This approach no doubt allows you to store names, prices and number of pages.
- But as you must have realized, it is an unwieldy approach that obscures the fact that you are dealing with a group of characteristics related to a single entity—the book.
- The program becomes more difficult to handle as the number of items relating to the book go on increasing
- For example, we would be required to use a number of arrays, if we also decide to store name of the publisher, date of purchase of book, etc.
- To solve this problem, C provides a special data type—the structure

```
main( )
{
    char  name[3] ;
    float  price[3] ;
    int  pages[3], i ;

    printf ( "\nEnter names, prices and no. of pages of 3 books\n" ) ;

    for ( i = 0 ; i <= 2 ; i++ )
        scanf ( "%c %f %d", &name[i], &price[i], &pages[i] );

    printf ( "\nAnd this is what you entered\n" ) ;
    for ( i = 0 ; i <= 2 ; i++ )
        printf ( "%c %f %d\n", name[i], price[i], pages[i] );
}
```

And here is the sample run...

```
Enter names, prices and no. of pages of 3 books
A  100.00  354
C  256.50  682
F  233.70  512

And this is what you entered
A  100.000000  354
C  256.500000  682
F  233.700000  512
```

Structure

- A structure contains a number of data types grouped together.
- These data types may or may not be of the same type.
- The following example illustrates the use of this data type.

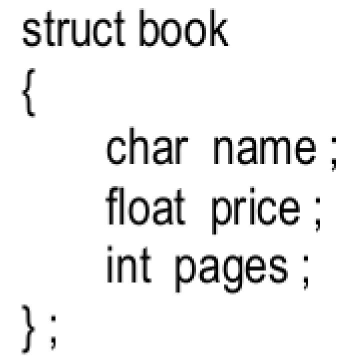
```
main( )
{
    struct book
    {
        char  name ;
        float  price ;
        int  pages ;
    } ;
    struct book  b1, b2, b3 ;

    printf ( "\nEnter names, prices & no. of pages of 3 books\n" ) ;
    scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;
    scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ;
    scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages ) ;

    printf ( "\nAnd this is what you entered" ) ;
    printf ( "\n%c %f %d", b1.name, b1.price, b1.pages ) ;
    printf ( "\n%c %f %d", b2.name, b2.price, b2.pages ) ;
    printf ( "\n%c %f %d", b3.name, b3.price, b3.pages ) ;
}
```

And here is the output...

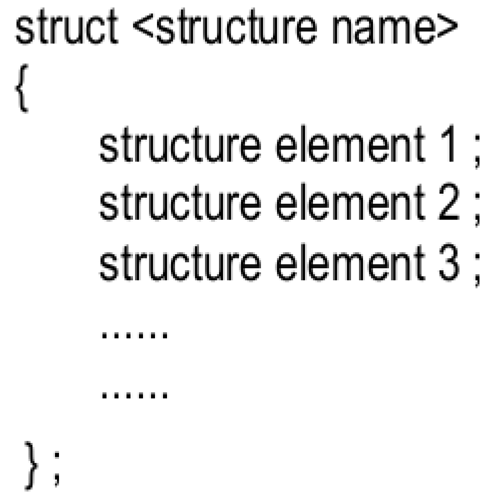
```
Enter names, prices and no. of pages of 3 books
A  100.00  354
C  256.50  682
F  233.70  512

And this is what you entered
A  100.000000  354
C  256.500000  682
F  233.700000  512
```

Declaring a Structure

- This statement defines a new data type called struct book.
- Each variable of this data type will consist of a character variable called name, a float variable called price and an integer variable called pages.
- The general form of a structure declaration statement is given below



```
struct book
{
      char  name ;
      float  price ;
      int  pages ;
} ;
```



```
struct <structure name>
{
      structure element 1 ;
      structure element 2 ;
      structure element 3 ;
      ......
      ......
} ;
```

- Once the new structure data type has been defined one or more variables can be declared to be of that type.
- For example the variables b1, b2, b3 can be declared to be of the type struct book, as,
- struct book b1, b2, b3 ;
- This statement sets aside space in memory.
- It makes available space to hold all the elements in the structure—in this case, 7 bytes—one for name, four for price and two for pages.
- These bytes are always in adjacent memory locations.
- If we so desire, we can combine the declaration of the structure type and the structure variables in one statement.

```
struct book
{
      char  name ;
      float  price ;
      int  pages ;
} ;
struct book  b1, b2, b3 ;
```

is same as...

```
struct book
{
      char  name ;
      float  price ;
      int  pages ;
} b1, b2, b3 ;
```

or even...

```
struct

{
      char  name ;
      float  price ;
      int  pages ;
} b1, b2, b3 ;
```

Initialization

- Like primary variables and arrays, structure variables can also be initialized where they are declared.
- The format used is quite similar to that used to initiate arrays.

```
struct book
{
      char  name[10] ;
      float  price ;
      int  pages ;
} ;
struct book  b1 = { "Basic", 130.00, 550 } ;
struct book  b2 = { "Physics", 150.80, 800 } ;
```

- Note the following points while declaring a structure type:
- The closing brace in the structure type declaration must be followed by a semicolon
- It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory.
- All a structure declaration does is, it defines the ‘form’ of the structure
- Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined.
- In very large programs they are usually put in a separate header file, and the file is included (using the preprocessor directive #include) in whichever program we want to use this structure type.

Accessing Structure Elements

- They use a dot (.) operator.
- So to refer to pages of the structure defined in our sample program we have to use,
- b1.pages
- Similarly, to refer to price we would use,
- b1.price
- Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

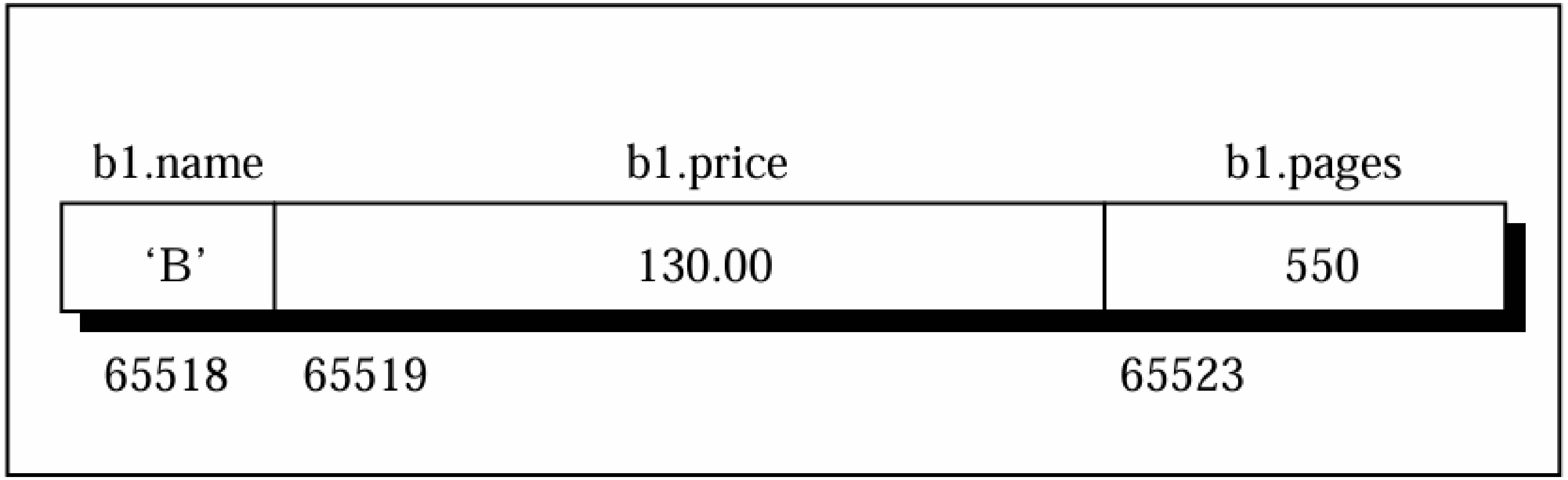
How Structure Elements are Stored

- Whatever be the elements of a structure, they are always stored in contiguous memory locations

```
/* Memory map of structure elements */
main( )
{
    struct book
    {
        char  name ;
        float  price ;
        int  pages ;
    } ;
    struct book  b1 = { 'B', 130.00, 550 } ;

    printf ( "\nAddress of name = %u", &b1.name ) ;

    printf ( "\nAddress of price = %u", &b1.price ) ;
    printf ( "\nAddress of pages = %u", &b1.pages ) ;
}
```

Here is the output of the program...

```
Address of name = 65518
Address of price = 65519
Address of pages = 65523
```


b1.name
b1.price
b1.pages
'B'
130.00
550
65518
65519
65523

Array of Structures

- In our sample program, to store data of 100 books we would be required to use 100 different structure variables from b1 to b100, which is definitely not very convenient.
- A better approach would be to use an array of structures.
- Notice how the array of structures is declared...
- struct book b[100] ;
- This provides space in memory for 100 structures of the type struct book.
- The syntax we use to reference each element of the array b is similar to the syntax used for arrays of ints and chars.
- For example, we refer to zeroth book’s price as b[0].price.
- Similarly, we refer first book’s pages as b[1].pages.

```
/* Usage of an array of structures */
main( )
{
    struct book
    {
        char  name ;
        float  price ;
        int  pages ;
    } ;

    struct book  b[100] ;
    int  i ;

    for ( i = 0 ; i <= 99 ; i++ )
    {
        printf ( "\nEnter name, price and pages " ) ;
        scanf ( "%c %f %d", &b[i].name, &b[i].price, &b[i].pages ) ;
    }

    for ( i = 0 ; i <= 99 ; i++ )
        printf ( "\n%c %f %d", b[i].name, b[i].price, b[i].pages ) ;
}
```

- In an array of structures all elements of the array are stored in adjacent memory locations.
- Since each element of this array is a structure, and since all structure elements are always stored in adjacent locations you can very well visualise the arrangement of array of structures in memory.
- In our example, b[0]’s name, price and pages in memory would be immediately followed by b[1]’s name, price and pages, and so on.

Additional Features of Structures

- The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.
- It is not necessary to copy the structure elements piece-meal.
- Obviously, programmers prefer assignment to piece-meal copying.
- Ability to copy the contents of all structure elements of one variable into the corresponding elements of another structure variable is rather surprising, since C does not allow assigning the contents of one array to another just by equating the two.
- As we saw earlier, for copying arrays we have to copy the contents of the array element by element.

```
main( )
{
    struct employee
    {
        char  name[10] ;
        int  age ;
        float  salary ;
    } ;
    struct employee  e1 = { "Sanjay", 30, 5500.50 } ;
    struct employee  e2, e3 ;

    /* piece-meal copying */
    strcpy ( e2.name, e1.name ) ;
    e2.age = e1.age ;
   e2.salary = e1.salary ;

   /* copying all elements at one go */
   e3 = e2 ;

   printf ( "\n%s %d %f", e1.name, e1.age, e1.salary ) ;
   printf ( "\n%s %d %f", e2.name, e2.age, e2.salary ) ;
   printf ( "\n%s %d %f", e3.name, e3.age, e3.salary ) ;
}
```

The output of the program would be...

```
Sanjay 30 5500.500000
Sanjay 30 5500.500000
Sanjay 30 5500.500000
```

- One structure can be nested within another structure.
- Using this facility complex data types can be created.
- Notice the method used to access the element of a structure that is part of another structure.
- For this the dot operator is used twice, as in the expression,
- e.a.pin or e.a.city
- Of course, the nesting process need not stop at this level.
- We can nest a structure within a structure, within another structure, which is in still another structure and so on... till the time we can comprehend the structure ourselves.

```
main( )
{
    struct address
    {
        char  phone[15] ;
        char  city[25] ;
        int  pin ;
    } ;

    struct emp
    {
        char  name[25] ;
        struct address  a ;
    } ;
    struct emp  e = { "jeru", "531046", "nagpur", 10 };

    printf ( "\nname = %s phone = %s", e.name, e.a.phone ) ;
    printf ( "\ncity = %s pin = %d", e.a.city, e.a.pin ) ;
}
```

And here is the output...

```
name = jeru phone = 531046
city = nagpur pin = 10
```

- Like an ordinary variable, a structure variable can also be passed to a function.
- We may either pass individual structure elements or the entire structure variable at one go.
- Let us examine both the approaches
- It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing.
- A better way would be to pass the entire structure variable at a time.
- This method is shown in the following program.

```
/* Passing individual structure elements */
main( )
{
    struct book
    {
        char  name[25] ;
        char  author[25] ;
        int  callno ;
    } ;
    struct book b1 = { "Let us C", "YPK", 101 } ;

    display ( b1.name, b1.author, b1.callno ) ;
}

display ( char  *s, char  *t, int  n )
{
    printf ( "\n%s %s %d", s, t, n ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

```
struct book
{
    char  name[25] ;
    char  author[25] ;
    int  callno ;
} ;

main( )
{
    struct book  b1 = { "Let us C", "YPK", 101 } ;
    display ( b1 ) ;
}

display ( struct book  b )
{
    printf ( "\n%s %s %d", b.name, b.author, b.callno ) ;
}
```

And here is the output...

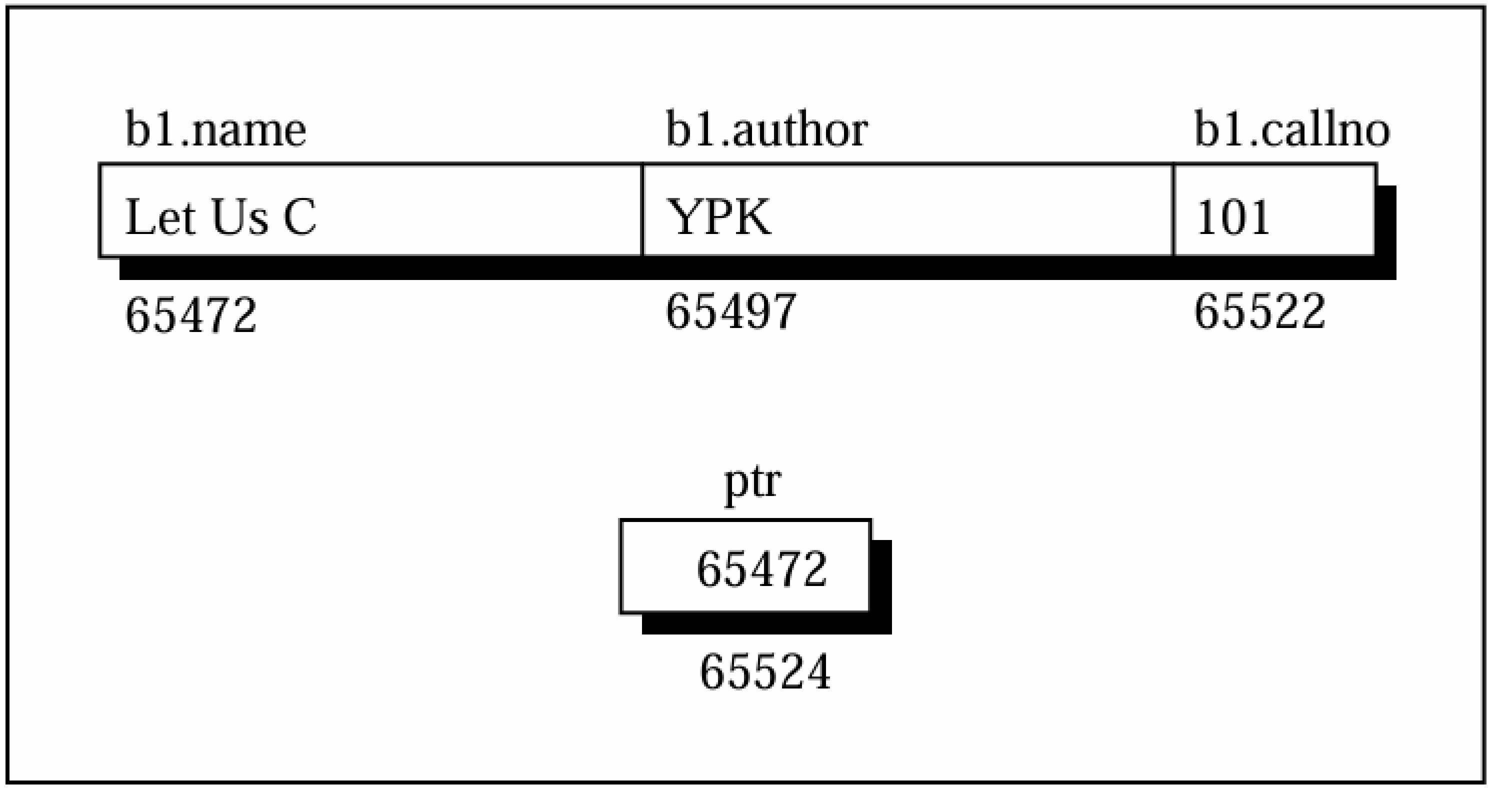
```
Let us C YPK 101
```

- Note that here the calling of function display( ) becomes quite compact,
- display (b1) ;
- Having collected what is being passed to the display( ) function, the question comes, how do we define the formal arguments in the function. We cannot say,
- struct book b1 ;
- because the data type struct book is not known to the function display( ).
- Therefore, it becomes necessary to define the structure type struct book outside main( ), so that it becomes known to all functions in the program.

Pointer to Structure

- The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to a struct.
- Such pointers are known as ‘structure pointers’.
- The first printf( ) is as usual. The second printf( ) however is peculiar.
- We can’t use ptr.name or ptr.callno because ptr is not a structure variable but a pointer to a structure, and the dot operator requires a structure variable on its left.
- In such cases C provides an operator ->, called an arrow operator to refer to the structure elements.
- on the left hand side of the ‘->’ operator there must always be a pointer to a structure.
- The arrangement of the structure variable and pointer to structure in memory is shown in the Figure

```
main( )
{
    struct book
    {
        char  name[25] ;
        char  author[25] ;
        int  callno ;
    } ;
    struct book  b1 = { "Let us C", "YPK", 101 } ;
    struct book  *ptr ;

    ptr = &b1 ;
    printf ( "\n%s %s %d", b1.name, b1.author, b1.callno ) ;
    printf ( "\n%s %s %d", ptr->name, ptr->author, ptr->callno ) ;
}
```


b1.name
b1.author
b1.callno
Let Us C
YPK
101
65472
65497
65522
ptr
65472
65524

Passing Address of Structure Variable

- We can pass the address of a structure variable to a function
- Again note that to access the structure elements using pointer to a structure we have to use the '->' operator.
- Also, the structure struct book should be declared outside main( ) such that this data type is available to display( ) while declaring pointer to the structure.

```
/* Passing address of a structure variable */
struct book
{
     char  name[25] ;
     char  author[25] ;
     int  callno ;
} ;

main( )
{
     struct book  b1 = { "Let us C", "YPK", 101 } ;
     display ( &b1 ) ;
}

display ( struct book  *b )
{
     printf ( "\n%s %s %d", b->name, b->author, b->callno ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Uses of Structures

- Database Management
- use of structures stretches much beyond database management
- They can be used for a variety of purposes like
- Changing the size of the cursor
- Clearing the contents of the screen
- Placing the cursor at an appropriate position on screen
- Drawing any graphics shape on the screen
- Receiving a key from the keyboard

Uses of Structures

- Checking the memory size of the computer
- Finding out the list of equipment attached to the computer
- Formatting a floppy Hiding a file from the directory
- Displaying the directory of a disk
- Sending the output to printer Interacting with the mouse