Structures

Scanned with CamScanner

Scanned with CamScanner

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). If data about say 3 such books is to be stored, then we can follow two approaches:

(a) Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.

(b) 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.

```
# include <stdio.h>
int main( )
{
    char name[ 3 ] ;
    float price[ 3 ] ;
    int pages[ 3 ], i ;
```

Scanned with CamScanner

Scanned with CamScanner

```
    printf ( "Enter 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 ] ) ;
    return 0 ;
}
```

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
```

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 goes 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.

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:

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

Scanned with CamScanner

Scanned with CamScanner

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

    printf ( "Enter 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 ( "And this is what you entered\n" ) ;
    printf ( "%c %f %d\n", b1.name, b1.price, b1.pages ) ;
    printf ( "%c %f %d\n", b2.name, b2.price, b2.pages ) ;
    printf ( "%c %f %d\n", b3.name, b3.price, b3.pages ) ;
    return 0 ;
}
```

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
```

This program demonstrates two fundamental aspects of structures:

(a) Declaration of a structure
(b) Accessing of structure elements

Let us now look at these concepts one-by-one.

Declaring a Structure

In our example program, the following statement declares the structure type:

```
struct book
{
```

Scanned with CamScanner

Scanned with CamScanner

```
    char  name ;
    float  price ;
    int  pages ;
} ;
```

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 <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.

For example,

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

is same as...

Scanned with CamScanner

Scanned with CamScanner

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

or even...

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

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 initialize arrays.

```
struct book
{
    char name[ 10 ] ;
    float price ;
    int pages ;
} ;
struct book b1 = { "Basic", 130.00, 550 } ;
struct book b2 = { "Physics", 150.80, 800 } ;
struct book b3 = { 0 } ;
```

Note the following points while declaring a structure type:

(a) The closing brace (}) in the structure type declaration must be followed by a semicolon (;).

(b) 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.

(c) 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

Scanned with CamScanner

Scanned with CamScanner

is included (using the preprocessor directive **#include**) in whichever program we want to use this structure type.

(d) If a structure variable is initiated to a value { 0 }, then all its elements are set to value 0, as in **b3** above. This is a handy way of initializing structure variables. In absence of this, we would have been required to initialize each individual element to a value 0.

Accessing Structure Elements

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed.

In arrays, we can access individual elements of an array using a subscript. Structures use a different scheme. 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. The following program would illustrate this:

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

Scanned with CamScanner

Scanned with CamScanner

```
    printf ( "Address of name = %u\n", &b1.name ) ;
    printf ( "Address of price = %u\n", &b1.price ) ;
    printf ( "Address of pages = %u\n", &b1.pages ) ;
    return 0 ;
}
```

Here is the output of the program...

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

Actually, the structure elements are stored in memory as shown in the Figure 17.1.

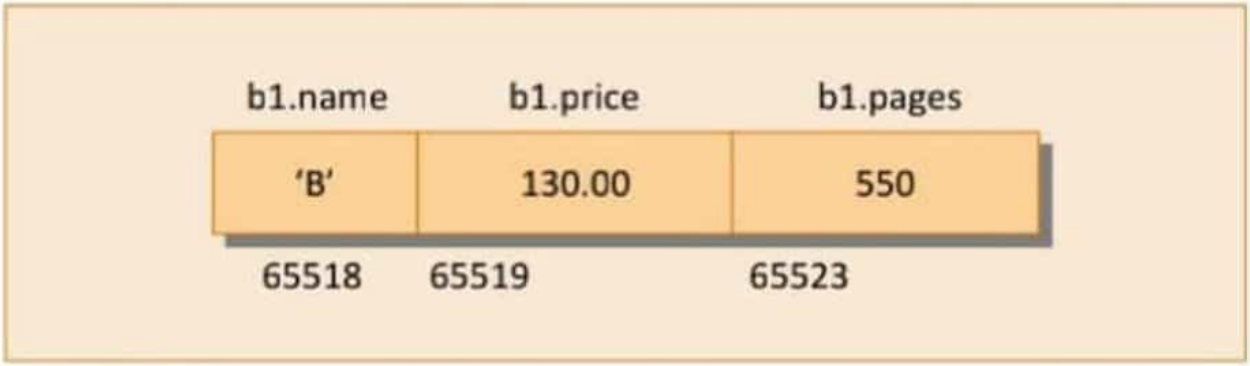


Figure 17.1

Array of Structures

Our sample program showing usage of structure is rather simple minded. All it does is, it receives values into various structure elements and output these values. But that's all we intended to do anyway... show how structure types are created, how structure variables are declared and how individual elements of a structure variable are referenced.

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. Following program shows how to use an array of structures:

```
/* Usage of an array of structures */
# include <stdio.h>
void linkfloat( ) ;
int main( )
{
```

Scanned with CamScanner

Scanned with CamScanner

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

    struct book  b[ 100 ] ;
    int  i ;

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

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

    return 0 ;
}

void linkfloat( )
{
    float a = 0, *b ;
    b = &a ;  /* cause emulator to be linked */
    a = *b ;   /* suppress the warning - variable not used */
}
```

Now a few comments about the program:

(a) 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**.

(b) The syntax we use to reference each element of the array **b** is similar to the syntax used for arrays of **int**s and **char**s. For example, we refer to zeroth book's price as **b[0].price**. Similarly, we refer first book's pages as **b[1].pages**.

Scanned with CamScanner

Scanned with CamScanner

(c) It should be appreciated what careful thought Dennis Ritchie has put into C language. He first defined array as a collection of similar elements; then realized that dissimilar data types that are often found in real life cannot be handled using arrays, therefore created a new data type called structure. But even using structures, programming convenience could not be achieved, because a lot of variables (**b1** to **b100** for storing data about hundred books) needed to be handled. Therefore, he allowed us to create an array of structures; an array of similar data types which themselves are a collection of dissimilar data types. Hats off to the genius!

(d) 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 visualize 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.

(e) What is the function **linkfloat( )** doing here? If you don't define it, you are likely to get an error "Floating-Point Formats Not Linked" with many C Compilers. What causes this error to occur? When parsing our source file, if the compiler encounters a reference to the address of a float, it sets a flag to have the linker link in the floating-point emulator. A floating-point emulator is used to manipulate floating-point numbers in functions like **scanf( )** and **atof( )**. There are some cases in which the reference to the **float** is a bit obscure and the compiler does not detect the need for the emulator. The most common is using **scanf( )** to read a **float** in an array of structures as shown in our program.

How can we force the formats to be linked? That's where the **linkfloat( )** function comes in. It forces linking of the floating-point emulator into an application. There is no need to call this function, just define it anywhere in your program.

Additional Features of Structures

Let us now explore the intricacies of structures with a view of programming convenience. We would highlight these intricacies with suitable examples.

(a) The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.

Scanned with CamScanner

Scanned with CamScanner

```
# include <stdio.h>
# include <string.h>
int 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.name = e1. name is wrong */
    e2.age = e1.age ;
    e2.salary = e1.salary ;

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

    printf ( "%s %d %f\n", e1.name, e1.age, e1.salary ) ;
    printf ( "%s %d %f\n", e2.name, e2.age, e2.salary ) ;
    printf ( "%s %d %f\n", e3.name, e3.age, e3.salary ) ;
    return 0 ;
}
```

The output of the program would be...

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

...contents of all structure elements of one variable

Scanned with CamScanner

Scanned with CamScanner

```
# include <stdio.h>
int 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 ( "name = %s phone = %s\n", e.name, e.a.phone ) ;
    printf ( "city = %s pin = %d\n", e.a.city, e.a.pin ) ;
    return 0 ;
}
```

And here is the output...

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

Scanned with CamScanner

Scanned with CamScanner

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. Such construction, however, gives rise to variable names that can be surprisingly self-descriptive, for example:

```
maruti.engine.bolt.large.qty
```

This clearly signifies that we are referring to the quantity of large sized bolts that fit on an engine of a Maruti car.

c) 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 shot. Let us examine both the approaches one-by-one using suitable programs.

```
/* Passing individual structure elements */
# include <stdio.h>
void display ( char  *, char  *, int ) ;
int 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 ) ;
    return 0 ;
}

void display ( char  *s, char  *t, int  n )
{
```

Scanned with CamScanner

Scanned with CamScanner

```
    printf ( "%s %s %d\n", s, t, n ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Observe that in the declaration of the structure, **name** and **author** have been declared as arrays. Therefore, when we call the function **display( )** using,

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

we are passing the base addresses of the arrays **name** and **author**, but the value stored in **callno**. Thus, this is a mixed call—a call by reference as well as a call by value.

It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements goes on increasing. A better way would be to pass the entire structure variable at a time. This method is shown in the following program:

```
# include <stdio.h>
struct book
{
    char  name[ 25 ] ;
    char  author[ 25 ] ;
    int  callno ;
} ;
void display ( struct book ) ;

int main( )
{
    struct book  b1 = { "Let us C", "YPK", 101 } ;
    display ( b1 ) ;
    return 0 ;
}

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

Scanned with CamScanner

Scanned with CamScanner

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 declare the structure type **struct book** outside **main( )**, so that it becomes known to all functions in the program.

d) 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'.

Let us look at a program that demonstrates the usage of a structure pointer.

```
# include <stdio.h>
int 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 ( "%s %s %d\n", b1.name, b1.author, b1.callno ) ;
    printf ( "%s %s %d\n", ptr->name, ptr->author, ptr->callno ) ;
    return 0 ;
}
```

Scanned with CamScanner

Scanned with CamScanner

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. Remember that on the left hand side of the '.' structure operator, there must always be a structure variable, whereas 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 17.2.

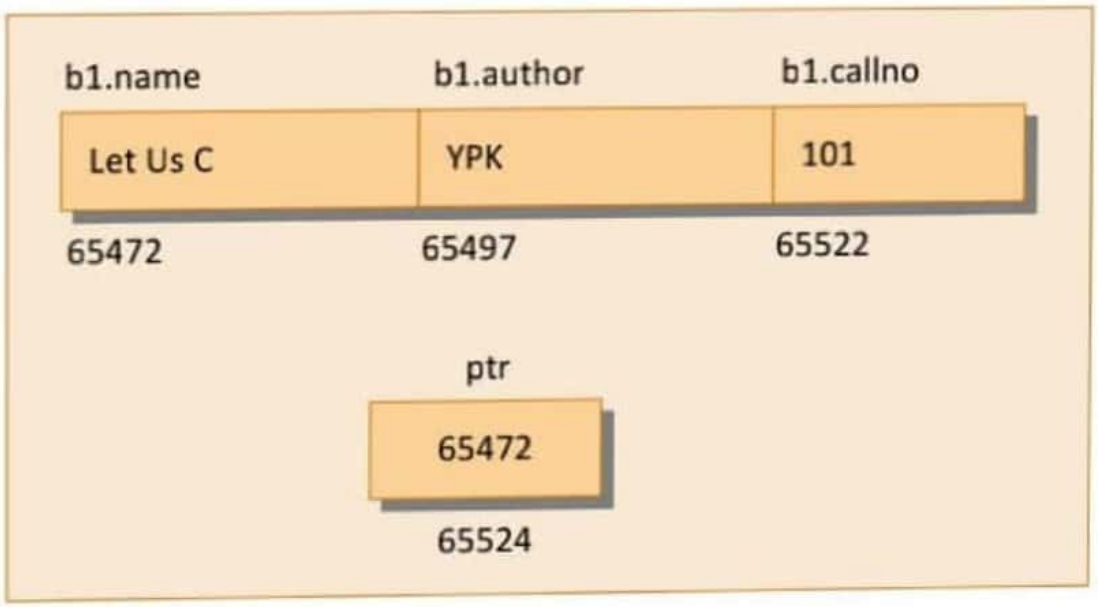


Figure 17.2

Can we not pass the address of a structure variable to a function? We can. The following program demonstrates this:

```
/* Passing address of a structure variable */
# include <stdio.h>
struct book
{
    char  name[ 25 ] ;
    char  author[ 25 ] ;
    int  callno ;
} ;
void display ( struct book  * ) ;

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

Scanned with CamScanner

Scanned with CamScanner

```
    return 0 ;
}

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

And here is the output...

```
Let us C YPK 101
```

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.

(e) Consider the following code snippet:

```
# include <stdio.h>
struct emp
{
    int a ;
    char ch ;
    float s ;
} ;
int main( )
{
    struct emp e ;
    printf ( "%u %u %u\n", &e.a, &e.ch, &e.s ) ;
    return 0 ;
}
```

If we execute this program using TC/TC++ Compiler we get the addresses as:

```
65518 65520 65521
```

Scanned with CamScanner

Scanned with CamScanner

Uses of Structures

Where are structures useful? The immediate application that comes to the mind is Database Management. That is, to maintain data about employees in an organization, books in a library, items in a store, financial accounting transactions in a company, etc. But mind you, use of structures stretches much beyond database management. They can be used for a variety of purposes like:

(a) Changing the size of the cursor
(b) Clearing the contents of the screen
(c) Placing the cursor at an appropriate position on screen
(d) Drawing any graphics shape on the screen
(e) Receiving a key from the keyboard
(f) Checking the memory size of the computer
(g) Finding out the list of equipment attached to the computer
(h) Formatting a floppy
(i) Hiding a file from the directory
(j) Displaying the directory of a disk
(k) Sending the output to printer
(l) Interacting with the mouse

And that is certainly a very impressive list! At least impressive enough to make you realize how important a data type a structure is and to be thorough with it if you intend to program any of the above applications.

Summary

(a) A structure is usually used when we wish to store dissimilar data together.

(b) Structure elements can be accessed through a structure variable using a dot (.) operator.

Scanned with CamScanner

Scanned with CamScanner

(c) Structure elements can be accessed through a pointer to a structure using the arrow (->) operator.

(d) All elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.

(e) It is possible to pass a structure variable to a function either by value or by address.

(f) It is possible to create an array of structures.

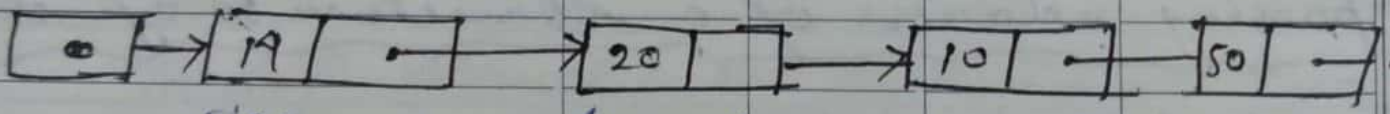
Scanned with CamScanner

Scanned with CamScanner

Self - Referential Structure :-

It is sometimes desirable to include with a structure, when a structure of member that is a pointer to the parent structure type, such structure is known as self - referential structure. eg :-

```
struct list {
    int number;
    struct list *next;
} node;
```



This is also known as linear Link list :-

Scanned with CamScanner

Scanned with CamScanner

11.7 UNIONS

Unions, like structures, contain members whose individual data types may differ from one another. However, the members within a union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory. They are useful for applications involving multiple members, where values need not be assigned to all of the members at any one time.

Within a union, the bookkeeping required to store members whose data types are different (having different memory requirements) is handled automatically by the compiler. However, the user must keep track of what type of information is stored at any given time. An attempt to access the wrong type of information will produce meaningless results.

Scanned with CamScanner

Scanned with CamScanner

The declaration and initialization of each member of union and variables of union is remains same as of structure. But initialization is possible at most member of the union at one time.

```
main()
{
  union id{
      char manuf[20];
      int size;
      char color;
      float cost;
  } shirt;
  printf("\n The size of the union %d",
                    sizeof(shirt));
  shirt.manuf = "Pantaloons";
  shirt.size = 44;
  shirt.color = 'R';
  shirt.cost = 375.00;
  printf("\n %s %d %c %.2f", shirt.manuf,
        shirt.size, shirt.color, shirt.cost);
  shirt.color = 'B';
  shirt.size = 46;
  shirt.cost = 500.00;
  printf("\n %s %d %c %.2f", shirt.manuf,
        shirt.color, shirt.size, shirt.cost);
}
```

O/p →

O/p → The size of the union : 20

Pantaloons (Rest of 3 will be garbage)

Garbage Garbage B Garbage

So, only one member of a union can be assigned a value at any one time.

Scanned with CamScanner

Scanned with CamScanner

11.6 SELF-REFERENTIAL STRUCTURES

It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type. In general terms, this can be expressed as

```
struct  tag  {
        member 1;
        member 2;
        . . . . .
        struct  tag  *name;
};
```

where *name* refers to the name of a pointer variable. Thus, the structure of type *tag* will contain a member that points to another structure of type *tag*. Such structures are known as *self-referential* structures.

EXAMPLE 11.29 A C program contains the following structure declaration.

```
struct  list_element  {
            char item[40];
            struct  list_element  *next;
};
```

This is a structure of type `list_element`. The structure contains two members: a 40-element character array, called `item`, and a pointer to a structure of the same type (i.e., a pointer to a structure of type `list_element`), called `next`. Therefore this is a self-referential structure.

Scanned with CamScanner

Scanned with CamScanner

Programming Tip:
The size of a union is equal to the size of its largest member.

Scanned with CamScanner
Scanned with CamScanner

8.9 ENUMERATED DATA TYPES

The enumerated data type is a user-defined type based on the standard integer type. An enumeration consists of a set of named integer constants. In other words, in an enumerated type, each integer value is assigned an identifier. This identifier (which is also known as an enumeration constant) can be used as a symbolic name to make the program more readable.

To define enumerated data types, we use the keyword `enum`, which is the abbreviation for `ENUMERATE`. Enumerations create new data types to contain values that are not limited to the values that fundamental data types may take. The syntax of creating an enumerated data type can be given as follows:

```
enum enumeration_name { identifier1,
identifier2, ....., identifiern };
```

The `enum` keyword is basically used to declare and initialize a sequence of integer constants. Here, `enumeration_name` is optional. Consider the following example, which creates a new type of variable called `COLORS` to store color constants.

```
enum COLORS {RED, BLUE, BLACK, GREEN,
YELLOW, PURPLE, WHITE};
```

Scanned with CamScanner

Scanned with CamScanner

The enum keyword is basically used to declare and initialize a sequence of integer constants. Here, enumeration_name is optional. Consider the following example, which creates a new type of variable called COLORS to store color constants.

```
enum COLORS {RED, BLUE, BLACK, GREEN,
YELLOW, PURPLE, WHITE};
```

Note that no fundamental data type is used in the declaration of COLORS. After this statement, COLORS has become a new data type. Here, COLORS is the name given to the set of constants. In case you do not assign any value to a constant, the default value for the first one in the list—RED (in our case), has the value of 0. The rest of the undefined constants have a value 1 more than its previous one. That is, if you do not initialize the constants, then each one would have a unique value. The first would be zero and the rest would count upwards. So, in our example,

```
RED = 0, BLUE = 1, BLACK = 2, GREEN = 3,
YELLOW = 4, PURPLE = 5, WHITE = 6
```

If you want to explicitly assign values to these integer constants then you should specifically mention those values shown as follows.

```
enum COLORS {RED = 2, BLUE, BLACK = 5, GREEN
= 7, YELLOW, PURPLE, WHITE = 15};
```

Scanned with CamScanner

Scanned with CamScanner

As a result of this statement, now RED = 2, BLUE = 3, BLACK = 5, GREEN = 7, YELLOW = 8, PURPLE = 9, WHITE = 15.

Look at the code which illustrates the declaration and access of enumerated data types.

```
#include <stdio.h>
main()
{  enum {RED=2, BLUE, BLACK=5, GREEN=7,
   YELLOW, PURPLE, WHITE=15};
   printf("\n RED = %d", RED);
   printf("\n BLUE = %d", BLUE);
   printf("\n BLACK = %d", BLACK);
   printf("\n GREEN = %d", GREEN);
   printf("\n YELLOW = %d", YELLOW);
   printf("\n PURPLE = %d", PURPLE);
   printf("\n WHITE = %d", WHITE);
   return 0;
}
```

Output

```
RED = 2
BLUE = 3
BLACK = 5
GREEN = 7
YELLOW = 8
PURPLE = 9
WHITE = 15
```

Scanned with CamScanner

Scanned with CamScanner