Chapter: 7

Custom (User-defined) data-types & Advanced Operators

Structure-Union, Bit-field, typedef & enumeration, bitwise operator, shift operator etc.

7.1 The Custom (User-defined) Data-types of C

Custom data types are the data types which are created by programmers. The ***built-in*** data types are ***int***, ***float***, ***char*** etc. Custom data types are created by **combining the built-in data types**.

In C we can define ***five*** kind of **custom data-types** :

[1] ***Structure***
[2] ***Union***
[3] ***Bit-field***
[4] ***Enumeration***
[5] ***Typedef***

- ***int*** , ***float***, ***char*** etc. are C's ***built-in*** data types (primary/fundamental data-type) and ***structure***, ***union*** etc. are C's ***combined***(user-defined) data-type.

7.2 STRUCTURE Basics

7.2.1 Defining structures

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.

A ***structure*** is an ***aggregate***(=collection) (or ***conglomerate***) ***data type*** that is composed of two or more related variables called ***members***. The members of a structure are also commonly referred to as ***fields*** or ***elements***. We'll use these terms interchangeably. Structures are defined in C using this ***general form***:

```
struct tag-name {
            type member_1;
            type member_2;
            type member_3;
            . . .
            . . .
            type member_N;
                  } variable-list;
```

- ***Unlike an array*** in which each element is of the ***same type***, **each member of a structure can have its own type**, which may differ from the types of the other members.
- The keyword ***struct*** tells the compiler that a ***structure type*** is being defined.
- The ***type*** of each ***member*** is a ***valid C type***.
- The ***tag-name*** is essentially the ***type name*** of the structure. It is the user defined ***data-type-name*** like : ***int***, ***char***. We use the ***type name*** as we've used ***int*** or ***char***.
- The ***variable-list*** is where actual instances (the variables which use the **structure data-type**) of the structure are declared.
- Either the ***tag-name*** or the ***variable-list*** is ***optional***, but **one must be present** (you will see why shortly).
- Generally, ***the information contained in a structure is logically related***. For example, consider a structure to hold a person's address. Another structure might be used to support an inventory program in which each item's name, retail and wholesale cost, and the quantity on hand are stored.

Difference between arrays and structures :

We have seen that arrays can he 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.

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.

[4] ***Structures*** help to organize complex data in a more meaningful way. It is a powerful concept that we nay often need to use in our program design.

Example : The structure shown here defines fields that can hold *card-catalog* information:

```
struct catalog {
            char name [40] ;     /* author name */
            char title[40] ;     /* title */
            char pub[40] ;       /* publisher */
            unsigned date;       /* copyright date */
```

```
    unsigned char ed;   /* edition */
        } card;
```

- Here, ***catalog*** is the ***type name*** of the structure. It is *not the name of a variable*.
- The only ***variable*** defined by this fragment is ***card***.
- It is important to understand that a structure declaration defines only a logical entity, which is a ***new data type***. It is not until variables of that type are ***declared*** than an object of that type actually exists. Thus, ***catalog is a logical template; card has physical reality***.

Table : How the card structure variable appears in memory (assuming 2-byre integers)

name [40]	...	40 byte
title[40]	...	40 byte
pub[40]	...	40 byte
date date	2byte	
ed	1byte	
Card = Total 123 byte.		

7.2.2 Declaring structure variables :

Once you have defined a structure type (with already existing variable-list or no variable-list), you can create additional variables of that type using this general form:

```
struct tag-name additional_variable-list;
```

Hence we can declare structure variable in two way either along with structure definition or separately after defining structure. Hence structure definition takes three forms :

Form : 1	***Form : 2***	***Form : 3***
struct tag-name { ***type member_1;*** ***type member_2;*** ***type member_3;*** ***. . .*** ***. . .*** ***type member_N;*** ***} variable-list;***	***struct {*** ***type member_1;*** ***type member_2;*** ***type member_3;*** ***. . .*** ***. . .*** ***type member_N;*** ***} variable-list;***	***struct tag-name {*** ***type member_1;*** ***type member_2;*** ***type member_3;*** ***. . .*** ***. . .*** ***type member_N;*** ***};***
▶ ***tag-name*** and ***variable-list*** are present. ▶ New ***additional*** variables can be declared along with ***old*** variable-list.	▶ No ***tag-name*** is present. ▶ ***Additional*** variables ***cannot*** be declared after defining structure.	▶ only ***tag-name*** is present. ▶ ***All variables*** can be declared after defining structure.
◘ ***struct tag_name var_list;*** is effective. ◘ It defines new ***additional*** variables.	◘ ***struct tag_name var_list;*** has no effect because no ***tag-name*** is present	◘ ***struct tag_name var_list;*** is effective. ◘ It defines ***all*** variables.

- ❒ For these reason either the ***tag-name*** or the ***variable-list*** is ***optional***, but **one must be present.** For example this statement declares three variables of type struct catalog:

```
struct catalog var1, var2, var3;
```

  This is why it is not necessary to declare any variables when the structure type is defined. You can declare them separately, as needed.

- ❒ If you know you only need a ***fixed number*** of ***structure variables***, you do *not need to specify the* ***tag-name*** (form - 2). For example, this code creates two structure variables, but the structure itself is unnamed:

```
struct { int a;
         char Ch;
       } var1, var2;
```

- ❒ In actual practice, we usually specify the tag name. And use "form-3". That is we don't declare variable with structure-definition. We'll declare all structure variable using : ***struct tag_name var_list;***

Defining structure :

```
struct tag-name {   type member_1;
                    type member_2;
                    type member_3;
                    . . .
                    . . .
                    type member_N;};
```

Declaring structure variables :

```
struct tag_name var_list;
```

7.2.3 Structure variable initialization :

Like any other data type, a structure variable can, be initialized at compile time. We can initialize a structure variable in two different ways:

- ***Along with structure declaration (without tag-name):*** We can initialize a structure variable directly when we ***declaring*** a ***structure***. In this case we don't need ***tag-name*** to mention. The general form is :

```
struct {      type member_1;
              type member_2;
              type member_3;
              . . .
              . . .
              type member_N;
    } variable_name = {member_1_value, member_2_value, . . ., member_N_value };
```

- ***Separately (split variable declaration) :*** We can initialize a structure variable ***using*** structure ***tag-name*** as we declared structure-variables before. This is the ***most used*** way. The general form is :

```
struct tag-name {   type member_1;
                    type member_2;
                    type member_3;
                    . . .
                    . . .
                    type member_N; } ;
                          . . . . . .
                          . . . . . .
struct tag-name variable_name = {member_1_value, member_2_value, . . ., member_N_value };
```

Rules and restrictions to initialize structure-variables : There are a few rules to keep in mind while initializing structure variables at compile-time.

[1] The compile-time initialization of a structure variable must have the following elements:
   i. The keyword ***struct***.
   ii. The structure ***tag-name***.
   iii. The name of the ***variable*** to be declared.
   iv. The assignment operator `=` .
   v. ***A set of values*** for the members of the structure variable, separated by commas and enclosed in ***curly braces***.
   vi. A ***terminating semicolon***.

[2] We cannot initialize ***individual members*** inside the structure template.

[3] The order of values enclosed in braces ***must match the order of members*** in the structure definition.

[4] 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.

[5] The ***uninitialized*** members will be assigned default values as follows:
   - Zero for integer and floating point numbers.
   - '\0' for characters and strings.

7.2.4 Accessing members of a structure & use of "." operator

To access a member of a structure, you must specify both the structure variable name and the member name, separated by a period. For example, using ***card***, the following statement assigns the ***date*** field the value ***1776*** : `card.date = 1776;` C programmers often refer to the ***period*** as ***the dot operator***.

- ***Output data from a member of a structure - variable :*** To output data from a member of a structure – variable we specify both the structure variable name and the member name, separated by a period inside the ***console output*** functions : ***`printf()`***, ***`putchar()`***, ***`puts()`***. For example: To print the copyright date of previous ***`catalog structure`***,

```
printf("Copyright date: %u", card.date);
```

- ***Input data to a member of a structure – variable :*** To input data to a member of a structure - variable we specify both the structure variable name and the member name, separated by a period inside the ***console input*** functions : ***`scanf()`***, ***`getchar()`***, ***`gets()`***. For example: To input the date, use a ***`scanf()`*** statement such as:

```
scanf("%u", &card.date);
```

  - Notice that the ***"&"*** goes before the ***structure name***, not before the ***member name***.

- ***I/O of string and individual character of a string to/from a member of a structure – variable :*** On a similar fashion, these statements input the author's name and output the title:

```
gets(card.name);
printf("%s", card.title);
```

  - To access an individual character in the `"title"` field, simply ***index*** `"title"`. For example, the following statement prints the third letter:

```
printf("%c", card.title[2]);
```

- ***Operations on individual members :*** 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. We can also apply ***increment*** and ***decrement*** operators to numeric type members. The ***precedence*** of the ***member operator*** (***dot operator***) is higher than all arithmetic and relational operators and therefore ***no parentheses*** are ***required***. For example : Consider

```
struct item{int m; float x; char c;} ver_1, ver_2;
```

We can perform the following operations:

```
if(ver_1.m > 4) ver_2.m++;        /* logical operator & increment operator*/
float sum = ver_1.x + ver_2.x;
```

7.2.5 Structures as arrays :

Structures can be arrayed in the same fashion as other data types. For example, the following structure definition creates a 100-element array of structures of type ***catalog***:

```
struct catalog cat[100];
```

- To access an individual structure of the array, you must index the array name. For example, the following accesses the first structure of catalog type structure array cat: ***cat[0];***
- To access a member within a specified structure, follow the index with a period and the name of the member you want. For example, the following Statement loads the ed field (or member) of structure cat[33] of type catalog with the value of 2: ***cat[33].ed = 2;***

7.2.6 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];
```

These elements can be accessed using appropriate subscripts. For example,

student[1].subject[2];

7.2.7 COPYING AND COMPARING STRUCTURE VARIABLES

You may assign the contents of one instance (variables) of a structure to another as long as they are both of the same type. For example, this fragment is perfectly valid:

```
struct s_type { int a; float f; } var1, var2;
var1.a = 10; var2.f = 100.23;    /*assigning values to members of var1*/
var2 = var1;                     /*copying values of var1 to members of ver2 */
```

After this fragment executes, ***var2*** will contain exactly the same thing as ***var1***.

- Two variables of the same structure type can be copied the same way as ordinary variables. If personl 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***. For Example these statements are valid:

```
person1.member_1 == person2.member_1;
person1.member_2 <= person2.member_2;
person1.member_3 != person2.member_3;
```

NOTE

A key concept to understand is that each instance of a structure contains its own copy of the members of the structure. For example,

```
struct tag_name var1, var2, var3;
```

the title field of ***var1*** is completely separate from the title field of ***var2***. In fact, the only relationship that ***var1***, ***var2***, and ***var3*** have with one another is that they are all variables of the same type of structure. There is no other linkage among the three.

7.2.8 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

[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.
- But it 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.*** Structures may be passed as ***parameters to functions*** just like any other type of value. A ***function*** may also ***return a structure***.

- ❒ 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 is to ***pass a pointer of 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.

Now we focus on the ***second method*** :

function returns structure : The general format of structure returning function

```
struct structure_type_name function_name ();
```

The called function takes the following form:

```
struct structure_type_name function_name ()
{
struct structure_type_name var_name;
. . .
Assignment of structure variable's members
. . .
return var_name;}
```

example : The following program, for example, loads the members of var1 with the values 100 and 123.23 and then displays them on the screen :

```
#include <stdio.h>
struct s_type { int i; double d; } ;
struct s_type f(void); /* function as a structure */
int main(void){struct s_type var1;
            var1 = f() ; /* calling the structure type function and assigned to var1 */
            printf("%d %f", var1.i, var1.d);
            return 0;}
struct s_type f(void){struct s_type temp;
                  temp.i = 100; temp.d = 123.23;
                  return temp;}
```

Structure as function parameter: 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 structure_type_name var_name)
{. . .
 . . .
 expression;}
```

example :

```
#include <stdio.h>
struct s_type { int i; double d; } ;
void f(struct s_type temp); /* structure as function parameter */
int main(void){struct s_type var1;
            var1.i = 99; var1.d = 98.6; /*assigning values to struct. variable var1 */
            f(var1); /* passing structure-variable to function*/
            return 0;}
/* defining function using structure-members */
void f(struct s_type temp){printf("%d %f", temp.i, temp.d);}
```

Remember following points

I. The called function must be declared for corresponding ***structure's type-name***, 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.

II. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same ***struct*** type.

III. The ***expression*** may be any simple variable or structure variable or an expression using simple variables.

IV. When a function returns a structure, it must be assigned to a structure of identical type in the calling function. And function must be declared as the corresponding structure type function.

V. The called functions must be declared in the calling function appropriately.

7.2.9 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 : ***sizof(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.
- If ***y[n]*** is an array variable of type ***struct*** then ***sizeof(y)*** would give the total number of bytes the array ***y[n]*** requires.

So using these information we can 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.

NOTE

[1] To know the size of a structure, you should use the sizeof compile-time operator. Do not try to manually add up the number of bytes in each field.

[2] We need sizeof() because : in some situations, the compiler may need to align certain types of data on even word boundaries. In this case, the size of the structure will be larger than the sum of its individual elements.

[3] When using sizeof with a structure type, you must precede the tag name with the keyword struct, as shown in this program:

```
#include <stdio.h>
struct s_type { int i; char ch; int *p; double d; } ver_s1, ver_sa[10];
int main(void){printf("s_type is %d bytes long", sizeof(struct s_type));
               printf("\n s_type variable ver_s1 is %d bytes long", sizeof(ver_s1));
               printf("\n s_type array variable ver_sa is %d bytes long", sizeof(ver_sa));
               return 0;}
```

7.2.10 Declare Pointer to Structure

We declare a pointer to a structure in the same way that we declare a pointer to any other type of variable with the same manner we declare pointer with the ***same structure-type*** of the variable. For example,

```
struct s_type {int i; char str[80];} s, *p;
```

the above fragment defines a ***structure*** called ***s_type*** and declares ***two variables***. The first, ***s***, is an *actual structure variable*. The second, ***p***, is *a pointer to structures of type* ***s_type***.

- Then, ***p = &s;*** assigns to ***p*** the address of ***s***.
- To access an individual element of ***s*** using ***p*** you cannot use the ***dot operator***. Instead, you must use the ***arrow operator*** as shown in the following

  ```
  p->i = 1
  ```

  This statement assigns the value ***1*** to element ***i*** of s through ***p***.

arrow operator : The arrow operator is formed using a minus sign followed by a greater-than sign. There must be no spaces between the two.

Why use pointer to structure : C passes structures to functions in their entirety. However, if the structure is very large, the passing of a structure can cause a considerable reduction in a program's execution speed. For this reason, when working with large structures, you might want to pass a pointer to a structure in situations that allow it instead of passing the structure itself.

REMEMBER :

- When accessing a member using a ***structure variable***, use the ***dot operator***.
- When accessing a member using a ***pointer***, use the ***arrow operator***.

NOTE : *Application of structure-pointer : Date and Time functions*

One very useful application of structure pointers is found in C's time and date functions. Several of these functions use a pointer to the current time and date of the system.

- The time and date functions require the header ***TIME. H*** for their prototypes. This header file also defines four ***types*** and two ***macros***.
- The type ***time_t*** is able to represent the system time and date as a ***long integer***. This is called the ***calendar time***. ***time_t*** represents the time and date of the system in an encoded implementation ***specific internal format***. To obtain the calendar time of the system, you must use the ***time()*** function, whose prototype is:

  time_t time(time_t *systime) ;

  The ***time()*** function returns the encoded calendar time of the system or ***-1*** if no system time is available. It also places this encoded form of the time into the variable pointed to by ***systime***. However, if ***systime*** is ***null***, the argument is ***ignored***.
- The ***structure type tm*** holds ***date*** and ***time*** broken down into its elements. The ***tm structure*** is defined as shown here:

```
struct tm {   int tm_sec;         /* seconds, 0-61 */
              int tm_min;         /* minutes. 0-59 */
              int tm_hour;        /* hours, 0-23 */
              int tm_mday;        /* day of the month, 1-31*/
```

```
int tm_mon;         /* months since Jan, 0-11 */
int tm-year;        /* years from 1900 */
int tm_wday;        /* days since Sunday, 0-6 */
int tm_yday;        /* days since Jan 1, 0-365 */
int tm_isdst;       /* Daylight Saving Time indicator */
};
```

The value of ***tm_isdst*** will be positive if Daylight Saving Time is in effect, 0 if it is not in effect, and negative if there is no information available. When the date and time are represented in this way, they are referred to as ***broken-down time***.

- Since the calendar time is represented using an implementation specified internal format, you must use another of C's time and date functions to convert it into a form that is easier to use. Called ***localtime().*** Its prototype is

  struct tm *localtime(time_t *systime);

  The ***localtime()*** function returns a pointer to the broken-down form of systime. The structure that holds the broken-down time is internally allocated by the compiler and will be overwritten by each subsequent call.

  This program demonstrates ***time()*** and ***localtime()*** by displaying the current time of the system:

```
#include <stdio.h>
#include <time.h>
int main(void)
{struct tm *systime;
 time_t t;
 t = time(NULL);
 systime = localtime(&t);
 printf("Time is %.2d:%.2d:%.2d\n", systime->tm_hour, systime->tm_min, systime->tm_sec);
 printf("Date: %.2d/%.2d/%.2d", systime->tm_mon+1, systime->tm_mday, systime->tm_year);
 return 0;}
```

  Here is sample output produced by this program:

```
Time is 10:32:49
Date: 03115/97
```

- The type ***clock_t*** is defined the same as ***time_t***. The header file also defines ***size_t***.
- The macros defined are ***NULL*** and ***CLOCKS_PER_SEC***.
- Another of C's time and date functions is called ***gmtime( )***. Its prototype is

  struct tm *gmtime(time_t *time);

  The ***gmtime()*** function works exactly like ***localtime()*** , except that it returns the Coordinated Universal Time (which is, essentially, Greenwich Mean Time) of the system.

7.2.11 NESTED STRUCTURES

we have only been working with structures whose members consist solely of C's ***basic types***. However, **members can also be other structures**. These are referred to as ***nested structures***. Here is an example that uses nested structures to hold information on the performance of two assembly lines, each with ten workers:

```
struct worker{char name[80];
              int avg_units_per_hour;
              int avg_errs_per_hour; };

struct asm_line{    int product_code;
                    double material_cost;
                    struct worker wkers[NUM_ON_ LINE];
              } line1, line2;
```

- To assign the value ***12*** to the ***avg_units_per_hour*** of the second ***wkers structure*** of ***line1*** , use this statement:

  line1.wkers[1].avg_units_per_hour = 12;

  As you see, the structures are accessed from the ***outer to the inner***. This is also the general case. **Whenever you have nested structures, you begin with the "*outermost*" and end with the "*innermost*".**

7.3 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.

- To reduce memory loss we use bit field.
- A bit field is a set of adjacent bits whose size can be from l 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. That is a bit-fields is a variation on a structure member . bit-fields is composed of one or more bits.
- Using a bit-field, you can access by name one or more bits within a byte or word. To define a bit-field, use this general form:

  type name : size;

  Here, ***type*** is either ***int*** or ***unsigned***. If you specify a ***signed*** bit-field, then the ***high-order bit*** is treated ***as*** a ***sign*** bit, if possible. The ***number of bits*** in the field is specified by ***size***. Notice that a ***colon*** separates the name of the bit-field from its size in bits.

- ***Bit-fields*** are useful when you want to pack information into the ***smallest possible space***. For example, here is a structure that uses bit-fields to hold inventory information.

```
struct b_type{ unsigned department: 3;  /* up to 7 departments */
               unsigned instock: 1;     /* 1 if in stock, 0 if out */
               unsigned backordered: 1; /* 1 if backordered, 0 if not */
               unsigned lead_time: 3;   /* order lead time in months */
             } inv [MAX_ITEM];
```

In this case one byte (which is half of size of int type field) can be used to store information on an inventory item that would normally have taken four bytes without the use of bit-fields.

The general form of bit-field structure is :

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

  - NOTE 1 : It is not necessary to completely define all bits ***within a byte*** or ***word***. For example, this is perfectly valid:

```
struct b_type { int a: 2;   int b: 3 ;} ;
```

  - NOTE 2 : Bit-fields are often used to store ***Boolean (true/false)*** data because they allow the efficient use of memory, remember, you can pack eight Boolean values into a single byte.

- 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.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name n			.....							Name 2			Name 1		

The C compiler is free to store bit-fields as it sees fit. However, usually the compiler will automatically store bit-fields in the smallest unit of memory that will hold them. Whether the bit-fields are stored high-order to low-order (left to right) or the other way around is implementation dependent. (However, many compilers use high-order to low-order.)

- 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.

  a) That is, the sum of lengths of all the fields in a structure should not be more than the size of a word.

  b) 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. Such fields provide ***padding*** within the word. Example:

```
unsigned : bit-length ;
```

  [4] There can be unused bits in a word.

  [5] We cannot take the address of a bit field variable. This means :

  a. we cannot use ***scanf*** to read values into bit fields.

  b. We can neither use ***pointer*** to access the bit fields.

  c. Bit fields cannot be ***arrayed***.

  [6] Bit fields should be assigned values that are within the range of their size. If we try to assign larger values, behavior would be unpredicted.

- It is possible ***to combine-normal structure elements with bit field elements***. Also we can mix bit-fields with other types of members in a structure's definition. for example :

```
struct b_type{ unsigned department: 3;  /* bit-field variable  */
               unsigned instock: 1;     /* bit-field variable  */
               unsigned backordered: 1; /* bit-field variable  */
               unsigned lead_time: 3;   /* bit-field variable  */
               char name[20];           /* normal variable */
               struct addr address;     /* structure variable */
             } inv[MAX_ITEM];
```

- ***Accessing members of bit-field :*** You refer to a bit-field just like any other member of a structure. The following statement, for example, assigns the value 3 to the ***department*** field of ***item 10*** (10th element of inv[n] array):

```
inv[9].department = 3;
```

Restrictions : Because the smallest addressable- unit of memory is a byte, you cannot obtain the address of a bit-field variable. That is ,

  - 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", &dpt);  /*assigning to other variable */
inv[9].department = dpt;      /*assigning other variable  to a bit-fild*/
```

- We cannot use ***pointer*** to access a bit-field.
- We cannot use ***array*** to a bit-field.

7.4 UNIONS

Unions are another user defined data type. Unions are a concept borrowed from structures and therefore follow the same syntax as structures. An union ,is defined much like a structure. Its general form is

```
union tag-name {
            type member_1;
            type member_2;
            type member_3;
            . . .
            . . .
            type member_N;
                  } variable-names;
```

Like a structure, either the ***tag-name*** or the ***variable-names*** may be present. Members may be of any valid C data type.

Variable declaration : we can declare variable using following or along with union-declaration (similar to structure) :

```
union tag-name variable_1, variable_2, . . . , variable_n;
```

- ☑ In C, a union is a ***single*** piece of memory that is shared by ***two*** or ***more*** variables.
- ☑ The variables that share the memory may be of ***different types***.
- ☑ However, ***only one variable*** may be in ***use*** at any one time.
- ☑ A ***union*** is defined much like a ***structure***.
- ☑ Like a structure, either the tag-name or the variable-names may be present.
- ☑ Members may be of any valid C data type.

NOTE

Why we need unions : In some cases we may need more than one variables but among them, we use only one variable at a time. So in such cases we use unions instead of structures.

7.4.1 Difference between structure and union

Though structure and unions are similar but there is major distinction between them in terms of storage(how they store data).

- 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***.
- The ***size*** of a ***structure*** is determined by the ***sum*** or total of the ***sizes of all*** of its ***members*** . The ***size*** of an ***union*** is the ***size*** of its ***largest ember***.

`union item {int m; float x; char c;} code;`

Storage of 4 bytes

1001	1002	1003	1004
c=1byte			
m=2byte	m=2byte		
x=4byte	x=4byte	x=4byte	x=4byte

Here x is the largest member having size 4 byte, hence size of the union is 4 byte.

`struct item{int m; float x; char c;} code;`

Storage of seven bytes

3001	3002	3003	3004	3005	3006	3007
c=1byte	m=2byte		x=4byte			

Here size of the structure = size of c + size of m + size of x
Hence, size of the structure is 7 byte.

Above declares a variable ***code*** of type ***union item***. The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

- 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.

NOTE

[1] It is important to understand that the ***size*** of a ***union*** is ***fixed*** at compile time and is large enough to accommodate the ***largest member*** of the union. Assuming 4-byte ***flaoat***, this means that ***code*** will be 4 bytes long. Even if ***code*** is currently used to hold an ***int*** value, it will still occupy 4 bytes of memory (though ***int*** is only 2 byte long).

[2] As is the case with structures, ***you should use the "sizeof" compile-time operator to determine the size of a union***. You should not simply assume that it will be the size of the largest element, because in some environments, the ***compiler may pad the union*** so that it aligns on a word boundary.

[3] Unions are very useful when you need to ***interpret data*** in two or more different ways.

7.4.2 Assigning values to union members

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. Where : `union item {int m; float x; char c;} sample;`

- To access a member of a ***union***, use the ***dot*** and ***arrow operators*** just as you do for structures. For example, this statement assigns 123.098 to x of ***sample***: ***`sample.x = 123.098;`***
- If you are accessing a union through a ***pointer***, you must use the ***arrow operator***. For example, assume that ***p*** points to ***sample***. The following statement assigns ***m*** the value 101: ***`p -> m = 101;`***
- ***Restrictions on accessing union members :***
  - 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;       /* int type variable m is used*/
code.x = 7859.36;   /* float type variable x is used and value of m is destroyed*/
printf("%d", code.m);      /* try to access destroyed variables value cause error*/
```

    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***.

Union variable initialization : 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*** (we cannot initialize a union with its second or third or other member).For example, ***`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***.

7.5 ENUMERATIONS

A list of ***named integer constants*** called an enumeration. These constants can then be used any place an integer can. To define an enumeration, use this general form:

`enum tag-name { enumeration list } variable-list;`

Either the ***tag-name*** or the ***variable-list*** is optional. The ***tag-name*** is essentially the ***type name*** of the enumeration (all type & variable declaration are same as structure). For example,

`enum color_type {red, green, yellow} color ;`

Here, an enumeration consisting of the constants ***red***, ***green***, and ***yellow*** is created. The enumeration tag is ***color_type*** and one variable, called ***color***, has been created.

- By default, the compiler assigns integer values to enumeration constants, beginning with ***0*** at the far left side of the list. Each constant to the right is one greater than the constant that precedes it. Therefore, in the color enumeration, ***red*** is ***0***, ***green*** is ***1***, and ***yellow*** is ***2***.
- However, you can override the compiler's default values by explicitly giving a constant a value ***new count begin from that value***. For example, in this statement `enum color_type {red, green=9, yellow} color ;`
  red is still ***0***, but ***green*** is ***9***, and ***yellow*** is ***10*** (*new counts begin from 9*).

☞ ***Split variable declaration of an enumeration :*** Once you have defined an enumeration, you can use its tag name to declare enumeration variables at other points in the program (similar to structure variable declaration) . For example `enum color_type mycolor;` declares ***mycolor*** as a ***color_type*** variable.

NOTE

[1] An enumeration is essentially an integer type and an enumeration variable can hold any integer value-not just those defined by the enumeration. But for clarity and structure, you should use enumeration variables to hold only values that are defined by their enumeration type. i.e. use 1, 2, 3, 4, 5, 6, etc.

[2] Two of the main uses of an enumeration are to help provide ***self-documenting*** code and to ***clarify the structure*** of your program.

7.6 typedef

In C you can create a ***new name*** for an existing ***type*** (i.e. we can rename ***char***, ***int***, ***float*** etc) using ***typedef***. The general form of ***typedef*** is

`typedef old-name new-name;`

This new name can be used to declare variables. For example., in the following program, ***smallint*** is a new name for a ***signed char*** and is used to declare ***i***.

```
#include <stdio.h>
typedef signed char smallint;     /* renaming the data-type signed char */
int main(void){     smallint i;   /*Renamed data-type*/
                    for(i=0; i<10; i++)
                    printf("%d", i);
                    return 0;}
```

- a ***typedef*** does ***not cause*** the original name to be ***deactivated***. For example, in the program, ***signed char*** is still a ***valid*** type.
- you can use several ***typedef*** statements to create many ***different***, ***new names*** for the ***same type***.

USE OF TYPEDEF

1. The first is to create ***portable programs***. Using a changed *data-type-name* instead of pre-defined *data-type-name* enables the opportunity to ***modify*** the variables *before compile* using just one statement. For example, if you know that you will be writing a program that will be executed on computers using ***16-bit integers*** as well as on computers using ***32-bit integers***, and you want to ensure that *certain variables are 16 bits long in both environments*, you might want to use a ***typedef*** when compiling the program for the ***16-bit*** machines as follows:

```
typedef int myint;
```

Assuming that you used ***myint*** to declare all integer values that you wanted to be ***16 bits long*** (which can be changed later).

Then, before compiling the code for a ***32-bit*** computer, you can change the ***typedef*** statement like this:

```
typedef short int myint;   /* changing 32 bit to 16 bit */
```

This works because on computers using ***32-bit*** integers. a ***short int*** will be ***16 bits long***.

2. The second reason you might want to use ***typedef*** is to help provide ***self-documenting*** code (*meaningful* or semantic *programming*). For example, if you are writing an inventory program, you might use this ***typedef*** statement.

```
typedef double subtotal;
```

Now, when anyone reading your program sees a variable declared as subtotal, he or she will know that *it is used to hold a subtotal*.

7.7 Bitwise and Shift Operators

Bitwise Operators : c contains four special operators that perform their operations on a ***bit-by-bit*** level. these operators are

1. bitwise ***AND*** : **&**
2. bitwise ***OR*** : **|**
3. bitwise ***XOR*** (eXclusive OR) : **^**
4. 1's ***complement*** : **~**

- These operators work with character and integer types; they cannot be used with floating-point types.
- The ***AND***, ***OR***, and ***XOR*** operators produce a result based on a comparison of corresponding bits in each operand.
  - ★ The ***AND*** operator sets a bit if both bits being compared are set.

```
  1010  0110
& 0011  1011
```

1	0	1	0	0	1	1	0
0	0	1	1	1	0	1	1
0	0	1	0	0	0	1	0

  - ★ The ***OR*** sets a bit if either of the bits being compared is set.

```
  1010  0110
| 0011  1011
```

1	0	1	0	0	1	1	0
0	0	1	1	1	0	1	1
1	0	1	1	1	1	1	1

  - ★ The ***XOR*** operation sets a bit when either of the two bits involved is ***1***, but not when both are ***1*** or both are ***0***.

```
  1010  0110
^ 0011  1011
```

1	0	1	0	0	1	1	0
0	0	1	1	1	0	1	1
1	0	0	1	1	1	0	1

⇨ Notice how the resulting bit is set, based on the outcome of the operation being applied to the corresponding bits in each operand.

- The ***1's*** complement operator is a ***unary operator*** that ***reverses the state*** of each bit within an integer or character.
  Eg: `~ 10010 = 01101.`

Shift Operators : C includes two operators ***not commonly found*** in other computer languages: the left and right ***bit-shift operators***. These operators may be applied only to character or integer operands. They take these general forms:

```
value << number-of-bits
Value >> number-of-bits
```

The integer expression specified by "***number-of-bits***" determines how many places to the left or right the bits within value are shifted. When bits are shifted off an ***end*** they are ***lost***.

▶ The ***left shift*** operator is **<<** . Each ***left-shift*** causes ***all bits*** within the specified value to be shifted ***left one position*** and a ***zero*** is brought in on the ***right***. a ***left shift*** is the ***same*** as ***multiplying*** the number ***by 2***.

▶ The ***right shift*** operator is **>>** . A ***right-shift*** shifts ***all bits*** to the ***right one position*** and brings a ***zero*** in on the ***left***. (Unless the number is negative, in which case a one is brought in). A ***right shift*** is equivalent to ***dividing*** a number ***by 2***.

⇨ Because of the internal operation of virtually all CPUs, shift operations are usually faster than their equivalent arithmetic operations.

For example :

Value	***Bitwise Presentation***								result	effect
a = 14	***0***	***0***	***0***	***0***	***1***	***1***	***1***	***0***	***14***	***no***
a **<<** 1	0	0	0	1	1	1	0	0	28	One bit left
a **>>** 1	0	0	0	0	0	1	1	1	7	One bit right

NOTE

1. The XOR operation has one interesting property. Given two values A and B, when the outcome of A ***XOR*** B is ***XOR***ed with B a second time (i.e. (***A*** ^ ***B***) ^ ***B*** = ***A***), A is produced. For example,

```
#include<stdio.h>
int main(void){int i;
                i = 100; printf("initial value of i: %d\n", i);
                i = i ^ 21987; printf("i after first XOR with 21987: %d\n", i);
                i = i ^ 21987; printf("i after second XOR with 21987: %d\n", i);
        return 0;}
```

OUTPUT

```
initial value of i: 100
i after first XOR with 21987: 21895
i after second XOR with 21987: 100
```

2. ***Difference between logical operators and bitwise operators :***
Logical operator gives 0 or 1 as output but bitwise operator gives any integer as output . For example consider following two examples ,
Say x=1, y=2. Then ***x&&y=1*** and ***x&y=0.***

Logical "AND" && : **x&&y = 1&&2 = 1&&1 = 1** , since 1 and 2 both positive.

Bitwise " AND " & :
$x = 1 \Rightarrow x = 0000\ 0001$
$y = 2 \Rightarrow x = 0000\ 0010$
$0000\ 0000 \Rightarrow 0$

Say x=3, y=4. Then **x||y=1** and **x|y=0.**

Logical "OR" ||: **x&&y = 3||4 = 1||1 = 1** , since 3 and 4 both positive.

Bitwise "OR" |:
$x = 3 \Rightarrow x = 0000\ 0011$
$y = 4 \Rightarrow x = 0000\ 0100$
$0000\ 0111 \Rightarrow 7$

3. There are no ***Logical*** **XOR** and **Complement of 1**. Also there is no ***BITWISE*** **negation (not)** operator.
4. One use of **Bitwise *XOR*** is ***encrypting*** password.

7.8 OPERATORs Advanced

7.8.1 The ternary operator " ? : "

A ***ternary operator*** requires ***three operands***. C contains one ternary operator: the ***?*** . The ***?*** operator is used to replace statements such as :

```
if(condition) var = exp1;
else  var = exp2;
```

The general form of the ***?*** operator is

```
var = condition ? exp1 : exp2 ;
```

Here, ***condition*** is an ***expression*** that evaluates to ***true*** or ***false***. If it is ***true***, ***var*** is assigned the value of ***exp1*** . If it is ***false*** , ,***var*** is assigned the value of ***exp2***.

The following program inputs a number and then converts the number into ***1*** if the number is ***positive*** and ***-1*** if it is ***negative***.

```
#include <stdio.h>
int main(void) {    int i;
                    printf("Enter a number: ");  scanf("%d", &i) ;
                    i=i>=0?1:-1;
                    printf("Outcome: %d ", , i);
                return 0;}
```

NOTE: The reason for the **?** operator is that a C compiler produce more efficient code using it instead of equivalent ***if/else*** statement.

7.8.2 The Comma Operator :

The comma operator has a very unique function: it tells the compiler to "***do this and this and this***". That is, the comma is used to ***string together*** several operations.

- The most common use of the comma is in the ***for loop***. In the following loop, the comma is used in the ***initialization portion*** to initialize two loop-control variables, and ***in the increment portion*** to increment i and j.

```
for(i=0, j=0 ; i+j<count ; i++, j++) ...
```

- The value of a comma-separated list of expressions is the rightmost expression. For example, the following assigns ***100*** to ***value***:

```
value = (count, 99, 33, 100);
```

The parentheses are necessary because the comma operator is lower in precedence than the assignment operator.

7.8.3 More Uses Of Assignment Operator :

You can ***assign several variables the same value*** using the general form

```
var _1 = var_2 = var_3 = ... = var_n = value ;
```

For example, `i = j = k = 100;` assigns ***i***, ***j***, and ***k*** the value ***100***. Professional programmers use such multiple-variable assignments.

Shorthand operators using "=" : Another variation on the assignment statement is sometimes called ***C shorthand***. In C, you can transform a statement like ***a = a + 3;*** into a statement like ***a += 3;***
In general, any time you have a statement of the form ***var = var op expression;***
you can write it in shorthand form as ***var = op expression;*** Here, ***op*** is one of **+ - * / % << >> & | ^** .

☞ NOTE : There must be ***no space between the operator and the equal sign***.

7.8.4 The precedence of all C - OPERATORS

Table shows the ***precedence*** of all the C ***Operators***.			
Precedence	***Operators***	***Precedence***	***Operators***
1 ***Highest***	**( ) [ ] -> .**	9	**^**
2	**! ~ + - ++ -- (type cast) * & sizeof**	10	**|**
3	*** / %**	11	**&&**
4	**+ -**	12	**||**
5	**<< >>**	13	**? :**
6	**< <= > >=**	14	**= += -+ *= /= etc**
7	**== !=**	15 ***Lowest***	**,**
8	**&**		