STRUCTURES AND UNIONS

INTRODUCTION TO STRUCTURE

As we know that Array is collection of the elements of same type , but many time we have to store the elements of the different data types.

Suppose Student record is to be stored, then for storing the record we have to group together all the information such as Roll, name, Percent which may be of different data types.

Ideally Structure is collection of different variables under single name.

Basically Structure is for storing the complicated data.

A structure is a convenient way of grouping several pieces of related information together.

Definition of Structure in C

Structure is composition of the different variables of different data types, grouped under same name.

```
typedef struct {
        char name[64];
        char course[128];
        int age;
        int year;
} student;
```

Some Important Definitions of Structures

Each member declared in Structure is called **member.**

```
char name[64];
char course[128];
int age;
int year;
```

are some examples of members.

Name given to structure is called as **tag**

Structure **member** may be of **different data type** including **user defined data-type** also

```
typedef struct {
        char name[64];
        char course[128];
        book b1;
        int year;
  } student;
```

Here book is user defined data type.

Declaring Structure Variable in C

In C we can group some of the user defined or primitive data types together and form another compact way of storing complicated information is called as Structure. Let us see how to declare structure in c programming language –

Syntax of Structure in C Programming

```
struct tag
{
  data_type1 member1;
  data_type2 member2;
  data_type3 member3;
};
```

Structure Alternate Syntax

```
struct <structure_name>
{
   structure_Element1;
   structure_Element2;
```

```
    structure_Element3;
    ...
    ...
};
```

Some Important Points Regarding Structure in C Programming:

Struct keyword is used to declare structure.

Members of structure are enclosed within opening and closing braces.

Declaration of Structure reserves **no space**.

It is nothing but the " **Template / Map / Shape** " of the structure .

Memory is created, very first time when the **variable is created /Instance** is created.

Different Ways of Declaring Structure Variable:

Way 1 : Immediately after Structure Template

```
struct date
{
    int date;
    char month[20];
    int year;
}today;

// 'today' is name of Structure variable
```

Way 2 : Declare Variables using struct Keyword

```
struct date
{
    int date;
    char month[20];
```

```
    int year;
};
```

```
struct date today;
```

where "**date**" is name of structure and "**today**" is name of variable.

Way 3 : Declaring Multiple Structure Variables

```
struct Book
{
    int pages;
    char name[20];
    int year;

}book1,book2,book3;
```

C Structure Initialization

When we declare a structure, memory is not allocated for un-initialized variable.

Let us discuss very familiar example of structure student , we can initialize structure variable in different ways –

Way 1 : Declare and Initialize

```
struct student
{
   char name[20];
   int roll;
   float marks;
}std1 = { "Pritesh",67,78.3 };
```

In the above code snippet, we have seen that structure is declared and as soon as after declaration we have initialized the structure variable.

std1 = { "Pritesh",67,78.3 }

This is the code for initializing structure variable in C programming

Way 2 : Declaring and Initializing Multiple Variables

```
struct student
{
  char name[20];
  int roll;
  float marks;
}
```

```
std1 = {"Pritesh",67,78.3};
std2 = {"Don",62,71.3};
```

In this example, we have declared two structure variables in above code. After declaration of variable we have initialized two variable.

```
std1 = {"Pritesh",67,78.3};
std2 = {"Don",62,71.3};
```

Way 3 : Initializing Single member

```
struct student
{
  int mark1;
  int mark2;
  int mark3;
} sub1={67};
```

Though there are three members of structure,only one is initialized , Then remaining two members are initialized with Zero. If there are variables of other data type then their initial values will be –

Data Type	Default value if not initialized
integer	0
float	0.00
char	NULL

Way 4 : Initializing inside main

```
struct student
{
   int mark1;
   int mark2;
   int mark3;
};

void main()
{
struct student s1 = {89,54,65};
- - - - --
- - - - --
- - - - --
};
```

When we declare a structure then memory won't be allocated for the structure. i.e only writing below declaration statement will never allocate memory

```
struct student
{
```

```
  int mark1;
  int mark2;
  int mark3;
};
```

We need to initialize structure variable to allocate some memory to the structure.

```
struct student s1 = {89,54,65};
```

Some Structure Declarations and It's Meaning :

```
struct
{
  int length;
  char *name;
}*ptr;
```

Suppose we initialize these two structure members with following values –

length = 30;

*name = "programming";

Now Consider Following Declarations one by One –

Member	Value	Address
length	30	3000
name	programming	3002

Example 1 : Incrementing Member

++ptr->length

"++" Operator is **pre-increment operator**.

Above Statement will increase the value of "**length**"

Example 2 : Incrementing Member

(++ptr)->length

Content of the length is fetched and then ptr is incremented.

Consider above Structure and Look at the Following Table:-

Expression	Meaning
++ptr->length	Increment the value of length
(++ptr)->length	Increment ptr before accessing length
(ptr++)->length	Increment ptr after accessing length
*ptr->name	Fetch Content of name
*ptr->name++	Incrementing ptr after Fetching the value
(*ptr->name)++	Increments whatever str points to
*ptr++->name	Incrementing ptr after accessing whatever str points to

Accessing Structure Members

Array elements are accessed using the Subscript variable, Similarly Structure members are accessed using dot [.] operator.

(.) is called as "Structure member Operator".

Use this Operator in between **"Structure name"** & **"member name"**

Live Example :

#include<stdio.h>

```
struct Vehicle
  {
  int wheels;
  char vname[20];
  char color[10];
}v1 = {4,"Nano","Red"};

int main()
{
printf("Vehicle No of Wheels : %d",v1.wheels);
printf("Vehicle Name         : %s",v1.vname);
printf("Vehicle Color        : %s",v1.color);
return(0);
}
```

Output :

Vehicle No of Wheels : 4

Vehicle Name : Nano

Vehicle Color : Red

Note :

Dot operator has Highest Priority than unary, arithmetic, relational, logical Operators

Initializing Array of Structure in C Programming

Array elements are stored in consecutive memory Location.

Like Array , Array of Structure can be initialized at compile time.

Way1 : Initializing After Declaring Structure Array :

```
struct Book
{
   char bname[20];
   int pages;
   char author[20];
   float price;
}b1[3] = {
        {"Let us C",700,"YPK",300.00},
        {"Wings of Fire",500,"APJ Abdul Kalam",350.00},
        {"Complete C",1200,"Herbt Schildt",450.00}
       };
```

Explanation :

As soon as after declaration of structure we initialize structure with the pre-defined values. For each structure variable we specify set of values in curly braces. Suppose we have 3 Array Elements then we have to initialize each array element individually and all individual sets are combined to form single set.

{"Let us C",700,"YPK",300.00}

Above set of values are used to initialize first element of the array. Similarly –

{"Wings of Fire",500,"APJ Abdul Kalam",350.00}

is used to initialize second element of the array.

Way 2 : Initializing in Main

```
struct Book
  {
   char bname[20];
   int pages;
   char author[20];
```

Some Observations and Important Points:

Tip #1 : All Structure Members need not be initialized

```
#include<stdio.h>
struct Book
{
   char bname[20];
   int pages;
   char author[20];
   float price;
}b1[3] = {
        {"Book1",700,"YPK"},
        {"Book2",500,"AAK",350.00},
        {"Book3",120,"HST",450.00}
       };
```

```
void main()
{

printf("\nBook Name   : %s",b1[0].bname);
printf("\nBook Pages  : %d",b1[0].pages);
printf("\nBook Author : %s",b1[0].author);
printf("\nBook Price  : %f",b1[0].price);

}
```

Output :

Book Name : Book1

Book Pages : 700

Book Author : YPK

Book Price : 0.000000

Explanation :

In this example , While initializing first element of the array we have not specified the price of book 1.It is not mandatory to provide initialization for all the values. Suppose we have 5 structure elements and we provide initial values for first two element then we cannot provide initial values to remaining elements.

{"Book1",700,,90.00}

above initialization is illegal and can cause compile time error.

Tip #2 : Default Initial Value

```
struct Book
{
   char bname[20];
   int pages;
```

```
    char author[20];
    float price;
}b1[3] = {
        {},
        {"Book2",500,"AAK",350.00},
        {"Book3",120,"HST",450.00}
       };
```

Output :

```
Book Name   :
Book Pages  : 0
Book Author :
Book Price  : 0.000000
```

It is clear from above output , Default values for different data types.

Data Type	Default Initialization Value
Integer	0
Float	0.0000
Character	Blank

Passing Array of Structure to Function in C Programming

Array of Structure can be passed **to function as a Parameter**.

Function can also return Structure as **return type**.

Structure can be passed as follow

Live Example :

```
#include<stdio.h>
#include<conio.h>
//-------------------------------------
struct Example
{
 int num1;
 int num2;
}s[3];
//-------------------------------------
void accept(struct Example sptr[],int n)
{
 int i;
 for(i=0;i<n;i++)
 {
 printf("\nEnter num1 : ");
 scanf("%d",&sptr[i].num1);
 printf("\nEnter num2 : ");
 scanf("%d",&sptr[i].num2);
 }
}
//-------------------------------------
void print(struct Example sptr[],int n)
{
 int i;
 for(i=0;i<n;i++)
```

```
 {
 printf("\nNum1 : %d",sptr[i].num1);
 printf("\nNum2 : %d",sptr[i].num2);
 }
}
//-----------------------------------
void main()
{
int i;
clrscr();
accept(s,3);
print(s,3);
getch();
}
```

Output :

```
Enter num1 : 10
Enter num2 : 20
Enter num1 : 30
Enter num2 : 40
Enter num1 : 50
Enter num2 : 60
Num1 : 10
Num2 : 20
Num1 : 30
Num2 : 40
```

Num1 : 50

Num2 : 60

Explanation :

Inside main structure and size of structure array is passed.

When reference (i.e ampersand) is not specified in main , so this passing is simple pass by value.

Elements can be accessed by using dot [.] operator

Pointer Within Structure in C Programming:

Structure may contain the **Pointer variable as member**.

Pointers are used to store the address of memory location.

They can be **de-referenced** by '*' operator.

Example :

```
struct Sample
{
  int  *ptr;  //Stores address of integer Variable
  char *name; //Stores address of Character String
}s1;
```

s1 is structure variable which is used to access the **"structure members"**.

```
s1.ptr  = &num;
s1.name = "Pritesh"
```

Here **num** is any variable but it's address is stored in the Structure member ptr (**Pointer to Integer**)

Similarly Starting address of the String "Pritesh" is stored in structure variable name(**Pointer to Character array**)

Whenever we need to print the content of variable **num** , we are dereferancing the pointer variable num.

```
printf("Content of Num : %d ",*s1.ptr);
printf("Name : %s",s1.name);
```

Live Example : Pointer Within Structure

```
#include<stdio.h>

struct Student
{
  int  *ptr;  //Stores address of integer Variable
  char *name; //Stores address of Character String
}s1;

int main()
{

int roll = 20;
s1.ptr   = &roll;
s1.name  = "Pritesh";

printf("\nRoll Number of Student : %d",*s1.ptr);
printf("\nName of Student        : %s",s1.name);

return(0);
}
```

Output :

```
Roll Number of Student : 20
```

Name of Student : Pritesh

Some Important Observations :

printf("\nRoll Number of Student : %d",*s1.ptr);

We have stored the address of variable 'roll' in a pointer member of structure thus we can access value of pointer member directly using de-reference operator.

printf("\nName of Student : %s",s1.name);

Similarly we have stored the base address of string to pointer variable 'name'. In order to de-reference a string we never use de-reference operator.

Array of Structure :

Structure is used to store the information of One particular object but if we need to store such 100 objects then Array of Structure is used.

Example :

```
struct Bookinfo
{
    char[20] bname;
    int pages;
    int price;
}Book[100];
```

Explanation :

Here Book structure is used to Store the information of one Book.

In case if we need to store the Information of 100 books then Array of Structure is used.

b1[0] stores the Information of 1st Book , b1[1] stores the information of 2nd Book and So on We can store the information of 100 books.

book[3] is shown Below

Accessing Pages field of Second Book :

Book[1].pages

Live Example :

```
#include <stdio.h>

struct Bookinfo
{
    char[20] bname;
    int pages;
    int price;
}book[3];

int main(int argc, char *argv[])
{
int i;

for(i=0;i<3;i++)
   {
   printf("\nEnter the Name of Book    : ");
   gets(book[i].bname);
   printf("\nEnter the Number of Pages : ");
```

```
    scanf("%d",book[i].pages);
    printf("\nEnter the Price of Book   : ");
    scanf("%f",book[i].price);
    }
printf("\n--------- Book Details ------------ ");
    for(i=0;i<3;i++)
    {
    printf("\nName of Book    : %s",book[i].bname);
    printf("\nNumber of Pages : %d",book[i].pages);
    printf("\nPrice of Book   : %f",book[i].price);
    }
return 0;
}
```

Output of the Structure Example:

```
Enter the Name of Book    : ABC
Enter the Number of Pages : 100
Enter the Price of Book   : 200
Enter the Name of Book    : EFG
Enter the Number of Pages : 200
Enter the Price of Book   : 300
Enter the Name of Book    : HIJ
Enter the Number of Pages : 300
Enter the Price of Book   : 500

--------- Book Details ------------
```

Name of Book : ABC

Number of Pages : 100

Price of Book : 200

Name of Book : EFG

Number of Pages : 200

Price of Book : 300

Name of Book : HIJ

Number of Pages : 300

Price of Book : 500

Union in C Programming :

In C Programming we have came across Structures. Unions are similar tostructure syntactically.Syntax of both is almost similar. Let us discuss some important points one by one –

Note #1 : **Union and Structure are Almost Similar**

```
union stud
{
  int roll;
  char name[4];
  int marks;
}s1;
```

```
struct stud
{
  int roll;
  char name[4];
  int marks;
}s1;
```

If we look at the two examples then we can say that both structure and union are same except Keyword.

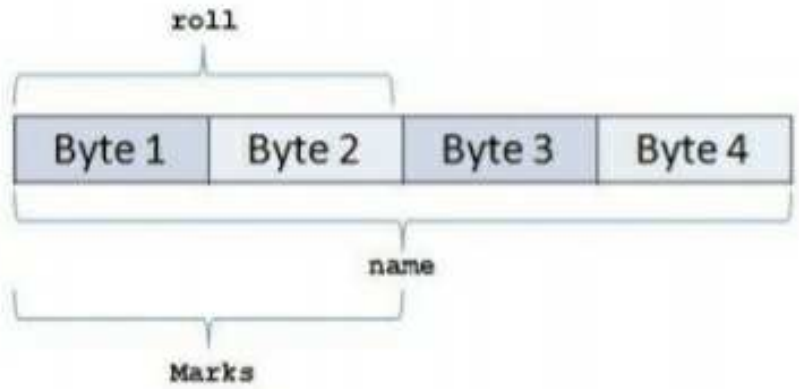
Note #2 : Multiple Members are Collected Together Under Same Name

```
int roll;
char name[4];
```

int marks;

We have collected three variables of different data type under same name together.

Note #3 : All Union Members Occupy Same Memory Area



For the union maximum memory allocated will be equal to the data member with maximum size. In the example character array 'name' have maximum size thus maximum memory of the union will be 4 Bytes.

Maximum Memory of Union = Maximum Memory of Union

Data Member

Note #4 : Only one Member will be active at a time.

Suppose we are accessing one of the data member of union then we cannot access other data member since we can access single data member of union because each data member shares same memory. By Using Union we can **Save Lot of Valuable Space**

Simple Example:

union u

{

char a;

int b;

}

How to Declare Union in C ?

Union is similar to that of Structure. Syntax of both are same but major difference between structure and union is '**memory storage**'.

In structures, **each member has its own storage location**, whereas all the members of union use the same location. Union contains many members of different types,

Union can handle **only one member at a time**.

Syntax :

```
union tag
{
  union_member1;
  union_member2;
  union_member3;
  ..
  ..
  ..
  union_memberN;
}instance;
```

Note :

Unions are Declared in the same way as a Structure.Only "**struct Keyword**" is replaced with **union**

Sample Declaration of Union :

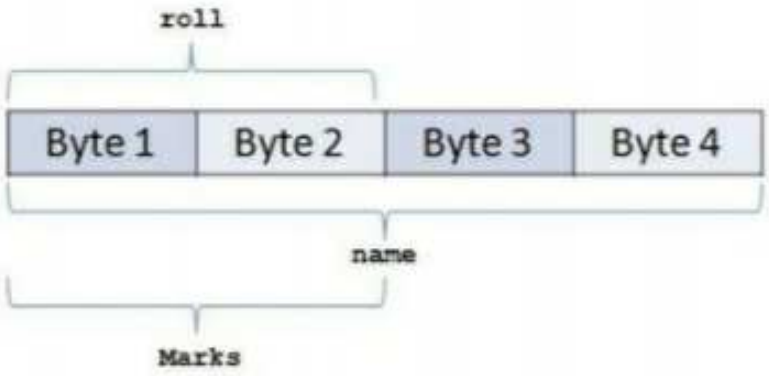
```
union stud
{
  int roll;
  char name[4];
  int marks;
}s1;<
```

```
union stud
{
int roll;
char name[4];
int marks;
}s1;
```

www.c4learn.com

Member	Memory Required
Roll	2
Name	4
Marks	2

How Memory is Allocated ?



So From the Above fig. We can Conclude –

Union Members that compose a union, **all share the same storage area within the computers memory**

Each member within a structure is assigned its own **unique storage area**

Thus unions are used to observe memory.

Unions are useful for **application involving multiple members**, where values need not be assigned to all the members at any one time.

C Programming accessing union members

While accessing union, we can have access to single data member at a time. we can access single union member using following two Operators –

Using DOT Operator

Using ARROW Operator

Accessing union members DOT operator

In order to access the member of the union we are using the dot operator. DOT operator is used inside printf and scanf statement to get/set value from/of union member location.

Syntax :

variable_name.member

consider the below union, when we declare a variable of union type then we will be accessing union members using dot operator.

```
union emp
{
int id;
char name[20];
}e1;
```

id can be Accessed by – union_variable.member

Syntax	Explanation
e1.id	Access id field of union
e1.name	Access name field of union

Accessing union members Arrow operator

Instead of maintain the union variable suppose we store union at particular address then we can access the members of the union using pointer to the union and arrow operator.

```
union emp
{
int id;
char name[20];
}*e1;
```

id can be Accessed by – union_variable->member

Syntax	Explanation
e1->id	Access id field of union
e1->name	Access name field of union

C Programs

Program #1 : Using dot operator

```
#include <stdio.h>
union emp
{
 int id;
 char name[20];
}e1;
int main(int argc, char *argv[])
{
   e1.id = 10;
   printf("\nID   : %d",e1.id);
   strcpy(e1.name,"Pritesh");
   printf("\nName : %s",e1.name);
   return 0;
}
```

Output :

ID : 10

Name : Pritesh

Program #2 : Accessing same memory

```
#include <stdio.h>

union emp

{

 int id;

 char name[20];

}e1;

int main(int argc, char *argv[])

{

   e1.id = 10;

   strcpy(e1.name,"Pritesh");

   printf("\nID   : %d",e1.id);

   printf("\nName : %s",e1.name);

   return 0;

}
```

Output :

```
ID   : 1953067600

Name : Pritesh
```

As we already discussed in the previous article of union basics, we have seen how memory is shared by all union fields. In the above example –

Total memory for union = max(sizeof(id),sizeof(name))

= sizeof(name)

= 20 bytes

Firstly we have utilized first two bytes out of 20 bytes for storing integer value. After execution of statement again same memory is overridden by character array so while printing the ID value, garbage value gets printed

Program #3 : Using arrow operator

```
#include <stdio.h>
union emp
{
 int id;
 char name[20];
}*e1;
int main(int argc, char *argv[])
{
   e1->id = 10;
   printf("\nID   : %d",e1->id);
   strcpy(e1->name,"Pritesh");
   printf("\nName : %s",e1->name);
   return 0;
}
```

Output :

```
ID   : 10
Name : Pritesh
```

Bit fiels:

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows −

```
struct {
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows −

```
struct {
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status;
```

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept −

```
#include <stdio.h>
#include <string.h>

/* define simple structure */
struct {
  unsigned int widthValidated;
  unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
struct {
  unsigned int widthValidated : 1;
```

```
  unsigned int heightValidated : 1;
} status2;
int main( ) {
  printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
  printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

The declaration of a bit-field has the following form inside a structure −

```
struct {
  type [member_name] : width ;
};
```

The following table describes the variable elements of a bit field −

Elements	Description
type	An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
member_name	The name of the bit-field.
width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows −

```
struct {
  unsigned int age : 3;
} Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example −

```
#include <stdio.h>
#include <string.h>
struct {
  unsigned int age : 3;
} Age;
int main( ) {
  Age.age = 4;
  printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
  printf( "Age.age : %d\n", Age.age );
  Age.age = 7;
  printf( "Age.age : %d\n", Age.age );
  Age.age = 8;
  printf( "Age.age : %d\n", Age.age );
  return 0;
}
```

When the above code is compiled it will compile with a warning and when executed, it produces the following result −

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
```

Age.age : 0

Typedef:

The C programming language provides a keyword called **typedef**, which you can use to give a type, a new name. Following is an example to define a term**BYTE** for one-byte numbers −

typedef unsigned char BYTE;

After this type definition, the identifier BYTE can be used as an abbreviation for the type **unsigned char, for example.**.

BYTE b1, b2;

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows −

typedef unsigned char byte;

You can use **typedef** to give a name to your user defined data types as well. For example, you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows −

```
#include <stdio.h>

#include <string.h>

 typedef struct Books {

   char title[50];

   char author[50];

   char subject[100];

   int book_id;

} Book;

 int main( ) {

   Book book;

    strcpy( book.title, "C Programming");

   strcpy( book.author, "Nuha Ali");

   strcpy( book.subject, "C Programming Tutorial");
```

```
  book.book_id = 6495407;

   printf( "Book title : %s\n", book.title);

  printf( "Book author : %s\n", book.author);

  printf( "Book subject : %s\n", book.subject);

  printf( "Book book_id : %d\n", book.book_id);

  return 0;

}
```

When the above code is compiled and executed, it produces the following result −

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book_id : 6495407

typedef vs #define

#define is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences −

typedef is limited to giving symbolic names to types only where as**#define** can be used to define alias for values as well, q., you can define 1 as ONE etc.

typedef interpretation is performed by the compiler whereas **#define**statements are processed by the pre-processor.

The following example shows how to use #define in a program −

```
#include <stdio.h>

 #define TRUE  1

#define FALSE 0

 int main( ) {

  printf( "Value of TRUE : %d\n", TRUE);

  printf( "Value of FALSE : %d\n", FALSE);
```

```
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of TRUE : 1
Value of FALSE : 0
```

Enumerated data type:

An enumeration is a user-defined data type consists of integral constants and each integral constant is give a name. Keyword **enum** is used to defined enumerated data type.

```
enum type_name{ value1, value2,...,valueN };
```

Here, ***type_name*** is the name of enumerated data type or tag. And ***value1***,***value2***,.....,***valueN*** are values of type ***type_name***.

By default, ***value1*** will be equal to 0, ***value2*** will be 1 and so on but, the programmer can change the default value.

```
// Changing the default value of enum elements
enum suit{
  club=0;
  diamonds=10;
  hearts=20;
  spades=3;
};
```

Declaration of enumerated variable

Above code defines the type of the data but, no any variable is created. Variable of type **enum** can be created as:

```
enum boolean{
  false;
  true;
```

```
};
enum boolean check;
```

Here, a variable check is declared which is of type **enum boolean**.

Example of enumerated type

```
#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};
int main(){
   enum week today;
   today=wednesday;
   printf("%d day",today+1);
   return 0;
  }
```

Output

4 day

You can write any program in C language without the help of enumerations but, enumerations helps in writing clear codes and simplify programming.

Dynamic memory allocation

The exact size of array is unknown untill the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under **"stdlib.h"** for dynamic memory allocation.

Function	Use of Function

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

The name malloc stands for "memory allocation". The function **malloc()**reserves a block of memory of specified size and return a pointer of type **void**which can be casted into pointer of any form.

Syntax of malloc()

ptr=(cast-type*)malloc(byte-size)

Here, ***ptr*** is pointer of cast-type. The **malloc()** function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

ptr=(int*)malloc(100*sizeof(int));

This statement will allocate either 200 or 400 according to size of **int** 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

ptr=(cast-type*)calloc(n,element-size);

This statement will allocate contiguous space in memory for an array of ***n***elements. For example:

ptr=(float*)calloc(25,sizeof(float));

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

free(ptr);

This statement cause the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
```

```
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
  }
  printf("Sum=%d",sum);
  free(ptr);
  return 0;
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int n,i,*ptr,sum=0;
  printf("Enter number of elements: ");
  scanf("%d",&n);
  ptr=(int*)calloc(n,sizeof(int));
  if(ptr==NULL)
  {
    printf("Error! memory not allocated.");
    exit(0);
  }
  printf("Enter elements of array: ");
  for(i=0;i<n;++i)
  {
    scanf("%d",ptr+i);
```

```
    sum+=*(ptr+i);
  }
  printf("Sum=%d",sum);
  free(ptr);
  return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, ***ptr*** is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int *ptr,i,n1,n2;
  printf("Enter size of array: ");
  scanf("%d",&n1);
  ptr=(int*)malloc(n1*sizeof(int));
  printf("Address of previously allocated memory: ");
  for(i=0;i<n1;++i)
     printf("%u\t",ptr+i);
  printf("\nEnter new size of array: ");
  scanf("%d",&n2);
  ptr=realloc(ptr,n2);
```

```
    for(i=0;i<n2;++i)
        printf("%u\t",ptr+i);
    return 0;
}
```