UNIT -4Syllabus:
Derived types – Structures – Basic Concepts, Nested structures, arrays of structures, structures and functions ,Unions, bit fields, C Programming examples.
Pointers- Basic concepts, pointers and functions, pointers and strings, pointers and arrays, pointers and structures, self-referential structures, example C programs.

1. Structures

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

1.1.Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

```
structstructure_name{
data_type struct_member1;
data_type struct_member2;
-----------------
data_typestruct_member n;
} [one or more structure variables separated by comma];
```

The **struct_name** is optional and each *struct_member* is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.
Here is the way you would declare the Book structure –

```
struct Books
{
        char title[50];
        char author[50];
        char subject[100];
        intbook_id;
}book;
```

1.2 Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program −

```
#include <stdio.h>
#include <string.h>
struct Books {
char  title[50];
char  author[50];
char  subject[100];
intbook_id;
};

int main( ) {

struct Books Book1;        /* Declare Book1 of type Book */
struct Books Book2;        /* Declare Book2 of type Book */
   /* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;
   /* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;
   /* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);
   /* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);
return 0;
}
```

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

```
Book 1 title : C Programming
Book 1 author :Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

2. Nested structures

Structure written inside another structure is called as nesting of two structures.Nested Structures are allowed in C Programming Language.We can write one Structure inside another structure as member of another structure.

Example:

```
struct date
{
int date;
int month;
int year;
};

struct Employee
  {
charename[20];
intssn;
float salary;
struct date doj;
}emp1;
```

2.1Accessing the nested elements

- Structure members are accessed using dot operator.
- ‘date‘ structure is nested within Employee Structure.
- Members of the ‘date‘ can be accessed using ’employee’
- emp1 &doj are two structure names (Variables)

2.2 Explanation of nested elements:

```
Accessing Month Field :  emp1.doj.month
Accessing day Field   :  emp1.doj.day
Accessing year Field  :  emp1.doj.year
```

2.3 Example

```
#include <stdio.h>
struct Employee
{
```

```
charename[20];
intssn;
float salary;
struct date
       {
int date;
int month;
int year;
}doj;
}emp = {"Pritesh",1000,1000.50,{22,6,1990}};

int main(intargc, char *argv[])
{
printf("\nEmployee Name   : %s",emp.ename);
printf("\nEmployee SSN    : %d",emp.ssn);
printf("\nEmployee Salary : %f",emp.salary);
printf("\nEmployee DOJ    : %d/%d/%d", \
emp.doj.date,emp.doj.month,emp.doj.year);

return 0;
}
```

Output:

```
Employee Name   : Pritesh
Employee SSN    : 1000
Employee Salary : 1000.500000
Employee DOJ    : 22/6/1990
```

3. Arrays of Structures:

We may declare the structure variable as an array by using index. Arrays of structures are passed using the following syntax,

```
structstructure_name{
data_type struct_member1;
data_type struct_member2;
-----------------
data_typestruct_member n;
} ;
structstructure_namestruct_variable[index_value];
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        struct student
        {
                intrno;
                char name[20];
                char course[20];
                float fees;
        };
        struct student s[20];
        clrscr();
        s[0].rno=10;
        gets(s[0].name);
        gets(s[0].course);
        s[0].fees=1000.50;
        printf("\n %d\t %s\t%s\t%f",s[0].rno,s[0].name,s[0].course,s[0].fees);
        getch();
}
```

4.Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

Program:

```
#include <stdio.h>
#include <string.h>
struct Books {
char  title[50];
char  author[50];
char  subject[100];
intbook_id;
};
/* function declaration */
voidprintBook( struct Books book );
int main( ) {
struct Books Book1;       /* Declare Book1 of type Book */
struct Books Book2;       /* Declare Book2 of type Book */
  /* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;
  /* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;
  /* print Book1 info */
printBook( Book1 );
  /* Print Book2 info */
printBook( Book2 );
return 0;
}
voidprintBook( struct Books book ) {
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);
}
```

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
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

5. Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable –

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows –

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the → operator as follows –

```
struct_pointer->title;
```

Let us re-write the above example using structure pointer.

```
#include <stdio.h>
#include <string.h>
struct Books {
char  title[50];
char  author[50];
char  subject[100];
intbook_id;
};
/* function declaration */
voidprintBook( struct Books *book );
int main( ) {
struct Books Book1;        /* Declare Book1 of type Book */
struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info by passing address of Book1 */
printBook(&Book1 );
   /* print Book2 info by passing address of Book2 */
printBook(&Book2 );
return 0;
}
voidprintBook( struct Books *book ) {
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);
}
```

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
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

6. Unions

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

6.1 Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows –

```
union [union tag] {
member definition;
member definition;
   ...
member definition;
} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str–

```
union Data {
int i;
float f;
charstr[20];
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union –

```
#include <stdio.h>
#include <string.h>
union Data {
int i;
float f;
charstr[20];
};

int main( ) {
union Data data;
printf( "Memory size occupied by data : %d\n", sizeof(data));
return 0;
}
```

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

```
Memory size occupied by data : 20
```

6.2 Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program −

```
#include <stdio.h>
#include <string.h>
union Data {
int i;
float f;
charstr[20];
};
int main( ) {
union Data data;
data.i = 10;
data.f = 220.5;
strcpy(data.str, "C Programming");
printf( "data.i : %d\n", data.i);
printf( "data.f : %f\n", data.f);
printf( "data.str : %s\n", data.str);
return 0;
}
```

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

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions –

```
#include <stdio.h>
#include <string.h>
union Data {
int i;
float f;
charstr[20];
};
int main( ) {
union Data data;
data.i = 10;
printf( "data.i : %d\n", data.i);
data.f = 220.5;
printf( "data.f : %f\n", data.f);
strcpy(data.str, "C Programming");
printf( "data.str : %s\n", data.str);
return 0;
}
```

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

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

7. Bit Fields

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

```
struct {
unsignedintwidthValidated;
unsignedintheightValidated;
} 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 {
unsignedintwidthValidated : 1;
unsignedintheightValidated : 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 {
unsignedintwidthValidated;
unsignedintheightValidated;
} status1;
/* define a structure with bit fields */
struct {
unsignedintwidthValidated : 1;
unsignedintheightValidated : 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
```

7.1 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 {
unsignedint 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 {
unsignedint 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
```

8. Pointers

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined –

```
#include <stdio.h>
int main () {
int  var1;
char var2[10];
printf("Address of var1 variable: %x\n", &var1  );
printf("Address of var2 variable: %x\n", &var2  );
return 0;
}
```

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

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

8.1What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

8.2 How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations –

```
#include <stdio.h>
int main () {
intvar = 20;   /* actual variable declaration */
int  *ip;        /* pointer variable declaration */
ip = &var;  /* store address of var in pointer variable*/
printf("Address of var variable: %x\n", &var);
   /* address stored in pointer variable */
printf("Address stored in ip variable: %x\n", ip );
   /* access the value using the pointer */
printf("Value of *ip variable: %d\n", *ip );
return 0;
}
```

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

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

8.3 NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>
int main () {
int  *ptr = NULL;
printf("The value of ptr is : %x\n", ptr  );
return 0;
}
```

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

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

8.4 Pointer arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer –

```
ptr++
```

After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If**ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

```
#include <stdio.h>
constint MAX = 3;
int main () {
intvar[] = {10, 100, 200};
int  i, *ptr;
  /* let us have array address in pointer */
ptr = var;
for ( i = 0; i < MAX; i++) {

printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
     /* move to the next location */
ptr++;
  }
return 0;
}
```

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

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

9. Pointers and Functions

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function –

```
#include <stdio.h>
#include <time.h>
voidgetSeconds(unsigned long *par);
int main () {
unsigned long sec;
getSeconds(&sec );
   /* print the actual value */
printf("Number of seconds: %ld\n", sec );
return 0;
}
voidgetSeconds(unsigned long *par) {
   /* get the current number of seconds */
   *par = time( NULL );
return;
}
```

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

```
Number of seconds :1294450468
```

The function, which can accept a pointer, can also accept an array as shown in the following example –

```
#include <stdio.h>
/* function declaration */
doublegetAverage(int *arr, int size);
int main () {
   /* an int array with 5 elements */
int balance[5] = {1000, 2, 3, 17, 50};
doubleavg;
```

```
   /* pass pointer to the array as an argument */
avg = getAverage( balance, 5 ) ;
   /* output the returned value  */
printf("Average value is: %f\n", avg );
return 0;
}
doublegetAverage(int *arr, int size) {
int  i, sum = 0;
doubleavg;
for (i = 0; i < size; ++i) {
sum += arr[i];
   }
avg = (double)sum / size;
returnavg;
}
```

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

```
Average value is: 214.40000
```

10. Pointers and Arrays

Before we understand the concept of arrays of pointers, let us consider the following example, which uses an array of 3 integers –

```
#include <stdio.h>
constint MAX = 3;
int main () {
intvar[] = {10, 100, 200};
int i;
for (i = 0; i < MAX; i++) {
printf("Value of var[%d] = %d\n", i, var[i] );
   }
return 0;
}
```

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

```
Value of var[0] = 10
Value of var[1] = 100
```

Value of var[2] = 200There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer −

```
int *ptr[MAX];
```

It declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows −

```
#include <stdio.h>
constint MAX = 3;
int main () {
intvar[] = {10, 100, 200};
int i, *ptr[MAX];
for ( i = 0; i < MAX; i++) {
ptr[i] = &var[i]; /* assign the address of integer. */
   }
for ( i = 0; i < MAX; i++) {
printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }
return 0;
}
```

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

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

11. Pointers and Strings

You can also use an array of pointers to character to store a list of strings as follows –

```
#include <stdio.h>
constint MAX = 4;
int main () {
char *names[] = {
      "Zara Ali",
      "Hina Ali",
      "Nuha Ali",
      "Sara Ali",
   };
int i = 0;
for ( i = 0; i < MAX; i++) {
printf("Value of names[%d] = %s\n", i, names[i] );
   }
return 0;
}
```

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

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

12. Self – referential structures

A self-referential structure is used to create data structures like linked lists, stacks, etc. Following is an example of this kind of structure:

```
structstruct_name
{
datatypedatatypename;
struct_name * pointer_name;
};
```

A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type. For example, a linked list is supposed to be a self-referential data structure. The next node of a node is being pointed, which is of the same struct type. For example,

```
typedefstructlistnode {
void *data;
structlistnode *next;
} linked_list;
```

In the above example, the listnode is a self-referential structure – because the *next is of the type structlistnode.