Unit 6 Structures, Unions & Pointers

Structure:

6.1 Introduction

A structure type is a type defined within the program that specifies the format of a record, including the names and types of its members, and the order in which they are stored. Once you have defined a structure type, you can use it like any other type in declaring objects, pointers to those objects, and arrays of such structure elements.

Unlike structure members, which all have distinct locations in the structure, the members of a union all share the same location in memory; that is, all members of a union start at the same address. Thus you can define a union with many members, but only one member can contain a value at any given time. Unions are an easy way for programmers to use a location in memory in different ways.

C supports the use of pointers, a very simple type of reference that records, in effect, the address or location of an object or function in memory. Pointers can be used to access data stored at the address pointed to, or to invoke a pointed-to function. Pointers can be manipulated using assignment and also pointer arithmetic. The run-time representation of a pointer value is typically a raw memory address (perhaps augmented by an offset-within-word field), but since a pointer's type includes the type of the thing pointed to, expressions including pointers can be type-checked at compile time. Pointer arithmetic is automatically scaled by the size of the pointed-to data type. Pointers are used for many different purposes in C. Text strings are commonly manipulated using pointers into arrays of characters. Dynamic memory allocation is performed using pointers.

A null pointer is a pointer value that points to no valid location (it is often represented by address zero). Dereferencing a null pointer is therefore meaningless, typically resulting in a run-time error. Null pointers are useful for indicating special cases such as no next pointer in the final node of a linked list, or as an error indication from functions returning pointers.

Objectives:

At the end of this unit, you will be able to:

- explain the general form of structure data type and its usage.
- explain pointer data type and its importance in problem solving.
- describes the application of pointers and structures with examples.
- discuss dynamic memory allocation schemes.

6.2 Structures

Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type. If we need to use a collection of different data type items it

is not possible using an array. When we require using a collection of different data items of different data types we can use a structure. Structure is a method of packing data of different types. A structure is a convenient method of handling a group of related data items of different data types. In C, the structure is declared with the reserved word **struct**.

Declaration of structure

Each and every structure must be defined or declared before it appears in the program. It does not occupy any memory space.

Syntax: **struct** <structurer name>

```
{
   <type1> <field/data1>
   <type2> <field/data2>
   <type3> <field/data3>
  ……….
  ……….
  <type n> <field/data n>
};
```

Syntax to declare structure type variable:

struct <structure name> <variable name>;

Example 1:

```
struct student
{
    int roll;
    char name[30];
    char address[30];
    char city[15];
    int marks;
};
struct student stu;
```

Initialization of structure

Initializing a structure data type, while declaration, is similar to that of a static data type.

Example 2:

struct student rec ={10,"Avinaba", "Ray Appartment.","Bangalore",560};

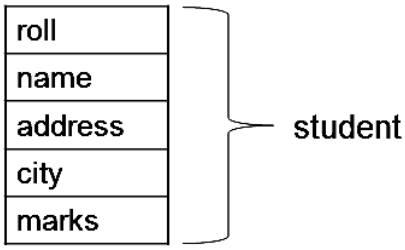
Sometimes we can create structure type variable with the keyword **typedef**

Syntax: **typedef struct** <structurer name>

```
{
   <type1> <field/data1>
   <type2> <field/data2>
   <type3> <field/data3>
  ……….
  ……….
  <type n> <field/data n>
}<variable>;
```

Example 3:

```
struct student
{
      int roll;
      char name[30];
      char address[30];
      char city[15];
      int marks;
};
typedef struct student rec;
```



Here, **student** is a structure data type and **rec** is a user-defined variable of type student that can be used to declare other variables of **student** type.

Dot (.) operator is used to access the member of a structure.

Example:

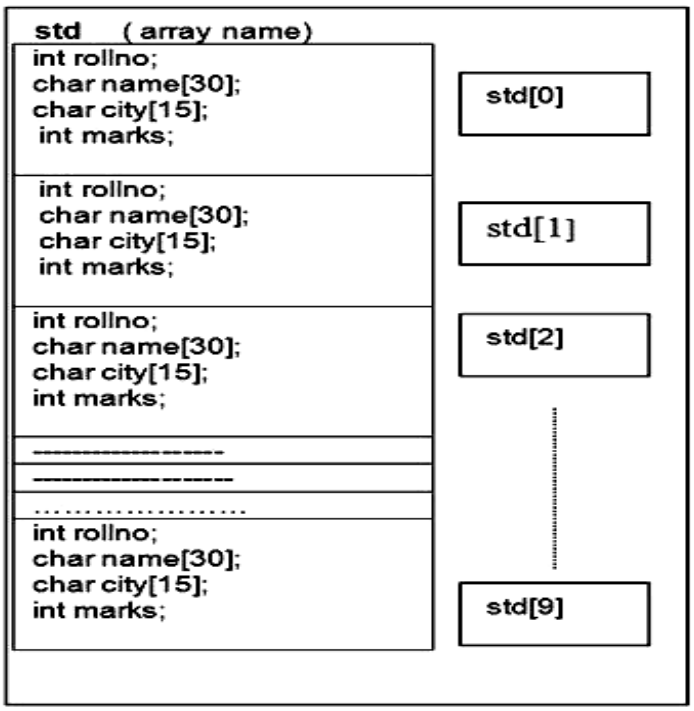
```
rec stu;
stu.roll=10;
```

6.2.1 Array of Structures

It is possible to define a array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college. We need to use an array than single variables. We can define an array of structures as shown in the following example.

Example 4:

```
struct student
 {
        int rollno;
        char name[30];
        char city[15];
        int marks;
} std[100];  /* Array defining */
```



Examples 5:

Q1) Write a program that prints a result sheet after inputting marks.

[File Name u6q1.c]

Solution:

```
/*************************************************************/
/*      PROGRAM TO PRINT THE RESULT SHEET USING STRUCTURE
*/
/*************************************************************/

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
typedef struct
{                   /* Delcearation of Structure  */
        char name[30];
        int roll;
        int eng;
        int math;
        int tot;
        char grd;
}rec;

void main()
{
void heading(void);
rec std[50];
int i,n,count=0;
char ch, ch1;
```

```
do{
        i=0;
        clrscr();
        printf("\t--------------------MAIN MENU-------------------\n");
        printf("\t1-Add Records\n");
        printf("\t2-List Records\n");
        printf("\t3-Exit Program\n");
        printf("\tYour choice please.....(1,2,3)  ");
        ch=getche();
        switch(ch)
        {
        case '1':
            i=count;
            do{
              clrscr();
              printf("Enter name :  ");       fflush(stdin);   gets(std[i].name);
              printf("Enter roll :  ");              scanf("%d",&std[i].roll);
              printf("Enter marks in Mathematics  : ");
              scanf("%d",&std[i].math);
    printf("Enter marks in English  : ");     scanf("%d",&std[i].eng);
              std[i].tot=std[i].math+std[i].eng;
            if (std[i].tot<60)
        std[i].grd='F';
            else if (std[i].tot<90)
        std[i].grd='C';
            else if (std[i].tot<120)
        std[i].grd='B';
             else if (std[i].tot<150)
        std[i].grd='A';
```

```
            else
          std[i].grd='*';
               i++;
               printf("\n\n Continue...(Y/N)");
               ch1=toupper(getch());
              }while(ch1=='Y');
            count=i;
          break;
           case '2':
           i=0;
            heading();
          do{
          printf("%d   %-30s     %5d     %5d     %5d     %5d          %c\n",i+1,
std[i].name,std[i].roll,std[i].eng,std[i].math,std[i].tot,std[i].grd);
          i++;
          }while(i<count);
          getch();
          break;
case '3':
          exit(0);
          break;
default:
          printf("\nInvalid choice.....");
          getch();
          }
          }while(1);
}
/*********************************************************/
void heading()
```

```
{
clrscr();
printf("\t\t\t\t List of Students\n");
printf("Sl-Name------------------------------Roll-----Eng-----Math---Total----
Grade\n");
}
/*******************************************************/
```

6.3 Union

A "union declaration" specifies a set of variable values and, optionally, a tag naming the union. The variable values are called "members" of the union and can have different types. Unions are similar to "variant records" in other languages.

A union type declaration is a template only. Memory is not reserved until the variable is declared.

If a union of two types is declared and one value is stored, but the union is accessed with the other type, the results are unreliable.

For example, a union of **float** and **int** is declared. A **float** value is stored, but the program later accesses the value as an **int**. In such a situation, the value would depend on the internal storage of **float** values. The integer value would not be reliable.

6.3.1 Defining Union Types

The definition of a union is formally the same as that of a structure, except for the keyword **union** in place of **struct**:

union [*tag_name*] { *member_declaration_list* };

The following example defines a union type named Data which has the three members i, x, and str:

```
union Data {
        int i;
        double x;
        char str[16];
              };
```

An object of this type can store an integer, a floating-point number, or a short string.

union Data var, myData[100];

This declaration defines **var** as an object of type union Data, and **myData** as an array of 100 elements of type union Data. A union is at least as big as its largest member. To obtain the size of a union, use the sizeof operator. Using our example, sizeof(var) yields the value 16, and sizeof(myData) yields 1,600.

All the members of a union begin at the same address in memory.

An object of the type union Data in memory

6.3.2 Initializing Unions

Like structures, union objects are initialized by an initialization list. For a union, though, the list can only contain one initializer. As for structures, C99 allows the use of a member designator in the initializer to indicate which member of the union is being initialized. Furthermore, if the initializer has no member designator, then it is associated with the first member of the union. A union object with automatic storage class can also be initialized with an existing object of the same type. Some examples:

```
union Data var1 = { 77 },
             var2 = { str = "Mary" },
             var3 = var1,
             myData[100] = { {x= 0.5}, { 1 }, var2 };
```

The array elements of myData for which no initializer is specified are implicitly initialized to the value 0.

You can access the members of a union in the same ways as structure members. The only difference is that when you change the value of a union member, you modify all the members of the union. Here are a few examples using the union objects var and myData:

```
var.x = 3.21;
var.x += 0.5;
strcpy( var.str, "Jim" );        // Occupies the place of var.x.
myData[0].i = 50;
for ( int i = 0; i < 50; ++i )
  myData[i].i = 2 * i;
```

As for structures, the members of each union type form a name space unto themselves. Hence in the last of these statements, the index variable i and the union member i identify two distinct objects.

You the programmer are responsible for making sure that the momentary contents of a union object are interpreted correctly. The different types of the union's members allow you to interpret the same collection of byte values in different ways. For example, the following loop uses a union to illustrate the storage of a double value in memory:

```
var.x = 1.25;
for ( int i = sizeof(double) - 1; i >= 0; --i )
  printf( "%02X ", (unsigned char)var.str[i] );
```

This loop begins with the highest byte of var.x, and generates the following output:

3F F4 00 00 00 00 00 00

6.4 Pointers

Pointer is a special type of variable which is used to hold the address of variable. If we use the pointer to operate the variable, then it is quite faster than the ordinary operation. Moreover uses of pointer minimize the complexity of the program. There are two types of symbols which are used to handle the pointer, & and *. Like a postcard, a pointer holds an address of a memory location.

Some points must be considered while using pointers

a) One pointer can hold only one address at a time.
b) Before use, pointer must be defined.
c) At the time of definition, the symbol * must be used as prefix.
d) Pointer holds only address, not the value.
e) Pointer when used in program after definition, then
   i) * means ‘value of’
   ii) & means ‘address of’
f) Pointer arithmetic is possible for addition and subtraction, but it has no meaning of multiplication and division.

The following example illustrates the above aspects.

```
int a,b,*p,*q;

a=10; b=25;

p=&a; q=&b; [address of a is assigned to p, address of b assigned to q]
```

Expression	Output
a	10
b	25
&a	100
&b	200
p	100
q	200
&p	500
&q	300
&(&a)	500
&(&b)	300
*(&a)	10
*(&b)	25

Example 6: Program to add of two numbers using pointer

```
void main()
{
        int a,b,*p,*q,c;
        p=&a; q=&b;
        printf (“Enter two numbers : “);
        scanf(“%d %d”,p,q);
        c=*p+*q;
        printf (“The sum is = %d”,c);
}
```

6.4.1 Dynamic Allocation of Memory

C Language requires the number of elements in an array to be specified at compile time. Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is called ***dynamic allocation of memory***. Language C has memory management functions so that memory can be allocated dynamically.

The three different functions used in dynamic memory allocation are:

a) malloc(), b) calloc(), c) realloc()

malloc(): Allocates request size of bytes and returns a pointer to the first byte of the allocated space. In the other sense, this function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer.

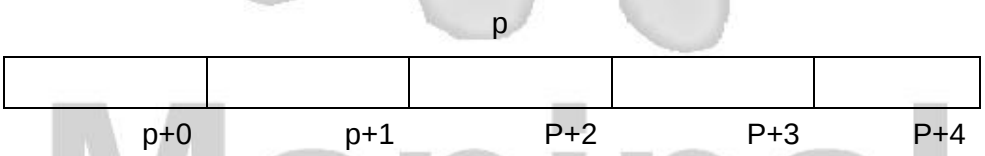
Syntax:

<pointer>=(type_cast *) malloc (size in bytes);

Example 7:

int n=5;

int *p= (int *) malloc (n*sizeof(int));



calloc(): Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory. In other sense, this function allocates a single block of storage spaces, calloc() allocates multiple blocks of storages, each of the same size, and then sets all the bytes to zero.

Syntax:

<pointer>=(type_cast *) calloc (n,size in bytes in each row);

Example 8:

```
int n=4,m=5,*p;
p=(int*) calloc(n,m*sizeof(int));
```

p+0+0	p+0+1	p+0+2	p+0+3	p+0+4
p+1+0	p+1+1	p+1+2	p+1+3	p+1+4
p+2+0	p+2+1	p+2+2	p+2+3	p+2+4
p+3+0	p+3+1	p+3+2	p+3+3	p+3+4

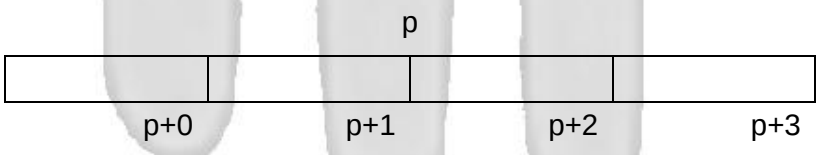
realloc(): Modifies the size of previously allocated space

Syntax:

<pointer>=(data type) realloc (new size in bytes);

Example 9:

```
int n=4,*p;
p=(int*) realloc(n*sizeof(int));
```



In case of realloc() function, the memory allocation can be changed i.e. increased or decreased. The previous value will preserve, if the new size is greater than the previous one.

We can use "→" to access the element of structure through pointer.

6.4.2 Deallocation of memory

To deallocate the memory space the function **free()** is used. The release of storage space becomes important when the storage is limited. When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for future user.

Syntax:

```
free(pointer name);
```

Example 10:

```
free(p);
```

6.5 Examples

Q2) Write a program to prepare a list of employees and students using union. [u6q2.c]

Solution

```
/*********************************************************************/
/*      Program to demo the function of UNION operation.          */
/*********************************************************************/
#include<stdio.h>
#include<string.h>
typedef union          /* Decleration of union */
{
  struct student
   {
        char sname[20];
        char sclass[10];
        int roll;
```

```
   }stu;
  struct employee
  {
        char reg[8];
        char ename[20];
        int  basic;
   }emp;
}record;
/***********************************/
void main()
{
record *rec[100];
char ch[100];
int i=0,j,y=5,condition=1;
clrscr();
do{
 clrscr();
 printf("1. Student.....\n");
 printf("2. Employee.....\n");
 printf("3  View All.....\n");
 printf("4. Exit From Program..\n");
 printf("\n\n\n  Your choice Please.:"); ch[i]=getch();
 rec[i]=(record *) malloc(sizeof(record));
                        /* Dynamic Memory Allocation */
 switch(ch[i])
 {
 case '1':    /* students record entry*/
   clrscr();
   gotoxy(30,3); printf("Students Records");
```

```
  gotoxy(10,5); printf("SL. NO.  %02d",i+1);
  gotoxy(10,10); printf("Student Name  : ");
  fflush(stdin); gets(rec[i]->stu.sname);
  gotoxy(10,12); printf("Student Roll  : "); scanf("%d",&rec[i]->stu.roll);
  gotoxy(10,14); printf("Student Class : ");
  fflush(stdin); gets(rec[i]->stu.sclass);
  i++;
  break;
 case '2':   /* employees record entry */
  clrscr();
  gotoxy(30,3); printf("Employee Records");
  gotoxy(10,5); printf("SL. NO.  %02d",i+1);
  gotoxy(10,10); printf("Employee Name   : ");
  fflush(stdin); gets(rec[i]->emp.ename);
  gotoxy(10,12); printf("Reg.  No        : ");
  fflush(stdin); gets(rec[i]->emp.reg);
  gotoxy(10,14); printf("Basic Pay (Rs.) : ");
  scanf("%d",&rec[i]->emp.basic);
  i++;
  break;
 case '3':
  clrscr(); printf("           ****** Output ******");
  printf("\n       Student Information ..... Employee Information\n");
  for(y=5,j=0;j<i;j++)
  {
   if(ch[j]=='1')  /* student's information view */
   {
    gotoxy(10,y);   printf("Name  : %s",rec[j]->stu.sname);
    gotoxy(10,y+1); printf("Roll  :%d",rec[j]->stu.roll);
```

```
    gotoxy(10,y+2); printf("Class :%s",rec[j]->stu.sclass);
    y=y+4;
   }
   if(ch[j]=='2')  /* employee's information view*/
   {
    gotoxy(40,y);   printf("Name      : %s",rec[j]->emp.ename);
    gotoxy(40,y+1); printf("Reg. No.  : %s",rec[j]->emp.reg);
    gotoxy(40,y+2); printf("Basic Pay : %d.00",rec[j]->emp.basic);
    y=y+4;
   }
  } /*end of for() */
  getch();
  break;
 case '4':                    /* Exit condition */
  condition=0;
  break;
 default:
   printf("Invalid Choice...."); getch();
 }
}while(condition);
}
/************************** end of main() **************************/
```

Q3) Write a program to accept 10 numbers and print in reverse order

Solution:

```
void main()
{
int a[10],*p,i,s=0;
p=&a[0];    /* assignment of pointer */
for(i=0;i<10;i++,p++)
{
```

```
    printf("Enter value "); scanf("%d",p);
   s = s + *p;
 }
printf("\n\n\n The outputs in reverse order \n\n\n");
p--;
for(i=0;i<10;i++)
    printf("%d+",*p--);
printf("\b=%d",s);
getch();
}
/*  End of program   */
```

6.6 Summary

A structure type is a type defined within the program that specifies the format of a record, including the names and types of its members, and the order in which they are stored. Unlike structure members, which all have distinct locations in the structure, the members of a union all share the same location in memory; that is, all members of a union start at the same address. C supports the use of pointers, a very simple type of reference that records, in effect, the address or location of an object or function in memory. This unit covers Structure, Union and application of pointers in program.

Self Assessment Questions

1. Structure is a collection of number of other types. (True / False)
2. Structure initialization is not possible. (True / False)
3. Union and structure are same except storage allocation. (True / False)
4. The pointer operation & is called ‘address of’. (True / False)
5. Dynamic allocation means allocation at specific space in memory. (True / False)

6.7 Terminal Questions

1. Define a Pointer. Discuss the advantages of using pointers.
2. Explain the pointer operators with examples.
3. Write C programs to illustrate the usage of pointers with (i) arrays and (ii) functions.
4. Define a Structure. Write Syntax, with appropriate example, to declare a Structure.
5. Write a C program to copy two strings using pointer.
6. Differentiate between a union and a structure.

6.8 Answer to Self Assessment and Terminal Questions

Self Assessment Questions

1. True
2. False
3. True
4. True
5. True

Terminal Questions

1. Section 6.4
2. Section 6.4
3. Section 6.4
4. Section 6.2
5. Section 6.3.2