Pointer variables in structures

Structure pointer points to the address of the Structure Variable in the memory block to which it points. This pointer can be used to access and change the value of structure members.

The structure ~~member~~ pointer points to the address of a memory block where the structure is being stored. we use structure pointer which tells the address of a structure in memory by pointing pointer variable ptr to the structure variable.

Declare a structure pointer

```
Struct structure_name *ptr;
```

Initialization of the structure pointer

```
ptr = &structure_variable;
```

[or]

we can also initialized a structure pointer directly during declaration of a pointer.

```
Struct structure_name *ptr = &structure_variable;
```

Here ptr is pointing to the address structure_variable of the structure.

Access structure member using pointer

1) use * ⇒ asterisk (or) indirection operator and dot (.) operator

2) use arrow (->) operator or membership operator

Example Program to access the structure member using structure pointer and the dot operator

```
#include <stdio.h>
# include <string.h>
struct subject
{
  char sub_name [30];
  int sub_id;
};
int main()
{
 struct subject sub;
 struct subject *ptr;

  ptr = &sub;

 strcpy (sub.sub_name, " Computer Science");
  sub. sub_id = 1201;
 printf (" Subject name : %s \t ", (*ptr).sub_name);
 printf (" \n subject Id : %d \t ", (*ptr).sub_id);

 return 0;
}
```

output

```
subject name: Computer science
subject Id  :  1201
```

Example Program to access the structure members using the pointer and arrow operator.

```
# include <stdio.h>
 // create Employee structure
 structe Employee
 {
     char name[30];
     int id;
 };

 struct Employee emp1, *ptr1;
 int main()
 {
     ptr1 = &emp1;
     printf ("Enter name and id of employee");
     scanf (" %s ", &ptr1-> name);
     scanf ("%d", &ptr1 -> id);

     printf (" Name : %s\n", ptr1 -> name).
     printf (" Id : %d\n", ptr1 -> id).
     return 0;
 }
```

Union :

Unions provide an efficient way of using the same memory locations for multiple purpose.

Union can be defined as a user-defined datatype which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members but only one member can contain a value at a particular point in time.

Example :

```
Union   abc
{
    int a ;
    char b ;
} var ;

int main()
{
  var . a = 66 ;
  Printf ("\n a = %d ", var.a);
  Printf ("\n b=%d ", var.b);
```

output

```
a = 66
b = 66
```

In the above code, union has two members a & b. Var is a variable of union abc type. Both a and b share the memory location.

Deciding the size of the union

The size of the union is based on the size of the largest member of the union.

Example:

```
union abc
{
   int a;        // 4 bytes
    char b;      // 1 byte
    float c;     // 4 bytes
      double d;  // 8 bytes
};
int main()
{
 printf(" size of union abc is %d ", sizeof(
                                       union abc));
 return 0;
}
```

output

Size of union abc is 8 bytes

In the above example double variable occupies the largest memory among all the four variables. so total 8 bytes will be allocated in the memory.

Accessing members of union using pointers

Access the members of the union through pointers by using (→) arrow operator.

```
#include <stdio.h>
union abc
{
   int a;
   char b;
};
int main()
{
union abc *ptr;
 union abc var;

  var.a =90;
  ptr =&var;
   printf("The value of a is : %d ", ptr->a);
   return 0;
}
```

output

```
The value of a is : 90
```

Compare Structure and Union

Structure	Union
1. Struct keyword is used to define a structure	Union keyword is used to define a union.
2. Unique memory location is assigned to every member.	All the data members share a memory location
3. change in the value of one data member does not affect other data members in the structure	change in the value of one data member affects the value of other data members.
4. we can initialize multiple members at a time.	we can initialize only the first member at once.
5. A structure can store multiple values of the different members.	A union stores one value at a time for all of its members
6. A structure's total size is the sum of the size of every data member.	A union's total size is the size of the largest data member.
7. Users can access or retrieve any member at a time.	Users can access or retrieve only one member at a time.
8. Structures are used when we need to store distinct values in a unique memory location.	Unions help to manage memory efficiently.

Bit fields

we have union and struct data types where we can declare user_defined data types. The size of the struct depends on data members. But sometimes we donot need such a huge size of the data type, because it occupies memory, and it creates a waste of memory.

Example:

```
#include<stdio.h>
struct dob
{
    int date;
    int month;
    int year;
};

int main()
{
printf("size of struct is %d\n", sizeof(struct
                                    (dob));

}
```

output

```
    size of struct is 12 bytes
```

But we need not store date, month & year in huge memory since we know the maximum values of date can be 31, month can be 12 and year can be of maximum digits.

So we use bit fields to save the memory.

In the bit field, we can explicitly give the width or the range to the data member in terms of bytes.

<u>Syntax for bit fields:</u>

datatype data-member : maximum_width_bits

<u>Example:</u>

```
#include <stdio.h>
struct dob
{ unsigned int date : 5;        // 1 byte
  unsigned int month : 4;       // 1 byte
  unsigned int Year : 12;       // 2 bytes
};
int main()
{
printf("size of the struct is %d\n", sizeof
                      (struct(dob));
}
```

<u>output:</u>

```
size of the struct is 4 bytes
```

So with the help of bit fields, we have saved 8 bytes (12 bytes (without using bit fields)

$$
\begin{array}{r}
12 \text{ bytes (without using bit fields)} \\
- \ 4 \text{ bytes (with using bit fields)} \\
\hline
8 \text{ bytes} \\
\hline
\end{array}
$$