Union in C

Union can be defined as a user-defined data type 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.

Union is a user-defined data type, but unlike structures, they share the same memory location.

Let's understand this through an example.

```
struct abc
{
  int a;
  char b;
}
```

The above code is the user-defined structure that consists of two members, i.e., 'a' of type **int** and 'b' of type **character**. When we check the addresses of 'a' and 'b', we found that their addresses are different. Therefore, we conclude that the members in the structure do not share the same memory location.

When we define the union, then we found that union is defined in the same way as the structure is defined but the difference is that union keyword is used for defining the union data type, whereas the struct keyword is used for defining the structure. The union contains the data members, i.e., 'a' and 'b', when we check the addresses of both the variables then we found that both have the same addresses. It means that the union members share the same memory location.

Let's have a look at the pictorial representation of the memory allocation.

The below figure shows the pictorial representation of the structure. The structure has two members; i.e., one is of integer type, and the another one is of character type. Since 1 block is equal to 1 byte; therefore, 'a' variable will be allocated 4 blocks of memory while 'b' variable will be allocated 1 block of memory.

The below figure shows the pictorial representation of union members. Both the variables are sharing the same memory location and having the same initial address.

In union, members will share the memory location. If we try to make changes in any of the member then it will be reflected to the other member as well. Let's understand this concept through an 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);
}
```

In the above code, union has two members, i.e., 'a' and 'b'. The 'var' is a variable of union abc type. In the **main()** method, we assign the 66 to 'a' variable, so var.a will print 66 on the screen. Since both 'a' and 'b' share the memory location, **var.b** will print '**B**' (ascii code of 66).

Deciding the size of the union

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

Let's understand through an example.

```
union abc{
int a;
char b;
float c;
double d;
};
int main()
{
  printf("Size of union abc is %d", sizeof(union abc));
  return 0;
}
```

As we know, the size of int is 4 bytes, size of char is 1 byte, size of float is 4 bytes, and the size of double is 8 bytes. Since the double variable occupies the largest memory among all the four variables, so total 8 bytes will be allocated in the memory. Therefore, the output of the above program would be 8 bytes.

Accessing members of union using pointers

We can access the members of the union through pointers by using the (->) arrow operator.

Let's understand through an example.

```c
#include <stdio.h>
union abc
{
    int a;
    char b;
};
int main()
{
    union abc *ptr; // pointer variable declaration
    union abc var;
    var.a= 90;
    ptr = &var;
    printf("The value of a is : %d", ptr->a);
    return 0;
}
```

In the above code, we have created a pointer variable, i.e., *ptr, that stores the address of var variable. Now, ptr can access the variable 'a' by using the (->) operator. Hence the output of the above code would be 90.

Why do we need C unions?

Consider one example to understand the need for C unions. Let's consider a store that has two items:

- Books
- Shirts

Store owners want to store the records of the above-mentioned two items along with the relevant information. For example, Books include Title, Author, no of pages, price, and Shirts include Color, design, size, and price. The 'price' property is common in both items. The Store owner wants to store the properties, then how he/she will store the records.

Initially, they decided to store the records in a structure as shown below:

```
struct store
{
   double price;
   char *title;
   char *author;
   int number_pages;
   int color;
  int size;
  char *design;
};
```

The above structure consists of all the items that store owner wants to store. The above structure is completely usable but the price is common property in both the items and the rest of the items are individual. The properties like price, *title, *author, and number_pages belong to Books while color, size, *design belongs to Shirt.

Let's see how can we access the members of the structure.

```
int main()
{
   struct store book;
   book.title = "C programming";
book.author = "Paulo Cohelo";
book.number_pages = 190;
book.price = 205;
printf("Size is : %ld bytes", sizeof(book));
```

```c
return 0;
}
```

In the above code, we have created a variable of type **store**. We have assigned the values to the variables, title, author, number_pages, price but the book variable does not possess the properties such as size, color, and design. Hence, it's a wastage of memory. The size of the above structure would be 44 bytes.

We can save lots of space if we use unions.

```c
#include <stdio.h>
struct store
{
    double price;
    union
    {
        struct{
        char *title;
        char *author;
        int number_pages;
        } book;

        struct {
        int color;
        int size;
        char *design;
        } shirt;
    }item;
};
 int main()
{
    struct store s;
    s.item.book.title = "C programming";
    s.item.book.author = "John";
    s.item.book.number_pages = 189;
    printf("Size is %ld", sizeof(s));
    return 0;
}
```

In the above code, we have created a variable of type store. Since we used the unions in the above code, so the largest memory occupied by the variable would be considered for the memory allocation. The output of the above program is 32 bytes. In the case of structures, we obtained 44 bytes, while in the case of unions, the size obtained is 44 bytes. Hence, 44 bytes is greater than 32 bytes saving lots of memory space.