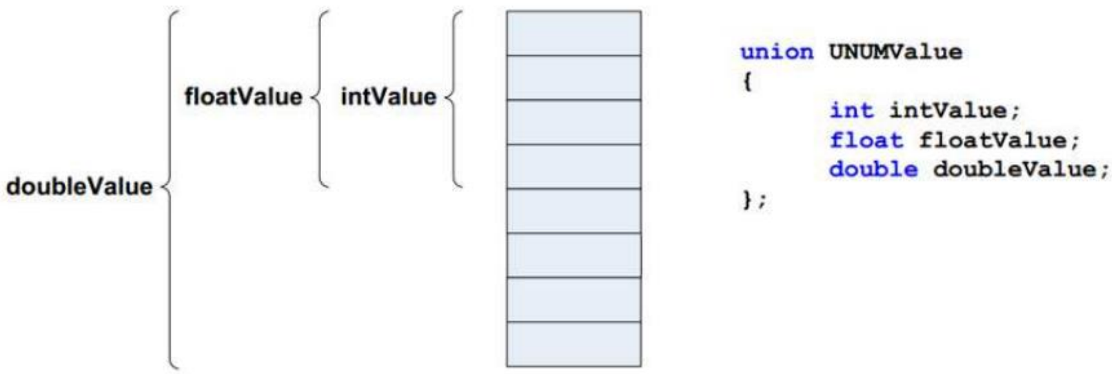
Unions in C

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.

All the members of a union share the same memory location. Therefore, if we need to use the same memory location for two or more members, then union is the best data type for that. The largest union member defines the size of the union.



Defining a Union

Syntax

Here is the syntax to define a **union** in C language −

```
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.

Accessing the Union Members

Syntax

Here is the syntax to access the members of a union in C language −

```
union_name.member_name;
```

Initialization of Union Members

You can initialize the members of the union by assigning the value to them using the assignment (=) operator.

Syntax

Here is the syntax to initialize members of union −

```
union_variable.member_name = value;
```

Example

The following code statement shows to initialization of the member "i" of union "data" −

```
data.i = 10;
```

Examples of Union

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

union Data{
   int i;
   float f;
   char str[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;
}
```

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

Here, we can see that the values of **i** and **f** (members of the union) show **garbage values** 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.

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

union Data{
   int i;
   float f;
   char str[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;
}
```

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

Size of a Union

The size of a union is the size of its largest member.

For example, if a union contains two members of **char** and **int** types. In that case, the size of the union will be the size of **int** because **int** is the largest member.

```c
#include <stdio.h>

// Define a union
union Data {
   int a;
   float b;
   char c[20];
};

int main() {
   union Data data;

   // Printing the size of the each member of union
   printf("Size of a: %lu bytes\n", sizeof(data.a));
   printf("Size of b: %lu bytes\n", sizeof(data.b));
   printf("Size of c: %lu bytes\n", sizeof(data.c));

   // Printing the size of the union
   printf("Size of union: %lu bytes\n", sizeof(data));

   return 0;
}
```

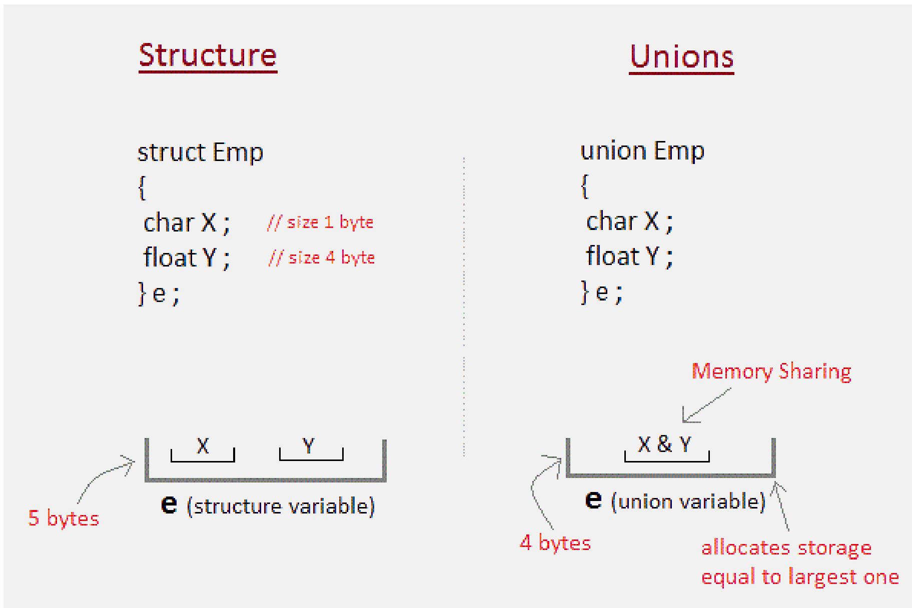
```
Size of a: 4 bytes
Size of b: 4 bytes
Size of c: 20 bytes
Size of union: 20 bytes
```

Difference between Structure and Union

Both structures and unions are **composite** data types in C programming.

The most significant difference between a structure and a union is the way they store their data.

A structure stores each member in separate memory locations, whereas a union stores all its members in the same memory location.



Here is a definition of union type called **myunion** –

```
union myunion{
   int a;
   double b;
   char c;
};
```

The definition of a union is similar to the definition of a structure. A definition of "struct type mystruct" with the same elements looks like this –

```
struct mystruct{
   int a;
   double b;
   char c;
};
```

The main difference between a struct and a union is the size of the variables.

The compiler allocates the memory to a struct variable, to be able to store the values for all the elements.

In **mystruct**, there are three elements – an int, a double, and char, requiring 13 bytes (4 + 8 + 1). Hence, **sizeof(struct mystruct)** returns 13.

VS

On the other hand, for a union type variable, the compiler allocates a chunk of memory of the size enough to accommodate the element of the largest byte size.

The **myunion** type has an int, a double and a char element. Out of the three elements, the size of the double variable is the largest, i.e., 8. Hence, **sizeof(union myunion)** returns 8.

Another point to take into consideration is that a union variable can hold the value of only one its elements.

When you assign value to one element, the other elements are undefined. If you try to use the other elements, it will result in some garbage.

Example 1: Memory Occupied by a Union

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

union Data{
   int i;
   float f;
   char str[20];
};

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

```
Memory occupied by Union Data: 20
```

Example 2: Memory Occupied by a Structure

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

struct Data{
   int i;
   float f;
   char str[20];
};

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

Output

This stucture will occupy 28 bytes (4 + 4 + 20). Run the code and check its output –

```
Memory occupied by Struct Data: 28
```

" من ضيع الأصول حرم الوصول ومن ترك الدليل ضل السبيل"