02_10_ C Union

C Union

- C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member.
- Union and structure in C are same in concepts, except allocating memory for their members.
- Structure allocates storage space for all its members separately.
- Whereas, Union allocates one common storage space for all its members

C Union

- **We can access only one member of union at a time**. We can't access all member values at the same time in union.
- But, **structure can access all member values at the same time**.
- This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.

C – Union

Type	Using normal variable	Using pointer variable

Syntax	union tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};
		union tag_name
		{
		data type var_name1;
		data type var_name2;
		data type var_name3;
		};

Example	union student
	{
	int mark;
	char name[10];
	float average;
	};
		union student
		{
		int mark;
		char name[10];
		float average;
		};

Declaring union variable	union student report;	union student *report, rep;
Initializing union variable	union student report = {100, "Mani", 99.5};	union student rep = {100, "Mani", 99.5}; report = &rep;
Accessing union members	report.mark report.name report.average	report -> mark report -> name report -> average

Why do we need C Unions?

- The purpose of union is to save memory by using the same memory region for storing different objects at different times. That's it.

- **It is like a room in a hotel**. Different people use it for non-overlapping periods of time. These people never meet, and generally don't need to know anything about each other.

- That's exactly what union does. If you know that several objects in your program hold values with non-overlapping value-lifetimes, then you can "merge" these objects into a union and thus save memory. Just like a hotel room has at most one "active" tenant at each moment of time, a union has at most one "active" member at each moment of program time. Only the "active" member can be read. By writing into other member you switch the "active" status to that other member.

Why do we need C Unions?

- Unions are particularly useful in Embedded programming or in situations where direct access to the hardware/memory is needed.

```
typedef union
{
    struct {
        unsigned char byte1;
        unsigned char byte2;
        unsigned char byte3;
        unsigned char byte4;
    } bytes;
    unsigned int dword;
} HW_Register;
```

- Then you can access the **reg** as follows:

```
HW_Register reg;
reg.dword = 0x12345678;
reg.bytes.byte3 = 4;
```

Why do we need C Unions?

- Unions allow data members which are mutually exclusive to share the same memory.

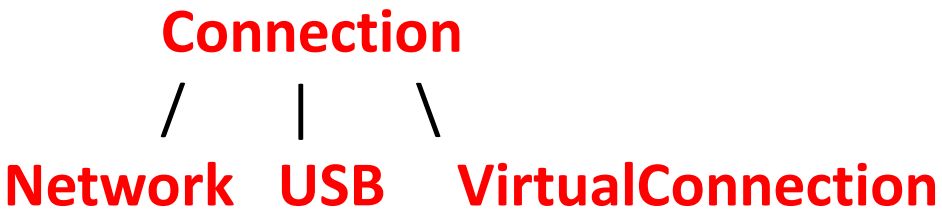
- This is quite important when memory is more scarce, such as in embedded systems.

```
union {
    int a;
    int b;
    int c;
} myUnion;
```

- This union will take up the space of a single int, rather than 3 separate int values. If the user set the value of a, and then set the value of b, it would *overwrite* the value of a since they are both sharing the same memory location.

Why do we need C Unions?

- I've seen it in a couple of libraries as a replacement for object oriented inheritance.
- E.g.



- If you want the Connection "class" to be either one of the above, you could write something like:

```
struct Connection
{
    int type;
    union
    {
        struct Network network;
        struct USB usb;
        struct Virtual virtual;
    }
};
```

C – Union

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

union student
{
            char name[20];
            char subject[20];
            float percentage;
};
```

```
Union record1 values example
Name :
Subject :
Percentage : 86.500000;

Union record2 values example
Name : Mani
Subject : Physics
Percentage : 99.500000
```

```
int main()
{
    union student record1;
    union student record2;

    // assigning values to record1 union variable
       strcpy(record1.name, "Raju");
       strcpy(record1.subject, "Maths");
       record1.percentage = 86.50;

       printf("Union record1 values example\n");
       printf(" Name       : %s \n", record1.name);
       printf(" Subject    : %s \n", record1.subject);
       printf(" Percentage : %f \n\n", record1.percentage);

    // assigning values to record2 union variable
       printf("Union record2 values example\n");
       strcpy(record2.name, "Mani");
       printf(" Name       : %s \n", record2.name);

       strcpy(record2.subject, "Physics");
       printf(" Subject    : %s \n", record2.subject);

       record2.percentage = 99.50;
       printf(" Percentage : %f \n", record2.percentage);
       return 0;
}
```

C – Union

- We can always access only one union member for which value is assigned at last. We can't access other member values.
- So, only "`record1.percentage`" value is displayed in output. "`record1.name`" and "`record1.percentage`" are empty.
- If we want to access all member values using union, we have to access the member before assigning values to other members as shown in `record2` union variable in this program.
- Each union members are accessed in `record2` example immediately after assigning values to them.
- If we don't access them before assigning values to other member, member name and value will be over written by other member as all members are using same memory.
- We can't access all members in union at same time but structure can do that.

C – Union

- In this program, union variable "**record**" is declared while declaring union itself as shown in the below program.

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

union student
{
            char name[20];
            char subject[20];
            float percentage;
}record;

int main()
{
            strcpy(record.name, "Raju");
            strcpy(record.subject, "Maths");
            record.percentage = 86.50;

            printf(" Name       : %s \n", record.name);
            printf(" Subject    : %s \n", record.subject);
            printf(" Percentage : %f \n", record.percentage);
            return 0;
}
```

```
Name :
Subject :
Percentage : 86.500000
```

Difference between structure and union in C

S.no	C Structure	C Union
1	Structure allocates storage space for all its members separately.	• Union allocates one common storage space for all its members. • Union finds that which of its member needs high storage space over other members and allocates that much space
2	Structure occupies higher memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.

4	Structure example:
	`struct student`
	`{`
	`int mark;`
	`char name[6];`
	`double average;`
	`};`
		Union example:
		`union student`
		`{`
		`int mark;`
		`char name[6];`
		`double average;`
		`};`

5	For above structure, memory allocation will be like below. int mark – 4B char name[6] – 6B double average – 8B Total memory allocation = 4+6+8 = 18 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

The Size of a Union

- You've been told that the members of a union share the same memory location. The size of a union is the same as the size of the largest member in the union.
- In contrast with a union, all members of a structure can be initialized together without any overwriting.
- This is because each member in a structure has its own memory storage.
- The size of a structure is equal to the sum of sizes of its members instead of the size of the largest member.
- **How do I know whether all these are true?** Well, I can prove it by measuring the size of a union or a structure.

```c
/* 20L03.c The size of a union */
#include <stdio.h>
#include <string.h>

main(void)
{
   union u {
       double x;
       int y;
    } a_union;

   struct s {
       double x;
       int y;
   } a_struct;

   printf("The size of double: %d-byte\n",
           sizeof(double));
   printf("The size of int:    %d-byte\n",
           sizeof(int));

   printf("The size of a_union:  %d-byte\n",
           sizeof(a_union));
   printf("The size of a_struct: %d-byte\n",
           sizeof(a_struct));

   return 0;
}
```

```
The size of double: 8-byte
The size of int:    2-byte
The size of a_union:  8-byte
The size of a_struct: 10-byte
```

End of 02_10