**Union**

In **C programming**, a **union** is a user-defined data type similar to a **structure**, but with one key difference: all members of a union share the same memory location. This means only **one member** can hold a value at a time, making unions useful for memory-efficient programming when working with variables that do not need to hold values simultaneously.

**Syntax for Declaring a Union**

union UnionName {

dataType member1;

dataType member2;

// More members...

};

Here:

* union is the keyword.
* UnionName is the name of the union.
* member1, member2, etc., are the variables inside the union, called **members**.

**Key Difference Between Structure and Union**

* In a **structure**, each member has its own memory.
* In a **union**, all members share the same memory location, and the size of the union is equal to the size of its largest member.

**Example of a Union in C**

Let’s create a union to store data about a variable that can be an integer, a float, or a character.

#include <stdio.h>

// Define the union

union Data {

int i;

float f;

char c;

};

int main() {

// Declare a union variable

union Data data;

// Assign an integer value

data.i = 10;

printf("Integer: %d\n", data.i);

// Assign a float value

data.f = 3.14;

printf("Float: %.2f\n", data.f);

// Assign a character value

data.c = 'A';

printf("Character: %c\n", data.c);

// Observe how assigning new values overwrites previous data

printf("\nMemory Overlap:\n");

printf("Integer after assigning char: %d\n", data.i);

printf("Float after assigning char: %.2f\n", data.f);

return 0;

}

**Output**

Integer: 10

Float: 3.14

Character: A

Memory Overlap:

Integer after assigning char: 65

Float after assigning char: 0.000000

**Explanation**

1. **Shared Memory:**
   * When data.i is assigned the value 10, the union uses its memory to store this integer.
   * When data.f is assigned 3.14, it overwrites the memory where the integer was stored.
   * Similarly, assigning data.c as 'A' overwrites the same memory location.
2. **Memory Efficient:**
   * The size of the union equals the size of its largest member.
   * In this example:
     + int might take 4 bytes, float 4 bytes, and char 1 byte.
     + The total size of the union is **4 bytes**, not the sum of all member sizes.
3. **Limitation:**
   * Only one member can hold a valid value at any time.

2)

#include <stdio.h>

union Job {

float salary;

int workerNo;

} j;

int main() {

j.salary = 12.3;

// when j.workerNo is assigned a value,

// j.salary will no longer hold 12.3

j.workerNo = 100;

printf("Salary = %.1f\n", j.salary);

printf("Number of workers = %d", j.workerNo);

return 0;

}

**Difference between unions and structures**

#include <stdio.h>

union unionJob

{

//defining a union

char name[32];

float salary;

int workerNo;

} uJob;

struct structJob

{

char name[32];

float salary;

int workerNo;

} sJob;

int main()

{

printf("size of union = %d bytes", sizeof(uJob));

printf("\nsize of structure = %d bytes", sizeof(sJob));

return 0;

}

**Output**

size of union = 32

size of structure = 40