**Union assignments**

**1. Refer the code below and comment on size of the given structure considering**

**a. Structure as union**

**b. Structure as struct**

**c. arr**

**d. uarr**

\_\_\_ Job

{

char name[32];

unsigned short ucount;

float salary;

int workerNo;

char \*orgname;

};

\_\_\_ Job myvar; //could of union or of struct

Struct Job arr[10];

Union Job uarr[10];

Ans: **1. Comment on the size of the given structure considering:**

The structure Job is defined as:

struct Job {

char name[32]; // 32 bytes

unsigned short ucount; // 2 bytes

float salary; // 4 bytes

int workerNo; // 4 bytes

char \*orgname; // 4 bytes (assuming 32-bit system)

};

We also have two arrays: struct Job arr[10]; // Array of 10 structs

union Job uarr[10]; // Array of 10 unions

**a. Structure as union:**

A **union** allows all its members to share the same memory space. The size of a union is determined by the largest member because all members overlap in memory. Therefore, the size of the union Job would be:

* The largest member is char name[32] (32 bytes).
* The other members (ucount, salary, workerNo, and orgname) occupy lesser sizes, but since they all share the same space, the size of the union will be the size of the largest member.

Thus, the **size of the union Job** is **32 bytes**.

For the array uarr[10], since each element is a union of size 32 bytes, the total size will be:

* Size of uarr[10] = 10 \* 32 = **320 bytes**.

**b. Structure as struct:**

A **struct** allocates separate memory for each member, so the size is the sum of the sizes of its members, but we also need to account for padding to align data properly. The typical alignment for int, float, and short is 4 bytes.

* char name[32] takes 32 bytes.
* unsigned short ucount takes 2 bytes, but there will be padding of 2 bytes after it to make the next member (float salary) 4-byte aligned.
* float salary takes 4 bytes.
* int workerNo takes 4 bytes.
* char \*orgname takes 4 bytes.

So, the total size of the struct would be:

* 32 (name) + 2 (ucount) + 2 (padding) + 4 (salary) + 4 (workerNo) + 4 (orgname) = **48 bytes**.

Thus, the **size of the struct Job** is **48 bytes**.

For the array arr[10], since each element is a struct of size 48 bytes, the total size will be:

* Size of arr[10] = 10 \* 48 = **480 bytes**.

**c. arr:**

arr is an array of struct Job, and we have already calculated that each struct Job occupies 48 bytes.

So, the size of arr[10] is **480 bytes**.

**d. uarr:**

uarr is an array of union Job, and we have already calculated that each union Job occupies 32 bytes.

So, the size of uarr[10] is **320 bytes**.

**2. Refer Job data structure in Q#1 above. Using uarr, perform below operations.**

**a. Read and store salary**

**b. Read and store workerNo**

**Comment on values of output if salary and workerNo are printed in order. Justify your statement.**

**to perform operations on the union Job array uarr, specifically reading and storing salary and workerNo.**

**a. Read and store salary:**

In a **union**, all members share the same memory location. If we store a value in one member, it will overwrite the memory space of all other members.

Let’s assume we store a value in salary:

uarr[0].salary = 50000.0;

This will store the value 50000.0 in the salary field, but since the union shares memory among all fields, the salary field will overwrite the memory space for the other fields.

**b. Read and store workerNo:**

Similarly, if we store a value in workerNo:

uarr[0].workerNo = 1234;

This will overwrite the same memory location where salary was stored.

**Comment on values of output:**

If you print both salary and workerNo after the above operations, both will give **incorrect or unexpected results** because they share the same memory. Here's why:

* After setting salary, the memory for salary is updated.
* When you store a value in workerNo, it overwrites the same memory location.

Therefore, when printing:

printf("Salary: %f\n", uarr[0].salary);

printf("WorkerNo: %d\n", uarr[0].workerNo);

* The output of salary will be 0.0 or some random value since the workerNo field is now overwriting the salary field.
* The output of workerNo will be the correct value (1234), but the value in salary will not be as expected.

This is because the union allows only one member to hold a valid value at any given time.

**3. Refer Job datastructure in Q#1 above. Assume that myvar is a structure variable. If I need to place 2 bytes (i.e 0x0102) as ucount using a char \*ptr then list all possible statements that can be used in \_\_\_\_\_.**

[Let solutions include cases such as

i. using base address of ucount

ii. using relative address of ucount w.r.t to base address of myvar]

int main()

{

char \*ptr = &myvar;

\_\_\_\_\_\_\_\_\_\_\_ = 0x01;

\_\_\_\_\_\_\_\_\_\_\_ = 0x02

}

The two possible ways to assign the value 0x0102 to the ucount field using a char \* pointer are:

1. **Using the base address of ucount:**

char \*ptr = (char\*)&myvar.ucount; // Point to ucount (cast to char pointer)

ptr[0] = 0x01; // Set the first byte (lower byte)

ptr[1] = 0x02; // Set the second byte (higher byte)

1. **Using the relative address of ucount with respect to the base address of myvar:**

char \*ptr = (char\*)&myvar; // Point to the base address of myvar

ptr += offsetof(struct Job, ucount); // Move ptr to the address of ucount

ptr[0] = 0x01; // Set the first byte (lower byte)

ptr[1] = 0x02; // Set the second byte (higher byte)

Both methods will correctly place the 2-byte value 0x0102 into the ucount field in little-endian byte order, assuming that ucount is a 2-byte short type.