Structures in C

Requirement: Store more than one different types in a single type group

Struct : user defined
↳ group of Primitive elements

Declaration Keyword

```
struct {
    char *engine, *fuel;
    int tank_cap;  int seats;   ] → members
} car1, car2;   two objects
```

car1.engine = "Any thing";

Dot operator is used to access the members

Types of Structs: Local & Global

You can use a structure tag to define a type of str

Ex

```
struct employee {      (employee: structure tag)
    char* name;
    int age, salary;
};

int manager() {
    struct employee manager;
    ...                 (employee: type, manager: name)
}
int main {
struct employee emp1, emp2;
```

Typedef Keyword:

→ Allows the user to create their own types
Not just Structs

Ex: typedef int INTEGER; ← New type
old type

```
INTEGER Var=100;
Printf("%d", Var); → output: 100
```

typedef Struct Car{ ← old type
...
} Car; ← New type

```
int main() {
    Car c1; → typedef
}
```
You don't need to use (Struct) on declaring a new variable

---

initializing & Accessing data members of struct

```
Struct abc {
    int p=23;
    int q=47;
}
```
✗ → Not allowed
You cannot initialize data members on defining the structs

initializing data members

Allowed:

```
Struct abc{
    int p;
    int q;
};
```

① `struct abc x = {23, 42};`

x → p = 23, q = 42

`struct abc y = {33, 36};`

y → p = 33, q = 36

Every object of a struct type is completely different

Accessing data members:

(Dot operator):

```
Struct abc {
    int p;
    int q;
} x, y;
```

→

```
x.p = 23;
x.q = 42;
y.p = 33;
y.q = 36;
```

Designated initialization:

You can initialize data members in any order

Ex:

```
struct abc {
    int x;
    int y;
    int z;
};
```

→ `struct abc a = {.y=2, .x=1, .z=3};`

dot operater to access data member

Array of structures

`struct Car c[2];` ← two objects of type Car

```
c[0].engine = "___";
c[1].engine = "___";
```

it is a structure But

We can say it is an object metaphorically

an object is more Complex

- data members
- functions

↓

object-oriented Programming Concept

Structure Pointer: (->) operator

```
Struct abc a = {0, 1}
Struct abc *Ptr = &a;
Printf("%d", Ptr->x); //o
```

Address of all Structure

ptr->x is equivalent to (*ptr).x

(*&a).x = a.x

Structure Padding:-

Memory allocation:

* on declaring an object of structure:

Contiguous block of memory is allocated to structure members in order

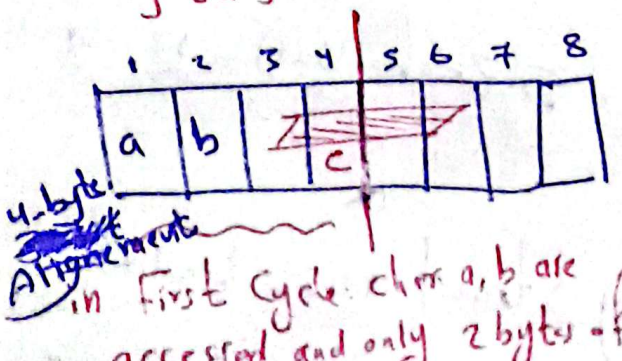
Ex:

```
Struct abc {
char a; // 1 byte
char b; // 1 byte
int c; // 4 bytes
} var;
```

Total: 6 bytes ✗ Wrong!

a Processor doesn't read 1 byte at a time.

↳ But it reads 1 Word at a time

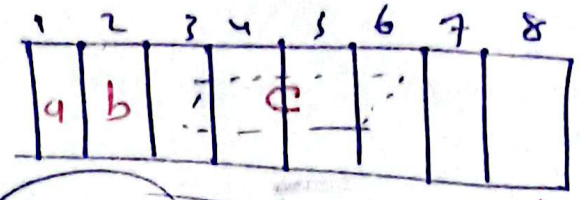


In First Cycle char a, b are accessed and only 2 bytes of int c

Second Cycle is required to Access the remaing 2 bytes of c

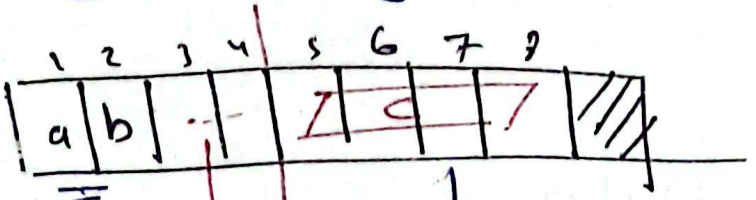
32-bit Processor: Access 4 bytes at a time

64-bit Processor: Access 8 bytes at a time



Problem: to access int c there is Wastage of CPU cycles

We can Solve this By Concept of Padding:-



empty rooms

Total: 1+1+2+2=8 bytes

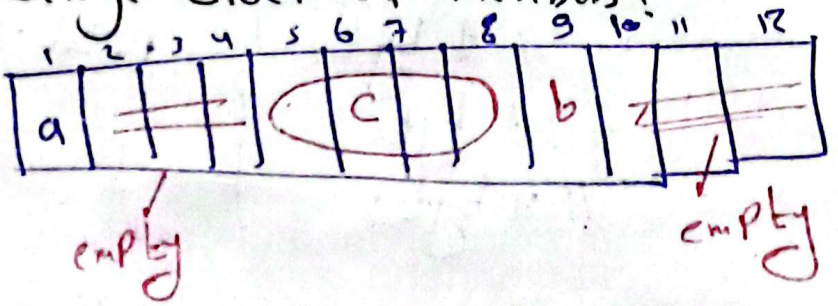
You can access char a, b in one cycle

You also can access int c in one cycle

Printf("%d", sizeof(Var)); // 8

So But if we change order of members?

```
Struct abc {
    char a;
    int b;
    char c;
} Var;
```



sizeof(Var) = 12 !

Note: Alignment is decided by struct most leading Variable (member): int 4 bytes in Previous ca-

Structure Packing:-

due to structure Padding; there is Waste of memory

Structure Packing X Structure Padding

Structure Packing → Save memory, Waste cycles

Structure Padding → Save CPU cycles, Waste memory

Pragma Pack(1) → turn off Padding

Pragma → special purpose directive turn on/off certain features

Ex

```
#Pragma Pack(1)
struct abc {
    char a;
    int b;
    char c;
} vars;

int main() {
    Printf("%d", sizeof(vars)); // 6 For any order
```

Ex:

```
#include <stdio.h>
Struct node {
    char x, y, z;
};
int main() {
  Struct node P = {'a', 'b', 'c'};
  struct node *ptr = &P;
```

structure Pointer → Pointer to the whole structure Not the first member but they are also equal [Same]

You can access data member by:

① -> operator : ptr -> x = 'f';

② Pointer type casting: [All members are of Same data type]

P.x: *((char *) ptr+0) = 'f';

P.y: *((char *) Ptr + 1) = 'u';

Pointer type casting From whole structure Pointer to char Pointer

Remember struct node *P[10]; → Array of structure Pointers