- structure are user defined data type in C.
- structure allows us to combine data of different types together
- It is used to create complex datatype which contain diverse information.

## Defining a structure

```
struct [structure_name]
{
data type var1;
data type var2;
data type var3;
}
[structure_variables];
```

Ex:

```
struct student{
int id;
int marks;
char fav_char;
} s1, s2, s3;
```

→ new datatype student has been created

```
#include <stdio.h>
struct Employee
{
int id;
char name[53];
float marks;
};
struct Employee e1, e2;

int main()
{
return 0;
}
```

```
#include <stdio.h>
struct Employee
{
int id;
char name[53];
float marks;
} e1, e2;

int main()
{
struct Employee tt;
return 0;
}
```

user defined datatype

→ data type

↳ variable name

## Initialization of a structure

```
#include <stdio.h>
struct Employee
{
int id;
float marks;
};

int main()
{
struct Employee e1;
e1.id = 12;
e1.marks = 34.12;
return 0;
}
```

→ create datatype

```
#include <stdio.h>
struct Employee
{
int id;
float marks;
};

int main()
{
struct Employee e1= {12, 34.12};

return 0;
}
```

## Accessing Structure members

- structure members are accessed using dot [.] operator
- (.) is called as "structure member operator".
- structure_name . member_name

## Type def key word

→ It is used to give alternate names to existing datatypes

both names remain valid

syntax

typedef <previous_name> <alias_name>

Ex:

```
type def unsigned long ul;
ul l1, l2, l3;
```

```
typedef struct student
{
int id;
int marks;
} std;
```

```
int main()
{
std s1, s2;
}
```

pointer

int *a, b;

↳ integer

solve the problem

```
type def int* intptr;
intptr a, b;
```

# Unions

- user defined data type
- difference lies in the fact that in structure each member has its own storage location whereas members of union uses a single shared memory location
- This single shared memory location is equal to the size of its longer data member
- Union cannot handle all members at once.

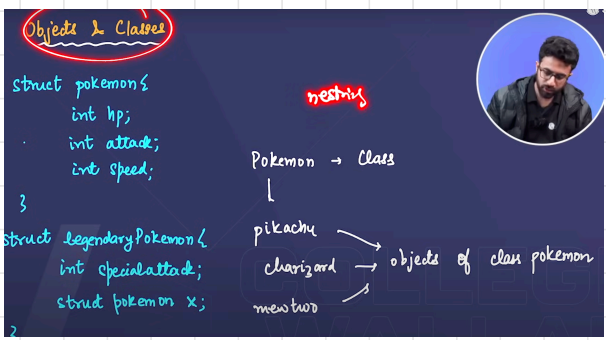
```
union student
{
    float marks;
    int id;
} u1;
```

4 bytes of memory location shared between id and marks

```
char ch[50];
ch = "deven";
```

} wrong

after declaration we can only access the individual characters and not change the whole array hence we use strcpy(str, str);




# Array of structures

```
typedef struct Pokemon{
    int hp;
    int attack;
    int speed;
    char tier;
} pokemon;
```

pokemon arr[] = array of pokemon

```
#include<stdio.h>
int main(){
    typedef struct pokemon{
        int hp;
        int speed;
        int attack;
        char tier;
    } pokemon ;

    pokemon arr[3]; // arr[0], arr[1], ... arr[2]
    arr[0].attack = 50;
    arr[0].hp = 100;
    arr[0].speed = 30;
    arr[0].tier = 'A';

    arr[1].attack = 150;
    arr[1].hp = 100;
    arr[1].speed = 130;
    arr[1].tier = 'S';
```

## Assigning the value of one structure variable to another of same type

```
pokemon a, b;
a = b;
```

→ copies all values of b into a

↳ creates deep copy of variables

* we cannot compare two user defined datatypes

```
if(a == b) → incorrect
```

## Nesting one structure within another structure

```c
int main(){
    typedef struct pokemon{
        int hp;
        int speed;
        int attack;
        char tier;
        char name[15];
    } pokemon ;
    typedef struct legendarypokemon{
        pokemon normal;
        char ability[10];
    } legendarypokemon ;

    return 0;
```

```c
legendarypokemon mewtwo;
strcpy(mewtwo.ability,"Pressure");
mewtwo.normal.hp = 150;
mewtwo.normal.attack = 180;
strcpy(mewtwo.normal.name,"Mewtwo");
mewtwo.normal.speed = 180;
mewtwo.normal.tier = 'S';
return 0;
```

```c
typedef struct godpokemon{
    legendarypokemon legend;
    int specialattack;
} godpokemon;
```

```c
godpokemon arceus;
arceus.specialattack = 300;
strcpy(arceus.legend.ability,"Turn Anyone to stone");
arceus.legend.normal.attack = 50
```

## Passing structures to a function

* structure should be declared globally so that it is accessible to all functions.

```c
void change(pokemon p){
    p.hp = 70;
    p.attack = 60;
    p.speed = 110;
    return;
}
int main(){
    pokemon pikachu;
    pikachu.hp = 60;
    pikachu.attack = 50;
    pikachu.speed = 100;
    change(pikachu);
    printf("%d\n",pikachu.hp);
    printf("%d\n",pikachu.attack);
    printf("%d\n",pikachu.speed);
```

→ passed by value i.e. a copy of the user defined datatype variable is created and changes are not reflected in the original variable.

- structures are passed by value

## Structure Pointers

```c
#include<stdio.h>
#include<string.h>
#include<stdbool.h>
typedef struct pokemon{
        int hp;
        int speed;
        int attack;
        char tier;
        char name[15];
} pokemon ;

int main(){
    pokemon pikachu;
    // int* x -> address of integer value
    pokemon* x = &pikachu;
    printf("%p",x);
```

→ structure pointer


To change values in the structure using pointer

$(*x).hp = 60;$ ≈ this is equivalent to pikachu.hp = 60; OR

important

$(*x).hp = 70$ can also be written as $x->hp = 70$

→ shorthand operator

very imp

Another way of initializing

```
    return;
}
int main(){
    pokemon pikachu = {60,70,100,'A',"Pikachu"};
    // pikachu.hp = 60;
    // pikachu.attack = 70;
    // pikachu.speed = 100;
    // pikachu.tier = 'A';
    // strcpy(pikachu.name,"Pikachu");

    // int* x -> address of integer value
```

should be of same order as in structure.

```
}
int main(){
    pokemon pikachu = {60,70,100};
    // pikachu.hp = 60;
    // pikachu.attack = 70;
    // pikachu.speed = 100;
    pikachu.tier = 'A';
    strcpy(pikachu.name,"Pikachu");

    // int* x -> address of integer value
```

→ half initialisation here

} remaining task here.

```
int main(){
    pokemon pikachu = {'A',"Pikachu"}
    pikachu.hp = 60;
    pikachu.attack = 70;
    pikachu.speed = 100;
    // pikachu.tier = 'A'
    // strcpy(pikachu.name,"Pikachu");

    int* x -> address of integer value
```

This gives error

- all members should be initialised in order cannot be skipped in between

| Aspect | User-Defined Data Types | Derived Data Types |
|---|---|---|
| Purpose | To create new data structures | To extend or manipulate existing types |
| Creation | Defined by the programmer | Derived from existing types |
| Examples | `struct`, `union`, `enum`, `typedef` | `array`, `pointer`, `function` |
| Memory Usage | Can vary (e.g., unions share memory) | Consistent with the base data types |

Use of struct: sometimes it is necessary to group datatype of different type together and arrays dont allow that so we use structs.

Structures: It is a user defined datatype that stores different related datatypes under a single name

Purpose: organize complex data

| Feature | Structure | Union |
|---|---|---|
| Memory | Each member has unique space | All members share space |
| Size | Sum of sizes of all members | Size of largest member |
| Usage | To hold multiple data points | Save memory for single data usage at different times |