Day: Date: / /

STRUCT

It is a datatype that is user defined.

- Everything in classes is also in struct
- All rules are same in classes and struct except a few e.g struct is initially public while class is initially private (will be discussed later).

"struct" is a reserve word.

General format for struct declaration:

```
struct  Identifier {

        // variable declarations:
        // These variables are called attributes

        };
```

styling Preffered:

- Identifier / TAG should start with a capital letter and variables should be of lower letters.

Yousaf

Day: Date: / /

- We use camel style but even the first letter is capitalized.

Member variables / states / attributes:

e.g

```
Struct Time {

        int h;
        int min;        - member variables
        int sec;
};
```

Peclared user defined variables:

1)
```
int main () {
        int x;
        Time t;
}
```

Here t represents : t

h	min	sec

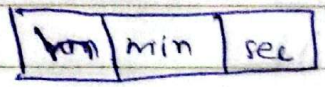
~~- so it can be code t might be same as an array but~~

Day: Date: / /

2) We can also declare variable when we create struct:

```
struct Time {
    int h;
    int min;
    int sec;
} b1, b2, t, t1;
```

3) What does t represent?



so t might be same as an array of size 3 of int datatype but we will also how it is different.

4) Making an array of our datatype:

```
int main(){
    Time t[3];   ✓ array created
}
```

Yousaf

Pg:

Date: / /

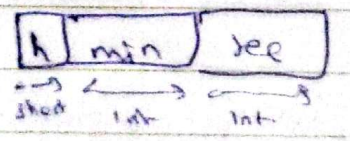
Differencing the struct and an array:

- Array is of single Data type.
- Struct can be of multiple data types.

eg:

```
Struct Time{

    short int h;
    int min;
    int sec;
};

int main(){

    Time t;
}
```

t:



Day:

Date: / /

Struct can also be different types of variables

```
Struct Students {

    string first_name;
    string last_name;
    int rollnumber;
    double gpa;

};
```

Initialling the user defined datatypes variable:

~~int a~~ - Considering the time example again.

```
int main(){

    Time t = {5, 30, 59};   ✓ allowed

    Time t1 {5, 30, 59};    ✓ allowed

    Time t2 = {5, 30};      ✓ allowed.

    Time t3 = {0, 5, 30, 7};   X Not allowed as we can only be given three as only 3 variables in Time.
```

Day: Date:

The dot [.] operator:

- use to access the member variables.
- Higher precendence than unary operator but lower than parenthesis or array subscript.
- It has higher precendence than * operator.

Usage:

```
Struct Time{

    int hour;
    int min;
    int sec;
};

int main(){

    Time t1 = {5, 30, 54};

    t.hour6; // makes this 6.
    t.sec = 20; // makes this 20.

}
```

Day: Date: / /

2)

```
int main() {
    Time t [3] = { {5,30,20} , { 15,0,0 } , { 7,0,0 } };

    t[2].min = 39;
}
```

Imp:- Why does t[2].min works but *(t+2).min error?.

[2] has higher precedence than *.
so t[2] happens first and then we go to min.

In *(t+2).min ; (t+2) happens then (t+2).min as . has higher precedence than * and then we are dereferencing it so it is wrong.

*((t+2).min) , this is an error.

so correction:

(*(t+2)).min ✓allowed.

Yousuf

Day: Date: / /

3)

```
struct Student {

    string f_name;
    string l_name;
    int gpa;

};

int main() {

    Student detail [10];

for (int i=0; i< 10 ; i++){

    cin >> detail [i]. f_name;
    cin >> detail [i]. l_name;
    cin >> detail [i]. gpa;

}
```

allowed.

Date: / /

4) imp

```
struct Student {

    char f_Name [20];
    char l_nam [20];
     double gpa;
};

int main () {

    Student S1 = { "ABC", "XYZ", 0.99};

    cin >> S1.f_name;  ✓ allowed.

    S1.f_nam = " haider";  ✗ Not allow

}
```

To make second expression allowed.

```
S1.f_nam [0] = 'h';
S1.f_nam [1] = 'A';
```

Yousaf

Day: Date:

5) Making pointers as member variables.

```
Struct Student {

    char * lname;
    char * fname;
    double gpa;
};

int main () {

    // Student s1 = { "ABC", "XYZ", 0.99 }; X
    now this is wrong as "ABC" is a string
    now an address for a pointer.

    char fname [ ] = "ABC";
    char lname [ ] = "XYZ";

    student s1 = { fname, lname, 2.7 };

    cout << s1 . fname ; // ABC
    cout << s1 . lname ; // XYZ

}
```

Yousaf

Day: Date: / /

Making better structs (Divide and conquer):

Instead of:

```
struct Datetime {
    int date;
    int month, year;

    int hour, min, sec;
}
```

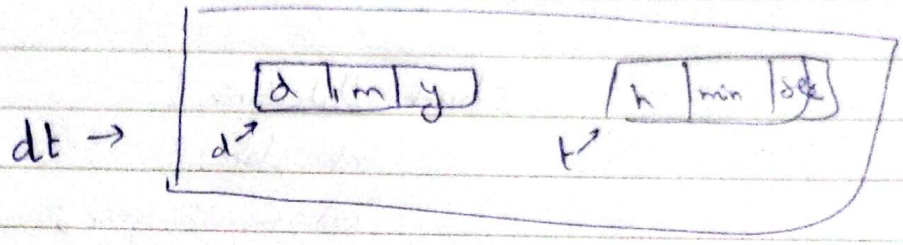
make:

```
struct Date {
    int date, month, year;
};

struct Time {
    int hour, min, sec
};

struct DateTime {
    Date d;
    Time t;
};
```

Day: Date: / /

Usage:

DateTime dt = { {20, 3, 2025}, {2} };



cout << dt.d.month; // couts 3.

Using these datatypes in function parameters:

- considering the same date time struct.

Pass by value:

```
void display ( DateTime td ){

    td.d.month = 5; // This is not being changed in main.
    cout<<td.d.month; // 5 as output

}
```

Pass by refernce

```
void display ( DateTime &td ){

}
```

- Now changes happen

Yousaf

Day: Date: / /

if we don't want to change in a function or don't want to change the value then:

```
void display ( const DateTime &td ) { };
```

Creating a pointer of user defined datatype.

Using the same DateTime:

```
int main () {
    Date time dt = { { }, { } }
    display ( & dt);
}

void display ( DateTime * td ) {
    cout << (*td).y.month ;
}
```

✓ correct

$ correct but (*td).y is a weird way to write so we use -> operator.

Yousaf

Day: Date: / /

"->" operator:

"->" represent "(*variable)."

usage:

e.g:

```
Struct Time {
    int h;
    int min, sec;
};

int main () {
    Time * t;

    (*t).h = 4;   > same thing
    t -> h = 4;

}
```

Yousaf

Day: Date: / /

But be careful:

The pointer is the user defined datatype; variable holds the member variable

eg

```
struct Time {
    int *hour;
    int min, sec;
};

int main() {

    Time b;

    // t -> hour = 3; X
```

This is wrong in 2 ways:

1)

t -> hour means:
(*b).hour,
but t is not a pointer.

2)

hour is a pointer and 3 is an int we cannot store that.

Correction

```
b.(*hour) = 3.
```

Yousaf

Day: Date: / /

A few important points

1) The "no"-name struct:

- A struct can be made without any identifier but its variables can only be defined where struct is made.

e.g

```
struct {

    int gpa;
    double marks;
    string name;
} t1, t2, g1, g, g[100];
```

These are now global variables and the only allowed from this struct.

Yousuf

Day:

Date: / /

2) You cannot declare a simple ~~integral~~ datatype inside the struct:

```
struct A{
    int x;
    A a;
}
```

not allowed

why?:

if a is made ; it ~~a~~ is made from A, then another a is made, that a makes another A. So this is infinite and wrong.

3) A pointer of same datatype are allowed:

```
struct A {
    int x;
    A * a;
}
```

allowed.

why?:

a is pointing to an A type, ~~so~~ it is pointer not a variable, so it not making A type.

Yousaf

Day:

Date: / /

Declaring Inline function:

eg

~~Struct Time~~

- We can make function inside a class.
- These function can be said to be the ~~attribute~~ actions of the attribute.
- These functions don't require the member variables as function parameter. They can simply use them.

e.g:

```
Struct Time {

    int hour, min, sec;

void display_24 ( ) {
    cout<< hour%24 << min<< sec; endl;
    }

void display12_am_pm ( ) {

    cout<< hour%12<<":"<<min<<":"<<sec;
    if  (hour >= 12) {
            cout<<" pm'<<endl; }
    else {
            cout<<" am"<<endl; }

}
```

Day: Date: / /

- In struct Time we can see that we can use member functions simply no need to pass by parameters.

Usage:

```
int main () {

    Time t;
    t.h = 15;
    t.min = 24;
    t.sec = 30;

    t.display4();

}
```

This give 15:24:30.

```
    Time t_ptr;

    t_ptr = &t;

    ptr → display ();

}
```

This again gives 15:24:30.

Important points:

1) We can declare everything in a struct in any order.

we can even declare variable of struct inside function in struct
It is allowed.

e.g.

```
struct Time {

void Display () {
    cout << hour << min << sec;
}

int hour, min, sec;
};
```

allowed