

```

A a = new A();
A b = new A(); // If there's another
A b = (a).clone();
----- X -----

```

pointers:

- variable that stores address of another variable
- i.e. - direct address of mem loc
- stores address rather than value
- "\*" used to denote pointer
- 12 ptr's are possible (12x by variable).

```

int a = 5;
      a   p
      5   100
100    200

```

```

int * p = &a;
      a   p
      5   100
100    200
datatype → ptr variable.

```

```

int * p - declaration
* p - dereference.

```

eg:

```

int a = 5;
      a   p   p1
      5   100  200
int * p = &a; 100  200  300
int ** p1 = &p;

```

```

pf(a, p, *p, p1, *p1, **p1)

```

```

o/p: 5, 100, 5, 200, 100, 5.

```

```

a=5   *p1 = 100

```

```

p=100  **p1 = 5.

```

```

*p = 5

```

```

p1 = 200
----- X -----

```

```

int main()
{ }

```

```

int a = 10;

```

```

void show (int x)
{ }

```

```

x++;

```

```

pf(x);

```

```

show(a); o/p: 11

```

```

} pf(a);
           10
----- X -----

```

```

int main()
{ }

```

```

int a = 10;

```

```

show (x)

```

```

void show (int x)
{ }

```

```

*x++ ; *x = 10 *x++ = 11

```

```

} pf(x); // 100 pf(*x) // 11

```

```

pf(a)
{ }

```

```

o/p: 100

```

```

11

```

```

11.
----- X ----- X -----

```

Structure:

- user defined datatypes

- collection of variables (can be of diff types) under single name

- malloc creation

- calloc - initializes 0 at creation

↓

```
#include < stdlib.h >
```

\* p = 100 - starting address

p+1 = 102

p+2 = 104

p+3 = 106.

1	2	3
---	---	---

Integers so

Increasing by 2 at every step.

Struct name

```

{ }
int a;
float b;
char c;
int d;
};
```

n =

h+2 depending

h+2 upon size

h+1 of bytes

h+2 pointer/cursor moves

Struct name n;

access: n.a, n.d like that.

----- X -----

Struct student

{ }

int marks;

char name [10];

{ }

int	char
-----	------

SI = 100

```
int main()
```

```
{ struct student s1;
```

```
struct student s2 = {90, "Devraj";
```

```
pf(&s2.mark); // 90.
```

```
struct student *s1;
```

```
struct student *s2 = {100, "Mahij";
```

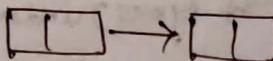
```
pf(s2->mark); // 100.
```

normal variable    ptr variable

s.name

s->name.

array of variables can be  
created:



```
int main()
```

```
{ int i;
```

```
struct student
```

$s[2]$ ;

// 1st student:

```
s[0].mark = 90;
```

```
s[0].name = "Rami";
```

// 2nd student // 3rd student.

(Or)

```
for (i=0; i<2; i++)
```

```
{
```

```
    &f("1.d", &s[i].mark);
```

```
    &f("1.s", &s[i].name);
```

```
}
```

```
pf same way.
```

% u - unsigned int.

pointers

declaration

mem allocation

getting i/p /scant / storing realme.

struct name

```
{ int a;
```

```
char b;
```

```
float c;
```

```
int main()
```

```
{
```

```
int i;
```

// struct name n [2];

```
for (i=0; i<2; i++)
```

```
{
```

```
&f("1.d", &a);
```

```
&f("1.s", &b);
```

```
{
```

```
&f("1.c", &c);
```

// struct name n [2] = { {1, 'a', 1.1},

{2, 'b', 2.2};

```
}
```

struct name \*p = n;

= no n[] just

n so enough

```
for (i=0; i<2; i++)
```

```
{
```

```
Pf("1.d\n", p+a);
```

```
Pf("1.s\n", p+b);
```

```
{ Pf("1.c\n", p+c); - p++;
```

Op:

we need to increase p by

1.

value (p++) by ending

only.

of loop or i++, p++ like

time.

Op:

1 2

9 8

1.1 2.2

100

200

300

400

500

600

700

800

900

1000

1100

1200

1300

1400

1500

1600

1700

1800

1900

2000

2100

2200

2300

2400

2500

2600

2700

2800

2900

3000

3100

3200

3300

3400

3500

3600

3700

3800

3900

4000

4100

4200

4300

4400

4500

4600

4700

4800

4900

5000

5100

5200

5300

5400

5500

5600

5700

5800

5900

6000

6100

6200

6300

6400

6500

6600

6700

6800

6900

7000

7100

7200

7300

7400

7500

7600

7700

7800

7900

8000

8100

8200

8300

8400

8500

8600

8700

8800

8900

9000

9100

9200

9300

9400

9500

9600

9700

9800

9900

10000

10100

10200

10300

10400

10500

10600

10700

10800

10900

11000

11100

11200

11300

11400

11500

11600

11700

11800

11900

12000

12100

12200

12300

12400

12500

12600

12700

12800

12900

13000

13100

13200

13300

13400

13500

13600

13700

13800

13900

14000

14100

14200

14300

14400

14500

14600

14700

14800

14900

15000

15100

15200

15300

15400

15500

15600

15700

15800

15900

16000

16100

16200

16300

16400

16500

16600

16700

16800

16900

17000

17100

17200

17300

17400

17500

17600

17700

17800

17900

18000

18100

18200

18300

18400

18500

18600

18700

18800

18900

19000

19100

19200

19300

19400

19500

19600

19700

19800

19900

20000

20100

20200

20300

20400

20500

20600

20700

20800

20900

21000

21100

21200

21300

21400

21500

21600

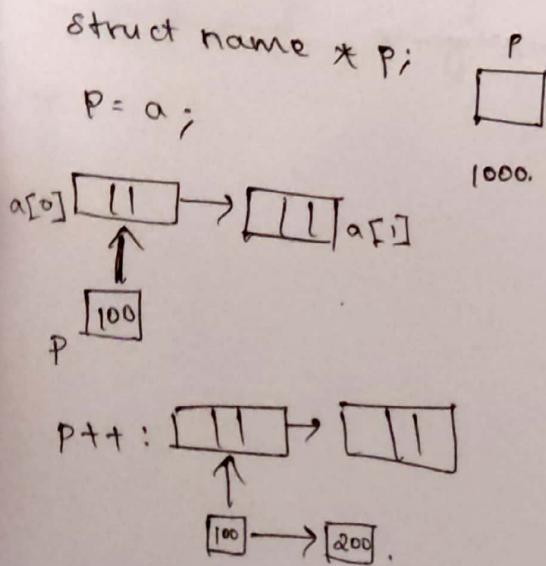
21700

```

struct name
{
    int a;
    char b;
    float c;
};

int main()
{
    struct name a[2];
    a[0] [ ] → [ ] a[i]
    100          200
}

```



1D and 2D array:

dynamic alloc use - mem

Is allocated at heap during runtime.

- C has library fn to req heap mem at runtime.

malloc - returns void ptr.

1D array:

```

int main()
{
    int n=5; // array size
    int i;
    int *a; // declaration
    a = (int *) malloc (n * sizeof(int)
                        typecast bcs // mem alch. );
}

```

```

for (i=0; i<n; i++)
{
    if (C + d, &(a+i)); // getting i/p
    for (j=0; j<n; j++)
    {
        Pf C + d, *(a+i));
    }
}

```

$a[i] \rightarrow$  value

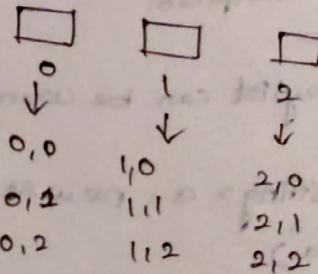
$a+i \rightarrow$  address.

2D array:

```

int main()
{
    int i, j, row, col, **a;
    if (C + fd fd, &row, &col);
    int **a = (int **) malloc (row * sizeof(int *));
    (row * sizeof(int *));
    (or) a = (int **) malloc (sizeof(int) * row);
    {
        *a = (int *) malloc (col *
        sizeof(int));
        → because in 2D array
        we get rows, st
    }
}

```



so we divide rows, that's why we specify "row" value in for loop.

$*a[i]$  → normal  $a[i]$  means address so we give it to rep value (dereference).

so now mem alloc for rows / col over

```

for (i=0; i<row; i++)
{
    for (j=0; j<col; j++)
        if (C + d, &a[i][j]);
}

```