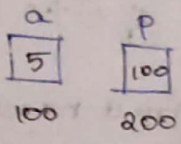
```
A a = new A();
A b = new A(); // ithuku bathi
A b = (a)a.clone();
```

— 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 (12 * by variable).

```
int a = 5;
int *p = &a;
```

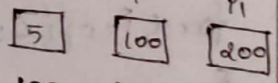


datatype → ptr variable.

int *p - declaration
*p - dereference.

eg:

```
int a = 5;
int *p = &a;
int **p1 = &p;
```

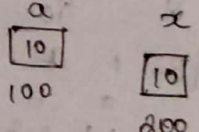


pf(a, p, *p, p1, *p1, **p1)
o/p: 5, 100, 5, 200, 100, 5.

a = 5 *p1 = 100
p = 100 **p1 = 5.
*p = 5
p1 = 200

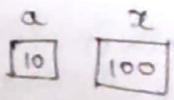
— X —

```
int main()
{
  int a = 10;
  void show (int x)
  {
    x++;
    Pf(x);
  }
  show(a); o/p: 11
} Pf(a);   10
```




```
int main()
{
  int a = 10;
  show (&a)
  void show (int *x)
  {
    *x++;   *x = 10 *x++ = 11
  } Pf(x); // 100  Pf(*x) // 11
Pf(a)
}
```

o/p: 100
11
11.



— X — X —

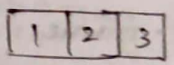
Structures:

- user defined datatypes
- collection of variables (can b of diff types) under single name
- malloc creation
- calloc - initialize '0' at creation

#include <stdlib.h>

*p = 100 - starting address

p+1 = 102
p+2 = 104
p+3 = 106.



integers so increasing by 2 at every step.

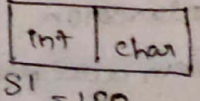
```
Struct name
{
  int a;
  float b;
  char c;
  int d;
};
Struct name n;
```

n = □
n+2 depending
n+2 upon size
n+1 of bytes
n+2 pointer/cursor moves

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

— X —

```
Struct student
{
  int mark;
  char name[10];
};
```



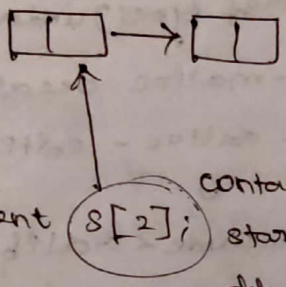
```
int main()
{
  struct student s1;
  struct student s2 = {90, "deva"};
  pf(s2.mark);  // 90.

  struct student *s1;
  struct student *s2 = {100, "mahi"};
  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 = "Ram";
// 2nd student  // 3rd student.
```

(or)

```
  for(i=0; i<2; i++)
  {
    sf("%d", &s[i].mark);
    sf("%s", &s[i].name);
  }
```

pf same vlag.

%u - unsigned int.

pointer:

declaration

mem allocation

getting i/p / scanf / storing values.

```
struct name
{ int a;
  char b;
  float c;
};
int main()
{
  int i;
// struct name n[2];
  for(i=0; i<n; i++)
  {
    sf("%d", &n.a);
    sf("%f", &n.c);
    sf("%c", &n.b);
  }
// struct name n[2] = {{1,'a',1.1},
                       {2,'b',2.2}};
  struct name *p = n;
```

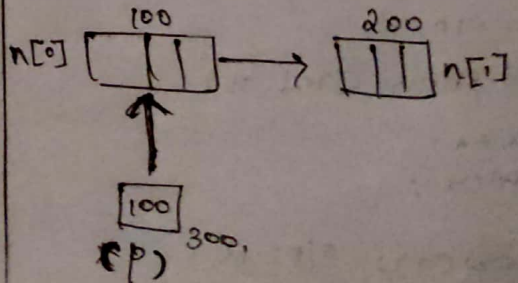
no n[] just n is enough

```
  for(i=0; i<2; i++)
  {
    pf("%d\n", p->a);
    pf("%c\n", p->b);
    pf("%f\n", p->c);
    p++;
  }
```

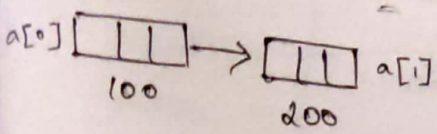
o/p:
1
a
1.1
only.

we need to increase ptr value, p++; by ending of loop or i++, p++ like this.

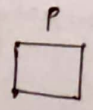
o/p: 1 2
a b
1.1 2.2

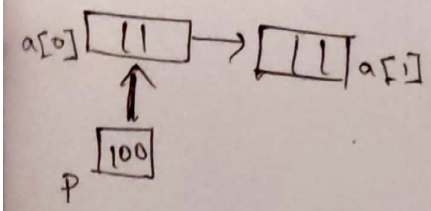


```
struct name
{
    int a;
    char b;
    float c;
};
int main()
{
    struct name a[2];
```

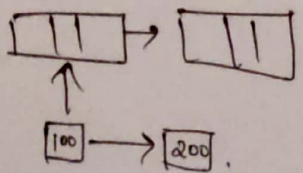



```
    struct name *p;
    p = a;
```





p++ :



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 alloctn.

```
for(i=0; i<n; i++)
{
    scf("%d", (a+i)); // no ampersand // getting i/p
}
for(i=0; i<n; i++)
{
    pf("%d", *(a+i));
}
}
```

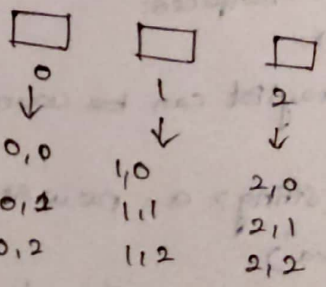
a[i] → value

a+i → address.

2D array:

```
int main()
{
    int i, j, row, col, **a;
    scf("%d %d", &row, &col);
    int **a = (int**) malloc
(row * sizeof(int*));
    (or) a = (int**) malloc(sizeof(int*)
for(i=0; i<row; i++)          *row);
{
    *(a+i) = (int*) malloc (col *
}                              sizeof(int));
```

→ because in 2D array we get rows, st



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

(a+i) → normal a+i means address so we giv '' to rep value (dereference).

so now mem alloc for rows & col over

for getting i/p: as usual

```
for(i=0; i<row; i++)
{
    for(j=0; j<col; j++)
        scf("%d", &a[i][j]);
```