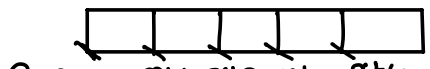
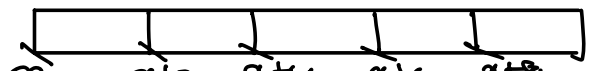
char c_arr[5];



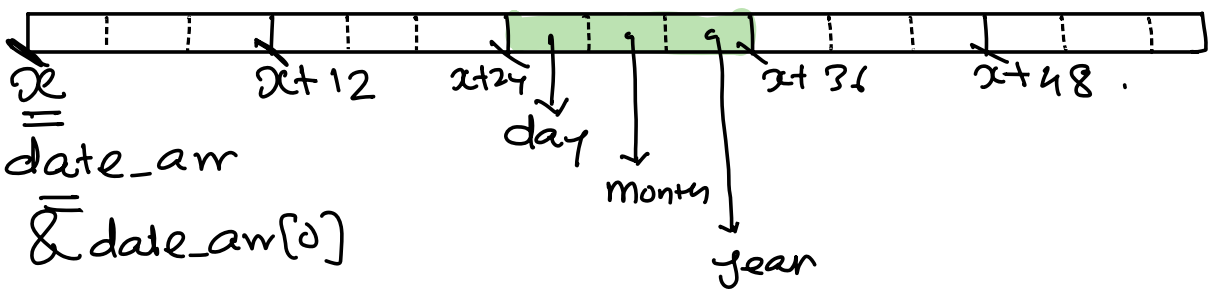
c_arr
=
&c_arr[0]
=
x

short s_arr[5];



x
=
s_arr
=
&s_arr[0]

Struct Date date_arr[5];



x
=
date_arr
=
&date_arr[0]

&s_arr[0] = x
&s_arr[1] = x+2
&s_arr[2] = x+4
&s_arr[3] = x+6
&s_arr[4] = x+8
&date_arr[0] = x
&date_arr[1] = x+12
&date_arr[2] = x+24
&date_arr[3] = x+36
&date_arr[4] = x+48

Let T be a data type.
Let arr be array N of T
Let i be the index ($0 \le i < N$).

```
T arr[N];
```

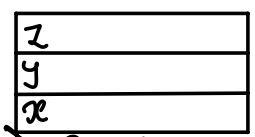
$$\&arr[i] = arr + i * sizeof(T)$$

$$arr \rightarrow \&arr[0]$$

---

c_arr == x1	&c_arr[2] == x1 + 2 * 1
s_arr == x2	&s_arr[2] == x2 + 2 * 2
i_arr == x3	&i_arr[2] == x3 + 2 * 4
d_arr == x4	&d_arr[2] == x4 + 2 * 8
date_arr == x5	&date_arr[2] == x5 + 2 * 12.

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

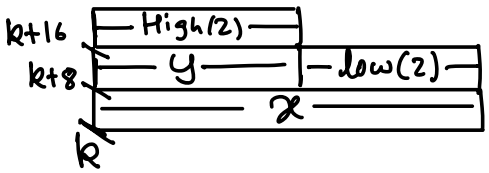


k = &inA.

&inA.x == k + 0
&inA.y == k + 4.
&inA.z == k + 8

```
struct B {
    double x;
    int y;
    double z;
};
```

```
struct B inB;
```



$8 + 4 + 8 = 20$ | &inB == k

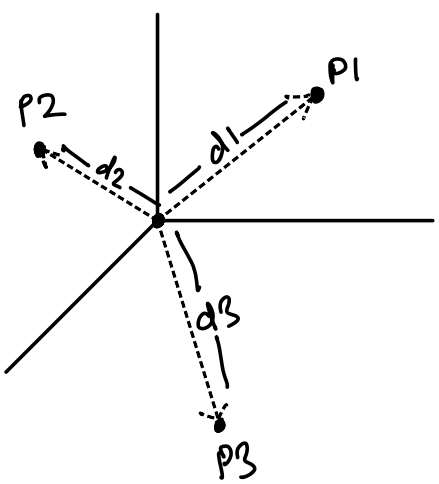
&inB.x == k + 0

&inB.y == k + 8

&inB.z == k + 12

Offsets of members with respect to base address of structure instance.

offset = Distance from some fixed location



offset of member of structure ==

Distance in bytes w.r.t. base address of any instance of that struct

---

```
Struct X {
    int a;
    char b;
    float c;
};
```

```
Struct inX;
```

offset of first member = 0

offset of k^{th} member

= Dependent on types of first (k-1) members in packed struct.

= Dependent on types of first (k-1) members + bytes left for padding done for keeping addresses of members aligned to 4 or 8 bytes!

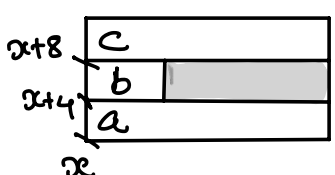
```
/* Packed struct */
Struct X {
    T1 mem_1;
    T2 mem_2;
    |
    Tn mem-n;
}
```

offset of mem_k $(1 \le k \le N)$

$= \text{Sizeof}(T1) + \text{Sizeof}(T2) + \cdots + \text{Sizeof}(T_{k-1})$

$= \sum_{r=1}^{r=n-1} \text{Sizeof}(T_r)$

```
Struct X
{
    int a;
    char b;
    float c;
} inX;
```



inX.

inX.a

inX.b

inX.c

inX . a

↓ ↓ ↓

Base Addr + offset (a within Struct X) = x + 0

of inX

&inX

inX . b

↓ ↓ ↓

Base Addr + offset (b within Struct X) = x + 4

of inX

&inX

inX . c

↓ ↓ ↓

Base Addr + offset (c within Struct X) = x + 8

of inX

&inX

```
Struct X
{
  int a;
  char b;
  float c;
} inX;
```

let Base address of inX be 1000.

M[1000:1003] → inX.a

M[1004] → inX.b

M[1008:1011] : inX.c

```
inX.c = 3.14f;
```

&inX. (1000) + offset (c, Struct X) (8)

M[1008] = 3.14 M[1008:1011] ← 3.14f

BE

D0		D31	1 byte word	2 byte word	4 byte word
D0	D7	$\overline{BE0}$	= 0	= 0	= 0
D8	D15	$\overline{BE_1}$	= 1	= 0	= 0
D16	D23	$\overline{BE_2}$	= 1	= 1	= 0
D24	D31	$\overline{BE_3}$	= 1	= 1	= 0

```
Struct T { T1 mem1;
           T2 mem2;
           :
           Tn mem-n;
} inT;
```

A) inT.mem_k = rhs; [write]

B) lhs = inT.mem_k [read]

1) Read / Write.
   ↓ B) ↓ A)

2) Base Addr. for read/write:

inT.mem_k

&inT + offset(mem_k, struct T) = Base-mem_k

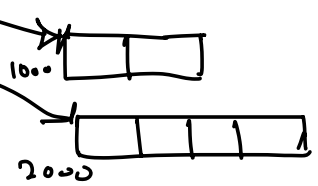
from Base_mem to Base_mem + ?

↓

Size of (Tk)

short* p1;

int * p2;



Assuming valid addresses in p1 & p2

*p1 = rhs;	*p2 = rhs;
lhs = *p1;	lhs = *p2;
M[1000:1003] ← rhs	M[2000:2003] ← rhs
lhs ← M[1000:1003]	lhs ← M[2000:2003].

p + k	*p = rhs;	arr[k]	inX.mem
p − k	lhs = *p;		

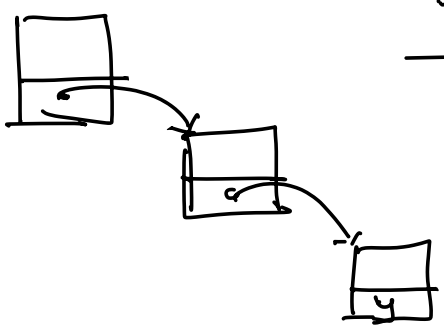
$$p \rightarrow mem \equiv (*p).mem$$

Struct x* pX = &inX;

&pX

pX → a	(*pX).a	*pX.a
pX → b	(*pX).b	(*pX).a
pX → c	(*pX).c	

pA → pB → pC → y *(*((*pA).pB).pC).y

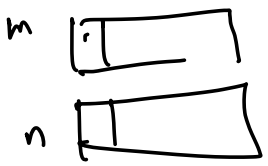


*px.a px.a

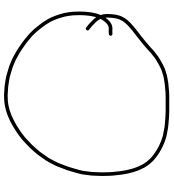
*(&px + offset(a, struct x))

```
int m = 10;
int n;
```

(n) = (m); (n) = 10; 10 = a;



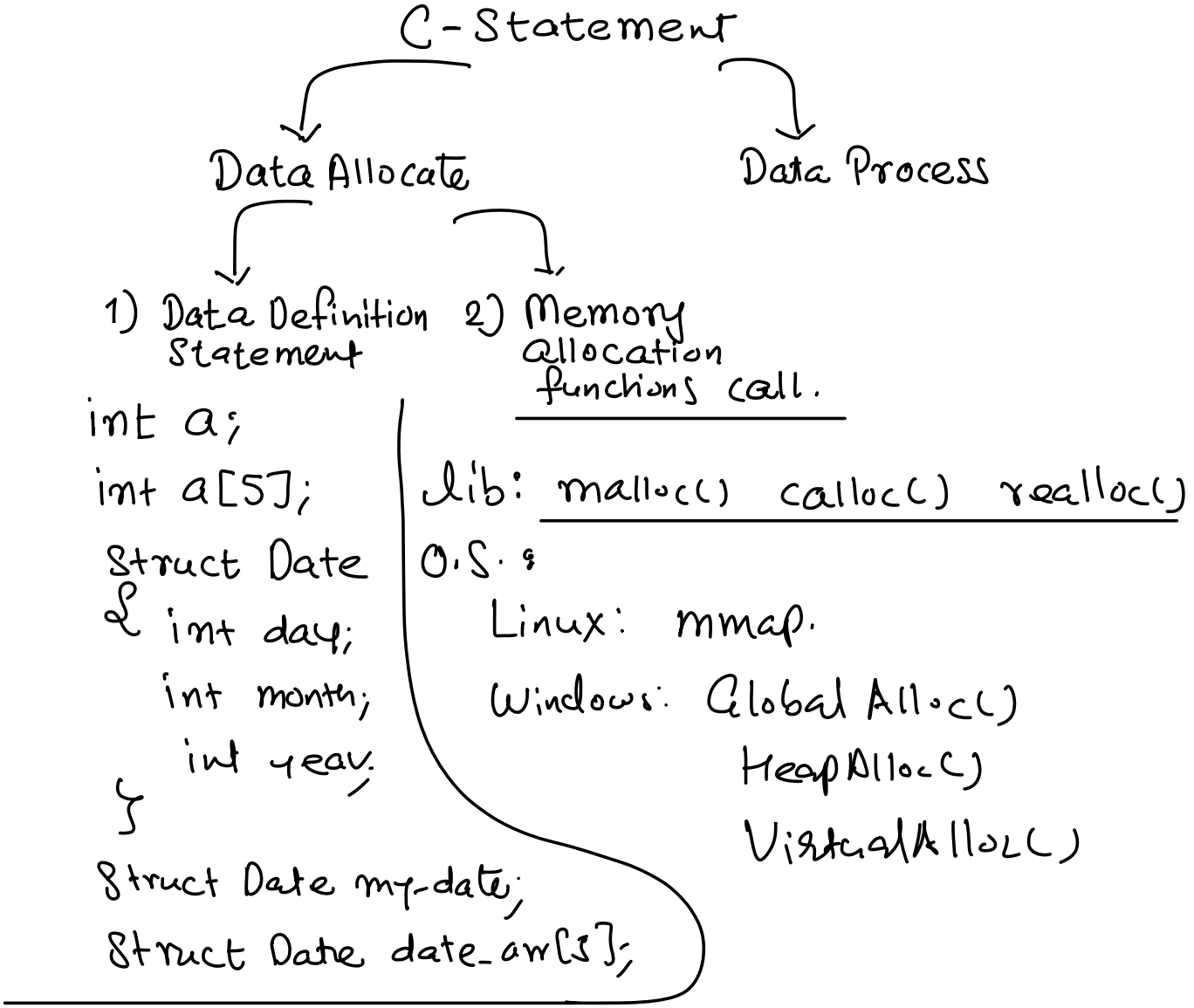

```
void test(void)
{  int m;
}
```



malloc()

```
void test(void)
{ int (n);
  int *p = &n;
  *p = 100;   n = 100;
   m = *p;    m = n;
  test_(&m)
}

void test_1(int *p)
{ *p
}
```


C-Statement
Data Allocate
Data Process
1) Data Definition Statement
2) Memory allocation functions call.
int a;
int a[5];
Struct Date
{ int day;
int month;
int year;
};
Struct Date my_date;
Struct Date date_arr[5];
lib: malloc() calloc() realloc()
O.S.:
Linux: mmap.
Windows: GlobalAlloc()
HeapAlloc()
VirtualAlloc()