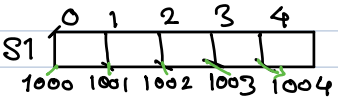
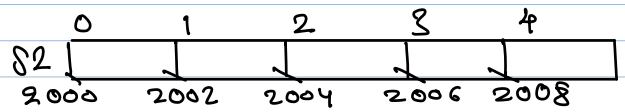
char S1[5];

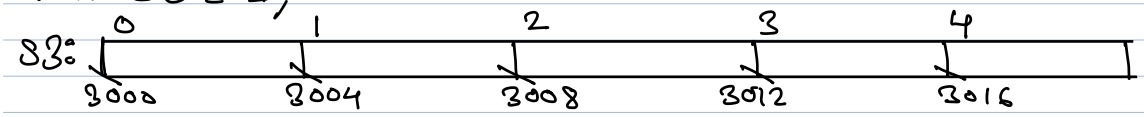


S1 = 1000

short S2[5];



int S3[5];



S1[2]	S1[2] = rhs;	lhs = S1[2];
S2[2]	S2[2] = rhs;	lhs = S2[2];
S3[2]	S3[2] = rhs;	lhs = S3[2];

S1 = 1000 [2] → 1002

S2 = 2000 [2] → 2004

S3 = 3000 [2] → 3008

S1 = 1000 S1[2] 1002 = 1000 + 2 * 1

S2 = 2000 S2[2] 2004 = 2000 + 2 * 2

S3 = 3000 S3[2] 3008 = 3000 + 2 * 4

T = Type.

T arr[N];

arr[i] means which location.

base addr. of arr + i * sizeof(T)

where T = Type of element of which array is made of.

arr[i] = rhs.

arr + i * sizeof(T) = base addr.

from this addr. sizeof(T)

lhs = arr[i]

arr + i * sizeof(T) = base addr.

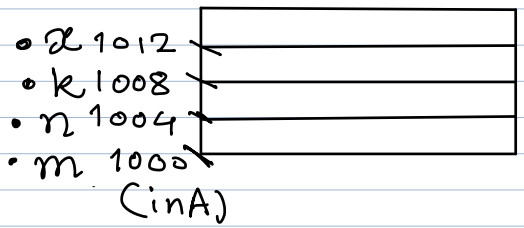
from base addr. sizeof(T) bytes read.

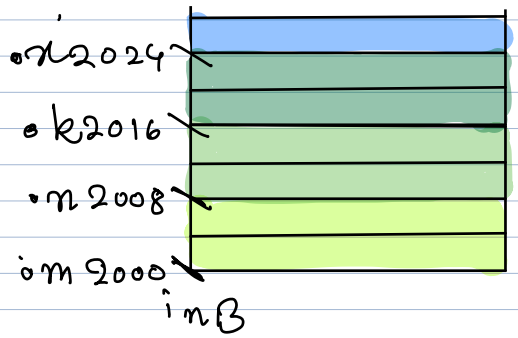
```
struct A
{ int m;
  int n;
  int k;
  float x;
};
```

```
struct B
{ double m;
  double n;
  double k;
  float x;
};
```

```
struct A inA;
inA.x = 3.14f;
```

```
struct B inB;
inB.x = 3.14f;
```





&inA.m = (1000)

&inA.n = (1004)

&inA.k = (1008)

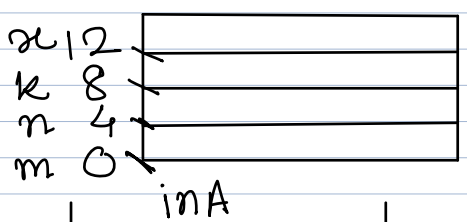
&inA.x = (1012)

&inB.m = (2000)

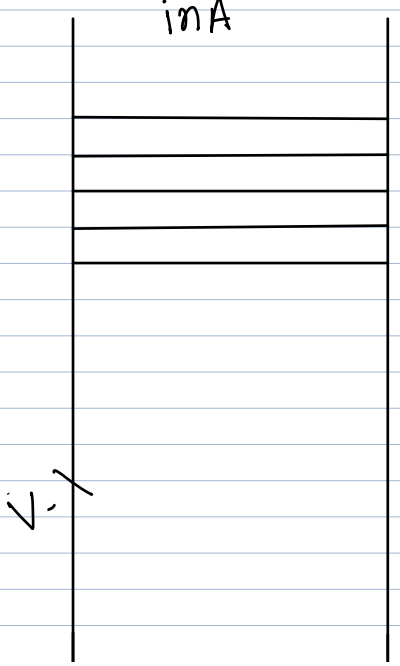
&inB.n = (2008)

&inB.k = (2016)

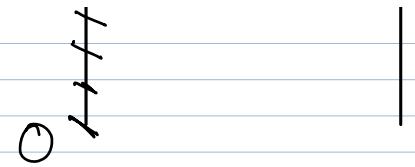
&inB.x = (2024)



The address every member gets after aligning structure instance at the start of memory is known as the 'Offset' of the member.



RAM / Virtual Addr.



struct A.

instance_of_A.member_of_A.

Base address of instance of A +
offset of member in struct A
= base addr. of member of A
in instance of A
from this base addr.
sizeof(type of (member_of_A))
bytes block can be read from
or written to!

inA.x = 3.14f

Base addr. of inA = 1000

offset of member 'x' in inA = 12.

Address of member 'x' in inA = 1000 + 12
= 1012.

from 1012 how many bytes can be read / written?

type of member 'x' in struct A.

float, sizeof (float) : 4.

∴ M[1012 : 1015]

lhs = inA.x;

inA.x = rhs;

Struct A* pA = &inA;

pA → x | (*pA).x

pA → pB → pC → mem. (*pA).x

(*(*((*pA).pB)).pC).mem

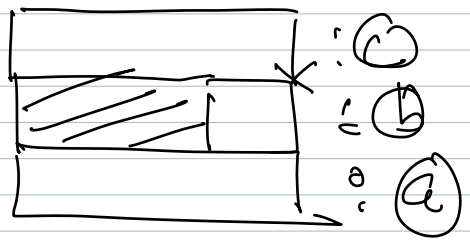
(*pA).x.

pA → x pB → x

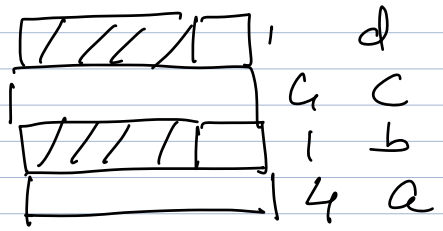
(*pA).x
1000 . x

(*qB).x
2000 c x

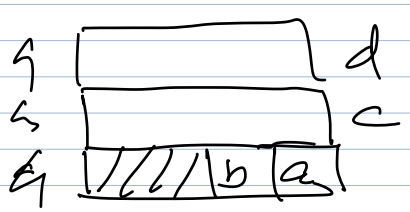
```
Struct C
{ int a; 10
  char b; 14
  float c; 15
};
```




```
Struct A
{ int a;
  char b;
  int c;
  char d;
}; 16
```




```
Struct B
{
  char a, b;
  int c, d;
};
```



Application Binary Interface

ABI | ABI elf i386

ABI elf x86_64.

Category I: Built-in data types.
& structure

Category II: pointer to array
& pointer to functions.

Category III: pointer to pointer.

Complicated Decl.

---

char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, long double.

struct {
=
};

struct Date ← tag of structure
{
int day;
int month;
int year;
};

T * p;

Rule 1: One data definition stmt. defines exactly one variable.

Rule 2: One should always start reading the C-decl/def from variable name.

p * T

p is pointer to T.

int *p; struct Date *pDate;

---

int n;
- int → type annotation.
- n → variable name.

T *v;

T*
- T* → type annotation.

Category - II :

1) Define an array 5 of int.

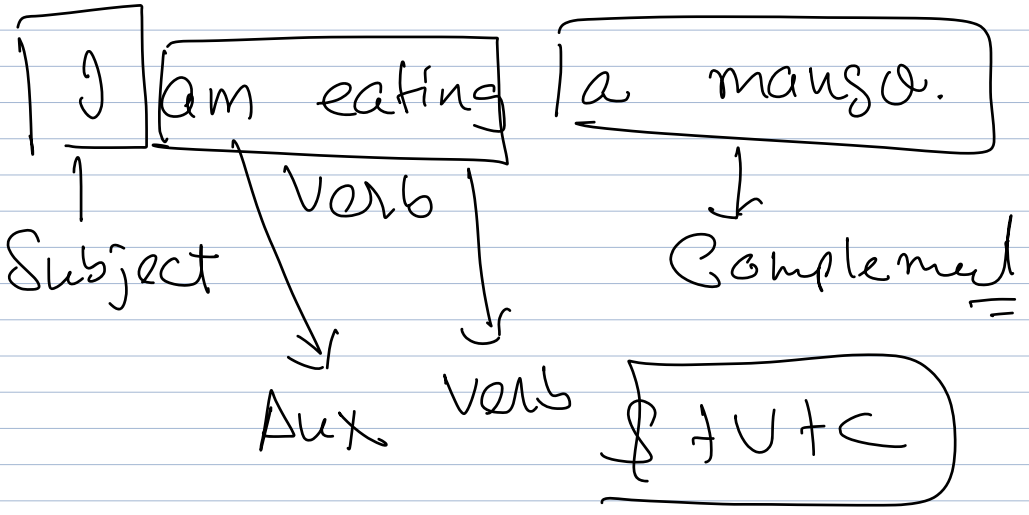
int arr[5]; int arr[5];

int [5];

2) Declare a function accepting two params and returning integer.

int my_func(int, int);

int (int, int)



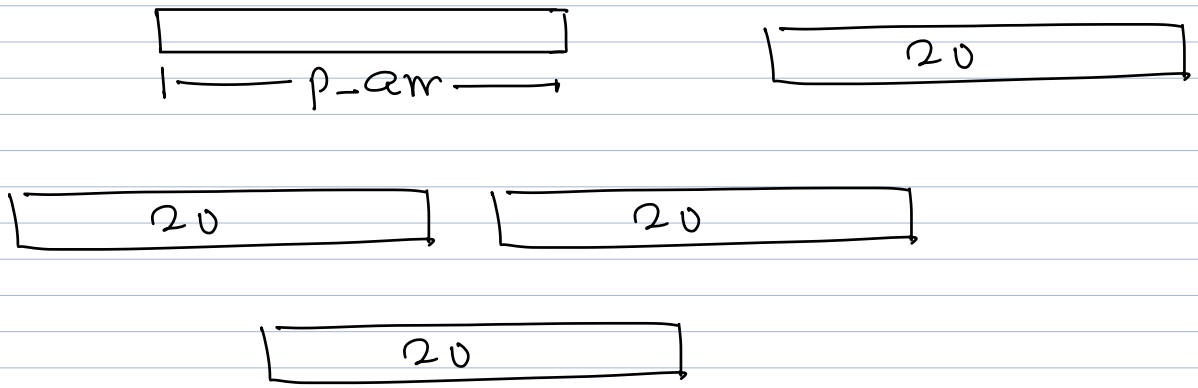
---

1) Define a pointer to an array 5 of integers.

int *parr [5] | int arr[5];

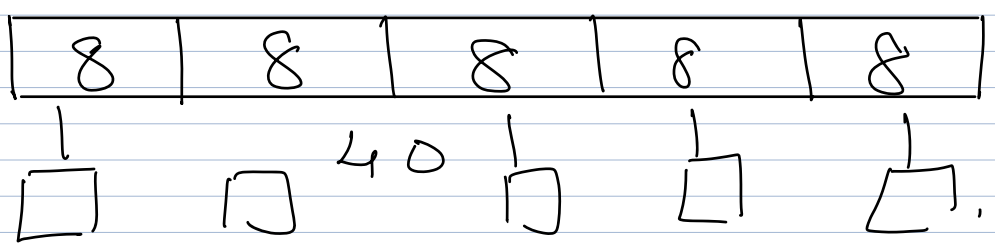
int *p_ar[5];

p_ar to be a point to
array 5 of int.



int *p_ar[5];

p_ar is an array 5 of int *

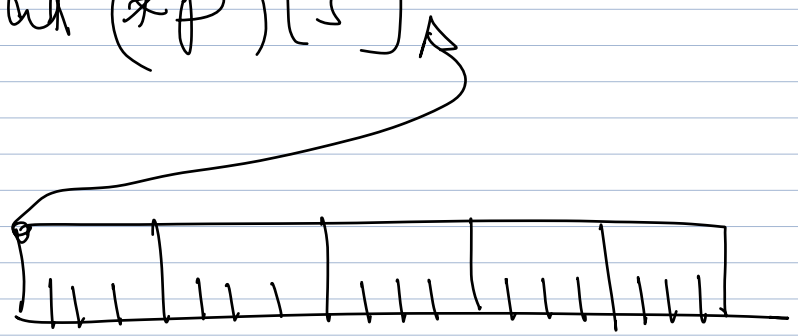


int *p_ar[5]; a+b *c

int (*p_ar)[5]; (a+b) *c

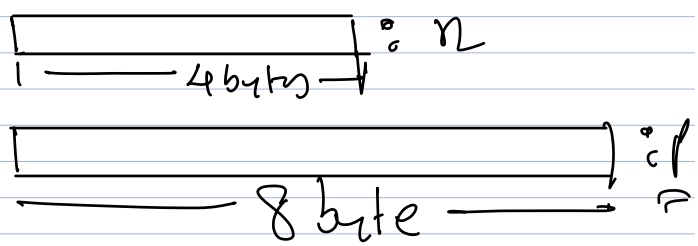
2) Define a pointer to function accepting two integers & returning an int.

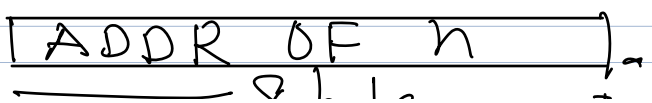
```
int (*p)(int, int);
    (*p)(      )
int (*p)[5]
```



Category III : pointer to pointer.

```
int n;
int* p;
p = &n
```





Date

&p

int n;

int *p = &n;

&p

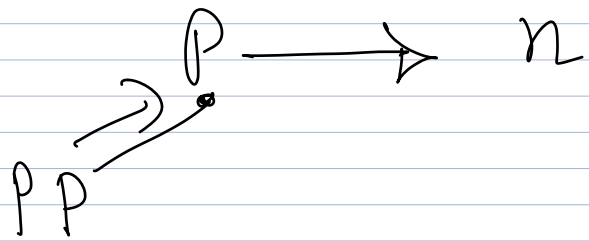
int *pp

int **pp;

pp = &p;

*pp

**pp



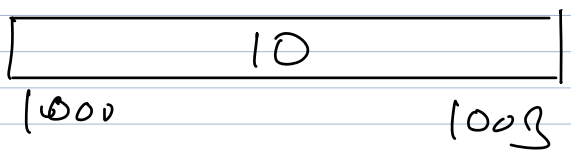
*p → n.

*(*pp)

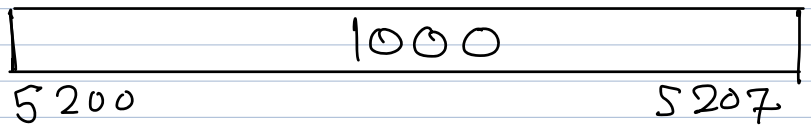
**pp.

---

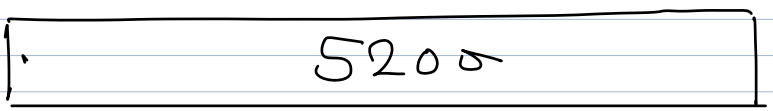
int n = 10;



int *p = &n;



int **pp; pp = &p;



*pp = 5200

= *(*pp)

---

int *p1 = 0;

char* p2 = 0;

double* p3 = 0;

Short* p4 = 0

Struct Date* p5 = 0;

p1 + 1 = Addr in (p1) + 1* Sizeof (pointee)
= 0 + 1*4 = 4

p2 + 1 = 0 + 1* Sizeof(char) = 0+1*1 = 1

p3 + 1 = 0 + 1* sizeof(double) = 0 + 1*8 = 8

p4 + 1 = 0 + 1 * sizeof(short) = 0 + 1*2 = 2

p5 + 1 = 0 + 1 * sizeof(struct Date)

= 0 + 1 * 12 = 12

---

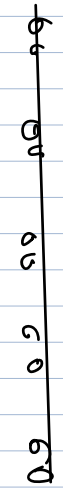
human.

kitten

Lion

Jiraffe

T-Rex



T * p = v

p + k = v + k * sizeof(T)

p - k = v - k * sizeof(T)

T arr[N];

&arr[k] = base addr of arr + k * sizeof(T)

p + k = v + k * sizeof(T)

int arr[5];

I want base addr. of array in p.

p = pointer to array element type.

int *p = base addr. of arr.

p + 0 == &arr[0]

p + 1 == &arr[1]

p + 2 == &arr[2]

p + 3 == &arr[3]

p + 4 &arr[4]

*(p + 0) == *(&arr[0]) == arr[0]

*(p + 1) == *(&arr[1]) == arr[1]

*(p + 2) == *(&arr[2]) == arr[2]

*(p + 3) == *(&arr[3]) == arr[3]

*(p+4) == *(&arr[4]) == arr[4]

int *p;

int *p = [ ];

int (*p_arr)[5]

int *p = &arr[0];

int arr[5];

arr ⟶ int [5]

arr[0] ⟶ int

&arr[0] ⟶ int *

---

int a[5];

int *p;

main()

{ p = &a[0];

for(i=0; i<5; ++i)

*(p+i)

}

---

int arr[5]; int (*p)[5]

int*.

int *p = arr; // &arr[0]
// addr
// type.

arr

int. [5]

int* = int (5)

---

*(p+0) == *(&arr[0]) == arr[0]

*(p+1) == *(&arr[1]) == arr[1]

*(p+2) == *(&arr[2]) == arr[2]

*(p+3) == *(&arr[3]) == arr[3]

*(p+4) == *(&arr[4]) == arr[4]

p[0] = *(arr+0)

p[1] = *(arr+1)

p[2] = *(arr+2)

p[3] = *(arr+3)

p[4] = *(arr+4)

```
void insertion_sort(int *p_arr, size_t N)
{  int i, j, key;
   for(j=1; j<N; ++j)
   {  key = p_arr[j];
      i = j-1;
      while(i>-1 && p_arr[i]>key)
      {
         *(p_arr+i+1) = *(p_arr+i);
         // p_arr[i+1] = p_arr[i];
         i = i-1
      };
      p_arr[i+1] = key;
}
```

```
main()
{ int a[8]={100,50,71,60,
            9,11,10,2};
    insertion_sort(a,8);
}
```

char*

1] Define an array 4 of pointer to function

1) Define an array 4 of pointer to function accepting two integers and returning and integer.

Start from deciding a variable name for the array, pfn_arr.

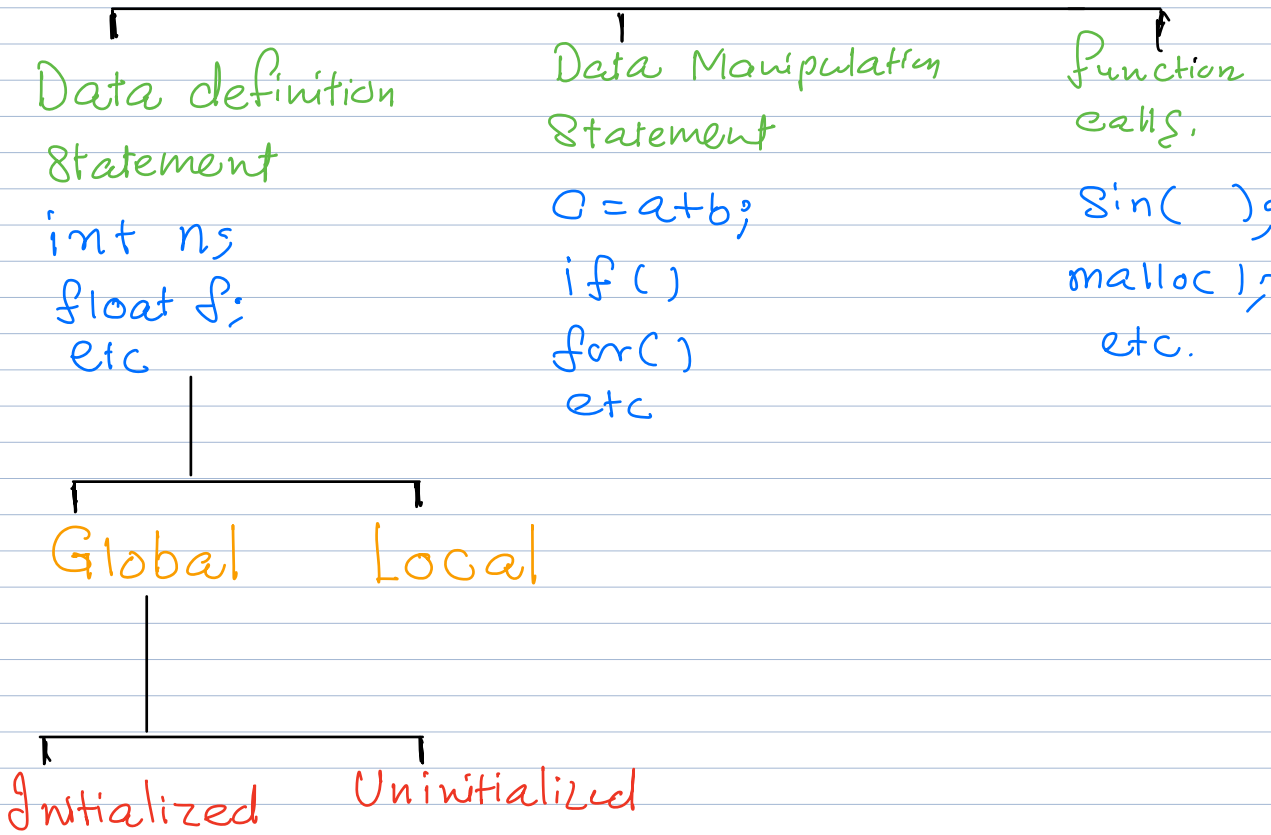
```
int (*pfn_arr[4]) (int, int);
```

```
Int main(void)
{

}
```

```
int (*pfn_arr[4])(int,int) =
  { cpa_add, cpa_sub, cpa_mul, cpa_div };
  int m=20, n=10, rs=0;
  int i;
  for(i=0; i<4; i++)
```

rj = ptn_anl ('m', n);

Classification of Statements in C.



1) Global initialized D.D.S.
2) Global uninitialized D.D.S.
3) Local Data Def. S.
4) Data Man. Stmt (local)
5) function call

SRC → Preprocessing → Preprocessed Src code

Preprocessed Source Code → Compile → Assembly Source code.

Assembly Source Code → Assembler → Object Code

Object Code → Linker → Executable code.

1) variable declare: to convey type of variable to compiler.

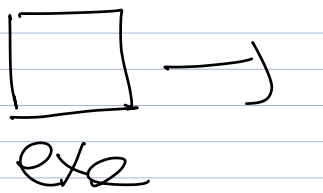
2) variable definition: Ask compiler to allocate a memory for variable.

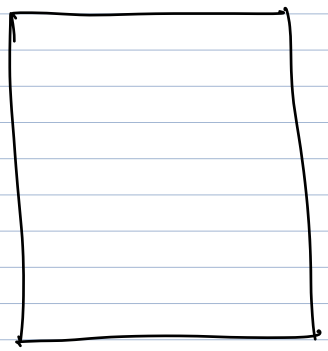
Definition Statement → Decl. also

int n=10; // Define ⇒ dec

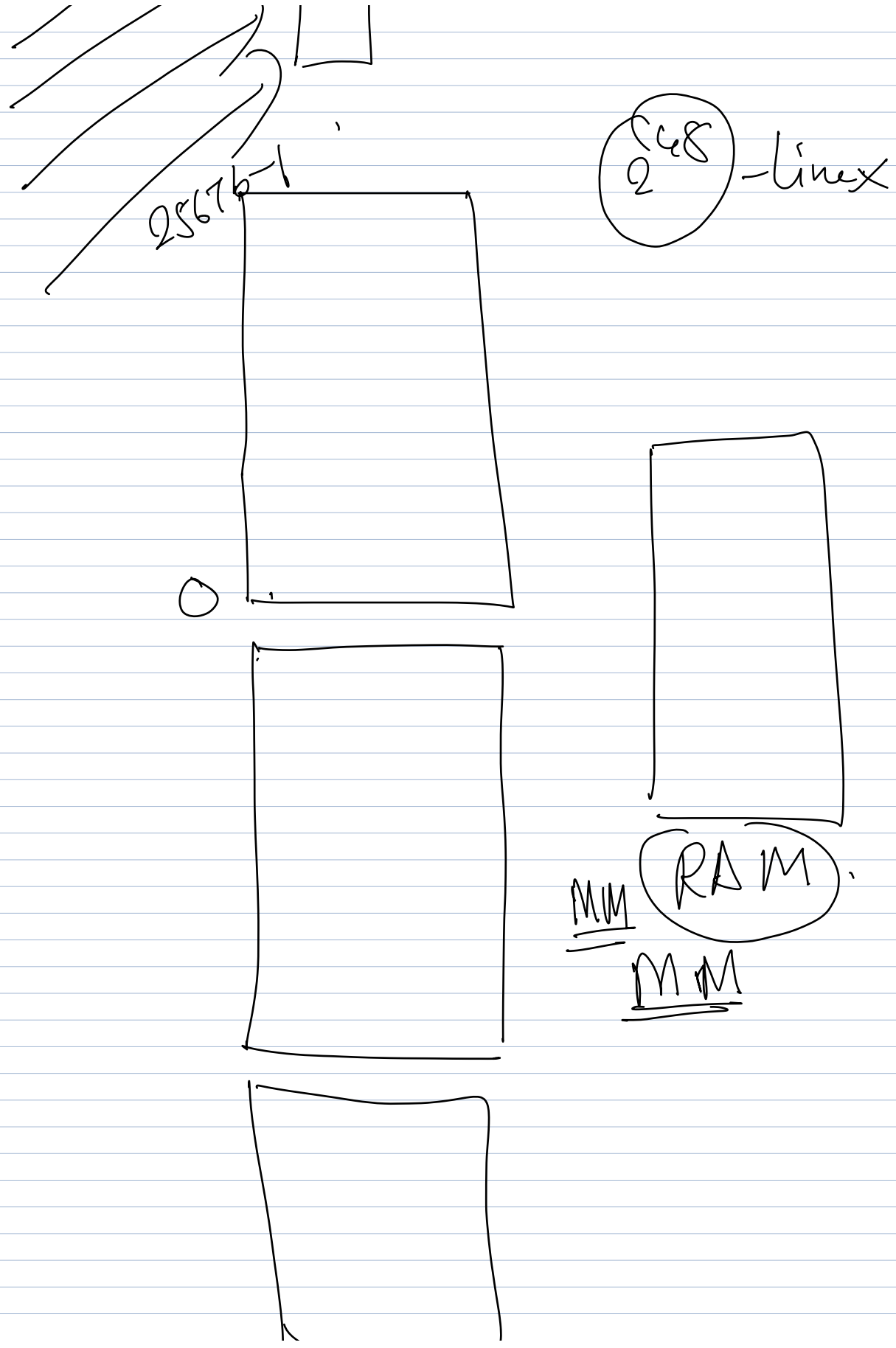
extern int n; ✗

```
int n;

void swap (int, int);

void swap (int n1, int n2)
{

}
```





APP / PROCESS
O.S. | Address range.


2567b-1
0
2^{48} -linux
RAM
MM
MM