A job ready bootcamp in C++, DSA and IOT

Pointers

Saurabh Shukla (MySirG)

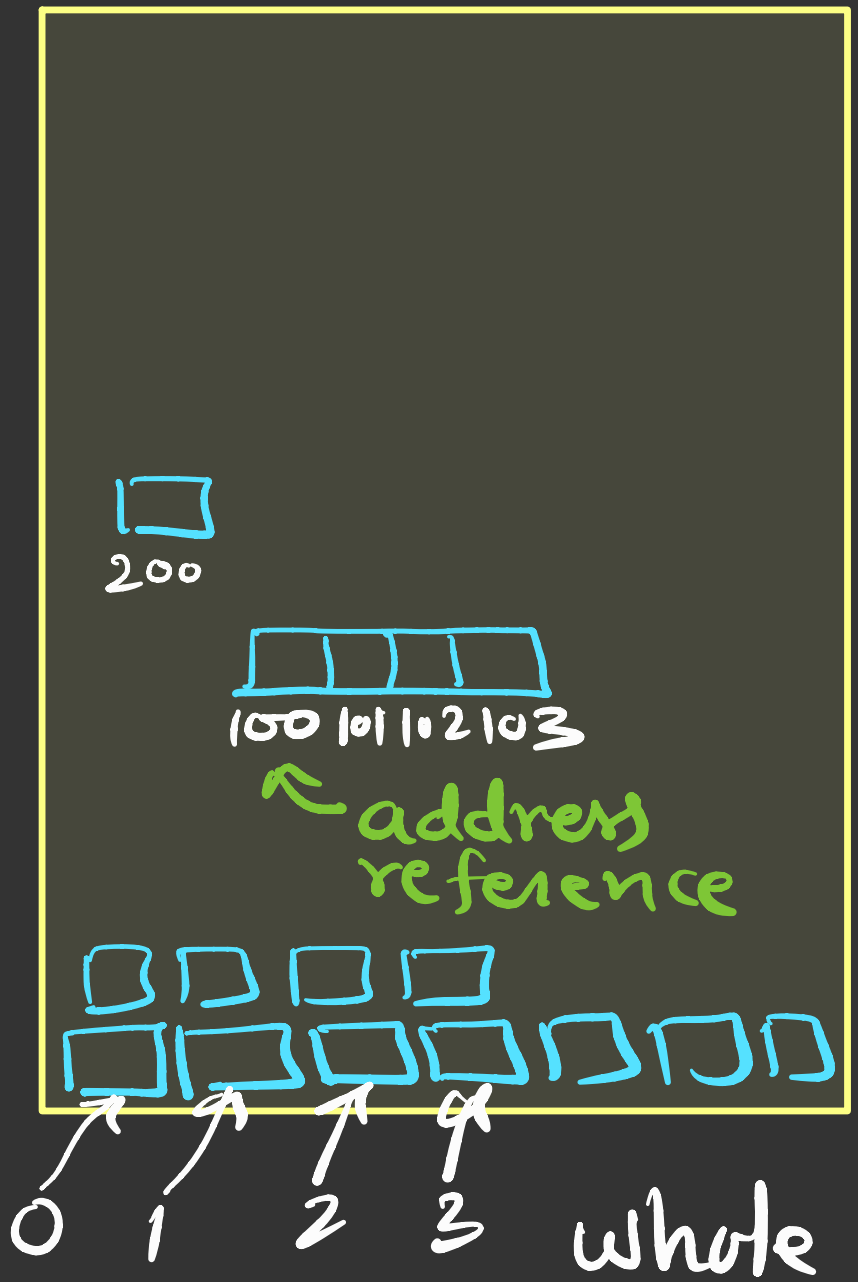
Agenda

1. Introduction to memory address
2. Referencing and Dereferencing operators
3. What is pointer?
4. Size of pointer
5. Base address
6. Data type of pointer
7. Extended concept of pointers
8. Pointer's Arithmetic
9. Call by reference
10. Pointers and arrays
11. Pointers and strings
12. Array of pointers
13. Pointer to array
14. Wild pointer
15. NULL pointer
16. Dangling pointer
17. Void pointer

Introduction to Memory Address

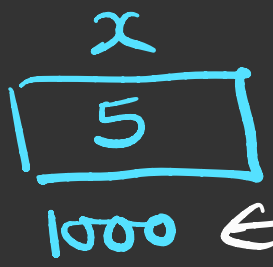
```
int x = 5;
char m = 'A';
```

logical address



Referencing and Dereferencing operators

```
int x=5;
printf("%d",&x); 1000
printf("%d", *&x ); 5
```



address
Reference
whole number
OS will decide

*&x ≈ x

&

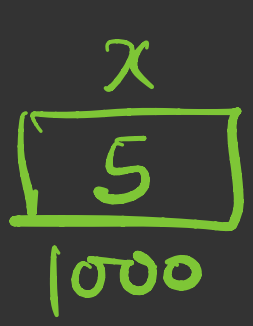
- Address of operator
- Referencing operator
- Unary operator
- & ___ ← variable

*

- Indirection operator
- Dereferencing operator
- Unary operator
- * ___ ← address

```
int x=5;
&x=7;
```

error




```
char str[20];
str = "MySirG";
```

address

```
strcpy(str, "MySirG");
```

&x is not a variable.
&x is just a way to represent address of variable x.
&x ⇒ 1000
&x is considered as a constant value

Constant =

error

```
int x = 5;
int *j;
j = &x;
```

j is a pointer variable

```
printf("%d %d %d", &x, j, *&x);
```

1000 1000 5

```
printf("%d %d %d", *j, x, &j);
```

5 5 2000

*j ≅ x

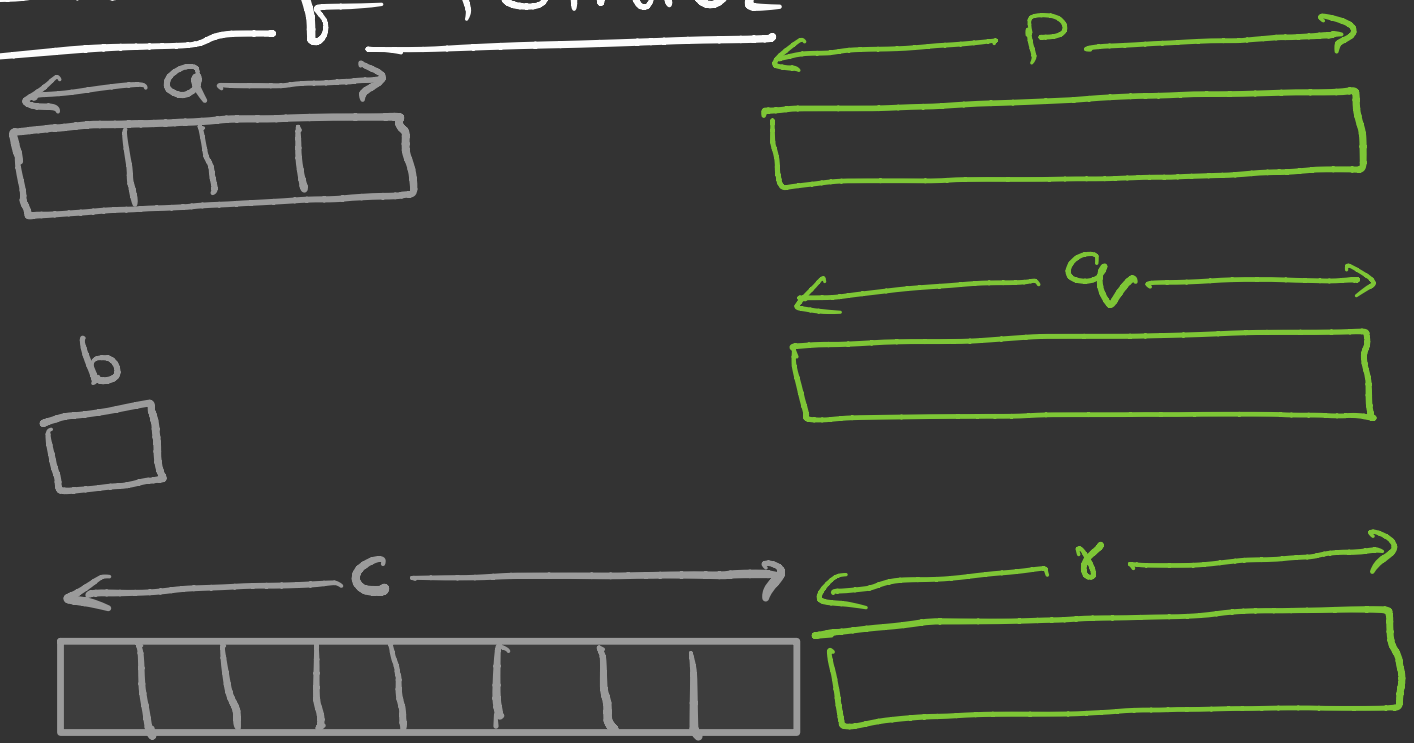
*j
*1000
x

What is a Pointer?

Pointer is variable, which contains address of another variable.

Size of Pointer

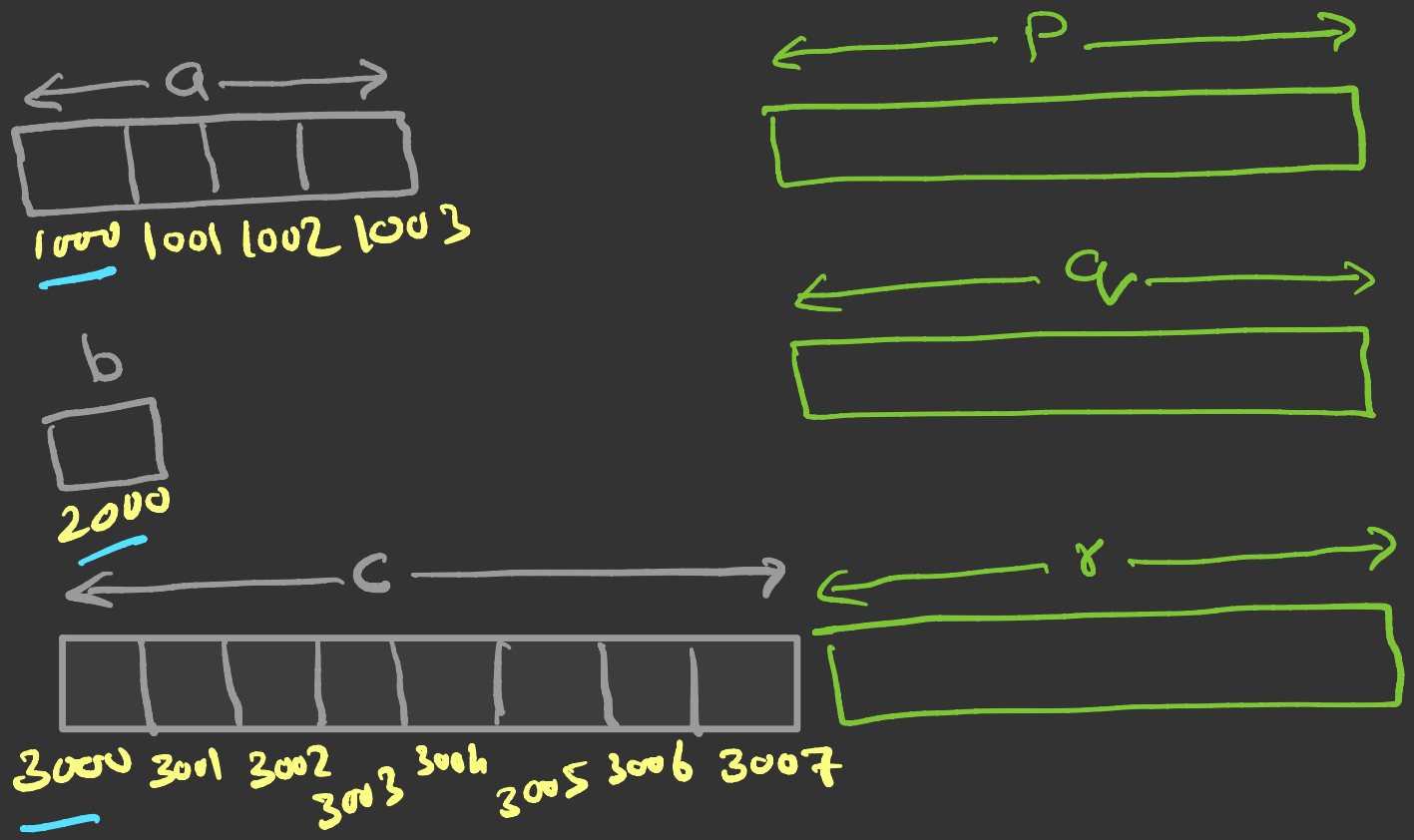
```
int a, *p;
char b, *q;
double c, *r;
```



- ordinary variables का size उसके data type पर depend करता है।
- size of pointer variable does not depend on it's data type

Base Address

```
int a, *p;
char b, *q;
double c, *r;
```

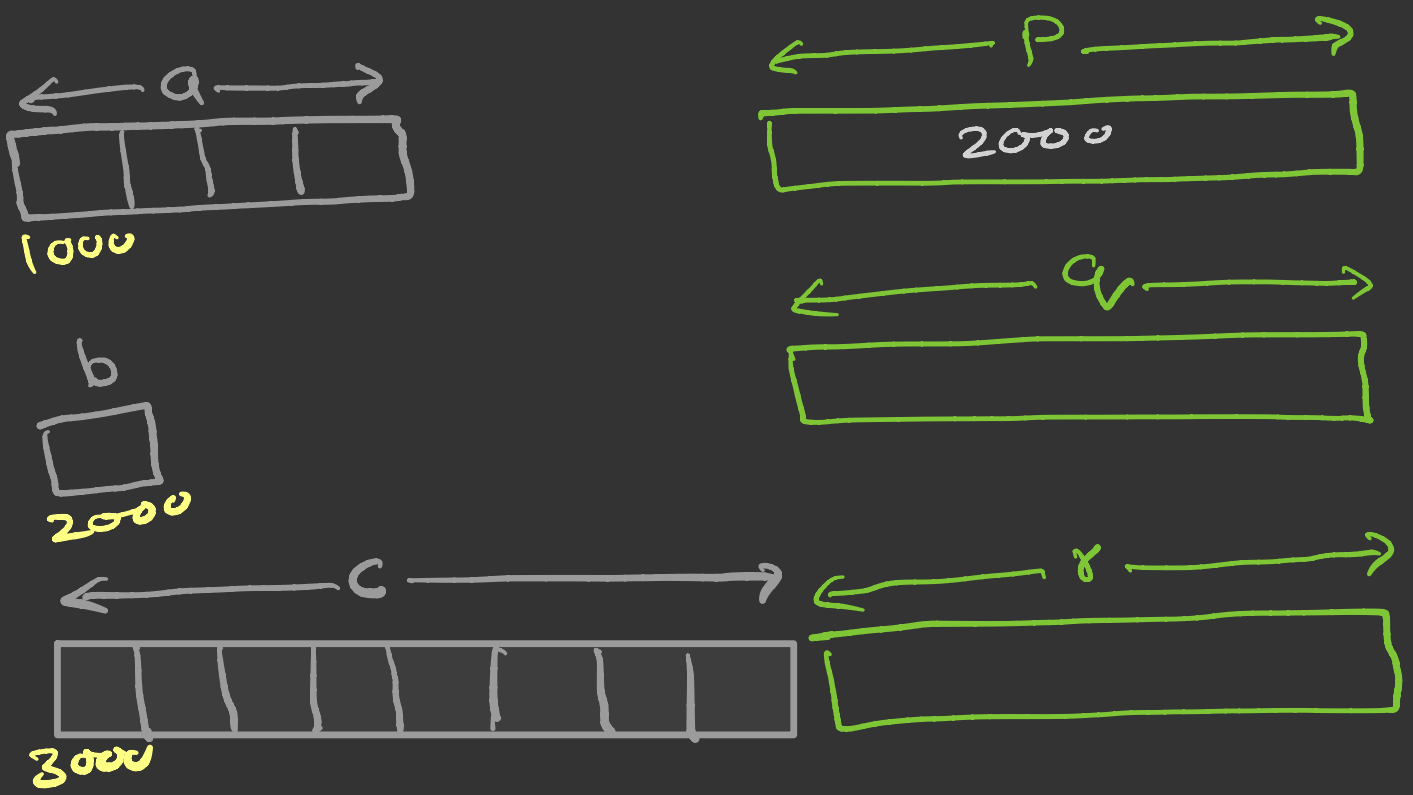


Base address

pointer variable always contains base address

Data Type of Pointer

```
int a, *p;
char b, *q;
double c, *r;
```

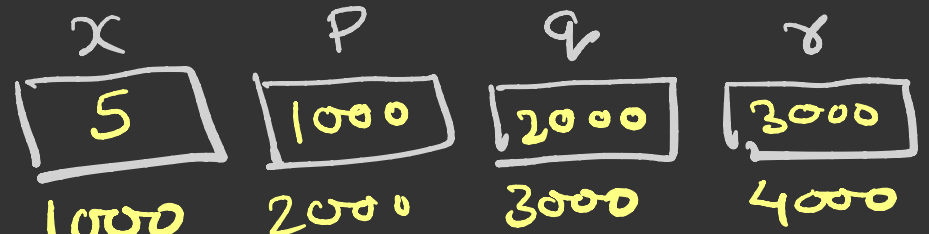


p = &b ; *p ≃ b

char *q ; q is not going to store character constant, but it is going to store address of char variable.

Extended Concept of Pointers

```
int x=5, *p, **q, ***r;
```




```
p=&x;
q=&p;
r=&q;
```

*p

*(*r)
*q
p

1000 4000 1000 1000

```
printf("%d %d %d %d", p, &r, *q, &x);
```

1000 3000 5 5

```
printf("%d %d %d %d", **r, &q, **&p, x);
```

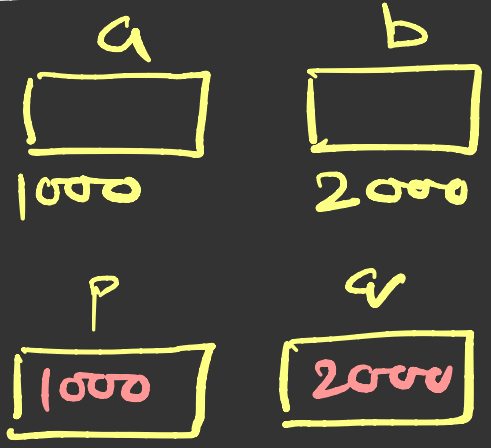
5 5 5 5

```
printf("%d %d %d %d", ***r, **q, *p, **&p);
```

```
int *p;
p = &p;   X
```

Pointer's Arithmetic

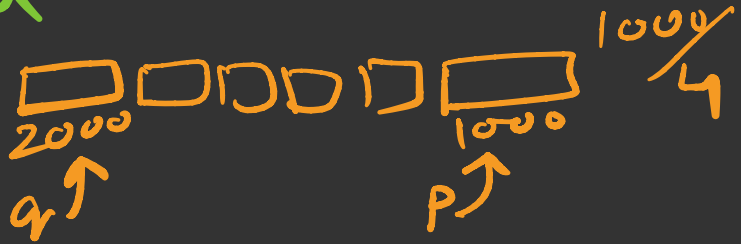
```
int a, b, *p, *q;
p = &a;
q = &b;
```



P+1 1004
P+2 1008
P+5 1020
q-1 1996
P-2 992
q-P 250

errors:
- P+q
- P*q
- P/q
- P*5
- P/4

valid:
- P+2
- P-5
- q-P



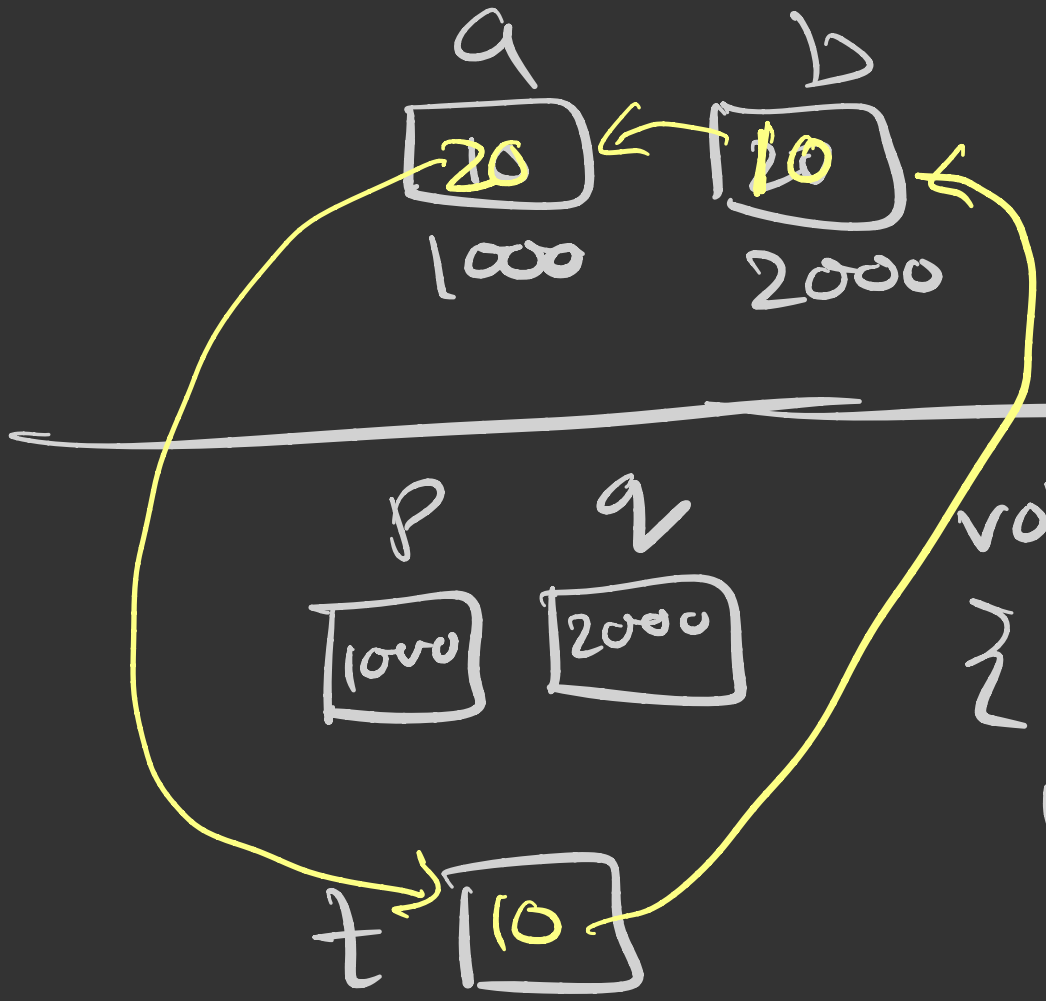
P+i => Address in P + sizeof(type of pointer) * i

P+5 => 1000 + 4*5
1000 + 20
1020

*p + *q

a + b

```
int main()
{
    int a, b;
    printf("Enter two numbers");
    scanf("%d %d", &a, &b);
    // call a function swap
    printf("%d %d", a, b);
    return 0;
}
```



swap (&a, &b)

```
void swap (int *p, int *q)
{
  int t ;
   t = *p ;
  *p = *q ;
  *q = t ;

}
```

Call by Reference

```
void swap(int *, int*);
int main()
{
    int a,b;
    printf("Enter two numbers");
    scanf("%d %d",&a,&b);
    swap(&a,&b);
    printf("%d %d", a, b);
    return 0;
}
void swap(int *p, int *q)
{
    int t;
    t = *p;
   *p = *q;
   *q = t;
}
```

call by reference

Formal argument

1. ordinary variable
2. pointer variable

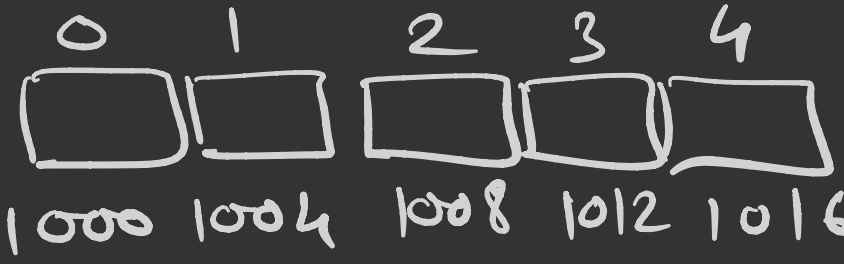
Why we use & in scanf() ?

```
scanf("%d", &x);
```

```
scanf( , *p)
```

Pointers and Arrays

```
int a[5];
```




```
int *p;
p = &a[0];
```

p+0	1000	&a[0]
p+1	1004	&a[1]
p+2	1008	&a[2]
p+i		&a[i]

```
fun(a);
```

*(p+0)	a[0]
*(p+1)	a[1]
*(p+i)	a[i]

```
fun(int *p)
```

a, p

a[i] ⟶ *(a+i) → *(i+a) → i[a]

a is a constant a= ✗

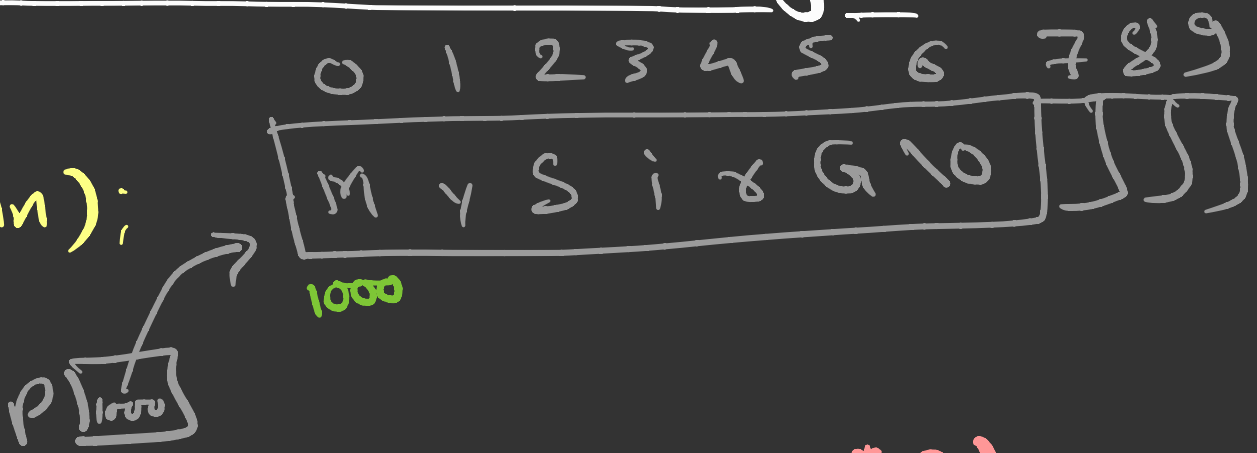
p is a variable p= ✓

a+i p+i ✓

p[i] → *(p+i) ⟶ *(i+p) → i[p]

Pointers and Strings

```
int l;
char str[10];
fgets(str, 10, stdin);
l = length(str);
```




```
int length(char *p)
{
    int i;
    for(i=0; *(p+i) ; i++);
    return i;
}
```

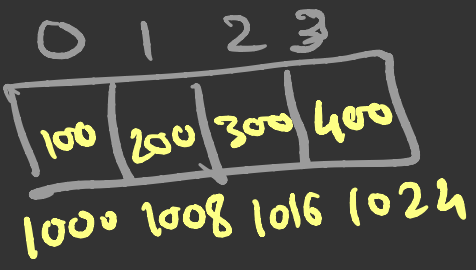
Array of Pointers

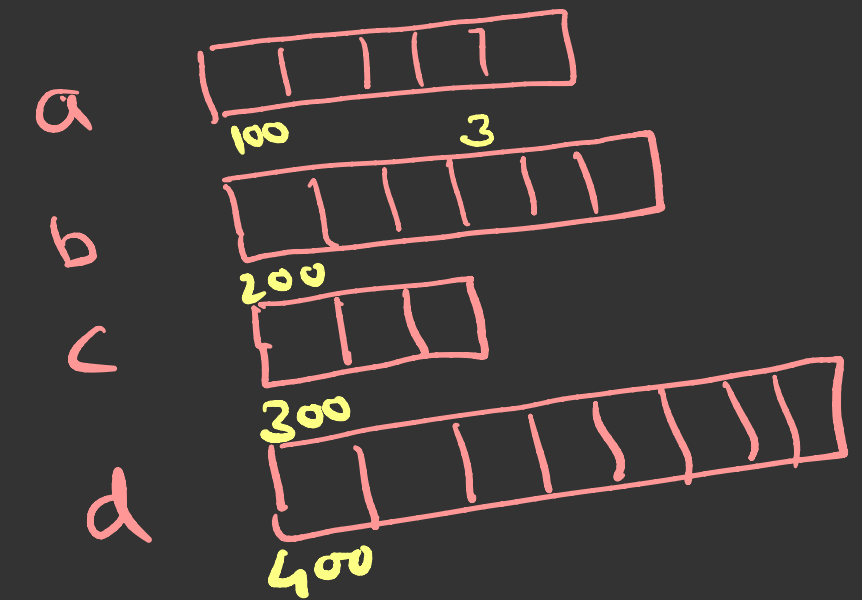
```
int *p[4];

int a[5], b[6], c[3], d[8];

p[0] = &a[0];
p[1] = &b[0];
p[2] = &c[0];
p[3] = &d[0]

input(p)
```





b[3]

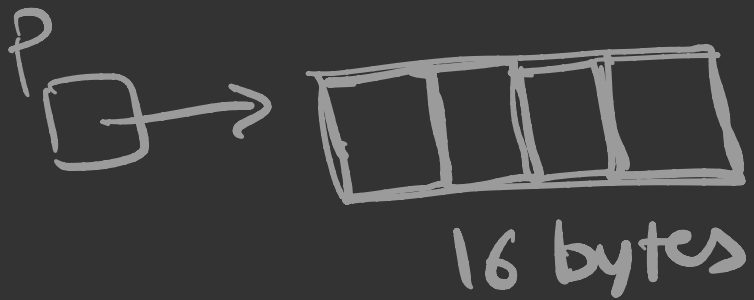
p[1][3]

p[i][j]

```
void input(int **ptr)
```

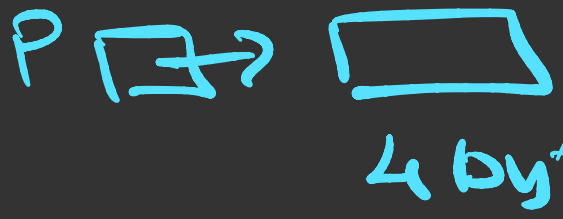
Pointer to Array

`int (*p)[4];` p is a pointer to an array of int type with 4 blocks.



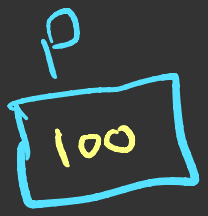
p+1 1016

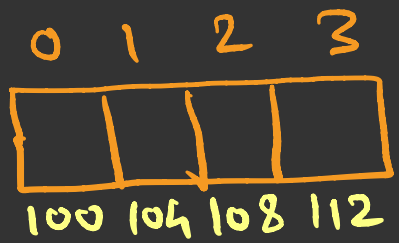
`int *p;`



p+1 1004

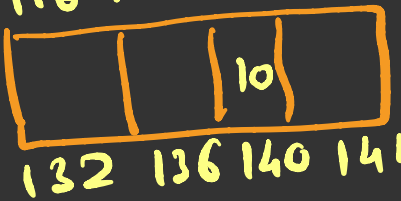
```
int (*p)[4];
int a[5][4];
p = a;
```



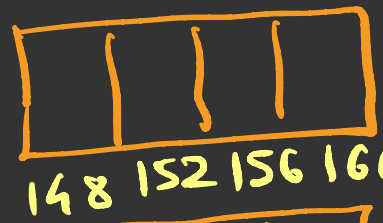


a[2][2]=10 1

*(P+2)[2]=10 2



((P+2)+2)=10 3



*(P[2]+2)=10 4

P[2][2]=10

Wild Pointer

An uninitialized pointer is a wild pointer.

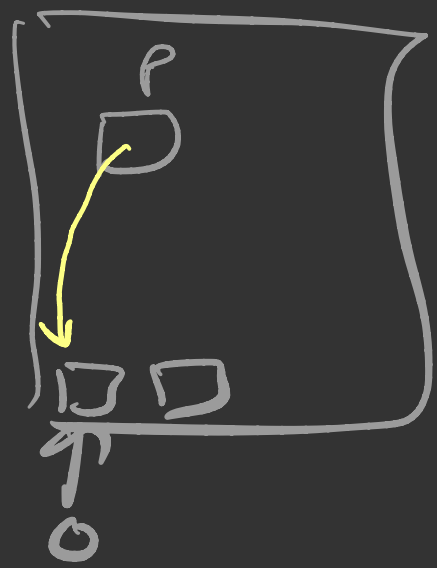
```
void f1()
{
        int *p;   <- wild pointer
        *p = 5;   <- illegal use of pointer
        ....

}
```

NULL Pointer

- A pointer containing NULL (special address) is known as NULL pointer.
- If a pointer containing NULL, we consider it as if it is not pointing to any location.
- As a safe guard to illegal use of pointers you can check for NULL before accessing pointer variable

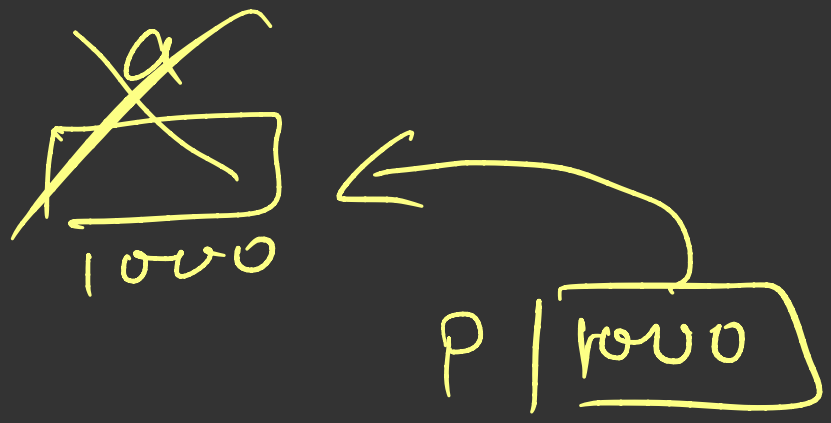
```
int  *p = NULL;
...
if (p != NULL)
{
     *p = 5;
}
```



Dangling Pointer

A pointer pointing to a memory location that has been deleted is called dangling pointer.

```
int * f1()
{
    int a;
    .....
    return &a;
}
void fun()
{
    int *p;
    p = f1();
    *p = 5;
}
```



p is dangling pointer

illegal use of pointer

```
void f1()
{
    int *p;
    {
        int x;
        p=&x;
        ....
    }
    *p=5;
}
```

Scope and life of x is limited to the block

After this line p becomes dangling pointer

illegal use of pointer

```
void f1()
{
    int *p;
    p = (int *) malloc (sizeof(int));
    ...
    ...
    free(p);
    *p = 5;
}
```

← p becomes a dangling pointer

← illegal use of pointer.

Void Pointer

- void pointer is a generic pointer that has no associated data type with it.
- void pointer can hold address of any type.

```
void *p;
int x;
p=&x;
```

```
void *p;
float y;
p=&y;
```

void pointers can not be dereferenced

```
*p=5;
```

```
*p=3.5f;
```

Error

- However void pointer can be dereferenced using typecasting.

```
*(int*)p = 5;
```

```
*(float*)p = 3.5f;
```

(int*)p

(Ram)Vash