Address of Operator (&) → used to print the memory location where the variable is present, represented by a hexadecimal number.

```
int x = 10;
cout << &x << endl;          // 0x7ffd50c0aaf0
                                  ↳ address of x

float y = 3.14;
cout << &y << endl;          // 0x7ffd50c0aaf4
                                  ↳ address of y
```

// It does not work for character variables
// as the << is overloaded such that when it sees a char*
it prints the actual char value/data.

```
char ch = 'A'
cout << &ch;          // A
```

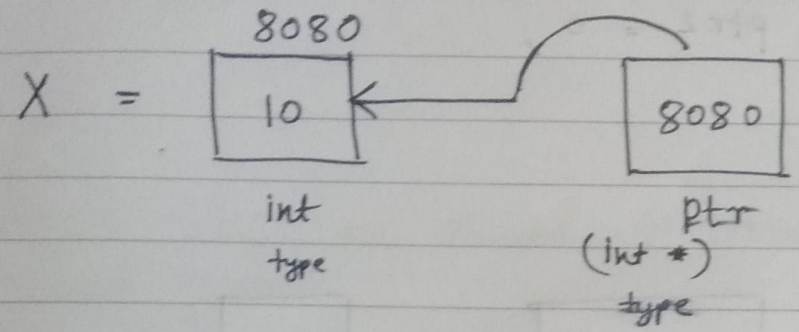
// However the << is only overloaded for char * so if we
// explicitly ~~char~~ type cast it will work.

// Explicit type casting from ~~char~~ char * to void *
// will work if we convert to int * as well.

```
cout << (void *) &ch;          // 0x7ffd50c0aaef
```

Pointers

→ variable that stores address of another variable.



cout << &x ; // Print

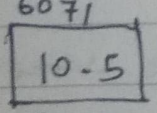
Datatype * variable_name // Store

Example -

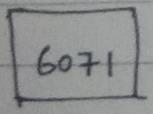
int *y = &x ; // Declare + initialization

int *y ; // Declare
y = &x ; // Assignment

Example -



float a = 10.5 ;
float * aptr = &a ;



double * ptr2; // No initialize

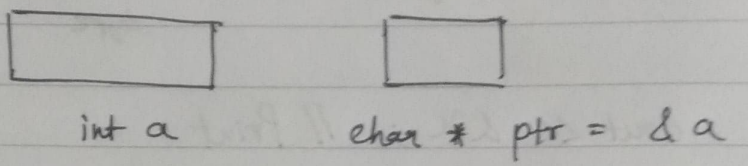
↓

has garbage value

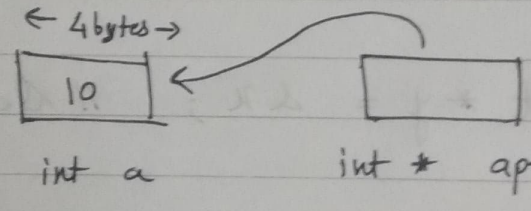
if we donot store anything we just assign 0

ptr2 = 0;

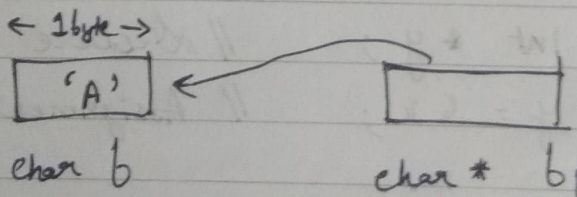
Avoid this



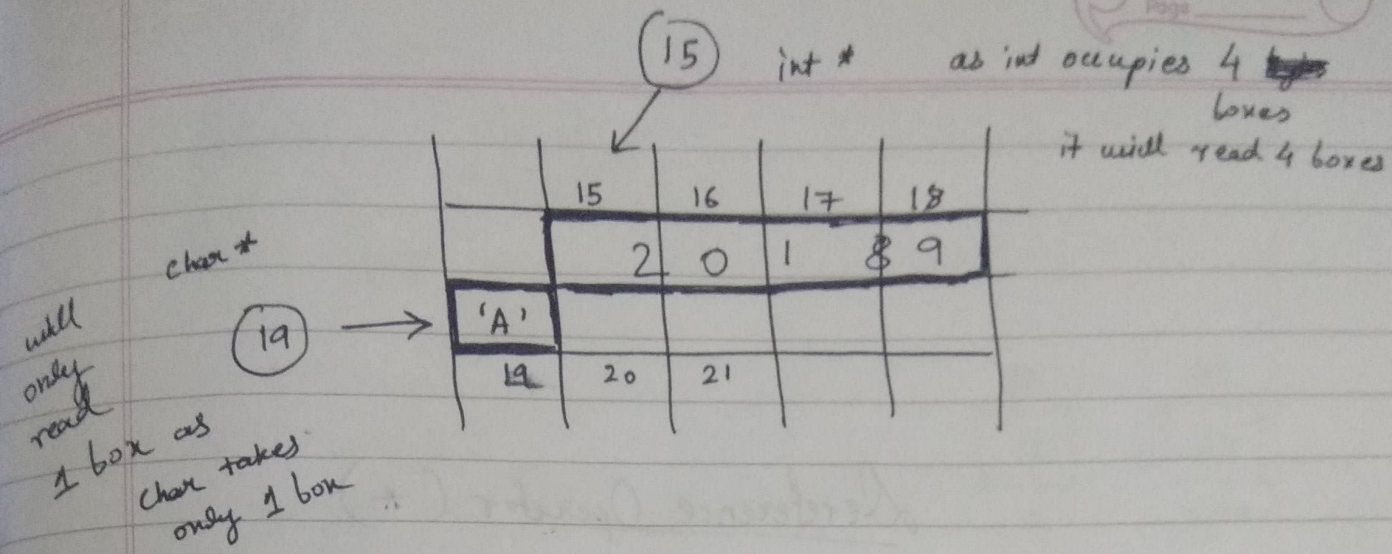
Size of a pointer variable



both have size 4 byte / 8 byte depending on system.



as size of address will always be same.



This is why we shouldn't store int in char* pointer.

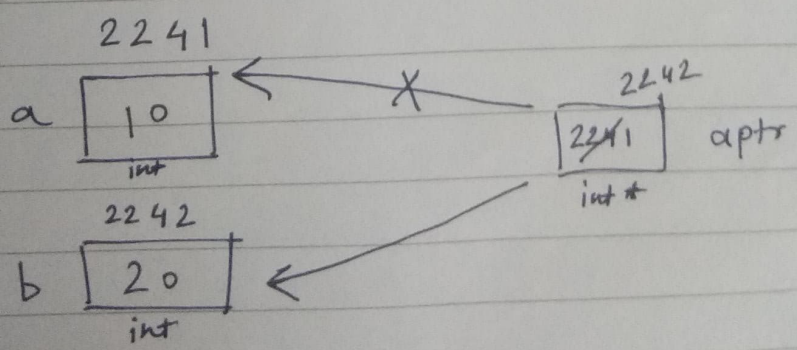
Re-assigning

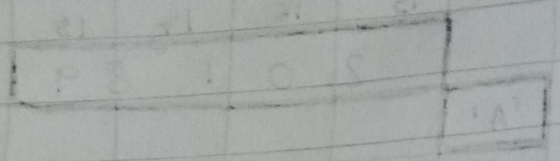
```
int a = 10, b = 20;

int * aptr = &a; // stores address of a
cout << &a << endl;
cout << aptr << end; // prints address of a

aptr= &b;          // re assigning address

cout << aptr << endl; // prints address of b
```





Dereference Operator (*)

& Bucket → Address

* Address → Bucket

Dereferencing any address gives the value of the bucket

```
int x = 10;
int * ptr = &x;

cout << *ptr;  // 10
```

```
Cout << *(ptr) + 1 ;  // 10 + 1 = 11

cout << *(ptr + 1) ;  maybe garbage of next address what ever value it gets.
```

Question

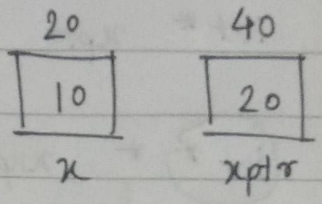
Predict Output :-

```
int x = 10;
int* xptr;
xptr = &x;

cout << &x << endl;
cout << xptr << endl;

cout << *(&x) << endl;
cout << *(xptr) << endl;

cout << *(&xptr) << endl;
cout << &(*xptr) << end;
```



Output

20
20
10
10
20
20

Pointer of pointer

```
int x = 10;

int * xptr = &x;

int ** xxptr = &xptr;
```

int * → data type
* → syntax

Functions - Pass by value

```
void increment (int a) {

    a = a + 1;
    Cout << "Inside Function" << a << endl;
}

int main()
{
    int a = 10;
    increment (a);
    cout << "Inside Main: " << a;
    return 0;
}
```

Output

11
10

As value of a was passed to func increment and that a inside increment was increased after the func terminated that a also got cleared.

Functions - Pass by reference using pointer

```
void increment (int* aptr) {

    *aptr = *aptr + 1;
    cout << "Inside Function" << *aptr << endl;
}

int main() {

    int a = 10;
    increment (&a);
    cout << "Inside Main: " << a;
    return 0;

}
```

Output

```
11
11
```