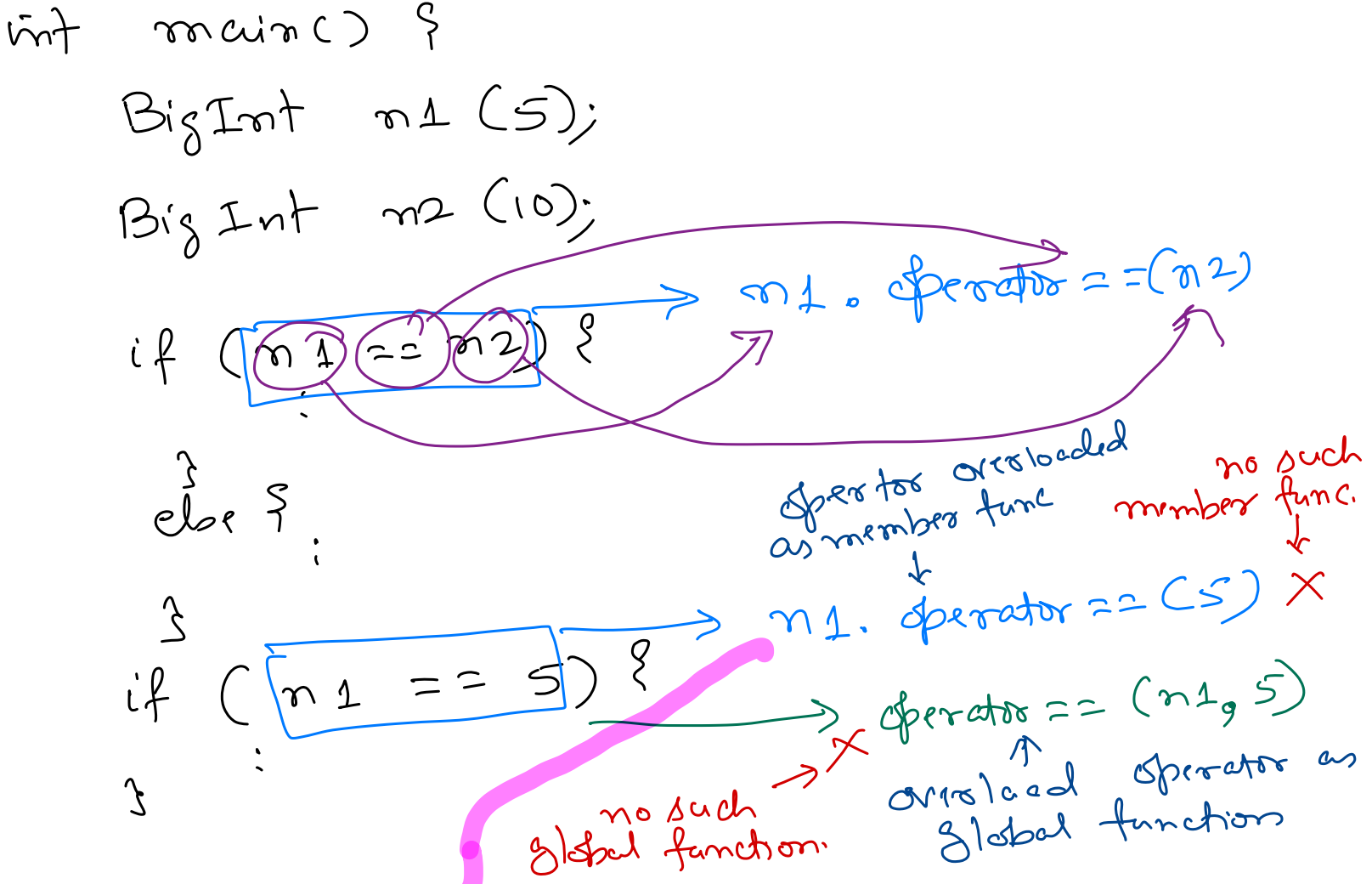

int main() {
BigInt n1(5);
BigInt n2(10);
if (n1 == n2) {
n1.operator==(n2)
} else {
}
if (n1 == 5) {
n1.operator==(5)
operator overloaded as member func
no such member func.
X
operator==(n1, 5)
overload operator as global function
no such global function.
X
}

```
class BigInt {
    :
public:
    :
    bool operator == (int no) {
        if (num == no)
            return true;
        return false;
    }
    :
};
```

```
BigInt n1(10);
int i = 5;
if (i == n1) {
    :
}
```

operator == (i, n1)

Global function.

```
bool operator==(int no, BigInt obj) {
    if (no == obj.num)
        return true;
    return false;
}
```

accessing private member in non-member function.

Sol 1 for error

```
bool operator == (int no, BigInt obj) {
    return (obj == no);
}
```

⇓

obj.operator == (no)

Sol 2

Make global function a friend of BigInt class.

```
class BigInt {
    :
    friend bool operator == (int, BigInt);
};
```

↑ is a friend of BigInt

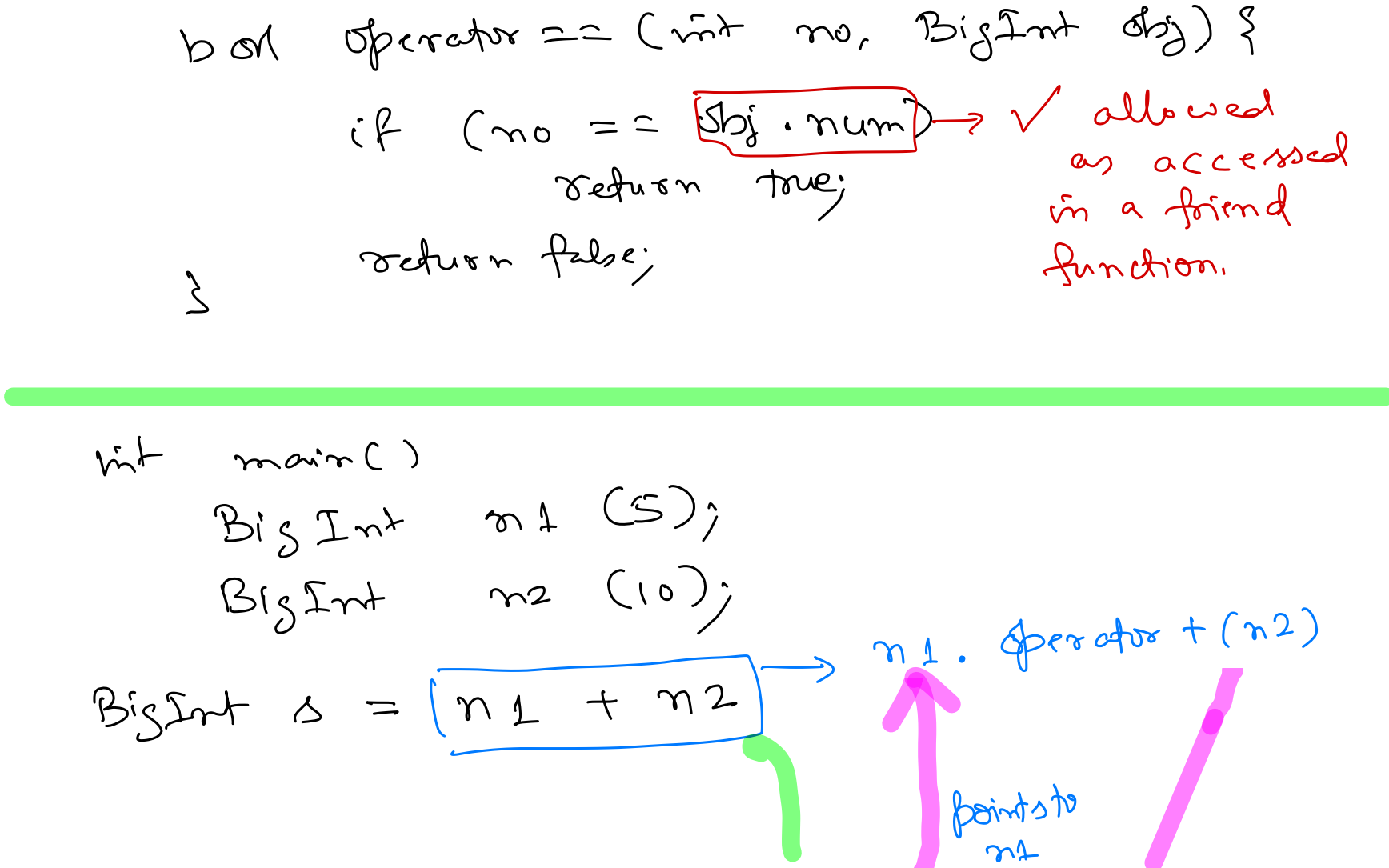
```
bool operator == (int no, BigInt obj) {
    if (no == obj.num)
        return true;
    return false;
}
```

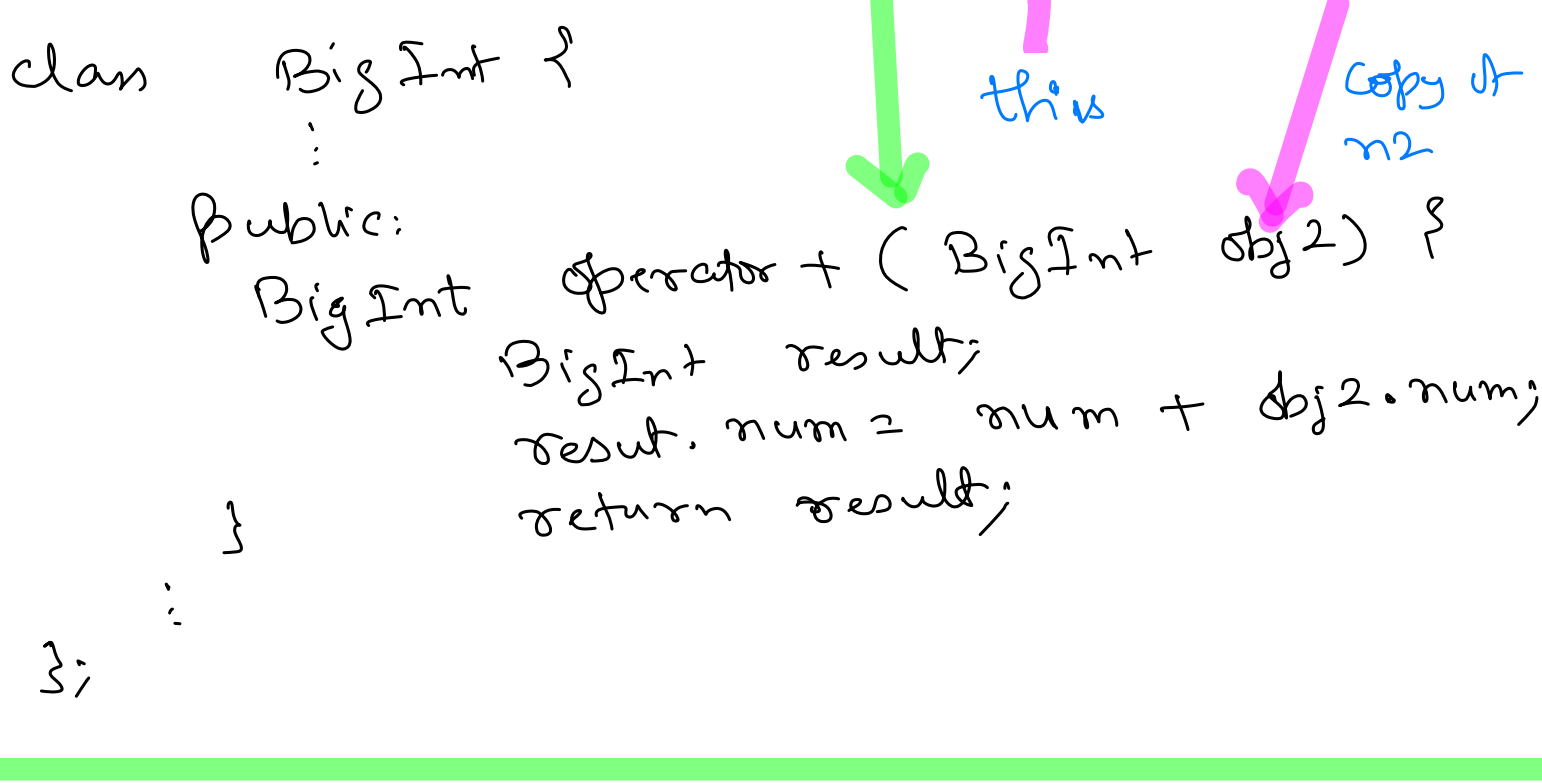
obj.num → ✓ allowed as accessed in a friend function.

```
int main()
{
    BigInt n1(5);
    BigInt n2(10);
    BigInt s = n1 + n2;
}
```

n1 + n2 → n1.operator+(n2)

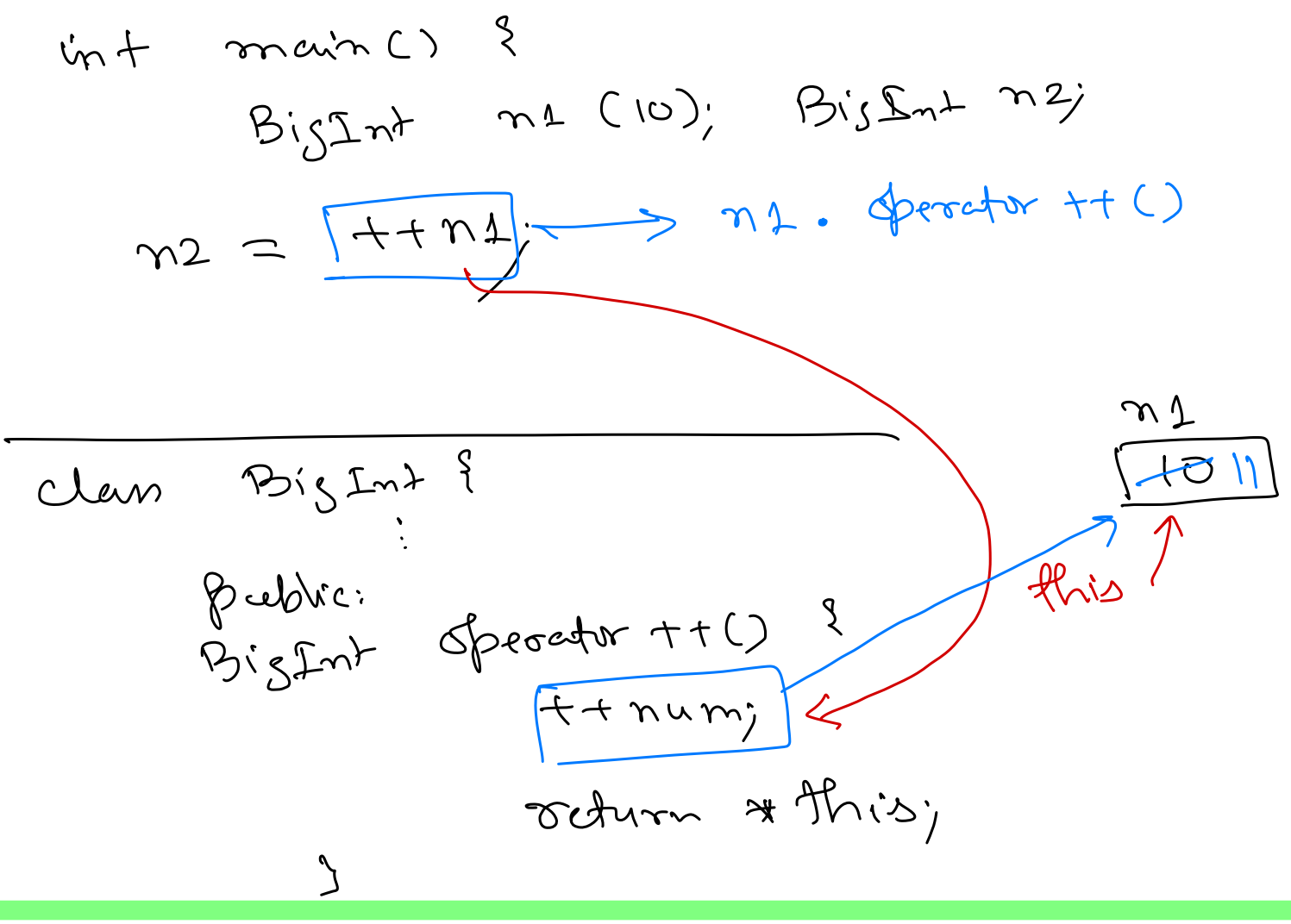
n1 points to





Unary operator overloading.

++ -- ! ~ + -


int main() {
BigInt n1(10); BigInt n2;
n2 = ++n1;
n1.operator++()
n1
10 11
this
class BigInt {
public:
BigInt operator++() {
++num;
return *this;
}

```
int main() {
    BigInt n1(10);
```

++n1; → n1.operator++()

n1++; → n1.operator++(0)

Pre increment

Post increment

some placeholder integer

```
class BigInt {
:
public:
    BigInt operator++(int) {
        BigInt result(*this);
        ++num;
```

↑ function argument name is optional in function definition

n1 → 10 ← this

```
        return result;
    }
};
```

result

10

```
BigInt result(*this);
```

define an object with name result

initialize object with the value

object pointed by this

object type = ?

BigInt

```
int i(10);
int j(i);
int *p = &i;
int R(*p);
```

```
i = 5;
j = i++;
```

```
BigInt n1(10);
BigInt n2(n1);
```

Call parametrised constructor taking one int as argument

Copy Constructor

```
class Array {
    int * data;
    const int size;
public:
    Array(int n) : size(n) {
        // size = n;
        data = (int *) malloc (sizeof(int) * n));
    }
    int& operator[](int i) {
        static int temp;
        if ((i < 0) || (i >= size)) {
            return temp;
```

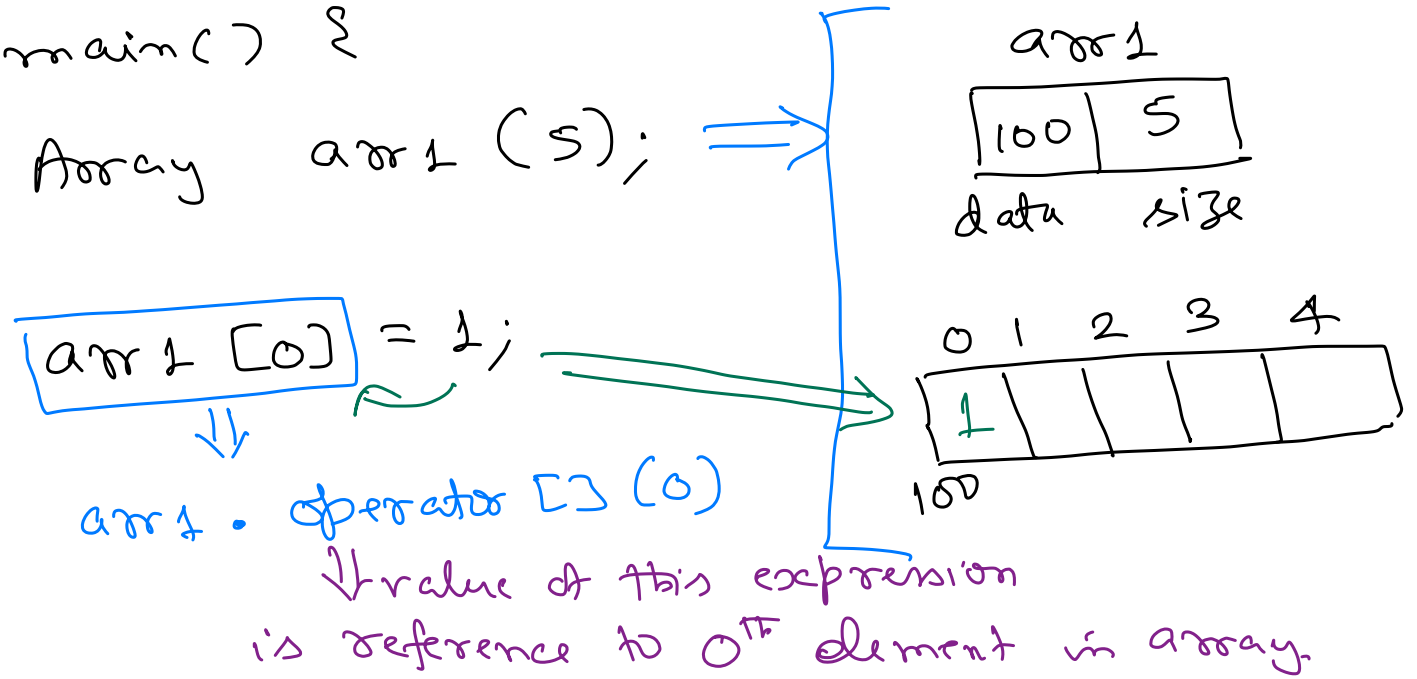
we don't want to change array size.

→ as size is const

to validate index

```
    }
}       return data[i];
~Array() { free(data); }
};
```

```
int main() {
    Array arr1(5);
    arr1[0] = 1;
}
```



expression	r-value	l-value ⇒ address
5	5	not defined 5 = 6; ✗
i	value stored in i	address of i i = 6; ✓
i + j	value of the expression	not defined i + j = 6; ✗
* p	contents of address stored in p	address stored in p * p = 6; ✓

```
int * f1( ) {
static int x;
    return &x;
}
```

⇒ `*f1() = 5;`

```
int f2() {
  static int y;
  return y;
}
```

$\Rightarrow$ `f2() = 6;`

Exercise

```
void f1 ( Array obj ) {
    obj [0] = 1;
    ++ obj [1];
}
```

```
int main() {
    Array arr1(5);
    f1(arr1);
    Array arr2(arr1);
    arr1[0] = 5;
    std::cout << arr1[0];
    std::cout << arr2[0];
}
```

new delete Vs malloc free

→ operators

→ new allocate memory for a type

→ functions

→ malloc allocates block of byte

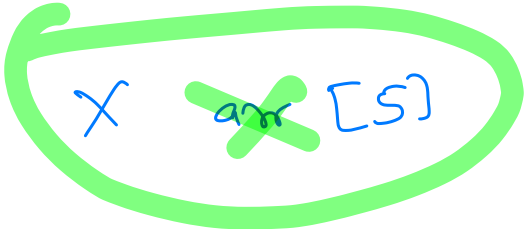
```
class X {
    :
};

X *p = (X*) malloc (sizeof(X));
    : allocate memory
free(p); ⇒ free the memory
X *q = new X();
```

⇒ allocate memory
→ call constructor

delete pj ⇒ → call destructor
- free the memory

X * obj_arr = new X [5];

→ allocate memory for array of 5 objects
→ call default constructor for each object.



not correct way to free memory for array of objects → delete obj_arr;

→ call destructor for first object

→ free memory for all objects

How to free memory for array of objects → delete [] obj-arr;

↓

→ call destructor for all objects

→ free memory for all objects