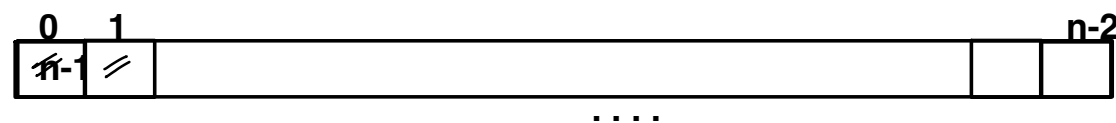


$$1 \text{ KB} = 1024 \text{ B}$$

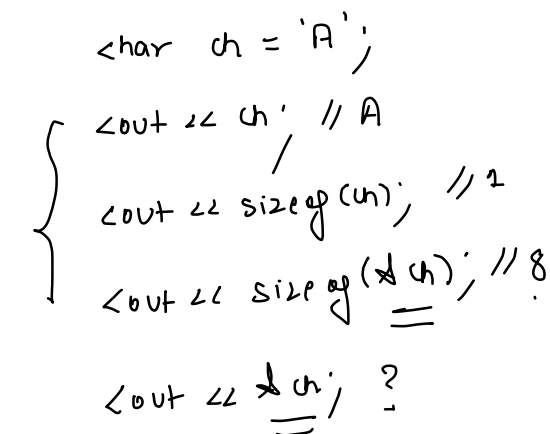
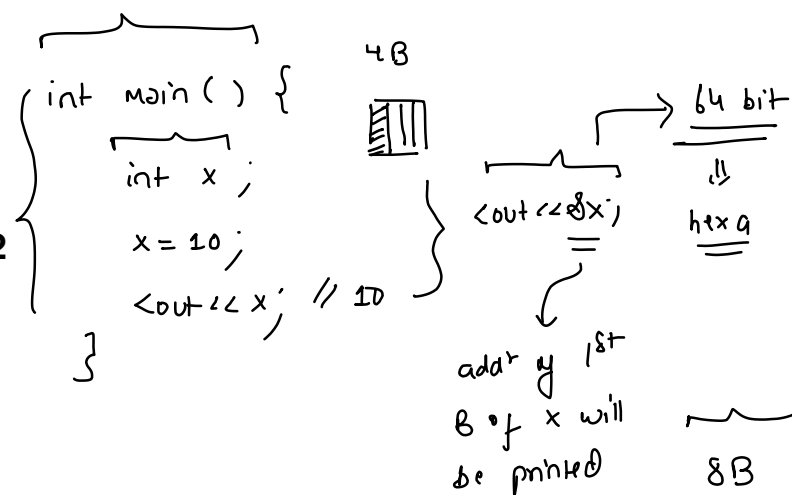
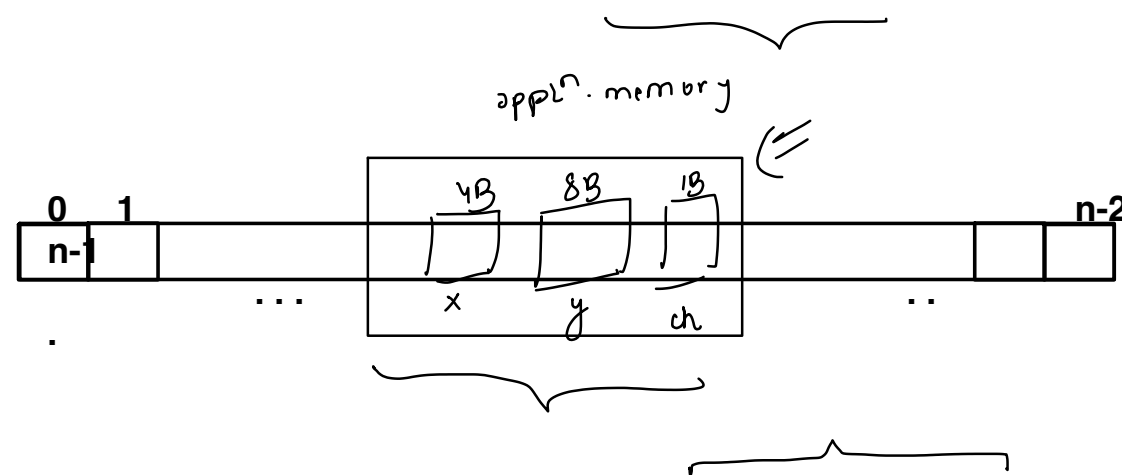
Introduction to Pointers in C++

- ⇒ We can visualize the system memory (**RAM**) as a sequence of memory cells such that each memory cell is of **1B (8 bits)** and is addressable.



- These addresses depending on the system (and the compiler) are either of **32-bits** i.e. **4B** or **64-bits** i.e. **8B**. Moreover, computer use the hexa-decimal i.e. the base-16 number system to represent addresses.

- When we run a C++ program, a portion of system memory is allocated for program execution which is known as **application memory**.



- To know the address of a byte in C++, we use the **address of (&)** operator.

double y = 3.14;
cout << &y;

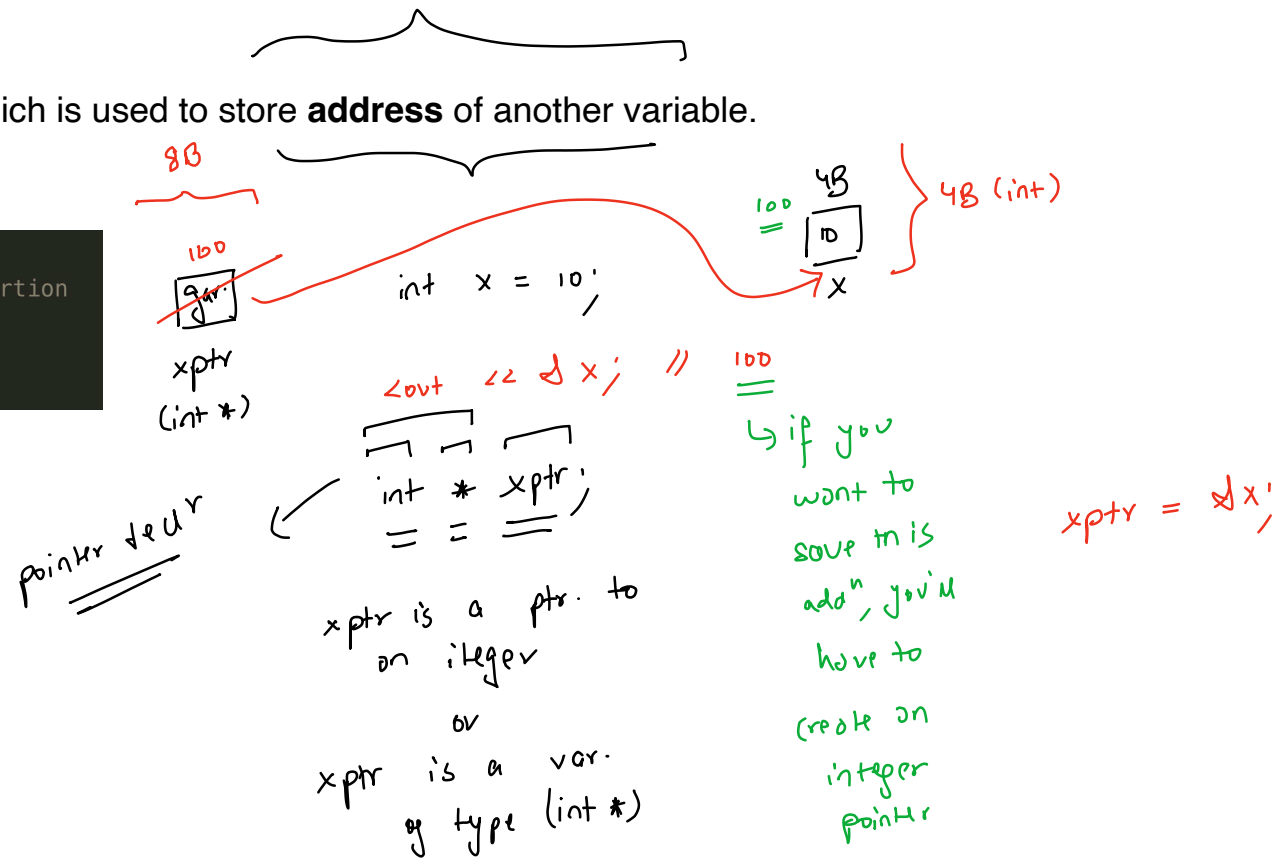
- When we print **address** of a **character** using the **cout <<**, C++ prints the contents of the memory stored at that address until it encounters the **null character ('\0')**.

- When we print **address** of a **character** using the **cout <<**, C++ prints the contents of the memory stored at that address until it encounters the **null character** (**'\0'**).

Pointers in C++

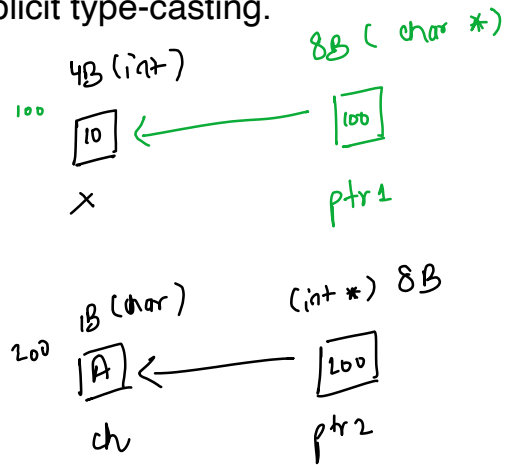
A **pointer** is a variable which is used to store **address** of another variable.

```
// syntax for pointer declartion  
type * name;
```



Since a pointer stores an address which are of **8B**, in C++, we can use a pointer of one type to store the address of a variable of another type via explicit type-casting.

```
int main() {  
    int x = 10;  
    char* ptr1 = (char*)&x;  
    char ch = 'A';  
    int* ptr2 = (int*)&ch;  
    return 0;  
}
```



Dereferencing a Pointer in C++

To dereference a pointer, i.e. to access the value of the variable whose address is stored by the pointer variable, we use the **dereference** operator (`*`)

```
int x = 10;
```

Diagram: A box labeled 'x (int)' at address 100 contains the value 10. A box labeled 'xptr (int*)' at address 8B contains the address 100. An arrow points from xptr to x.

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

```
cout << *xptr; // 10
```

Diagram: A box labeled 'xptr' at address 8B contains the address 100. An arrow points from xptr to the box labeled 'x' at address 100, which contains the value 10.

```
cout << x; // 10
```

```
int main() {
    int x = 516;
    char* ptr = (char*)&x;
    cout << (int)*ptr << endl;
    return 0;
}
```

Diagram: A box labeled 'x' at address 100 contains the value 516. A box labeled 'ptr (char*)' at address 8B contains the address 100. An arrow points from ptr to x.

516 = 512 + 4

only the 1st B is read and is interpreted as an integer

Diagram: A 32-bit memory layout for variable x. The first 4 bits (bits 31-28) are 0. Bits 27-24 are 1, 0, 0, 0. Bits 23-20 are 0, 0, 0, 0. Bits 19-16 are 0, 0, 1, 0. Bits 15-12 are 0, 0, 0, 0. Bits 11-8 are 0, 0, 0, 0. Bits 7-4 are 0, 0, 0, 0. Bits 3-0 are 1, 0, 0, 0. The first byte (bits 31-24) is 0. The second byte (bits 23-16) is 1000. The third byte (bits 15-8) is 0001. The fourth byte (bits 7-0) is 0000. The first byte is labeled 'char*' and the second byte is labeled 'ptr'.

```
cout << (int)*ptr;
```

```
int x = 10;
```

```
int y = 20;
```

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

```
int * yptr = &y;
```

```
cout << x + y; // 30
```

```
cout << (*xptr) + (*yptr); // 30
```

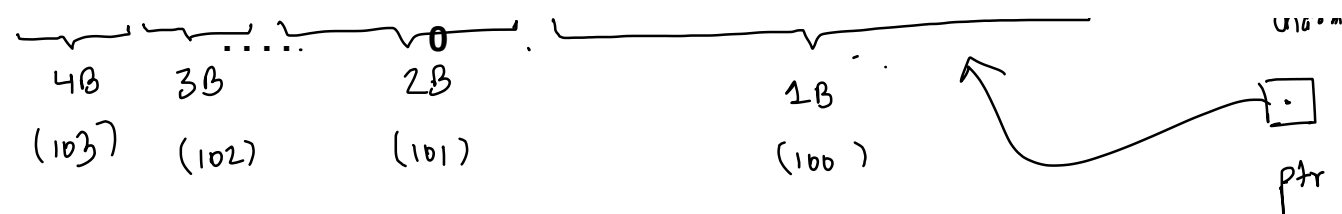
Diagram: A box labeled 'x' at address 100 contains the value 10. A box labeled 'y' at address 20 contains the value 20. A box labeled 'xptr' at address 10 contains the address 100. A box labeled 'yptr' at address 20 contains the address 20. Arrows point from xptr to x and from yptr to y.

255 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1

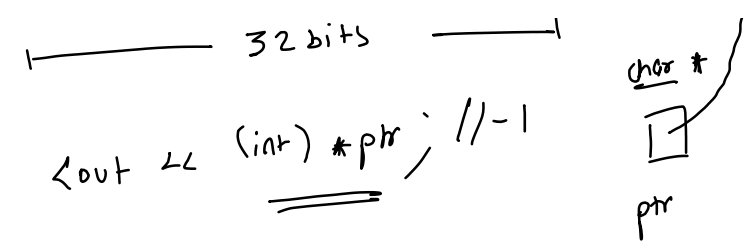
Diagram: A 32-bit memory layout for variable ptr. The first 31 bits are 1, and the last bit is 0. The first 31 bits are labeled 'sign bit' and the last bit is labeled '0'. The first 31 bits are labeled '32 bits'.

```
cout << (int)*ptr; // -1
```

Diagram: A box labeled 'ptr' at address 8B contains the address 100. An arrow points from ptr to the box labeled 'x' at address 100, which contains the value -1.

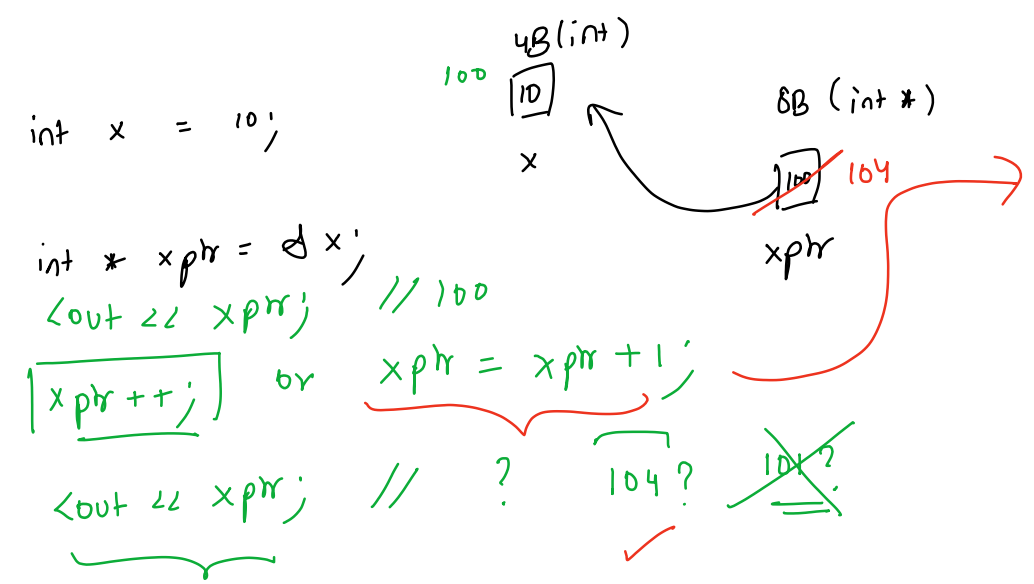


<out << (int) * ptr;

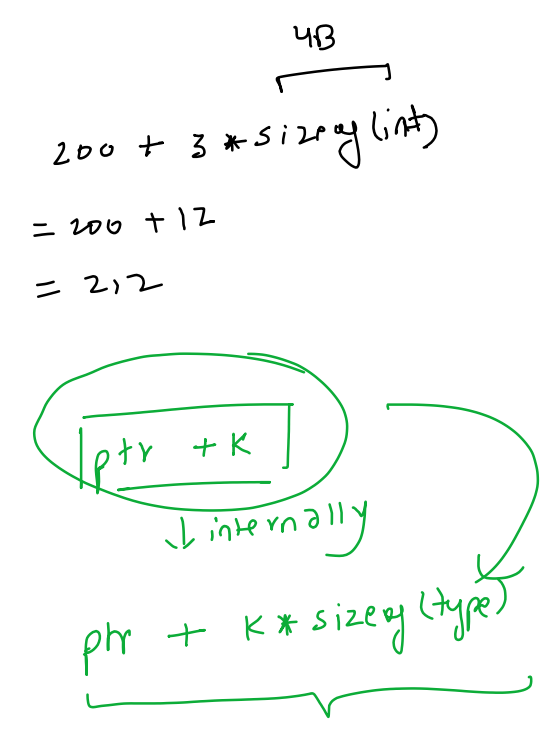
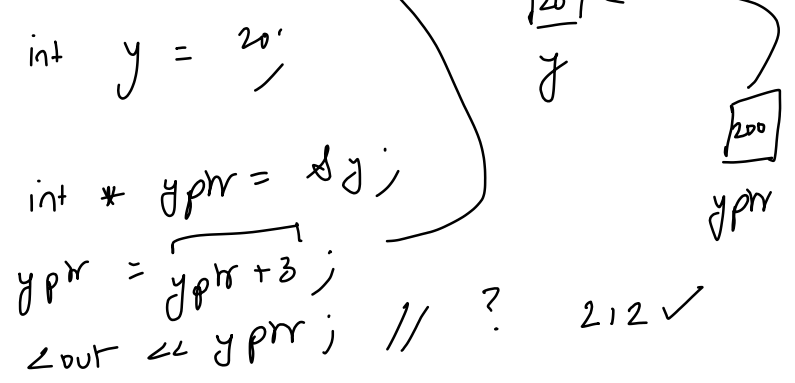


⇒ Pointer Arithmetic

A pointer variable only supports arithmetic **addition** and **subtraction** operations. Moreover, in C++, we can also subtract two pointer variables.



$$\begin{aligned}
 &= x_{ptr} + 1 * \text{sizeof}(int) \\
 &= 100 + 1 * 4 \\
 &= 104
 \end{aligned}$$



$$\begin{aligned}
 &200 + 3 * \text{sizeof}(int) \\
 &= 200 + 12 \\
 &= 212
 \end{aligned}$$

$y_{ptr} = y_{ptr} + 3;$
 $\text{cout} << y_{ptr};$ // ? 212 ✓

$ptr + k * \text{sizeof}(type)$

Pointers and Functions

```

void increment(int a) {
    a++;
}

int main() {
    int a = 0;
    increment(a);
    cout << a << endl; // 0
  }

```

Diagram: A box labeled 'a' containing '0' is shown above the function call. An arrow points from the parameter 'a' in the function signature to the variable 'a' in the function body.

```

void increment(int& a) {
    a++;
}

int main() {
    int a = 0;
    increment(a);
    cout << a << endl; // 1
  }

```

Diagram: A box labeled 'a' containing '0' is shown above the function call. An arrow points from the parameter 'a' in the function signature to the variable 'a' in the function body. Another arrow points from the variable 'a' in the main function to the parameter 'a' in the function body.

```

void increment(int* ptr) {
    (*ptr)++;
}

int main() {
    int a = 0;
    increment(&a);
    cout << a << endl;
  }

```

Diagram: A box labeled 'ptr' containing '100' is shown above the function call. A box labeled 'a' containing '0' is shown below the function call. An arrow points from the parameter 'ptr' in the function signature to the variable 'ptr' in the function body. Another arrow points from the variable 'ptr' in the main function to the parameter 'ptr' in the function body. A third arrow points from the variable 'a' in the main function to the parameter 'ptr' in the function body.

Pass by addr
 or
 passing addr by value

⇒ Void or Generic Pointers

A generic or void pointer (void*) can be used to address of a variable of any type.

```
int main() {
    char ch = 'a';
    void* ptr = &ch;
    cout << ptr << endl;
}
```

1B (char)

100

ch

{void*} 8B

ptr

100

garbage

100

ptr

In C++, void pointers cannot be **dereferenced**.

A use case of a **void*** is that it can be used to pass generic arguments to a function.

```
void increment(void* pvoid, int size) {
    if(size == sizeof(int)) {
        int* pint = (int*)pvoid; (*pint)++;
    } else if(size == sizeof(char)) {
        char* pchar=(char*)pvoid; (*pchar)++;
    }
}

int main () {
    int x = 0;
    char y = 'a';
    increment(&x, sizeof(x));
    increment(&y, sizeof(y));
    cout << x << ", " << y << endl;
    return 0;
}
```

double z=3.14;

incr(&z, sizeof(z))

⇒ Null Pointers

A **null pointer** is a pointer variable that contains the *null pointer value* and therefore it points to nowhere.

```
int main() {
    int* p = nullptr;
    int* q = 0;
    int* r = NULL;
}
```

#define NULL 0
↓
false

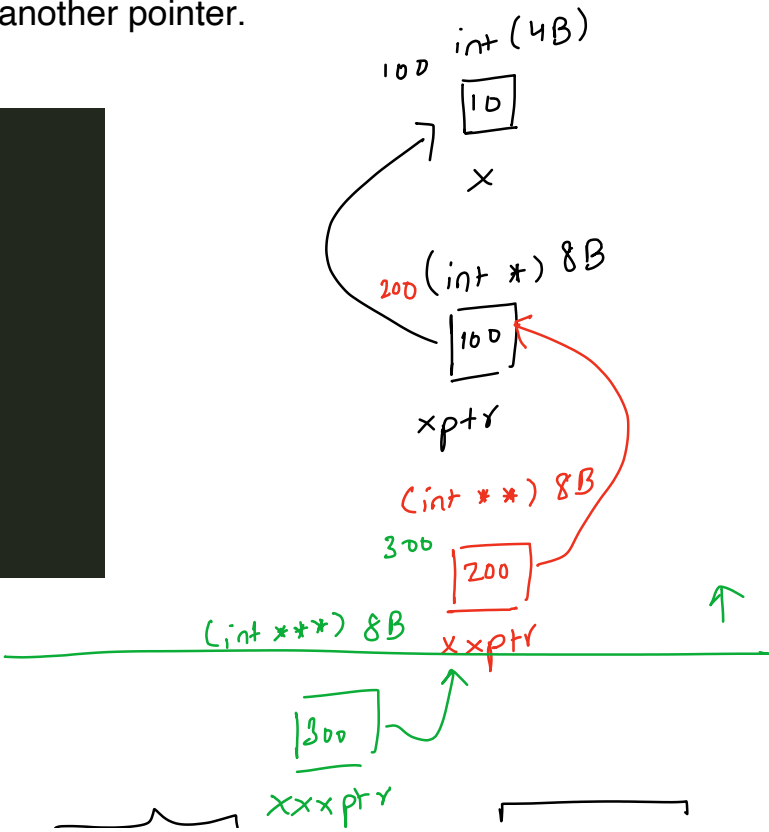
DSA
false

In C++, null pointers cannot be dereferenced.

⇒ Pointer to Pointer

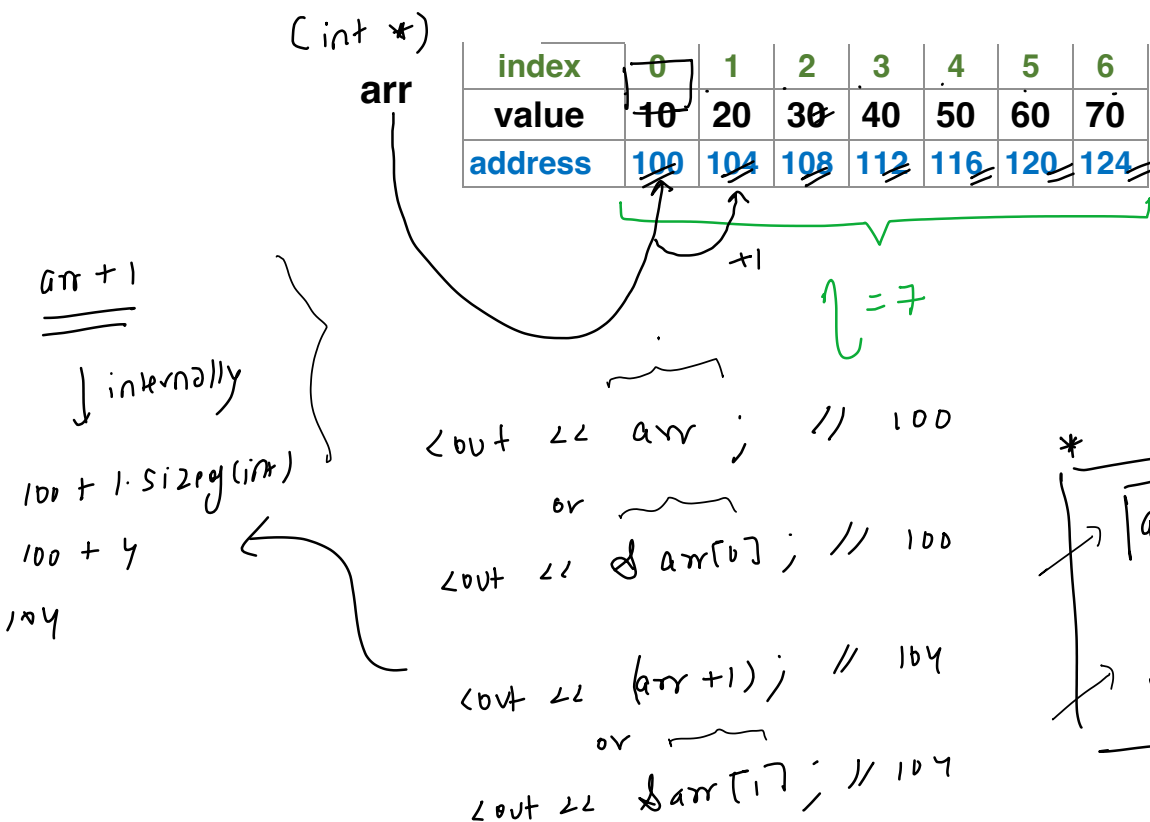
In C++, we can create a pointer to point to another pointer.

```
int main() {
    int x = 10;
    int* xptr = &x;
    int** xxptr = &xptr;
    int*** xxxptr = &xxptr;
    return 0;
}
```



Pointers and Arrays

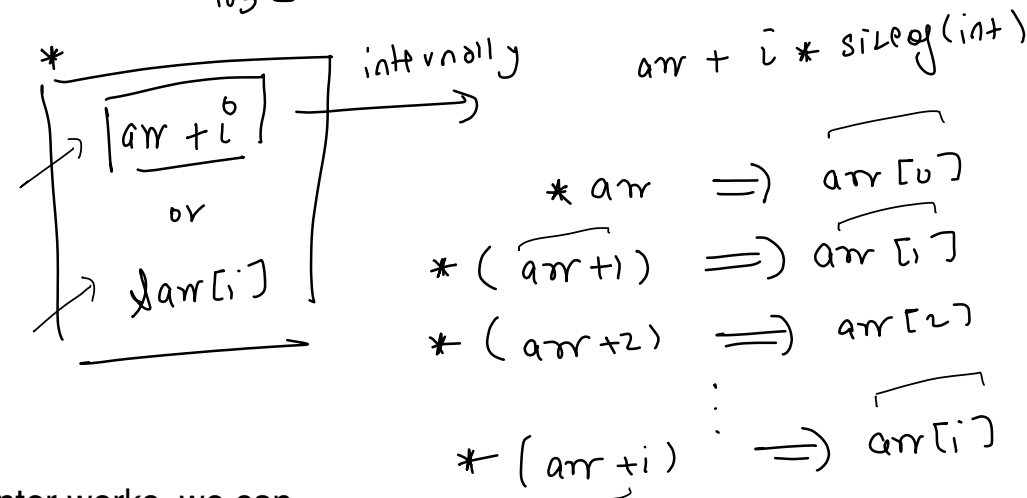
In C++, we can think of **name** of an array as a **pointer** to the element at the **0th** index.



int arr[] = {10, 20, 30, 40, 50, 60, 70};

28B

100
101
102
103 } 4B => 0th idx



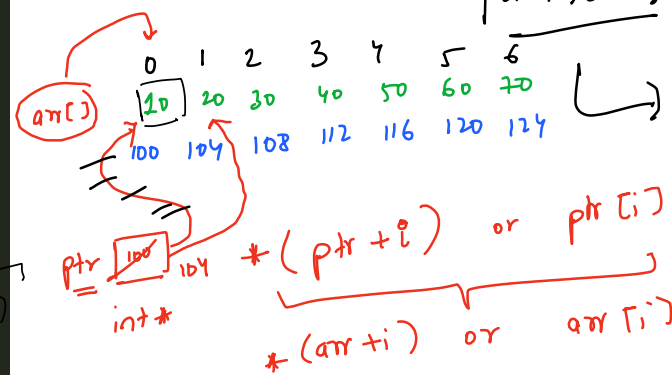
$$arr + i * \text{sizeof(int)}$$

Since the name of an array in C++ works exactly how a pointer works, we can assign an array name to a pointer.

arr = arr + i // error
arr++; // error
ptr++;

(int *)

```
int main() {
    int arr[] = {10, 20, 30, 40, 50, 60, 70};
    int n = sizeof(arr) / sizeof(int);
    int* ptr = arr; or arr[0]
    for(int i=0; i<n; i++) {
        cout << ptr[i] << " "; // or *(ptr+i)
    }
    return 0;
}
```



arr[i] equi to *(arr + i)

it is a const time op

time spent
here is
independent
of arr size

The **difference** between a pointer and array name is that while a pointer can be assigned a different address, the array name cannot be assigned anything.