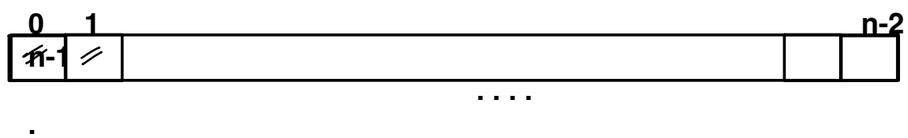
1 KB = 1024 B

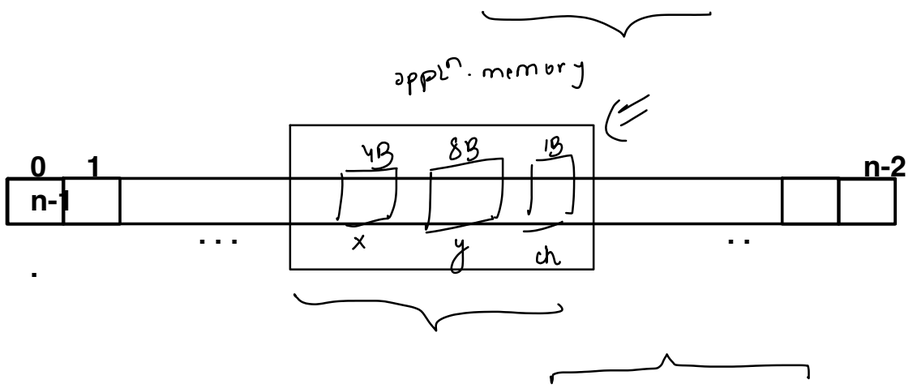
Introduction to Pointers in C++

- 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.**




```
int main() {
    int x;
    x = 10;
    cout << x; // 10
}
```

4B

```
cout << &x;
```

64 bit

hex a

addr of 1st B of x will be printed

```
char ch = 'A';
cout << ch; // A
cout << sizeof(ch); // 1
cout << sizeof(&ch); // 8
cout << &ch; ?
```

8B

3.14

y

```
double y = 3.14;
cout << &y;
```

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

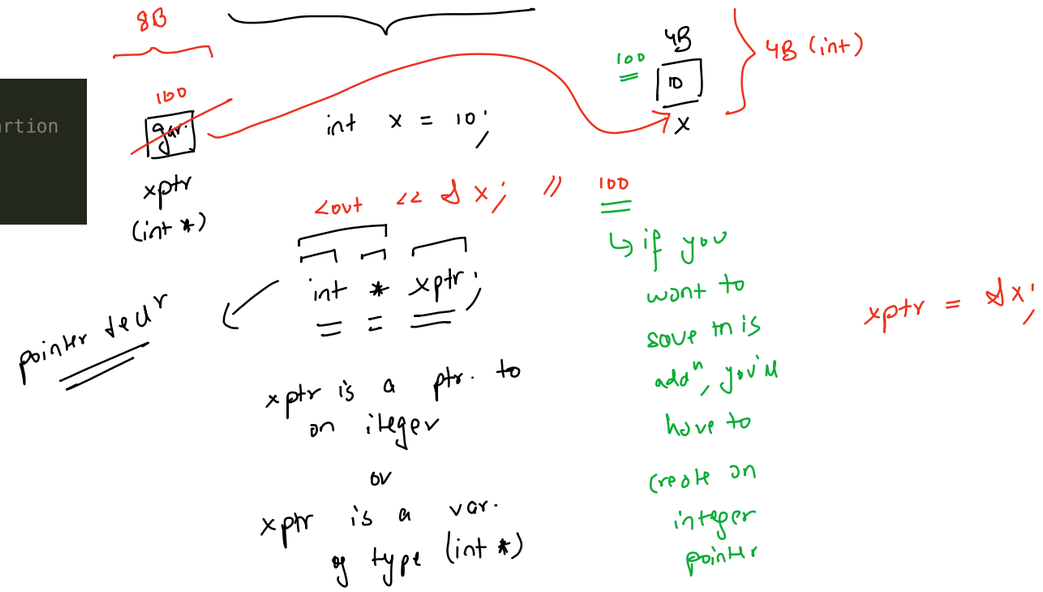
- When we print **address** of a **character** using the **cout <<**, C++ prints the contents of the memory stored at that address until it encounters the **null character** (**'\0'**).

- 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++

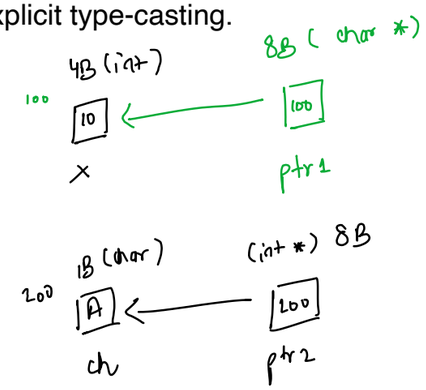
A **pointer** is a variable which is used to store **address** of another variable.

```
// syntax for pointer declartion

type * name;
```



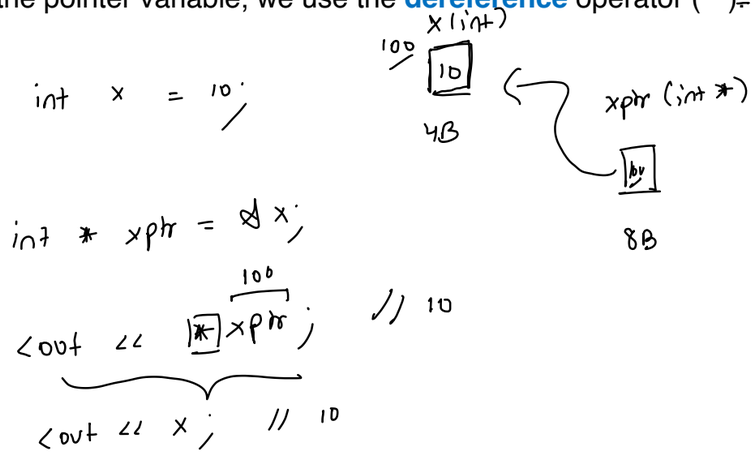
Since a pointer stores an address which are of **8B**, in C++, we can use a pointer of one type to store the address of a variable of another type via explicit type-casting.

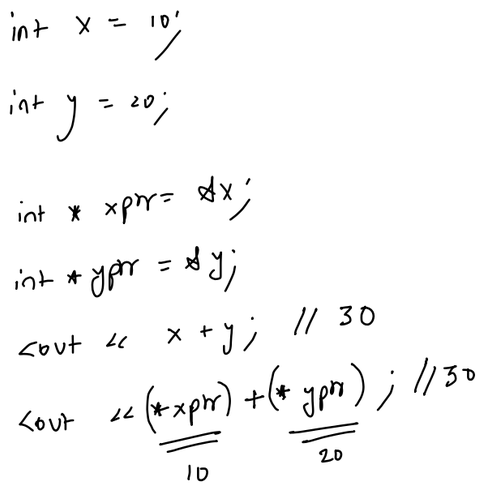
```
int main() {

    int x = 10;
    char* ptr1 = (char*)&x;

    char ch = 'A';
    int* ptr2  = (int*)&ch;

    return 0;
}
```



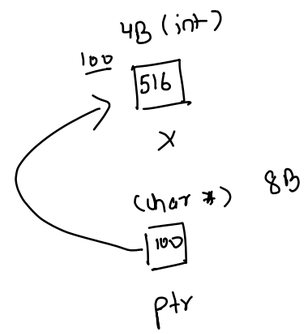
Dereferencing a Pointer in C++

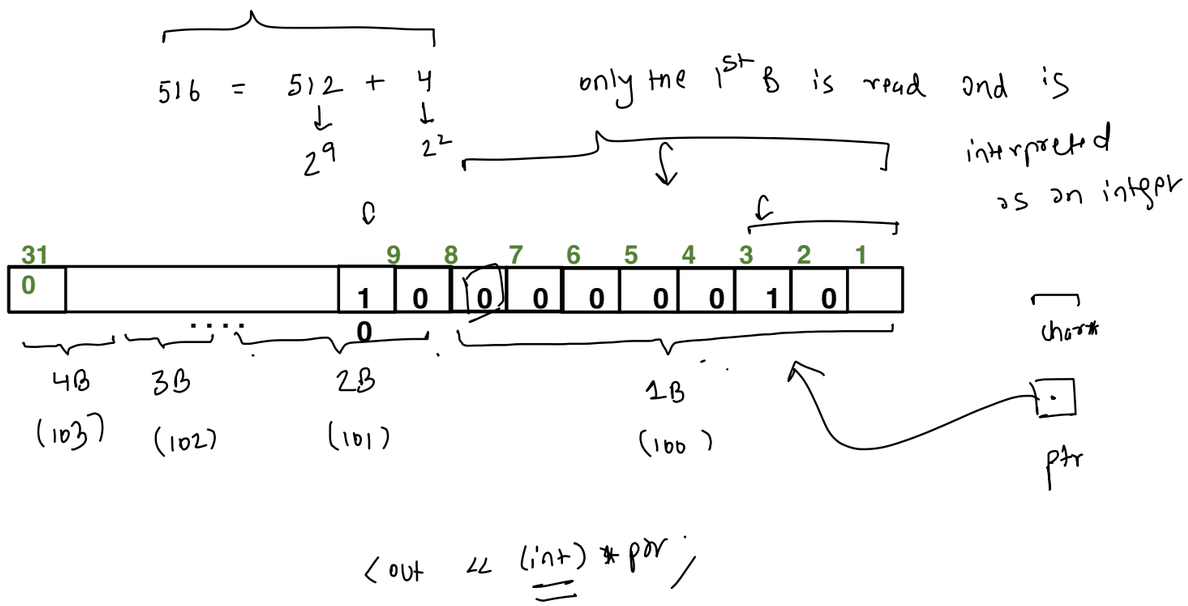
To dereference a pointer, i.e. to access the value of the variable whose address is stored by the pointer variable, we use the **dereference** operator (*):

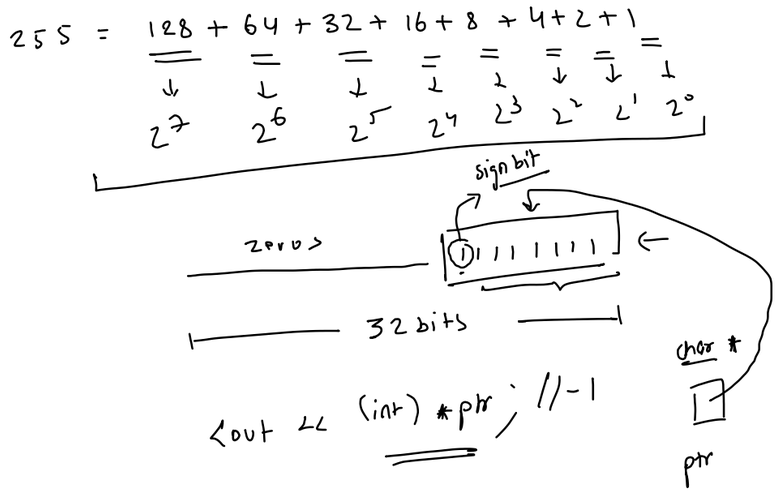


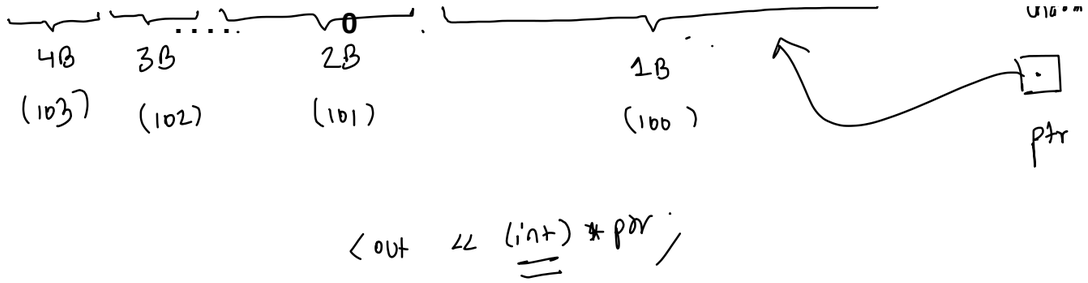


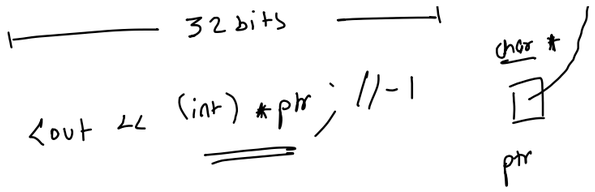

```
int main() {

    int x = 516;
    char* ptr = (char*)&x;
    cout << (int)*ptr << endl;

    return 0;
}
```





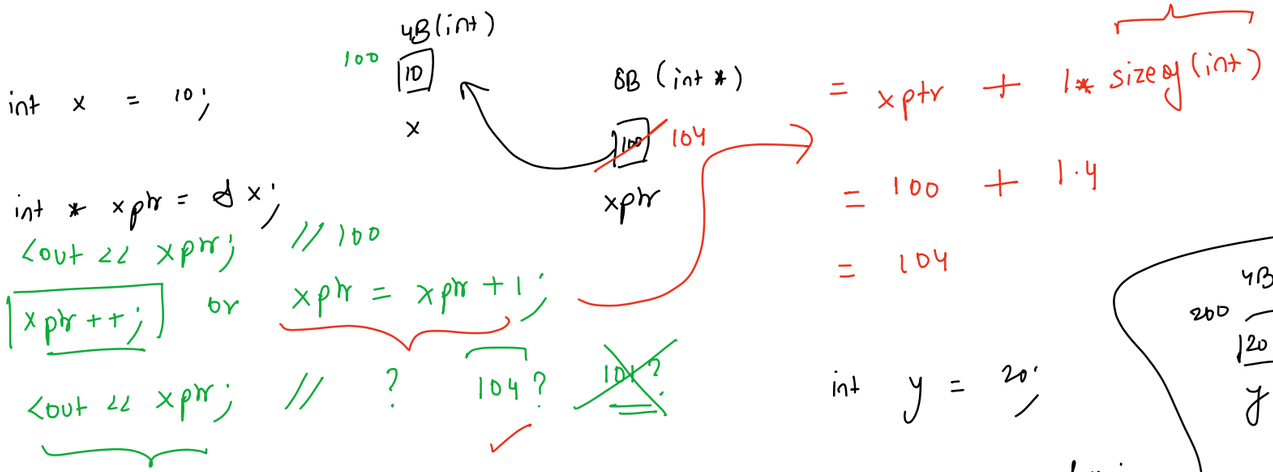


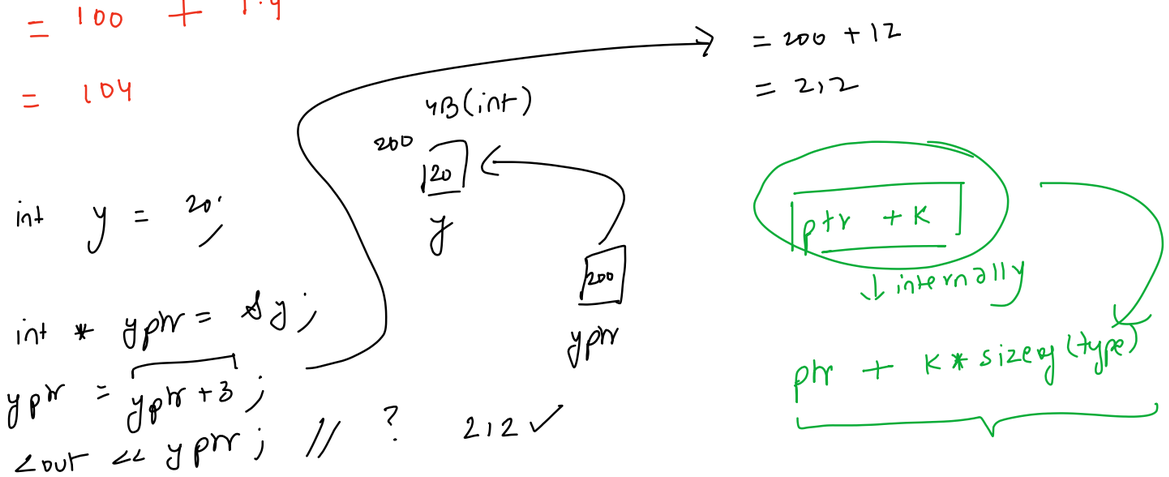


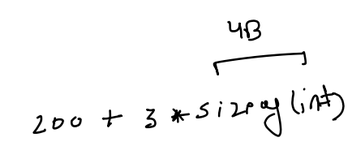


Pointer Arithmetic

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







ptr + K * sizeof(type)

Pointers and Functions

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

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

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

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

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

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

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;
}
```

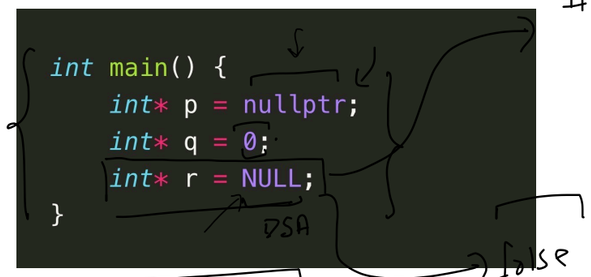
In C++, void pointers cannot be **dereferenced**.

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(psize == 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;
}
```

Null Pointers

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

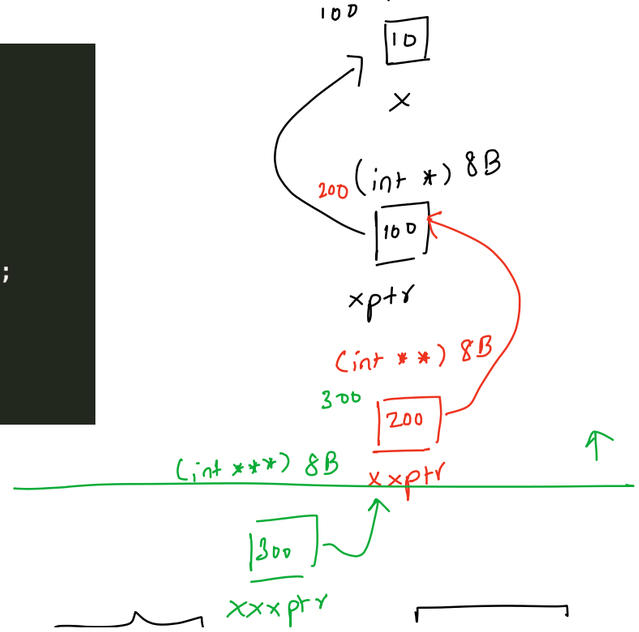


In C++, null pointers cannot be dereferenced.

Pointer to Pointer

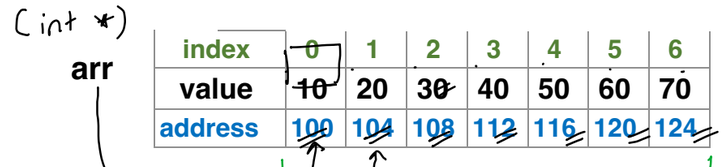
In C++, we can create a pointer to point to another pointer.

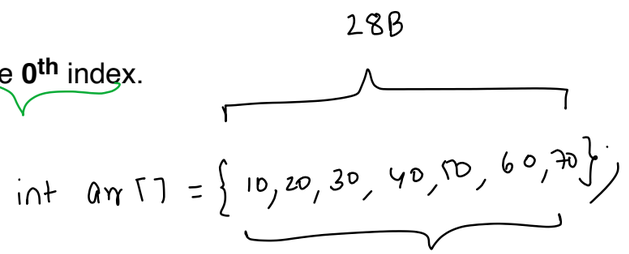
```
int main() {

    int x = 10;
    int* xptr = &x;
    int** xxptr = &xptr;
    int*** xxxptr = &xxptr;

    return 0;
}
```



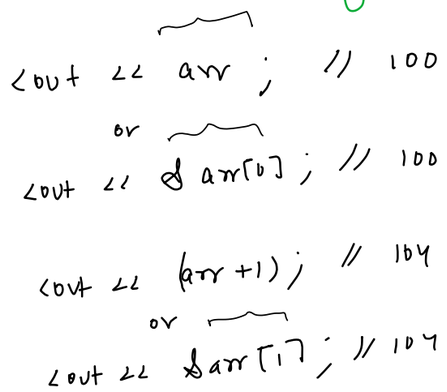
Pointers and Arrays

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

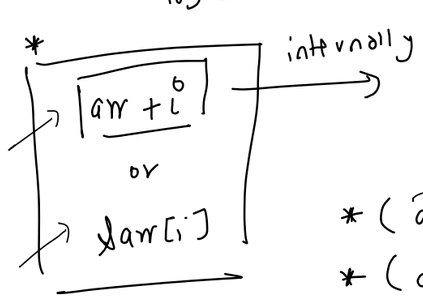




arr + 1
↓ internally
100 + 1 · sizeof(int)
100 + 4
104



100 101 102 103 } 4B ⇒ 0th idx



arr + i * sizeof(int)

*arr ⇒ arr[0]
*(arr+1) ⇒ arr[1]
*(arr+2) ⇒ arr[2]
⋮
*(arr+i) ⇒ arr[i]

10
20
30
⋮

internally

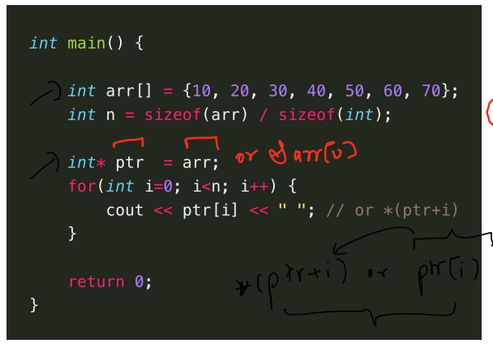
arr + i * sizeof(int)
= =

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

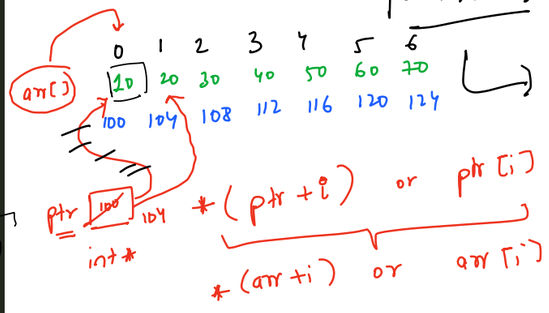
arr = arr+i // error

arr++; // error

ptr++;



(int*)



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

↳ it is a const time op

*(ptr+i) or ptr[i]

*(arr+i) or arr[i]

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.