PIC 10A 2B

TA: Bumsu Kim

UCLA

Today…

- Pointers

- HW7 and HW8

UCLA

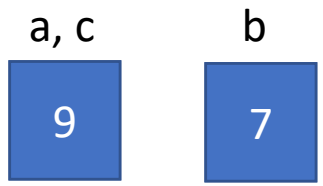
Pointers

- Recall that a reference to a variable is just “another name” of the other

```
class T {
        // ...
};
T tvar;                 // variable of type T
T& ref_tvar = tvar;             // a reference to a type T variable
```

- And they both refer to the same “address” in the memory

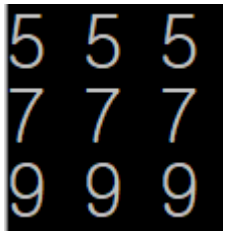
```
int a = 5;
int b = a;
int& c = a;
cout << a << ' ' << b << ' ' << c << endl;
b = 7;
cout << a << ' ' << b << ' ' << c << endl;
c = 9;
cout << a << ' ' << b << ' ' << c << endl;
```

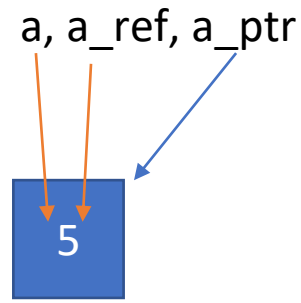


Pointers

- So, a reference, in fact, refers to the value in some address
- Pointers are used to “point” to those address

```
int a = 5;
int& a_ref = a;
int* a_ptr = &a; // address of a
cout << a << ' ' << a_ref << ' ' << *a_ptr << endl;
a_ref = 7;
cout << a << ' ' << a_ref << ' ' << *a_ptr << endl;
*a_ptr = 9;
cout << a << ' ' << a_ref << ' ' << *a_ptr << endl;
```





- Note that, they store the “address” so you can’t do something like

```
a_ptr = a;
```

a value of type "int" cannot be assigned to an entity of type "int *"

Search Online

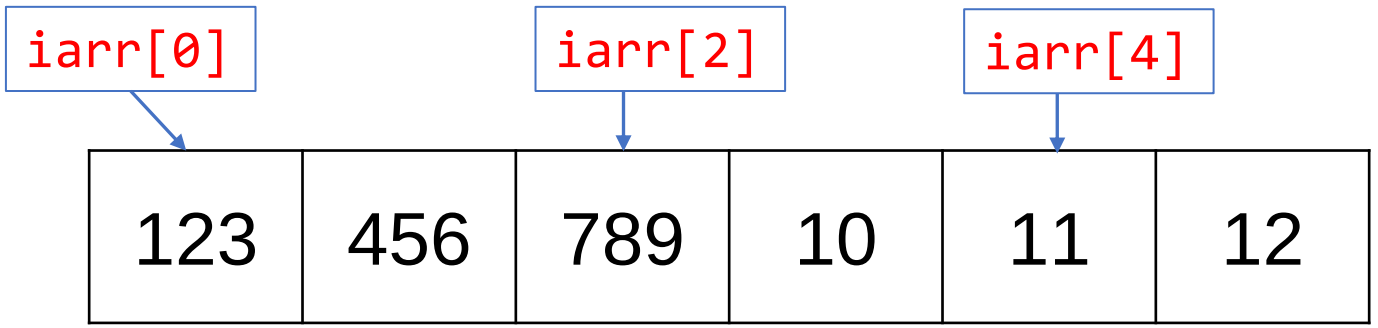
UCLA

Pointer Arithmetic

- To access the value stored in the address where your pointer points, use the ***dereference operator*** (*)

```
int a = 5;
int* a_ptr = &a; // address of a
*a_ptr = 9;            // dereference to access the value
cout << a << ' ' << *a_ptr << endl; // again, dereference to access the value
```

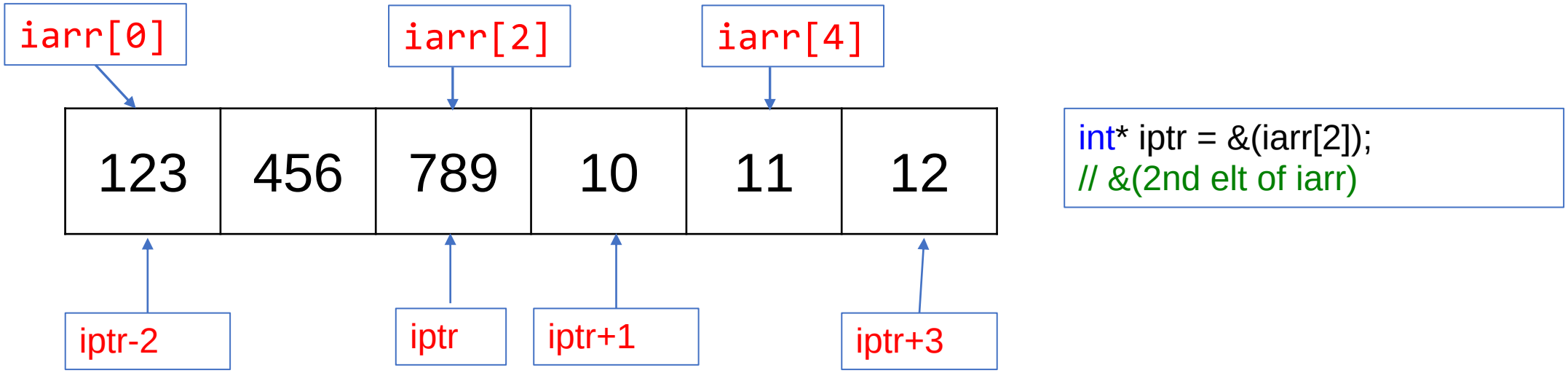
- You can also “move around” the address using “pointer arithmetic”
- Let’s say you have an `int` array of size 6:



UCLA

Pointer Arithmetic

- An `int` array of size 6 may look like (in the memory):




```
int* iptr = &(iarr[2]);
// &(2nd elt of iarr)
```

- Let iptr be a pointer pointing to the 2nd element of the array, then you can move around by adding/subtracting numbers from its address
- This is called "pointer arithmetic" (just like "integer arithmetic" but no *,/,%, etc.)
  - +, -, and ++, -- are allowed
  - Additionally, comparisons (==, <, >) are also available

UCLA

Arrays vs Pointers

```
size of a: 20
size of aptr: 8
```

- Arrays "decay" to pointers if it needs to be
- But still, they are DIFFERENT!

```
int a[] = { 1,2,3,4,5 };
int* aptr = a;
cout << "size of a: " << sizeof(a) << endl
<< "size of aptr: " << sizeof(aptr) << endl;
```

- You can compute the length of the array using "`sizeof(a)/sizeof(int)`"

- But then why do you still need to pass the array size to a function?
  - e.g. consider the following exercise from the textbook:

(This is essentially Exercise E6.8 of the book)
Write a program equals.cpp that implements the function

```
bool equals(int a[], int a_size, int b[], int b_size)
```

UCLA

Arrays vs Pointers

- It is because an array ***decays*** to a pointer when it is passed to a function
- You can never pass an array (by value), nor return an array
  - Passing arrays is not an error, but it decays to a pointer
  - Returning an array is an error!

- *Remark*: In fact, arrays can be passed by reference

```
int f(int(&a)[3]) {
    cout << "*size of this array: " << sizeof(a) << endl;
    return 0;
}
```

  - This function can get an array of size 3, and the output is `*size of this array: 12`

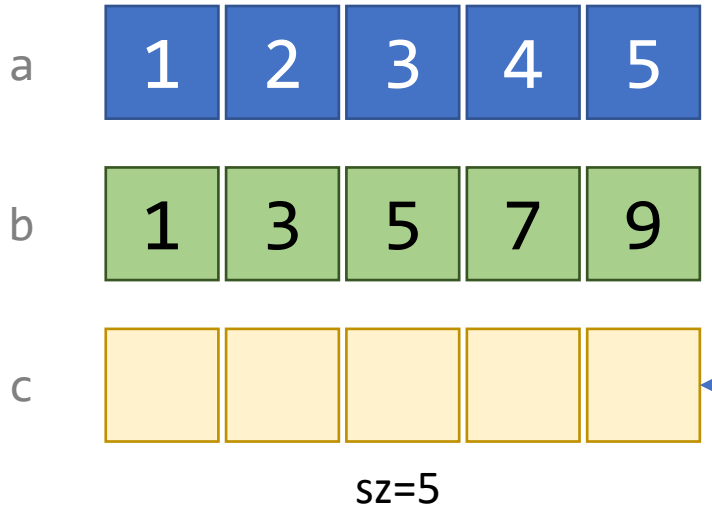
Arrays vs Pointers

- Since the implicit conversion [array → pointer] is allowed, you can use arrays in every expression where pointers are expected

```
void add_arrays(int a[], int b[], size_t sz, int *c) {
    for (size_t i = 0; i < sz; ++i) {
        *(c++) = *(a++) + *(b++);
    }
}
```

- How does the function work?

Arrays vs Pointers

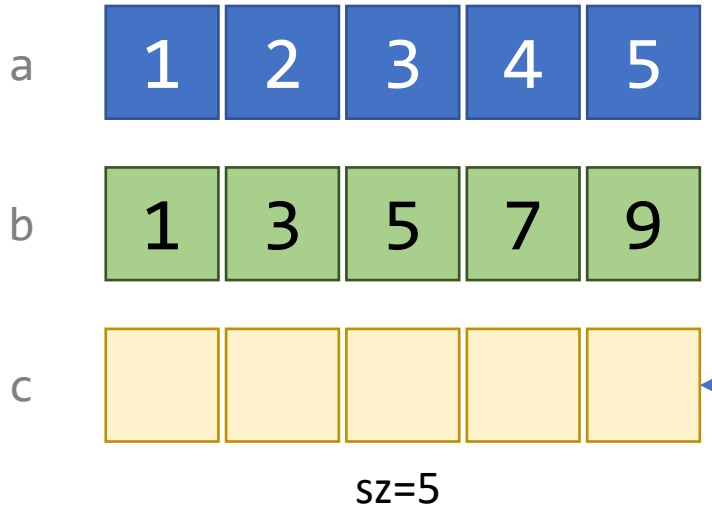



```
void add_arrays(int a[], int b[], size_t sz, int *c) {
    for (size_t i = 0; i < sz; ++i) {
        *(c++) = *(a++) + *(b++);
    }
}
```

At least "sz" places must be reserved!
(otherwise it may corrupt your data and result in undefined behavior)

- Pointers move forward by ++
- The post-increment operator returns the original value
- Access the value stored in an address via the dereference operator *

Arrays vs Pointers




```
void add_arrays(int a[], int b[], size_t sz, int *c) {
    for (size_t i = 0; i < sz; ++i) {
        *(c++) = *(a++) + *(b++);
    }
}
```

At least "sz" places must be reserved!
(otherwise it may corrupt your data and result in undefined behavior)

- Pointers move forward by ++
- The post-increment operator returns the original value
- Access the value stored in an address via the dereference operator *

HW7 and HW8 Questions?

- HW7: Right Triangle and Moth
  - First decide which member variables to use
  - Methods are very straightforward to implement

- HW8: Replace Number
  - 1. store the “old value”
  - 2. change the value that p refers to if necessary
  - 3. return the old value

- HW8: Find Number
  - 1. Use a for loop to check if we find the match
  - 2. Return the pointer (using the pointer arithmetic), or nullptr if not found