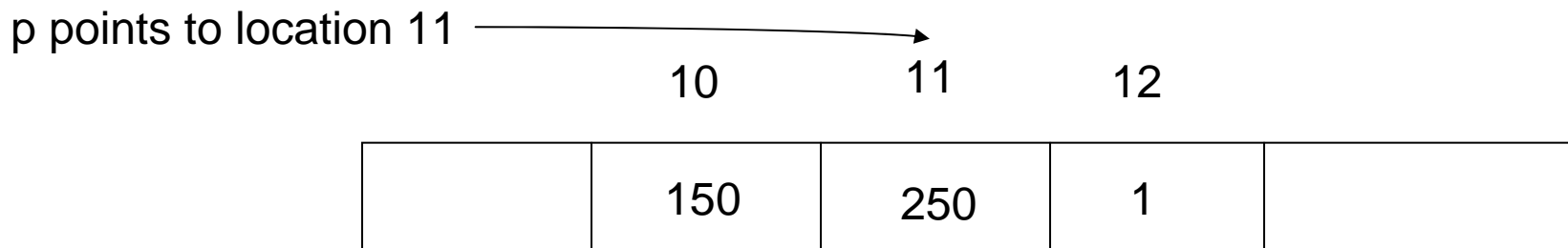


C++

pointers

Pointers

- A pointer is a variable storing the location or address of a memory cell



Pointer operators

- $*P$ is the variable pointed to by P
- $\&X$ is the address of X
- $\&$ is the **address of** operator
- $*$ is the **dereferencing** operator
- Suppose $P1 = \&X$ and $P2 = \&Y$, then:
 - $P1 = P2$ means that $P1$ now points to Y
 - $*P1 = *P2$ means that X has the same content as Y

Pointer operators - example

- `int X; // X is an integer`
`int* P; // P is a pointer to an integer`
`P = &X; // P stores the address of X`
`*P = 5; // sets the value of the memory location pointed to by P to 5`
- At the end of the example: `X == 5` and P points to X

Classical error with pointer

- `int* P, Q; // P is a pointer but not Q`
- Instead of: `int *P, *Q; // P and Q are pointers`
- Use **typedef** to avoid such error:
`typedef int* IntPtr; // now IntPtr is a new name for the type int*`
`IntPtr P, Q; // P and Q are both pointers`

Static and dynamic memory allocation

- The following code allocates **memory statically** (i.e. at compile time):

```
int X, Y;  
int* P;
```

- Memory allocated during execution is known as **dynamically allocated memory**:

```
P = new int; // P points to a memory cell  
containing an integer
```

Dynamic memory allocation

- Dynamically allocated memory is allocated in a special area of memory called **heap**
- In C++ dynamically allocated memory is obtained with **new**
- If **new** can not allocate sufficient memory, it throws a **bad_alloc** exception or returns **NULL**; (default is to throw an exception)

Example

```
// ...
int* p;
try {
    p = new int;
} catch (bad_alloc&) {
    std::cout << "error allocating memory"
              << std::endl;
}
// ...
```

```
#include <new>
// ...
int* p;
p = new (nothrow) int;
if (p == NULL) {
    std::cout << "error allocating memory"
              << std::endl;
}
// ...
```

Delete

- The heap has a limited amount of space, so unused allocated memory needs to be returned to the heap
- If P is a pointer, the associated memory can be deleted by:
 `delete(P);`
- This statement deletes the memory area P points to.
 - **It does not modify P .**
 - After executing `delete(P)`, the value of P is **undefined**.

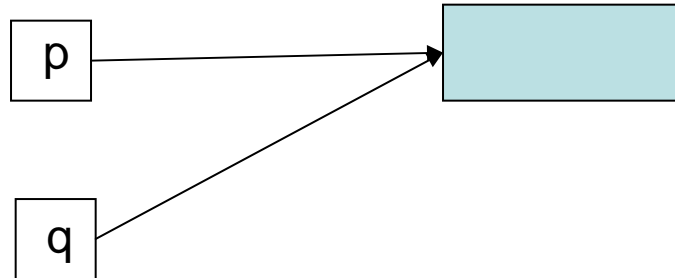
Malloc and free

- new and delete are specific for C++
- malloc and free, the functions used to allocate and free memory in the heap in C are also available in C++
- It is **not recommended** to use malloc and free.

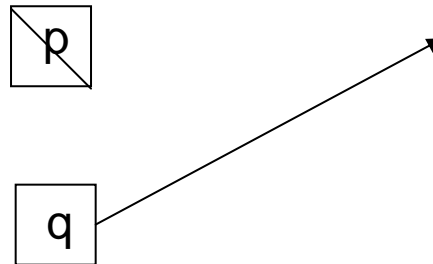
Common problems with pointers

- **Dangling pointer:**

```
int* p; int* q;  
p = new int;  
q = p;
```



```
delete p;  
p = null;
```



- This leaves q in an undefined state

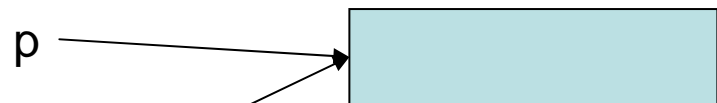
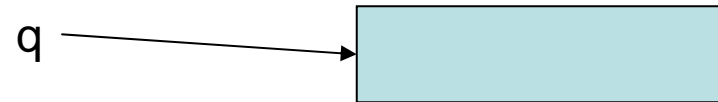
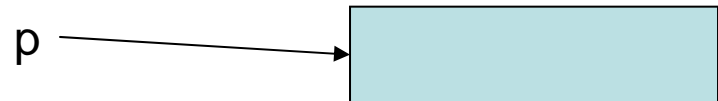
Common problems with pointers

- Memory leak: it happens when we lose the address of space allocated in the heap

- Example:

```
int* p; int* q;  
p = new int;  
q = new int;
```

```
q = p;
```



Common problems with pointers

- The two previously mentioned problems with pointers are sources of most of the corrupted memory problems when programming in C++ (and C as well)
- There exist some tools that help developers tracking such problems: purify, valgrind
- Try to limit the usage of pointers in the code

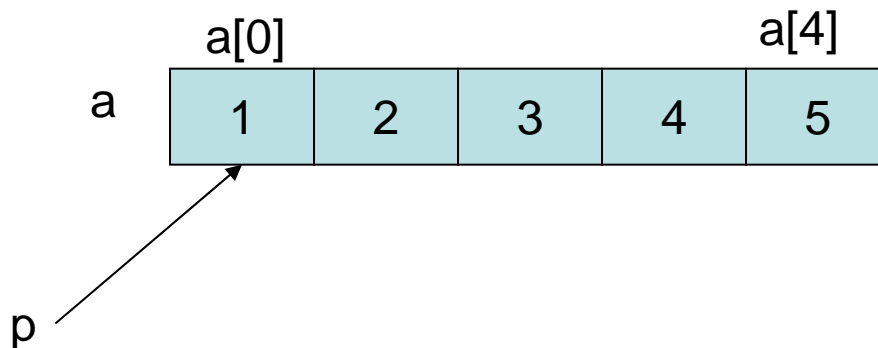
Arrays and pointers

- To some extent we can think of an array name as a pointer to the beginning of the array in memory:

```
int a[5] = {1, 2, 3, 4, 5};
```

```
int* p;
```

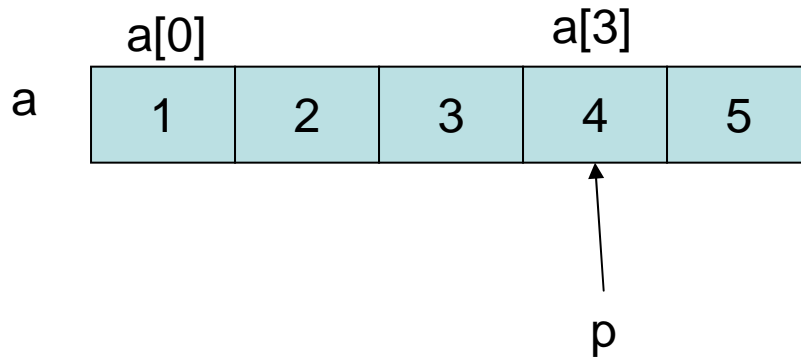
```
p = a; // p points to a[0]
```



```
p[1] = 11;  
// now a[1] == 11
```

Arrays and pointers

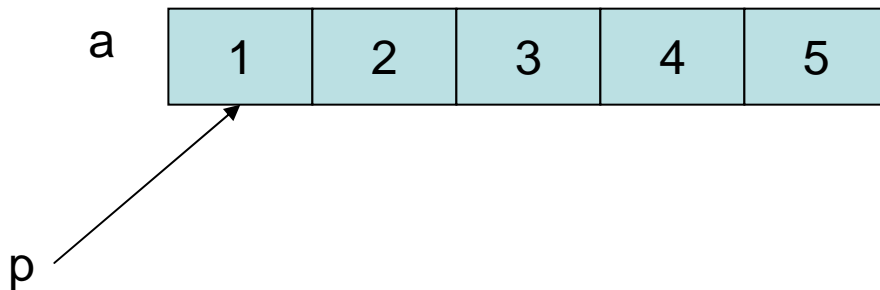
- It is of course possible to start from a different location:
 $p = \&(a[3]);$
- Now: $p[0]$ would refer to $a[3]$, $p[1]$ to $a[4]$, etc



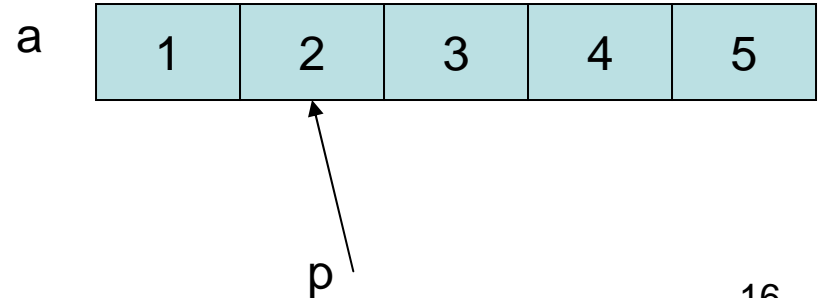
Arithmetic on pointers

- Arithmetic on pointers is different (than for example) arithmetic on integers
- `Type* p;`
`p = p + i; // now p stores the address of p + i * sizeof(Type)`
- `p++;` // corresponds to the address `p + sizeof(Type)`
- Examples:

`p = a`



`p++;`



Dynamic allocation of Arrays

- **p = new T[n]** will allocate an array of n objects of type T. It returns a pointer to the beginning of the array.
- `delete[] p` will destroy the array to which p points and return the memory to the heap.
p must point to the beginning of the allocated array otherwise the result of `delete[] p` is undefined.
- Always use `delete[]` to free memory allocated for an array. **DO NOT USE DELETE** in that case.