LECTURE 10 · CSE 333 · Summer 2026

C++: objects

Discussion: Teach me some slang! I'll try to use it lecture. (EDIT: nobody did)

Reminders

Assessments:

- Exercise 4 will be due on Friday at 11:59pm
  - Includes C and C++
- HW2 is due next Thursday at 11:59pm
- Exercise 5 will be released on Friday.

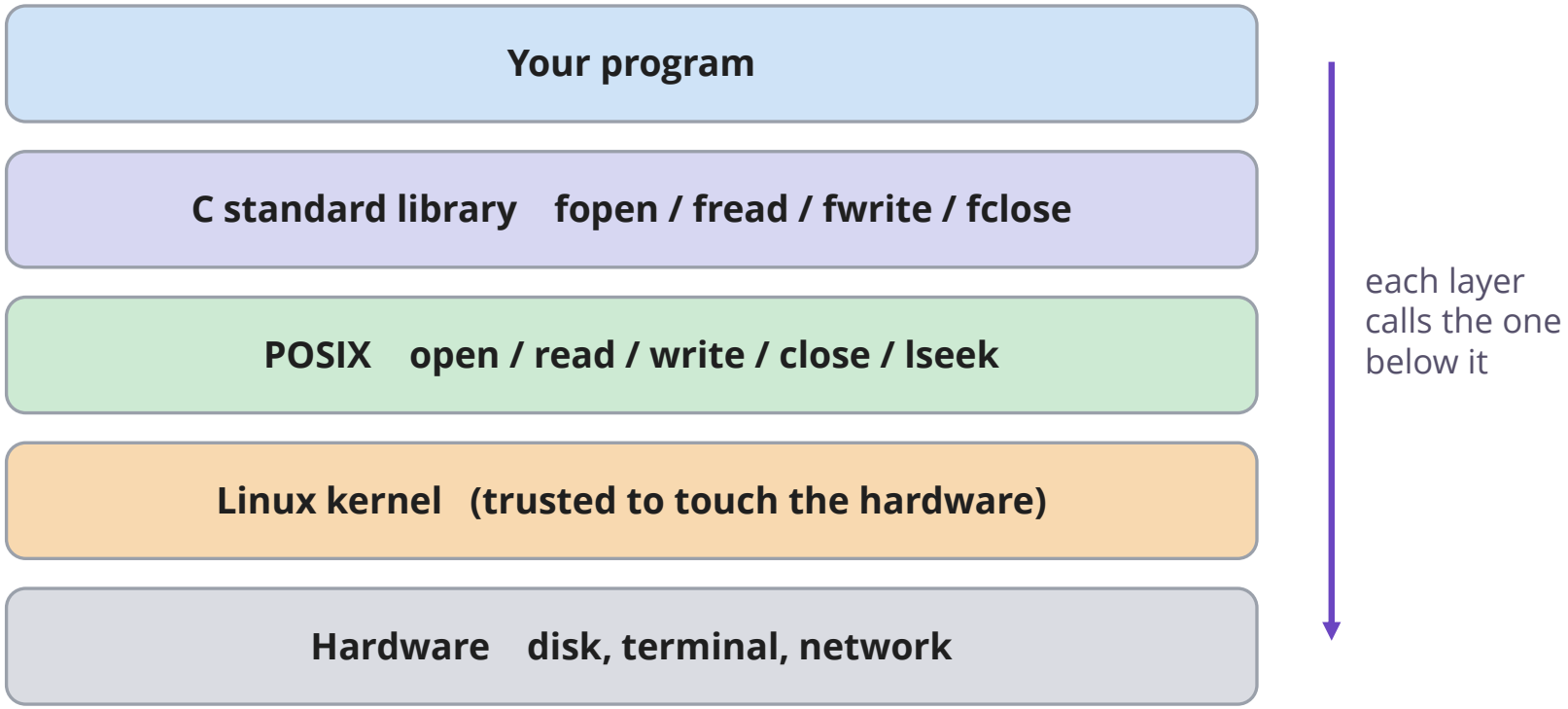
General:

- Working to find some industry guest lecturers for you!
- Working to see if we can do a final project (not exam)
- Working to try and fit in some “how AI is used in industry”
- Less than 2 weeks till the midterm

Review

REVIEW

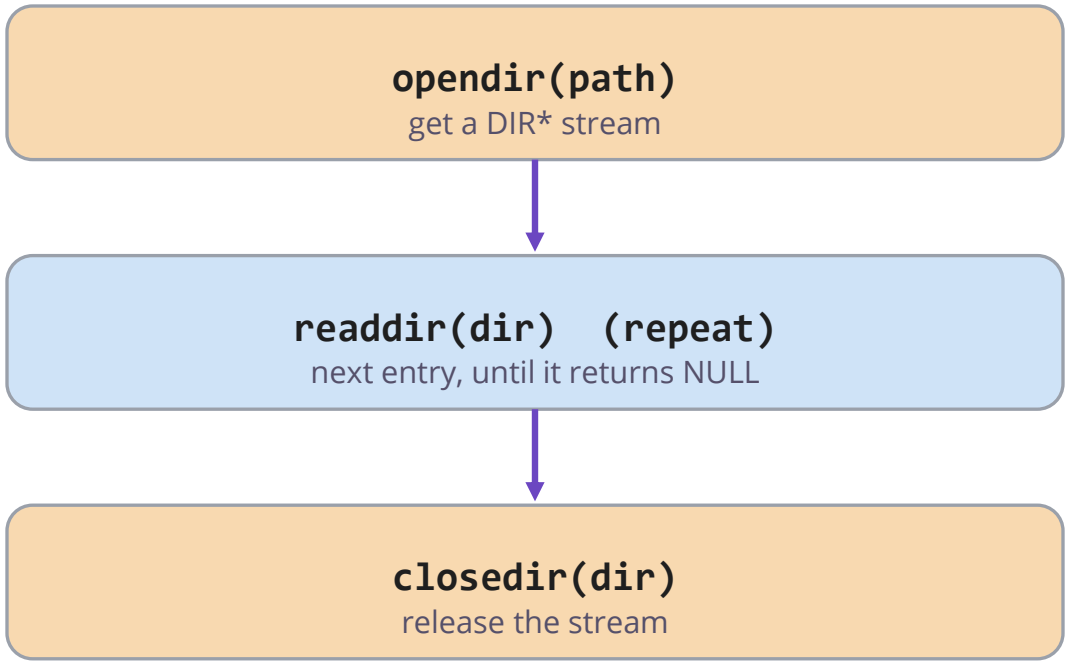
The I/O software stack



REVIEW

A directory is just a special file

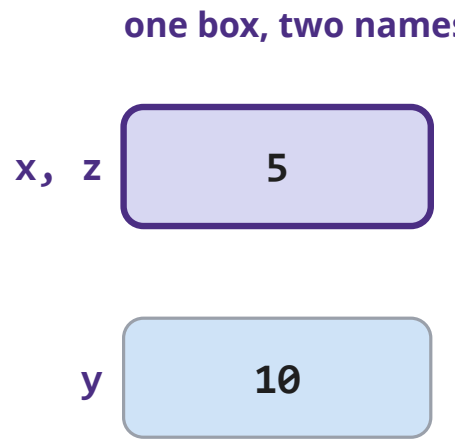
- A directory is a file whose contents are a list of entries (names plus some metadata)
- So POSIX gives you directory calls that mirror open / read / close
- **You do not read the bytes yourself:** the library hands you one entry at a time



Step 1 of 4

Reference (in C++)

```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int& z = x;   // binds the name z to x
4    z += 1;         // sets z (and x) to 6
5    x += 1;         // sets x (and z) to 7
6    z  = y;         // copies y's VALUE into x
7    z += 1;         // sets z (and x) to 11
8    return EXIT_SUCCESS;
9  }
```



❖ **int& z = x;** makes z another name for x.

❖ There is no separate box for z, and no address to store.

REVIEW

Pointers vs references

```
1  void IncPtr(int* p) { *p += 1; }   // pointer version
2  void IncRef(int& r) {  r += 1; }   // reference version
3
4  int main(int argc, char** argv) {
5    int a = 5;
6    IncPtr(&a);   // caller: &a,  body: *p
7    IncRef(a);    // caller:  a,  body:  r
8    // a is now 7
9    return EXIT_SUCCESS;
10 }
```

❖ **Both functions do the same thing**: add 1 to the caller's a.

❖ **Pointer:** the caller writes **&a**, the body writes ***p**.

❖ **Reference:** the caller writes **a**, the body writes **r**. No stars, no ampersands.

Pass around big objects cheaply

```
1  struct Image {                  // a big struct
2    int  width, height;
3    char pixels[1000000];         // ~1 MB of data
4  };
5
6  void Save(Image img);           // copies ~1 MB!
7  void Save(Image& img);          // no copy
```

❖ **Pass by value** copies the whole struct into the parameter. Here that is a 1 MB copy on every call.

❖ Pass by **const&** hands the function a read-only alias: no copy, and the compiler forbids modifying it.

❖ **This is the most common reference idiom in C++:** big inputs go by const reference.

In practice… we normally use this instead

```
1  struct Image {                  // a big struct
2    int  width, height;
3    char pixels[1000000];         // ~1 MB of data
4  };
5
6  void Save(Image img);           // copies ~1 MB!
7  void Save(const Image& img);    // no copy, no mutate
```

❖ **Pass by value** copies the whole struct into the parameter. Here that is a 1 MB copy on every call.

❖ Pass by **const&** hands the function a read-only alias: no copy, and the compiler forbids modifying it.

❖ **This is the most common reference idiom in C++:** big inputs go by const reference.

const can be weird!

```
const Account* p;
```

p is a pointer to a const Account

data locked, pointer free

`p->balance=0;` error `p=&other;` ok

```
Account* const p;
```

p is a const pointer to an Account

pointer locked, data free

`p->balance=0;` ok `p=&other;` error

```
const Account* const p;
```

const pointer to a const Account

both locked

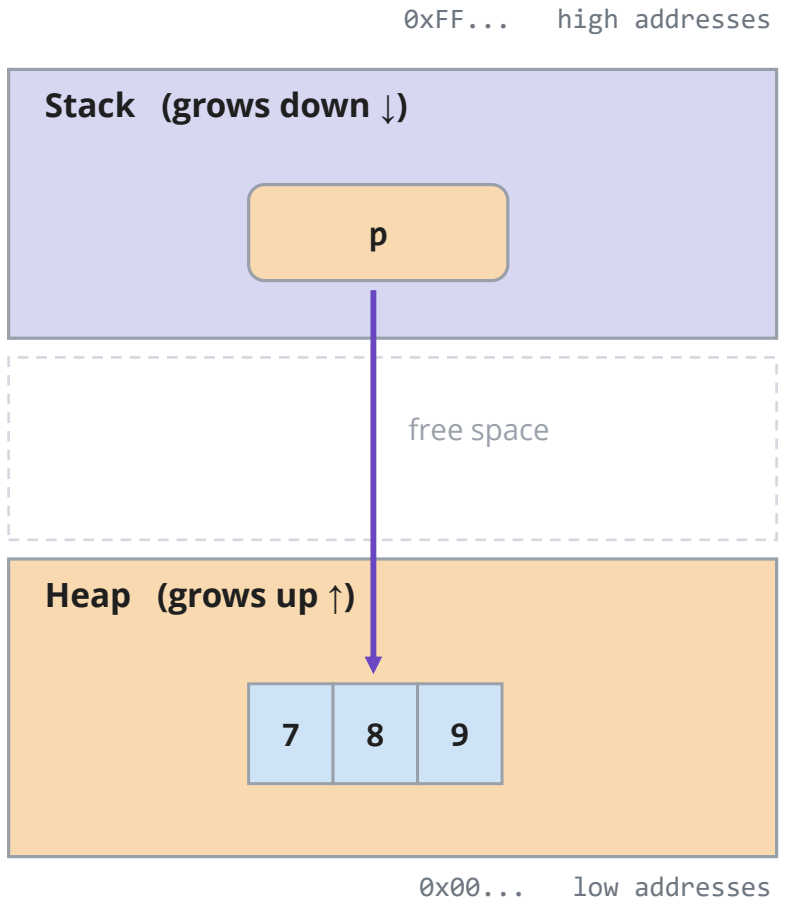
`p->balance=0;` error `p=&other;` error

Heap

new and delete

Stack vs. Heap

```
1  int  x = 5;            // x lives on the stack
2  int* p = new int[3];   // p on the stack,
3                         // the array on the heap
```



- **Locals** live on the **stack**, freed automatically at end of scope.
- **new / new[]** carve memory from the **heap**, which stays until you delete it.
- The pointer sits on the stack; **what it points to sits on the heap.**

new and delete

```
1  // one value on the heap
2  int* n = new int(333);   // allocate an int, set to 333
3  int* m = new int;        // allocate an int, UNINITIALIZED
4
5  delete n;                // give the memory back
6  delete m;
```

- **new T** allocates space for a T on the heap and returns a **T***.
- **delete p** gives that memory back to the heap.
- **new / delete** are typed: no cast, no sizeof, unlike malloc.

Takeaway: `new` allocates a typed block on the heap. `delete` gives it back to the system. Do NOT mix with malloc/free

malloc vs new, in one line each

new (C++)

- Returns a **typed pointer** (no cast needed).
- Can **initialize** the value: new int(0).
- Throws **bad_alloc** on failure (never NULL).
- Released with **delete / delete[]**.

malloc (C)

- Returns **void*** (you cast / assign it).
- **No initialization:** just raw bytes.
- Returns **NULL** on failure (you must check).
- Released with **free**.

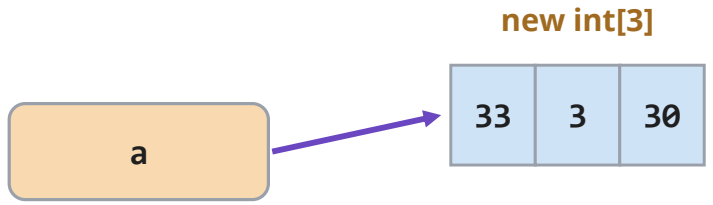
new throws, it does not return NULL

```
1  int* n = new int(333);      // ok: one heap int
2
3  // On failure, new THROWS std::bad_alloc.
4  // It never returns NULL, so there is no
5  // NULL check to write (unlike malloc).
6  int* big = new int[HUGE];   // no NULL check
7
8  delete n;
```

- **On failure new throws** std::bad_alloc.
- **It never returns NULL,** so the malloc-style NULL check is dead code here.
- **An exception unwinds;** you do not test the returned pointer at all.

new[] and delete[]

```
1  // dynamically allocated array of 3 ints
2  int* a = new int[3];
3
4  a[0] = 33;  a[1] = 3;  a[2] = 30;
5
6  delete[] a;   // note the [] !
```



- **type* a = new type[n]** allocates n contiguous elements on the heap.
- **Array new pairs with** delete[], not plain delete.

new[] and delete[]

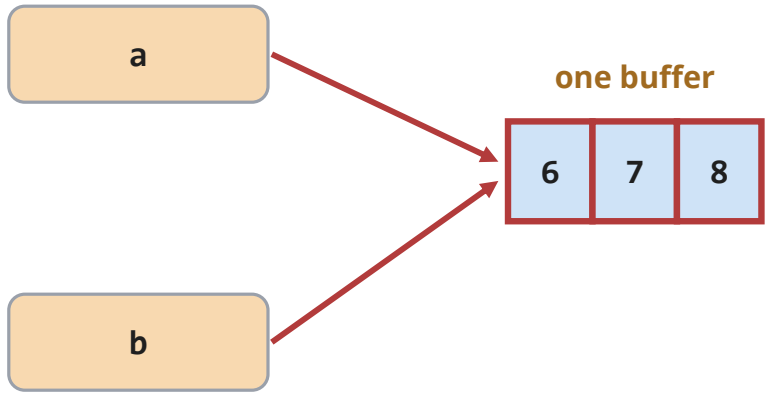
```
1  // C++11 brace initialization
2  int* a = new int[3]{1, 2, 3};
3
4  // no initializer: values are garbage
5  int* b = new int[3];
6
7  delete[] a;   // match new[] with delete[]
8  delete[] b;
```

delete[] vs delete

- **delete[]** frees the WHOLE array; plain delete frees only one element.
- **Wrong form is undefined behavior:** delete on a new[] array, or delete[] on a new array.
- The compiler usually cannot catch this: both are just a pointer. It is on you to match them.

int* b = a shares the buffer

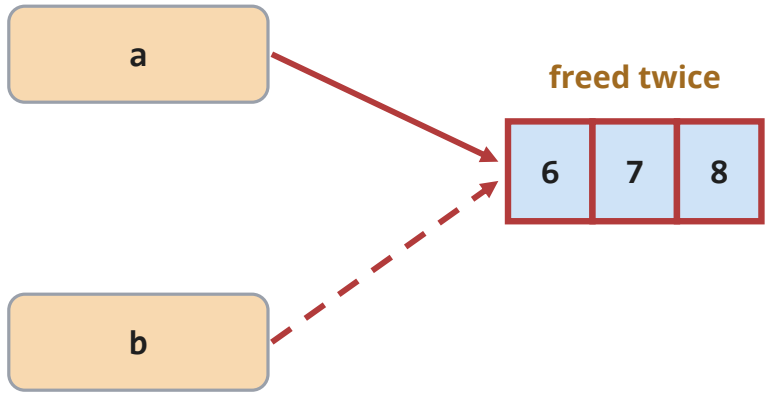
```
1  int* a = new int[3]{6, 7, 8};
2
3  int* b = a;     // copies the POINTER, not the values
4  b[0] = 0;       // now a[0] is 0 too  (aliasing!)
5
6  delete[] a;
7  delete[] b;     // SAME array -> double free
```



Predict first: a and b now share one buffer. What happens at the two deletes?

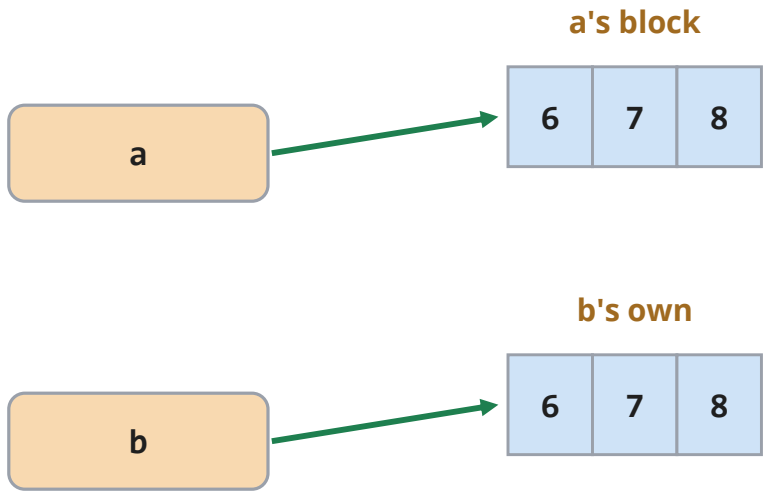
Sharing means a double free

- **delete[] a** frees the block. Fine so far.
- **delete[] b** frees the SAME array again.
- **Double free** = undefined behavior, heap corruption.
- **The fix:** give b its own buffer. Copy DEEP, not shallow.



Deep copy: b gets its own block

```
1  int* a = new int[3]{6, 7, 8};
2
3  int* b = new int[3];      // b gets its OWN arrays
4  for (int i = 0; i < 3; i++)
5    b[i] = a[i];            // copy the values across
6
7  delete[] a;
8  delete[] b;               // two arrays, two frees
```



❖ **Allocate a second block,** then copy the values across. Two owners, two clean deletes.

❖ A deep copy allocates its own buffer and copies the values, so each pointer owns and frees its own block.

Dangling pointers

```
1  int* a = new int[3]{6, 7, 8};
2  delete[] a;          // a is now a DANGLING pointer
3
4  a[0] = 5;            // use-after-free: undefined behavior
5  delete[] a;          // double free: undefined behavior
6
7  a = nullptr;         // habit: null it after delete
```

After delete, the pointer is dead

- **delete[] a** frees the block, but a still holds the old address.
- **Using a now** (read, write, or delete again) is undefined behavior.
- **Habit:** set a = nullptr after delete, so misuse crashes loudly.

Cleaning up an array of pointers

```
1  char** words = new char*[3];    // array of 3 char*
2  words[0] = new char[6];         // each string: its own
3  words[1] = new char[4];         // heap block
4  words[2] = new char[8];
5
6  for (int i = 0; i < 3; i++) {
7    delete[] words[i];   // 1. free each string first
8  }
9  delete[] words;         // 2. then free the array
```

Two levels, two deletes

- **words** is a heap array whose elements are themselves heap pointers.
- **Free inside-out:** delete[] each string first…
- …then **delete[] words** for the array itself.
- **Delete the array first** and you leak every string it pointed to.

Objects (!)

struct vs class in C++

```
1  struct Pair {    // members PUBLIC
2    int a;         // by default
3    int b;
4  };
```

```
1  class Pair {     // members PRIVATE
2    int a;         // by default
3    int b;
4  };
```

- **In C++ they are almost the same thing.** Both can have members, methods, constructors, everything.
- **The only language difference:** default access. struct starts public, class starts private.
- **Style convention (Google / CSE 333):**
  - **struct** for plain passive data, no invariants
  - **class** for data plus behavior that guards it
- We will write classes for the rest of the unit.

Interface in the .h, definitions in the .cc

Point.h (the interface)

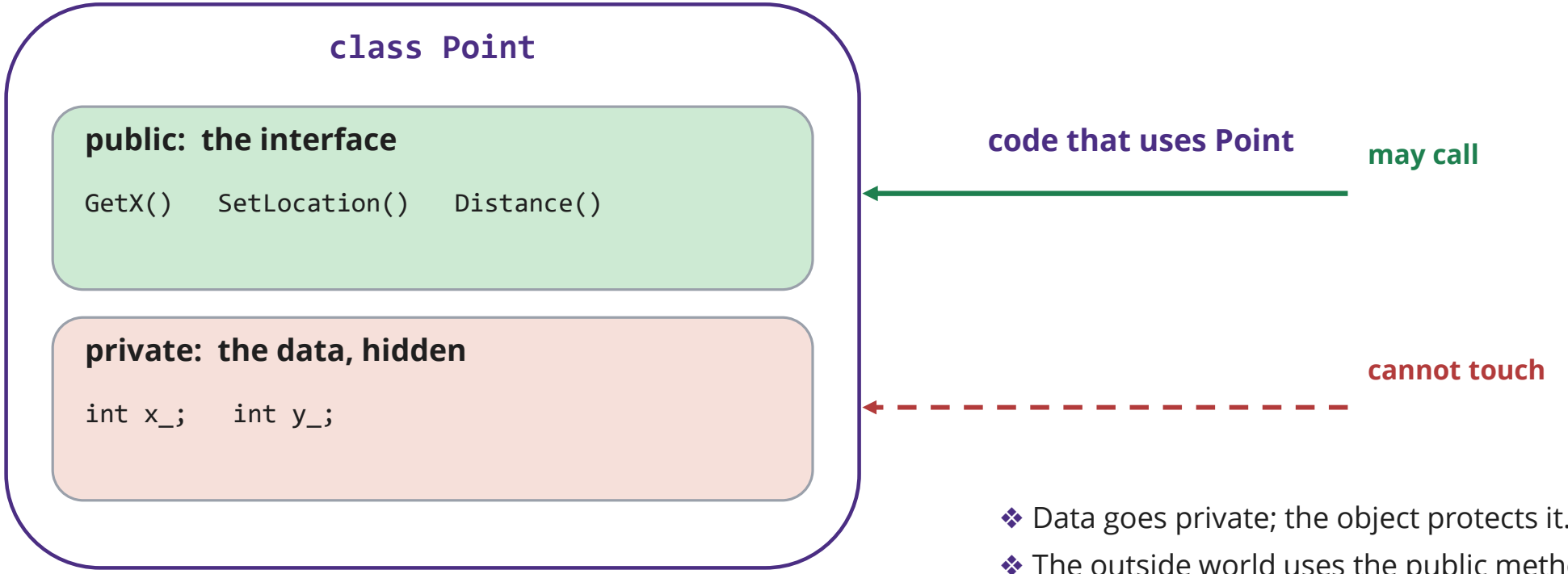
```
1  #ifndef POINT_H_
2  #define POINT_H_
3
4  class Point {
5   public:
6    Point(int x, int y);               // constructor
7    int GetX() const { return x_; }  // inline getter
8    int GetY() const { return y_; }
9    double Distance(const Point& p) const;
10   void SetLocation(int x, int y);
11
12  private:
13   int x_;   // data member
14   int y_;
15 };
16
17 #endif  // POINT_H_
```

Point.cc (the definitions)

```
1  #include "Point.h"
2  #include <cmath>
3
4  Point::Point(int x, int y) {
5    x_ = x;
6    y_ = y;
7  }
8
9  double Point::Distance(const Point& p) const {
10   double dx = x_ - p.GetX();
11   double dy = y_ - p.GetY();
12   return sqrt(dx * dx + dy * dy);
13 }
```

- **retType Class::Method(...)** ties a definition back to its class.
- Short getters/setters can be defined inline in the header.
- NOTE: the getters are const

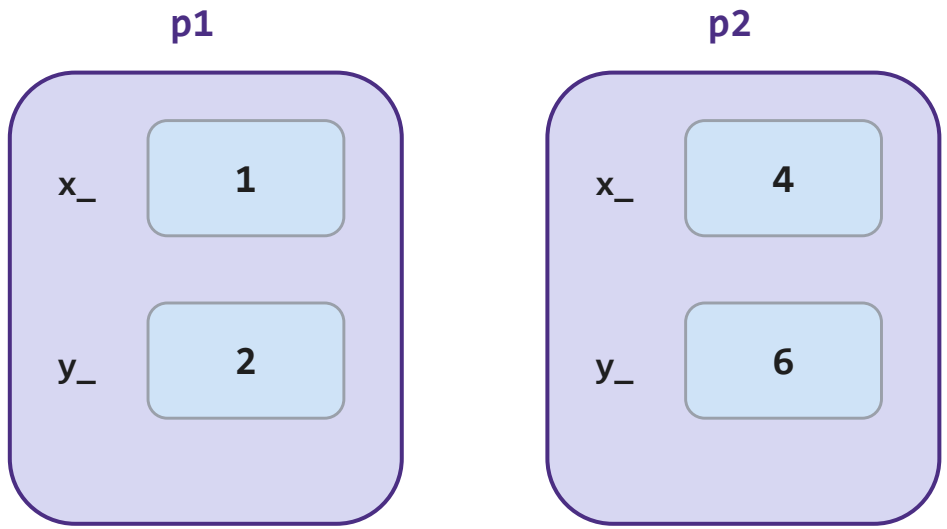
public vs private



- Data goes private; the object protects it.
- The outside world uses the public methods only.

An object is variables within a variable

```
1  Point p1(1, 2);
2  Point p2(4, 6);
```



- Each object gets its own copy of the data members, laid out back to back.
- **p1.x_** and **p2.x_** are different storage, just like two ints.
- Methods are shared code; they act on whichever object you call them on.

const objects and const methods

```
1  void PrintPoint(const Point& p) {
2    cout << p.GetX() << ", " << p.GetY();
3    //  ^ OK: GetX() is a const method
4
5    p.SetLocation(0, 0);  // compiler error!
6    //  ^ SetLocation is non-const
7  }
```

- Mark a method **const** when it does not modify the object.
- A **const** object can call only **const** methods.
- **p.SetLocation(...)** -> compile error.

The default ctor you get, and then lose

```
1  class Point {
2   public:
3    Point(int x, int y);   // we declared a ctor...
4   private:
5    int x_;
6    int y_;
7  };
8
9  Point p(3, 4);    // OK
10 Point q;          // ERROR: no default ctor anymore
```

- **Write NO constructor,** and the compiler synthesizes a default (no-arg) ctor for you.
- **Declare ANY constructor yourself,** and that free default ctor goes away.
- **Point q;** now fails to compile: there is no no-arg ctor.
- **Fix:** add **Point();** or **Point() = default;** if you still want it.

Takeaway: Declaring any constructor removes the free synthesized default. Add one back explicitly if you still need Point q;.

Initialization lists

```
1  // assignment in the body: default-build, then set
2  Point::Point(int x, int y) {
3    x_ = x;   // x_ was already built, now assigned
4    y_ = y;
5  }
6
7  // initialization list: build members directly
8  Point::Point(int x, int y) : x_(x), y_(y) { }
```

- The **: member(value)** list runs BEFORE the body.
- **Assignment version:** each member is default-built first, then overwritten. Two steps.
- **Init-list version:** each member is built directly from the value. One step.
- Required for **const** members and references, which cannot be assigned.

Takeaway: Prefer the initialization list: it constructs each member once, directly, instead of build-then-assign.

Members init in declaration order

```
1  class IntArray {
2   public:
3    explicit IntArray(size_t size)
4        : data_(new int[size_]),   // uses size_...
5          size_(size) { }          // ...set only here!
6   private:
7    int* data_;      // declared FIRST
8    size_t size_;    // declared SECOND
9  };
```

❖ The list says **data_(...)** then **size_(size)**, so it looks like size_ is ready.

❖ **It is not.** Members init in DECLARATION order: data_ FIRST, size_ second.

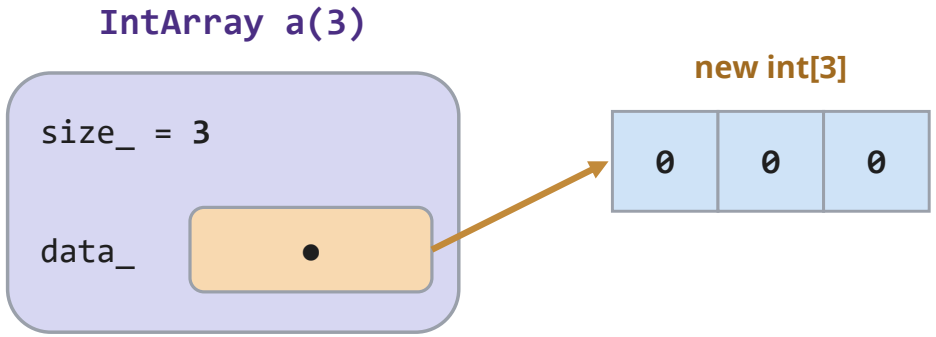
❖ So **new int[size_]** reads size_ before it is set. Garbage size, wild allocation.

❖ **Fix:** declare size_ before data_ (as our IntArray does), and list the initializers in that same order.

Takeaway: The init list order is ignored; members initialize in the order they are DECLARED. Declare size_ before data_ so the buffer sees a real size.

IntArray: a class that owns the heap

```
1  class IntArray {
2   public:
3    explicit IntArray(size_t size);
4    size_t Size() const { return size_; }
5    int At(size_t i) const { return data_[i]; }
6    void Set(size_t i, int v) { data_[i] = v; }
7   private:
8    size_t size_;    // declared FIRST
9    int* data_;      // heap buffer we OWN
10 };
11
12 IntArray::IntArray(size_t size) : size_(size) {
13   data_ = new int[size];   // who frees this?
14 }
```



- The object holds a **pointer**, not the array. The array lives on the heap.
- **So: who calls delete[]** on that buffer? And when?
- This is incorrect code – we'll come back to this…

Predict first: if we copy this object, what happens to the one heap buffer? We will come back to this…

Default

C++ defaults

Point.c (what you see)

```
1  class Point {
2   private:
3    int x_;   // data member
4    int y_;
5  };
6
7
8
9
10
11
12
13
14
15
16
17
```

Point.c (what you get)

```
1  class Point {
2    Point() = default;                                  // constructor
3    Point(const Point& other) = default;                // copy constructor
4    Point& operator=(const Point& other) = default;     // copy assignment
5    Point(Poin&t&& other) = default;                    // move constructor
6    Point& operator=(Point&& other) = default;          // move assignment
7    ~Point() = default;                                 // destructor
8
9   private:
10   int x_;   // data member
11   int y_;
12 };
13
14
15
16
17
```

Concept #1: The Rule of Three

If your class needs any one of these, it almost certainly needs all three. They all manage the same owned resource.

1. Destructor

```
~IntArray()
```

release the resource

2. Copy constructor

```
IntArray(const IntArray&)
```

deep-copy on construction

3. Assignment operator

```
operator=(const IntArray&)
```

free old, then deep-copy

Takeaway: The Rule of Three: if you write a destructor, copy constructor, or operator=, you almost certainly need all three. Ask first, who owns the resource?

Destructors

Destructors

```
1  class IntArray {
2   public:
3    explicit IntArray(size_t size);  // acquire
4    ~IntArray();                     // release
5    // ...
6   private:
7    int* data_;
8    size_t size_;
9  };
10
11 IntArray::~IntArray() {
12   delete[] data_;   // give the heap back
13 }
```

❖ **~IntArray()** runs automatically when the object dies.

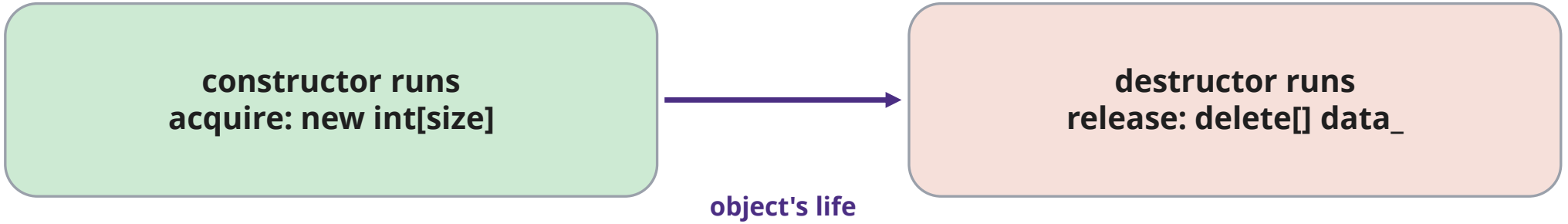
- a stack object: at the end of its scope
- a heap object: when you delete it

❖ **Its job:** release what the object owns, here **delete[] data_**. No return type, no params, one per class.

❖ **The private buffer can't leak:** cleanup is automatic, no delete[] at the call site.

Concept #1: RAII

RAII = Resource Acquisition Is Initialization. Acquire the resource in the constructor; release it in the destructor. The object's lifetime IS the resource's lifetime.



❖ Clients never call delete[] by hand because your object's destructor cleans up for you, even if an exception is thrown.

Takeaway: RAII: the object owns the resource, and the resource lives and dies with the object. This idea makes smart pointers inevitable.

Copy Constructors

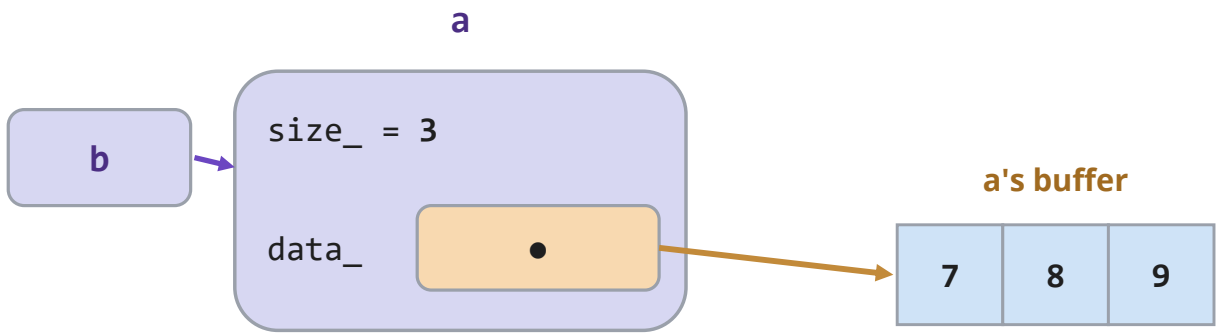
When do copies happen?

```
1  IntArray a(3);
2
3  IntArray b = a;       // 1. copy-initialization
4  IntArray c(a);        //    also a copy
5
6  Consume(a);           // 2. pass BY VALUE -> copy
7
8  IntArray d = Make(); // 3. return BY VALUE -> copy
```

- **A copy constructor builds a new object** from an existing one:
  - **IntArray(const IntArray& other)**
- It runs more often than you think:
  - initializing one object from another
  - passing an object BY VALUE
  - returning an object BY VALUE
- So the copy ctor has to be correct, even if you never call it by name.

A reference is just another name

```
1  IntArray a(3);
2  a.Set(0, 7);  a.Set(1, 8);  a.Set(2, 9);
3
4  IntArray& b = a;   // b: another NAME for a
```

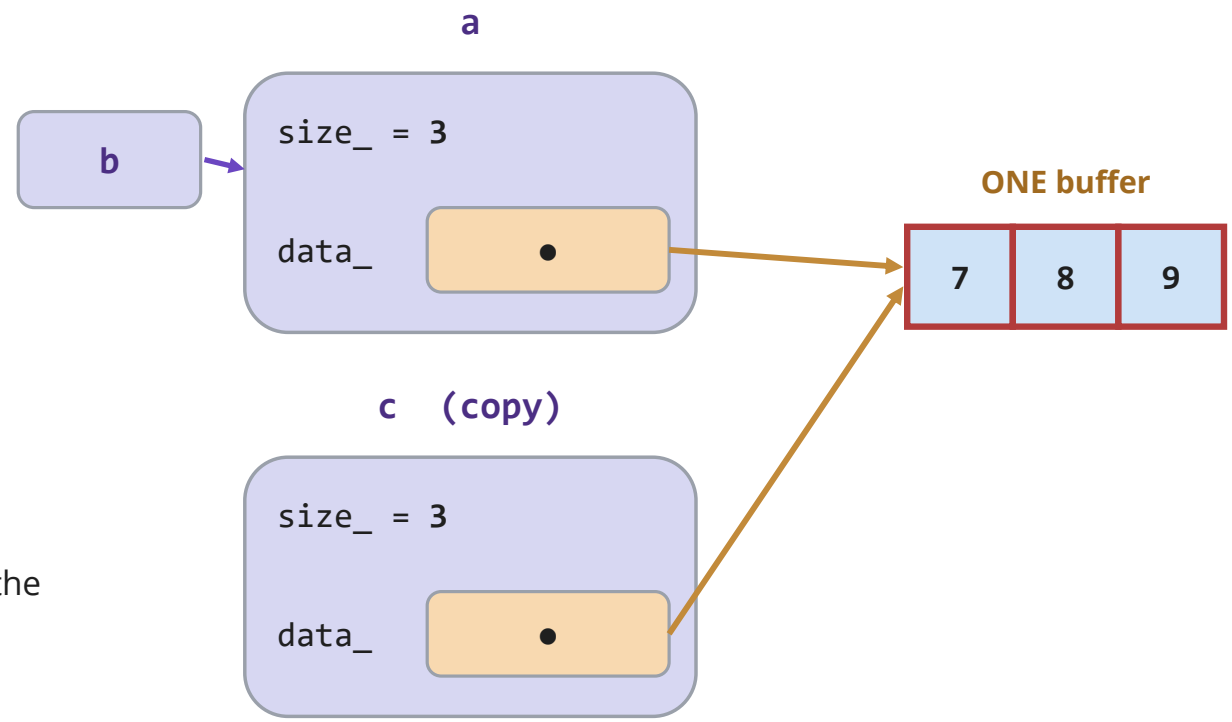


❖ **A reference makes NO new object and NO new buffer.** b is literally another name for a.

❖ **b.data_** IS a.data_. Still exactly one object and one buffer.

A copy shares the same buffer

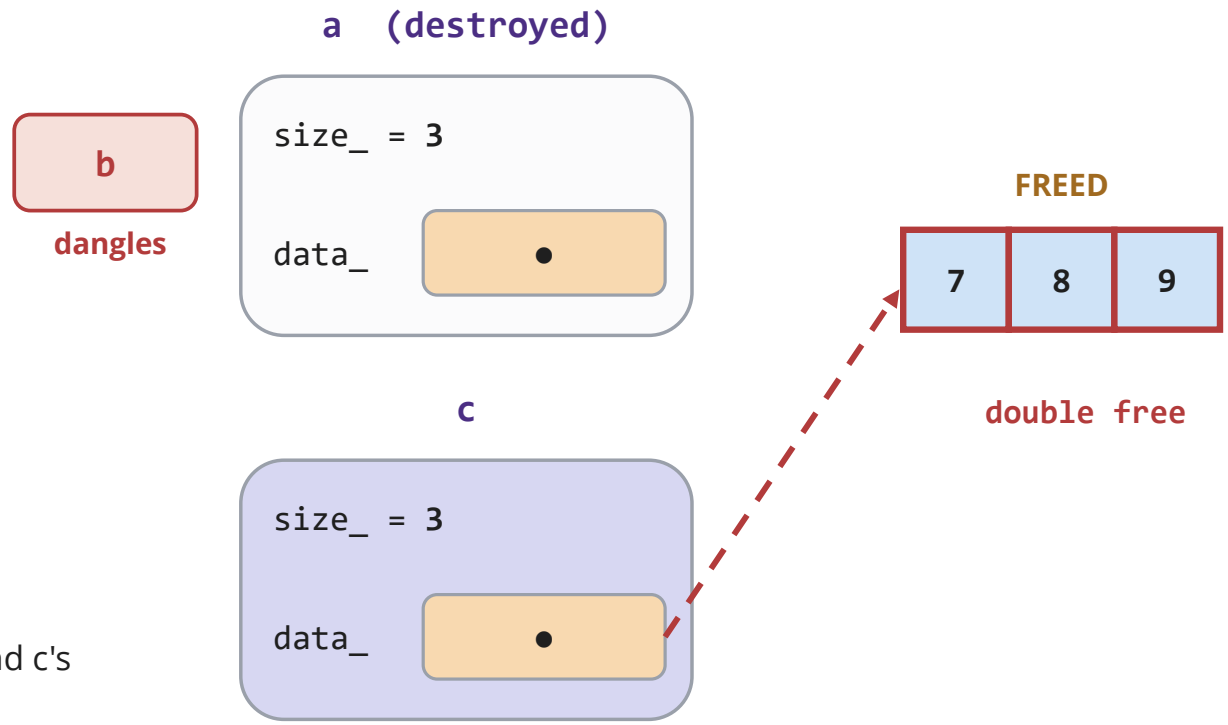
```
1  IntArray a(3);
2  a.Set(0, 7);  a.Set(1, 8);  a.Set(2, 9);
3
4  IntArray& b = a;   // reference: same object
5  IntArray  c = a;   // copy: shallow by default
```



- The default copy copies each member. data_ is a pointer, so it copies the ADDRESS, not the buffer.
- **Now a.data_ == c.data_**: two separate objects, one shared buffer.

The original dies, the copy dangles

```
1  IntArray a(3);
2  IntArray& b = a;    // reference
3  IntArray  c = a;    // shallow copy
4
5  // a is destroyed: ~IntArray() does delete[]
6  //   b       -> dangling reference
7  //   c.data_ -> freed memory, freed AGAIN
```



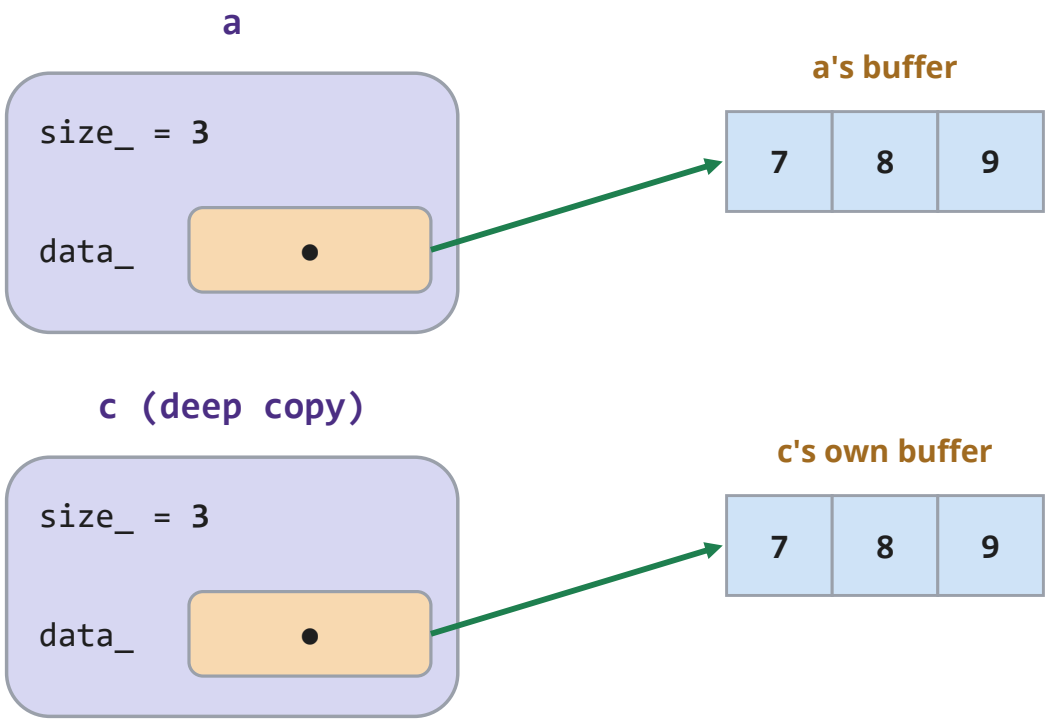
- b was only a name for a, so it now refers to a destroyed object.
- **c is a real object; c.data_ points at freed memory:** use-after-free, and c's destructor frees it a SECOND time.

The fix

The fix: a deep copy

```
1  IntArray::IntArray(const IntArray& other)
2      : size_(other.size_) {
3    data_ = new int[size_];         // our own buffer
4    for (size_t i = 0; i < size_; i++) {
5      data_[i] = other.data_[i];    // copy the values
6    }
7  }
```

❖ **Allocate a fresh buffer,** then copy the elements over. Each object owns its own array.



Reminders

Assessments:

- Exercise 4 will be due on Friday at 11:59pm
  - Includes C and C++
- HW2 is due next Thursday at 11:59pm
- Exercise 5 will be released on Friday.

General:

- Working to find some industry guest lecturers for you!
- Less than 2 weeks till the midterm

Questions?

Thanks for coming - see you next lecture!