CS 112

Introduction to Data Structures

WEEK 04

Vectors: Dynamic Arrays

Eric Araújo

Calvin University · Fall 2026

This Week

1. Why a plain array isn't enough
2. `new` and `delete`
3. The stack and the heap
4. Building a dynamic array
5. Growing when full
6. Destructors: now you need one
7. Copy constructors: shallow vs deep **KEY**

Arrays: The Good and the Bad

Good

- Simple to declare
- Indexing is fast: one multiply and add
- Contiguous, so the cache loves them

Bad

- Size fixed at **compile time**
- Can't grow when you need one more
- No idea how many slots are actually used

> Python's `list` grows on demand. This week we build that, and find out what it costs.

Memory You Ask For

```
int *arr = new int[capacity];  // ask the OS for space

arr[0] = 42;                   // use it like an array

delete [] arr;                 // give it back
arr = nullptr;                 // don't keep the address
```

- `new` returns the **address** of the block
- The size can be a variable: decided while the program runs
- `delete []` for arrays, plain `delete` for single objects

> Every `new` needs exactly one matching `delete`. Forget it and you leak; do it twice and you corrupt the heap.

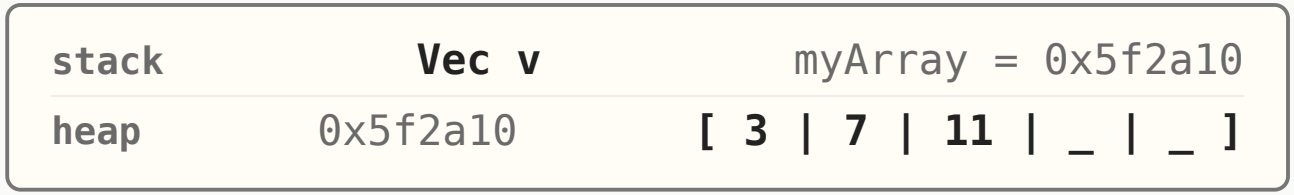
Stack and Heap

Stack

- Locals and parameters
- Freed automatically at end of scope
- Size known at compile time
- Small and fast

Heap

- Whatever `new` hands you
- Freed only when *you* say so
- Size decided at runtime
- Large, slightly slower



> The object lives on the stack; the elements live on the heap. That split is the whole design.

A Dynamic Array

```
class Vec {
 public:
  Vec();
  Vec(const Vec &original);      // copy constructor
  ~Vec();                        // destructor
  unsigned getSize() const;
  void     append(const Item &it);
  Item&    operator[](unsigned i);
 private:
  Item     *myArray;             // heap block
  unsigned  mySize;              // slots in use
  unsigned  myCapacity;          // slots available
};
```

size is how many items you have. **capacity** is how many fit before it must grow. Keeping them apart is what makes appending cheap most of the time.

Appending When There's Room

```
void Vec::append(const Item &it) {
    myArray[mySize] = it;
    ++mySize;
}
```

before	[3 | 7 | _ | _]	size 2, cap 4
after	[3 | 7 | 11 | _]	size 3, cap 4

> Two operations, no matter how big the array is. **Constant time.**

Appending When It's Full

```
void Vec::append(const Item &it) {
  if (mySize == myCapacity) {
    unsigned cap = (myCapacity == 0) ? 1 : myCapacity*2;
    Item *bigger = new Item[cap];
    for (unsigned i = 0; i < mySize; ++i) {
      bigger[i] = myArray[i];      // copy everything
    }
    delete [] myArray;             // release the old
    myArray = bigger;
    myCapacity = cap;
  }
  myArray[mySize] = it;
  ++mySize;
}
```

Now it copies all *n* items: **linear time**. Doubling means this happens rarely, more on that next week.

TALK TO YOUR NEIGHBOR · TTYN

Why double the capacity instead of adding one slot?

- **A.** Doubling uses less memory
- **B.** Adding one would make every append copy the whole array
- **C.** The heap only hands out sizes that are powers of two
- **D.** It doesn't matter, both are equally fast

Now You Need a Destructor

```
Vec::~Vec() {
    delete [] myArray;
    myArray = nullptr;
}
```

C++ writes a destructor for you, but it only destroys the *members*. It destroys the pointer, not what the pointer points at.

> **The rule:** if your class calls `new`, your class must define a destructor. Otherwise every `Vec` that dies leaves its heap block behind.

TALK TO YOUR NEIGHBOR · TTYN

When does a destructor run?

A. Only when you write `delete`

B. When a local object reaches the end of its scope, or is deleted

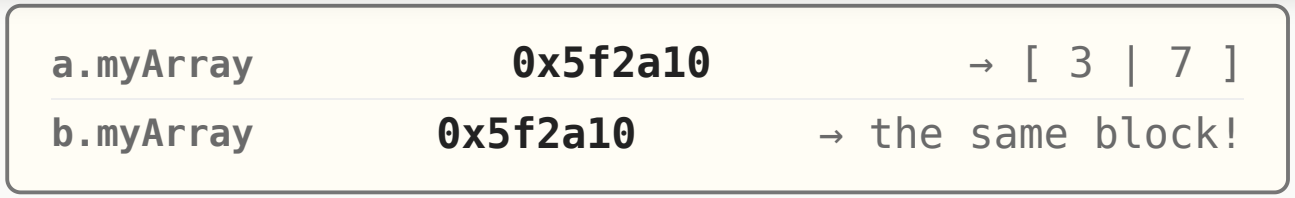
C. At the end of `main()`, for every object

D. When the garbage collector decides

The Copy That Bites

```
Vec a;
a.append(3);
a.append(7);

Vec b = a;        // what exactly got copied?
```



The default copy duplicates the **pointer**, not the data. Two objects now own one block, a **shallow copy**.

TALK TO YOUR NEIGHBOR · TTYN

What goes wrong with that shallow copy?

A. Nothing: sharing the block saves memory

B. Changing `b[0]` also changes `a[0]`

C. Both destructors delete the same block, undefined behaviour

D. Both B and C

Deep Copy: the Copy Constructor

```
Vec::Vec(const Vec &orig) {
  mySize     = orig.mySize;
  myCapacity = orig.myCapacity;
  myArray    = new Item[myCapacity];   // our OWN block
  for (unsigned i = 0; i < mySize; ++i) {
    myArray[i] = orig.myArray[i];      // copy the values
  }
}
```

> Same shape as the growth code: allocate, copy, own. Now `a` and `b` are independent, and each destructor frees its own block.

Three Times It Copies

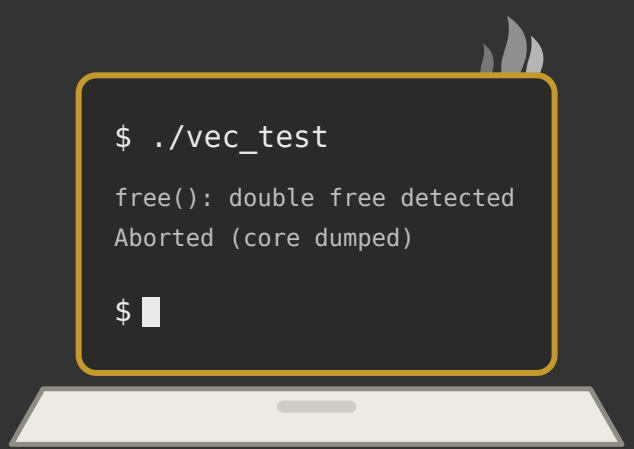
```
Vec b = a;              // 1. explicit copy

void show(Vec v);       // 2. pass-by-value
show(a);

Vec makeVec() {         // 3. return by value
    Vec local;
    return local;
}
```

> Case 2 is why week 1 said *pass big objects by const reference*. Every by-value call to `show()` copies the whole heap block.

Demo



Watching a shallow copy destroy itself.

(two objects, one array, zero survivors)

Returning a Reference: `operator[]`

```
Item& Vec::operator[](unsigned i) {
    return myArray[i];          // a reference, not a copy
}

v[2] = 99;       // works because v[2] IS the slot
```

> Return `Item` and you get a copy, assigning to it would change a temporary and vanish. Returning `Item&` makes the call an *alias* for the element.

The `this` Pointer

Python

```python
def set_first(self, v):
    self.my_first = v
```

C++

```cpp
void setFirst(Item v) {
  this->myFirst = v;
}
```

> C++ passes the object silently as `this`: a pointer to the object the method was called on. You rarely write it, but it's there, and `this->x` is just `(*this).x`.

Week 04 Recap

- `new` gets heap memory at runtime; every `new` needs one `delete`
- **size** vs **capacity** is what makes append cheap most of the time
- Growing means allocate, copy, delete, repoint: linear, so we double
- If your class calls `new`, it needs a **destructor**
- The default copy is **shallow**: aliasing plus a double delete
- Write a copy constructor that allocates its own block

Next: a05 →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**