CS 112

Introduction to Data Structures

WEEK 05

Generic Containers

Eric Araújo

Calvin University · Fall 2026

This Week

1. The problem with `typedef`
2. Class templates
3. Turning a class into a template
4. STL `vector`
5. Iterators
6. How long do operations take?
7. Amortized constant time KEY

One Class, One Type

```
typedef double Item;       // in Vec.h

Vec scores;                // holds doubles ✓
Vec names;                 // also doubles ✗
```

Change the `typedef` and *every* `Vec` in the program changes with it. Want a `Vec` of doubles and a `Vec` of strings in one program? You'd need two copies of the class with different names.

Python never had this problem, a list holds anything. C++ needs types at compile time, so it needs another way.

A Class That Takes a Type

```
template <typename Item>
class Vec {
 public:
  void append(const Item &it);
 private:
  Item     *myArray;
  unsigned  mySize;
};

Vec<double>  scores;      // a Vec of doubles
Vec<string>  names;       // a Vec of strings
```

> You instantiate a *class* to get an object. You instantiate a *class template* to get a class. `Vec<double>` and `Vec<string>` are two different, separately compiled classes.

Turning a Class Into a Template

- Build and debug it first with `typedef double Item;`: get it working
- Replace the typedef with `template <typename Item>` above the class
- In the `.cpp`, put that same line above **every** method
- Change each `Vec::` to `Vec<Item>::`
- Move the implementation into the header, then `#include` it

```
template <typename Item>
void Vec<Item>::append(const Item &it) { ... }
```

> Debug once as a concrete class, then generalise. Templated compiler errors are famously unfriendly, don't meet them and your own bugs at the same time.

TALK TO YOUR NEIGHBOR · TTYN

Why does a class template's implementation go in the header?

- **A.** Style, the committee preferred it that way
- **B.** The compiler must see the code to generate a class for each type used
- **C.** Headers compile faster than source files
- **D.** So the linker can find it

You Already Have One: `vector`

```
#include <vector>

vector<int> scores;

scores.push_back(90);        // append
scores.push_back(85);

cout << scores.size();       // 2
cout << scores[0];           // 90
```

> The STL is a library of class templates: `vector`, `list`, `stack`, `queue`, `set`, `map`. You've just built a simplified `vector`: from here on you may use the real one.

Iterators

```
vector<string>::iterator it;

for (it = names.begin(); it != names.end(); ++it) {
    cout << *it << endl;      // dereference, like a pointer
}

for (const string &name : names) {    // the modern way
    cout << name << endl;
}
```

- `begin()`: points at the first element
- `end()`: points *just past* the last one
- Every STL container is walked the same way

> An iterator behaves like a pointer on purpose: `*it` to read, `++it` to advance.

Counting the Work: Indexing

```
v[i]    →    *(myArray + i * sizeof(Item))
```

- one multiply, one add, one dereference
- three operations: **whatever *i* is**
- whatever the size of the array is

> Work that doesn't grow with *n* is **constant time**. Indexing an array is the classic example.

Counting the Work: Append

Room left

```
myArray[mySize] = it;
++mySize;
```

2 operations → **constant**

Full

```
allocate 2n
copy n items
delete old
```

≈ 3n + 5 → **linear**

> So which is it? The honest answer is "usually cheap, occasionally expensive", and we need a way to talk about that.

Amortized Constant Time

Start empty and append 16 items, doubling each time we fill up:

Append #	Capacity before	Copies
1	0	0
2	1	1
3	2	2
5	4	4
9	8	8
the other 11	none	0

> 15 copies across 16 appends, under one copy each *on average*. Spread over the whole sequence, appending is **amortized constant time**: rarely expensive, cheap in the long run.

TALK TO YOUR NEIGHBOR · TTYN

What is the time complexity of `append()` on a dynamic array?

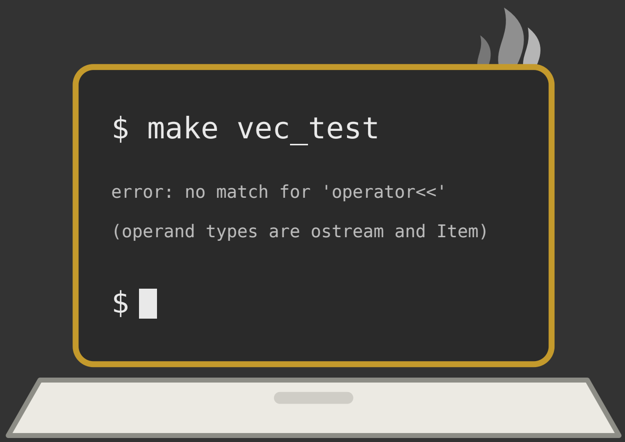
- **A.** Constant; it's just two operations
- **B.** Linear; it might copy everything
- **C.** Amortized constant, usually two operations, occasionally n
- **D.** It depends on the Item type

TALK TO YOUR NEIGHBOR · TTYN

Inserting at the *front* of a dynamic array is…

- **A.** Constant time
- **B.** Amortized constant time
- **C.** Linear time, every element must shift up one
- **D.** Faster than appending

Demo



One template, many types.

(the template compiles fine. Item does not.)

Cost So Far

Operation	Dynamic array	Why
`v[i]`	constant	address arithmetic
`append()`	amortized constant	doubling spreads the copy
`insert()` at front	linear	everything shifts
`remove()` from middle	linear	everything after shifts
copy constructor	linear	copies every element
`getSize()`	constant	we stored the count

> Keep this table. Next week we build a structure with almost the opposite profile, and the comparison is the point.

Week 05 Recap

- A **class template** takes a type the way a function takes an argument
- `Vec<int>` and `Vec<string>` are two separately generated classes
- Template implementations live in the **header**
- Every STL container is walked with `begin()` and `end()`
- Indexing is constant; append is **amortized constant**; inserting at the front is linear

Next: a06 →

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