Vectors

STL Containers

- With the exception of the string class, all the STL containers are templated
  - The types they hold must be specified at compile time
  - You can indicate nearly any type to be used in the container
    - If you define your own type, you might have to do some work for some container ops

STL Containers

Sequential Containers	Associative Containers
vector<T>	map<T, U>
list<T>	unordered_map<T, U>
deque<T>	set<T>
string	

Sequential containers have order to their elements, associative containers do not!

You also must have the `#include <vector>` (replace with appropriate container as needed)

Template type T

- The “standard” name that C++ programmers use for the template type variable is T. Thus you will see in the documentation things like
  - `vector<T>`
  - `list<T>`

Differences

- These containers have different characteristics that make them suitable for various operations:
  - `vector`: fast random access, only fast to add / delete at the end
  - `list`: fast insert / delete at any point. Fast to traverse in either direction
  - `deque` (deck): Double-ended queue. Fast random access, add/delete front or back

Handle their own memory

- Containers also have internal methods that allow them to grow or shrink in size during runtime
  - This is a big deal. You got used to this in Python, but in C++ it is work to dynamically handle memory. STL makes that easier.

vector<T>: definition

- Examples:
  - `vector<double> temperatures;`
  - `vector<int> project_points;`
  - `vector<string> names;`
- Like we did with templated functions, we can have templated classes. The difference is that we **must** say the type.
- After that, the new class instance can **only** work with that type (no mixing!)

Example

- `vector<int> i`
- `vector<string> s`
- `vector<double> d`

The angle bracket describes the type that will be used by the class template when making a variable (instance of that class with the template type

Remember, class template is a pattern

- The class definition has every type represented by a variable (for example, `T`)
- When you make an instance / variable of the class, instantiate the class with the `T` type substituted for the `T` type
- The class instance is made with all the types substituted properly

Size vs Capacity

- Because each container manages their own memory, they can grow under demand
- Methods that reflect this
  - `size`: how much the container presently holds
  - `capacity`: how much it could hold before it has to grow and manage memory

Definition (Constructor)

- Create a vector of size and capacity zero
  - `vector<int> sample;`
- Create a vector of capacity 5, size 5, with each initialized to the default value (0 for int)
  - `vector<int> sample(5);`
- Create a vector of capacity 5, size 5, and each with initial value 1
  - `vector<int> sample(5, 1);`
- Initialize the elements between `{}`
  - `vector<int> sample{1, 2, 3, 4, 5}`

Definition

```
vector<int> sample(5); // filled with default value
```

sample	0	0	0	0	0

```
vector<int> sample(5, 1);
```

sample	1	1	1	1	1

```
vector<int> sample{1, 2, 3, 4, 5};
```

sample	1	2	3	4	5

vector<T> member functions

- `v.capacity()` // v can store before growing
- `v.size()` // v currently contains
- `v.empty()` // true iff size == 0
- `v.reserve(n)` // grow capacity to n
- `v.push_back(value)` // append value to end
- `v.pop_back()` // remove last value of v (no return)

Notes

- `v.size()` is useful because `v.size() - 1` is the index of the last element in `v`
- `v.empty()` is equivalent to `v.size() == 0`
- `v.reserve()` is not used often since `v.push_back(n)` implicitly increases the capacity of `v`. Allocates more memory for future use.

Access front and back

- `v.front()`
  - The element at the front of the vector
  - First element, no change to vector
- `v.back()`
  - The element at the back of the vector
  - Last element, no change to vector

Basic add, push_back

- Like we saw with strings, the method to add something to the end of a vector is `push_back`
- This is the primary way to add to a vector, as they are optimized to add elements at the end

Delete from the end, pop_back

- Access to a vector is from the end, so we have the `pop_back` method
- Does not return the value it removed, just removes it
- If you wanted to know, you need to check `.back()` first!

Operators

- Subscript: `v[i] or v.at(i)`
  - Cannot use subscript to **append**
  - To append, use `v.push_back(item)` so capacity increases
- Assignment: `v1 = v2`
  - Copy each element!
- Equality: `v1 == v2`
- Comparison: `v1 < v2`
  - Lexicographical comparison like string

`[]` or `.at()` does not add elements

- The only way to get elements into a vector is
  - construct it with elements
  - `push_back` elements
- `[]` or `.at` can reference an existing element, change an existing element, **but not add** new elements

for iteration

- Can iterate with a for loop
  - auto is convenient here again. It is the type of each element in the vector

```
for (auto element : vec)
    cout << element << ", ";
```

- Trailing comma is irritating, how to fix?

Other operators

- `vector <int> v = {1, 2, 3};`
  - `v.front()`, first value, here is 1
  - `v.back()`, last value, here is 3
  - `v.clear()`, clear elements. Now `v.size() == 0`
  - `v.assign(3, 10)` puts 3 values of 10 into the vector. Now `v.size() == 3`

Some more

- Swap the contents of two vectors
  - Same size **not** required

```
vector<int> v1(3, 100);
vector<int> v2(2, 10);
v1.swap(v2);
for (auto a : v2)
    cout << a << endl; // 3 100s
```

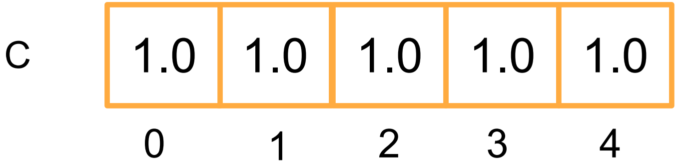
Can’t just print a vector

- Like most containers, you cannot just print a vector.
- You have to iterate through each element and print it out ☹️
- More on this in a minute

2-D Vectors

Review vector<T> constructor

```
vector<double> A;
const int MAX = 5;
vector<double> B(MAX);
vector<double> C(MAX, 1.0);
```



2D vector<T> in Two Steps

- Form row

```
const int COLS = 4;
vector<double> initialRow(COLS, 0.0);
```

initialRow	0.0	0.0	0.0	0.0
	0	1	2	3

For vector of rows

```
const int ROWS = 3;
vector<vector<double>> table(ROWS, initialRow);
```

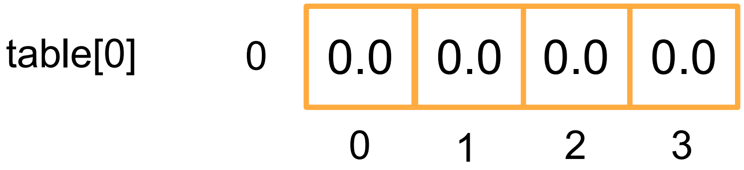
2D vector<T> table

```
vector<double> initialRow(COLS, 0.0);
vector<vector<double>> table(ROWS, initialRow);
```

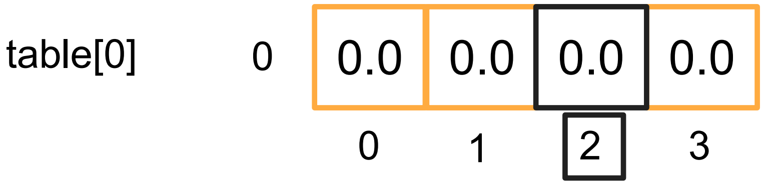
0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
	0	1	2	3

Subscript

- First row: table[0]



- Element: table[0][2]



2D vector<T> one step

- `const int ROWS = 3;`
- `const int COLS = 4;`
- ```
  vector<vector<double>>
          table(ROWS, vector<double>(COLS, 0.0));
  ```

- Note the unnamed row vector (constructor).
  ```

Readable

```
using TableRow = vector<double>;
using Table = vector<TableRow>;

Table aTable; // empty table
const int ROWS = 3, COLS = 4;
Table theTable(ROWS, TableRow(COLS, 0.0));
```

Operations

- `.size()`
  - Rows in Table: `theTable.size();`
  - Columns in row r:
    - `theTable[r].size()`
    - Allows for variable-sized rows

push_back()

- Add a row

```
theTable.push_back(TableRow(COLS, 0.0));
```

- Add a column

```
for (int row = 0; row < theTable.size(); row++)
   theTable[row].push_back(0.0);
```

Example: Output

```
void print_table(const Table & aTable) {
    for (int row = 0; row < aTable.size(); row++)
        for (int col = 0; col < aTable[row].size(); col++)
            cout << aTable[row][col];
        cout << endl;
}
```

Pass as a parameter

- Pass the type (probably as a reference)

```
int func(vector<vector<long>> & v) {
    … do some stuff
}
```

Range Based Loops

Make copies of each element

```
vector<int> my_ints {1, 2, 3};
for (auto x : my_ints) {
    x += 2;
} // my_ints is still {1, 2, 3};
```

Make references of each element

```
vector<int> my_ints {1, 2, 3};
for (auto& x : my_ints) {
    x += 2;
} // my_ints is now {3, 4, 5};
```

Make const references of each element

```
vector<int> my_ints {1, 2, 3};
for (const auto& x : my_ints) {
    // x += 2; generates an error
    cout << c;
}
```