Vectors, iterators

roger murdock: We have clearance Clarence.
captain oveur: roger Roger what's our vector Victor

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 container ops
## STL Containers

<table>
<thead>
<tr>
<th>Sequential containers</th>
<th>Associative Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector&lt;T&gt;</td>
<td>map&lt;T,U&gt;</td>
</tr>
<tr>
<td>list&lt;T&gt;</td>
<td>unordered_map&lt;T,U&gt;</td>
</tr>
<tr>
<td>deque&lt;T&gt;</td>
<td>set&lt;T&gt;</td>
</tr>
<tr>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

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

### 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 the below:

```
vector<T> and list<T>
```
These containers have different characteristics that make them suitable for various operations:

**vector**: fast random access, only fast to add/delete at the vector 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

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 some work to dynamically handle memory. STL makes that easy, but we will see ourselves later.
Concentrate on the vector

Bjarne Stroustrup, inventor of C++:

"Fundamentally, if you understand vector, you understand C++"

Example:

```cpp
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 a variable/instance 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.

Vectors

Definition (Constructor)

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

Vector<int> sample(5,1);

Vector<int> 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 of vector
- `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 in strings, the method to add something to the end of the 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 available the `pop_back` method.

Does not return the value it removed, just removes it. If you wanted to know, you needed to check `.back()` first!
Operators

- Subscript: v[i] or v.at(i)
  - cannot use subscript to *append*
  - to append use v.push_back(i)
    so capacity increases
- Assignment: v1 = v2
  - copy each element!
- Equality: v1 == v2
- Comparison: v1 < v2
  - lexicographical comparison like string

This is obvious but worth saying.
The only way to get elements into a vector is:
- init it with elements
- push_back elements

[] or .at() does not add elements

[] or .at can reference an existing element, change an existing element, *but not add* new elements
for iteration

Can iterate with a for iterator

- auto is convenient here again. It is the type of each element in the vector

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

Trailing comma is irritating, how to fix?

Vectors

Other operators

```cpp
vector<int>v = {1, 2, 3}
- v.front(), first value, here 1
- v.back(), last value, here 3
- v.clear(), clear elements. Now v.size() == 0
- v.assign(3,10) put 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
2d structures

2D vectors
Review vector<T> constructors

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

```
   C
   1.0 1.0 1.0 1.0 1.0
  0   1   2   3   4
```

2D vector<T> in Two Steps

- Form Row
  ```cpp
  const int COLS = 4;
vctor<double> initialRow(COLS, 0.0);
  ```
  ```
  initialRow
  0.0 0.0 0.0 0.0
  0   1   2   3
  ```

- Form Vector of Rows
  ```cpp
  const int ROWS = 3;
vctor<vector<double>> table(ROWS, initialRow);
  ```
  ```
  2d
  ```
### 2-D vector<T> Table

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

#### Subscript

- **First Row:** `table[0]`

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
```

- **Element:** `table[0][2]`

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td><strong>-</strong></td>
<td>0.0</td>
</tr>
</tbody>
</table>
```
2-D 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

```cpp
void Print (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)
```cpp
int func(vector<vector long> &v){
    ...do some stuff
}
```