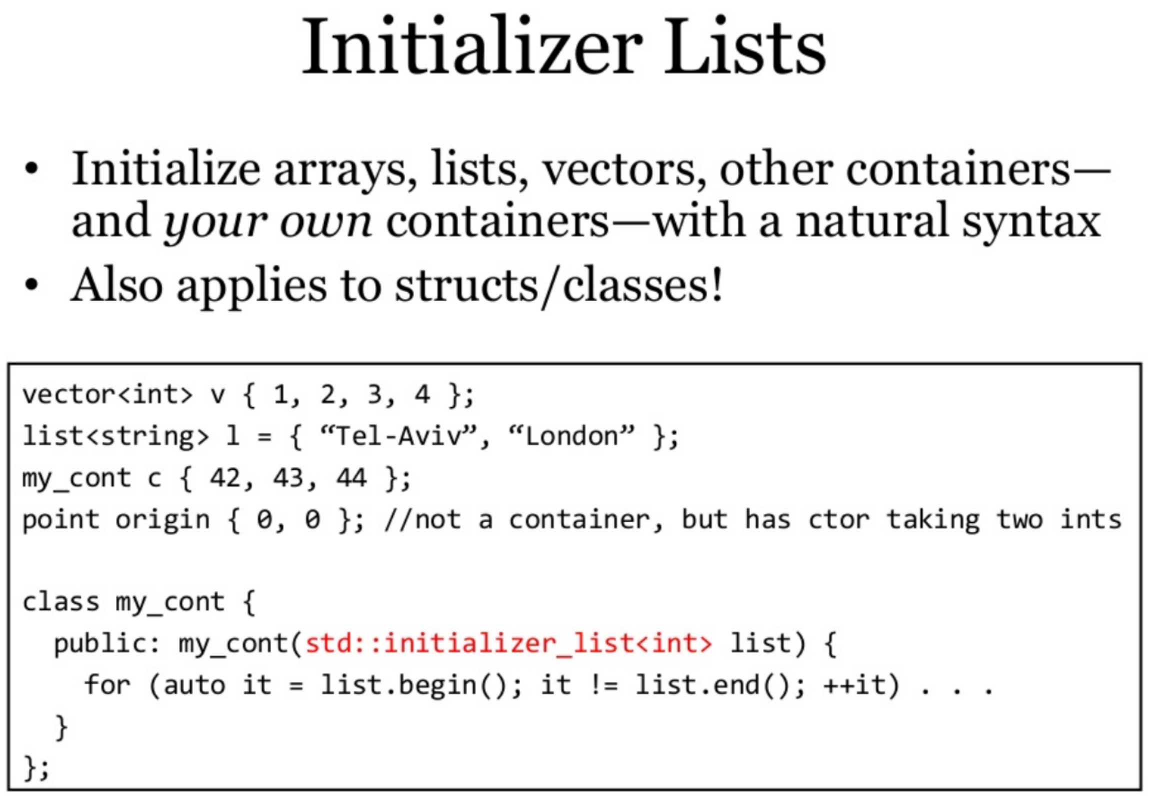
<u>INITIALIZER LIST</u>



Initializer Lists

- Initialize arrays, lists, vectors, other containers—and *your own* containers—with a natural syntax
- Also applies to structs/classes!

```
vector<int> v { 1, 2, 3, 4 };
list<string> l = { “Tel-Aviv”, “London” };
my_cont c { 42, 43, 44 };
point origin { 0, 0 }; //not a container, but has ctor taking two ints

class my_cont {
  public: my_cont(std::initializer_list<int> list) {
    for (auto it = list.begin(); it != list.end(); ++it) . . .
  }
};
```

<u>Before C++11 it was easy to initialize an array with with default elements like,</u>

// Initializing array with default values int arr[]= {1,2,3,4,5};

```
// Initializing array with default values
int arr[]= {1,2,3,4,5};
```

But there was no way no to initialize other containers like vector, list and map etc.

It is also used to initialized the members of the class in constructor.

Why do we need to use it?

Basically copying and pasting from Bjarne Stroustrup's *"The C++ Programming Language 4th Edition"*:

List initialization does not allow narrowing (§iso.8.5.4). That is:

- An integer cannot be converted to another integer that cannot hold its value. For example, char to int is allowed, but not int to char.
- A floating-point value cannot be converted to another floating-point type that cannot hold its value. For example, float to double is allowed, but not double to float.
- A floating-point value cannot be converted to an integer type.
- An integer value cannot be converted to a floating-point type.

Example:

```
void fun(double val, int val2) {

    int x2 = val; // if val==7.9, x2 becomes 7 (bad)

    char c2 = val2; // if val2==1025, c2 becomes 1 (bad)

    int x3 {val}; // error: possible truncation (good)

    char c3 {val2}; // error: possible narrowing (good)

    char c4 {24}; // OK: 24 can be represented exactly as a char (good)

    char c5 {264}; // error (assuming 8-bit chars): 264 cannot be
                   // represented as a char (good)

    int x4 {2.0}; // error: no double to int value conversion (good)

}
```

The *only* situation where = is preferred over {} is when using `auto` keyword to get the type determined by the initializer.

Example:

```
auto z1 {99}; // z1 is an initializer_list<int>
auto z2 = 99; // z2 is an int
```

Conclusion

Prefer {} initialization over alternatives unless you have a strong reason not to.