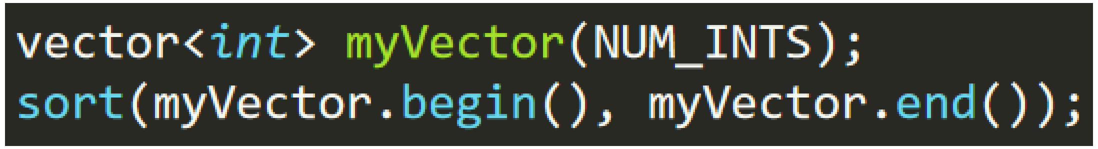
CPP Programming: Session 8

Anupama Chandrasekhar (September 27 2018)

STL : The Standard Template Library

- The STL was invented by Alex Stepanov and it offers an efficient way to store, access and view data and is designed to scale from few to millions of elements.



```
vector<int> myVector(NUM_INTS);
sort(myVector.begin(), myVector.end());
```

- If we did not have an easy way to represent a list of objects, like the STL container **`vector`**, we would have to deal with memory allocation and freeing that is needed while allocating dynamic memory … not fun!

Overview of STL

- STL is logically divided into six pieces:
  - **Containers**: E.g.: list of elements can be stored in an vector, associative collection of key/value pairs in a map etc.
  - **Iterators:** To traverse the containers
  - **Algorithms**: STL algorithms operator over ranges of data provided by the iterators, there are algorithms for searching, sorting, reordering, permuting etc.
  - **Adapters:** ADTs built on containers
  - **Functors :** Customizations
  - **Allocators:** Customizations

Why STL? Motivational Example

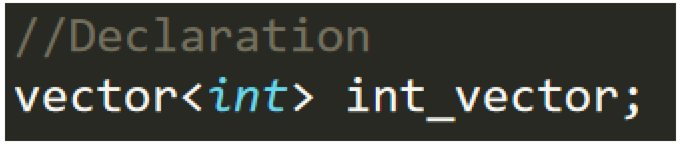
- When you know exactly how much data your program needs, we can use static allocations. But often that is not the case.
- Take a simple example:
  - Write a program that reads three integers from a user and prints them out in sorted order.
  - Problems we need to solve:
    - How would we store the numbers that the use enters?
    - How would we sort them?
    - See 1_sort3numbers.cpp
  - Another problem is scalability .. What if we had to sort four numbers.

STL : **vector**

- Fortunately, the STL provides us a with a versatile tool called the ***vector*** that allows us to store sequences of elements using a single variable.
- **vector** can be used to store a sequence of elements of the same type.

Value	137	42	2718	3141	410
Index	0	1	2	3	4

- That is you can have a **vector** of ints, **vector** of **string**s or a **vector** of your own type.



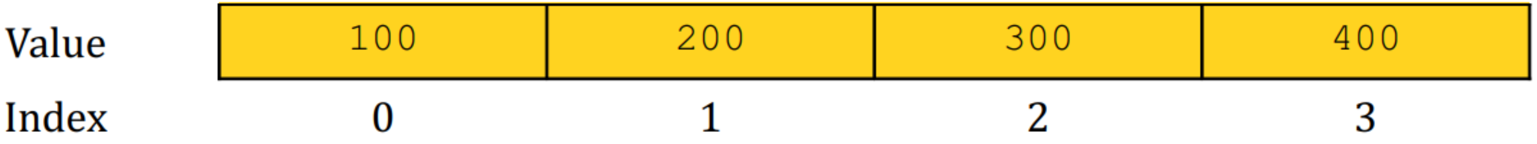
```
//Declaration
vector<int> int_vector;
```

STL: **`vector`**

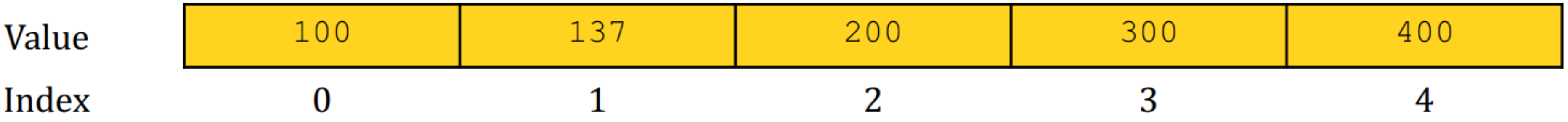
- Using a **`vector`**. See 2_vector.cpp
- Initially empty, you can use **`push_back`** to put elements in.
- For sorting, let's implement a simple sorting algorithm
  - Find the smallest element in a list
  - Put that element in the front of the list
  - Repeat until all elements are in place
- Things to note:
  - Always define the template parameter while declaring a **`vector`**: **`vector<int>`**, **`vector<string>`** etc.
  - Prefer to pass by reference for efficiency
  - Use **`size_t`** for indexing as an index is always positive and **`size_t`** can hold the largest size.

STL: `vector`

- Another way to do selection sort



Value	100	200	300	400
Index	0	1	2	3

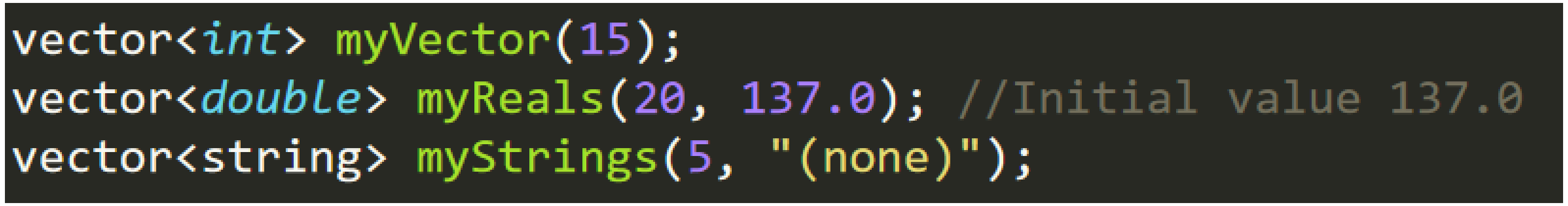


Value	100	137	200	300	400
Index	0	1	2	3	4

- Every time we get a new element insert it in the sorted position and maintain a sorted list

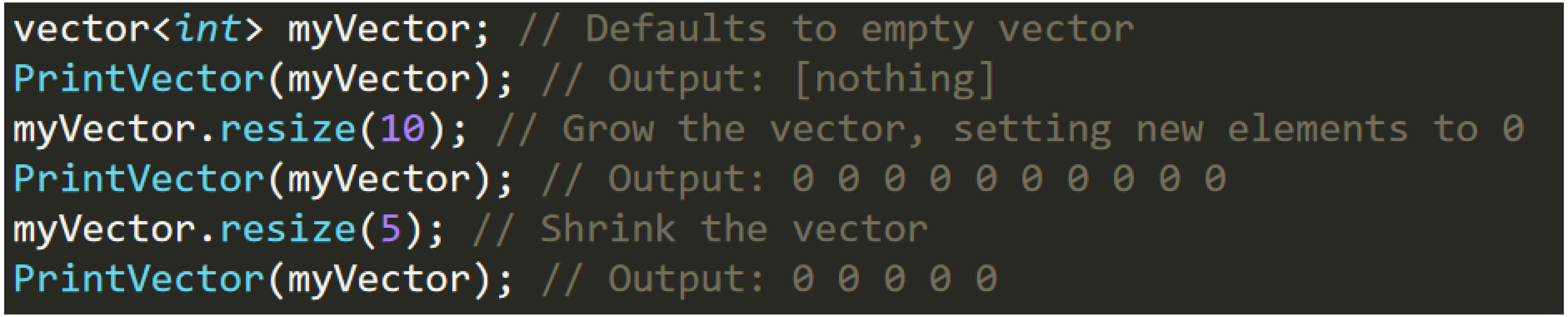
STL: **vector**

- Constructing/Initializing a **vector**:



```
vector<int> myVector(15);
vector<double> myReals(20, 137.0); //Initial value 137.0
vector<string> myStrings(5, "(none)");
```

- Resizing a **vector**:



```
vector<int> myVector; // Defaults to empty vector
PrintVector(myVector); // Output: [nothing]
myVector.resize(10); // Grow the vector, setting new elements to 0
PrintVector(myVector); // Output: 0 0 0 0 0 0 0 0 0 0
myVector.resize(5); // Shrink the vector
PrintVector(myVector); // Output: 0 0 0 0 0
```

STL: **`vector`** (Summary)

API	Description
Constructor: `vector<T> ()`	`vector<int> myVector;` Constructs an empty vector.
Constructor: `vector<T> (size_type size)`	`vector<int> myVector(10);` Constructs a vector of the specified size where all elements use their default values (for integral types, this is zero).
Constructor: `vector<T> (size_type size, const T& default)`	`vector<string> myVector(5, "blank");` Constructs a vector of the specified size where each element is equal to the specified default value.
`size_type size() const;`	`for(int i = 0; i < myVector.size(); ++i) { ... }` Returns the number of elements in the vector.
`bool empty() const;`	`while(!myVector.empty()) { ... }` Returns whether the vector is empty.
`void clear();`	`myVector.clear();` Erases all the elements in the vector and sets the size to zero.

STL : **`vector`** (Summary)

API	Description

`T& operator [] (size_type position);`
`const T& operator [] (size_type position) const;`
`T& at(size_type position);`
`const T& at(size_type position) const;`
	`myVector[0] = 100;`
	`int x = myVector[0];`
	`myVector.at(0) = 100;`
	`int x = myVector.at(0);`
	Returns a reference to the element at the specified position. The bracket notation [] does not do any bounds checking and has undefined behavior past the end of the data. The at member function will throw an exception if you try to access data beyond the end.

`void resize(size_type newSize);`
`void resize(size_type newSize, T fill);`
	`myVector.resize(10);`
	`myVector.resize(10, "default");`
	Resizes the vector so that it's guaranteed to be the specified size. In the second version, the `vector` elements are initialized to the value specified by the second parameter. Elements are added to and removed from the end of the vector, so you can't use `resize` to add elements to or remove elements from the start of the vector.

`void push_back();`	`myVector.push_back(100);` Appends an element to the `vector`.

`T& back();`
`const T& back() const;`
	`myVector.back() = 5;`
	`int lastElem = myVector.back();`
	Returns a reference to the last element in the `vector`.

STL : `vector` (Summary)

`T& front();`
`const T& front() const;`
	`myVector.front() = 0;`
	`int firstElem = myVector.front();`
	Returns a reference to the first element in the vector.

`void pop_back();`	`myVector.pop_back();` Removes the last element from the vector.

`iterator begin();`
`const_iterator begin() const;`
	`vector<int>::iterator itr = myVector.begin();`
	Returns an iterator that points to the first element in the vector.

`iterator end();`
`const_iterator end() const;`
	`while(itr != myVector.end());`
	Returns an iterator to the element *after* the last. The iterator returned by end does not point to an element in the vector.

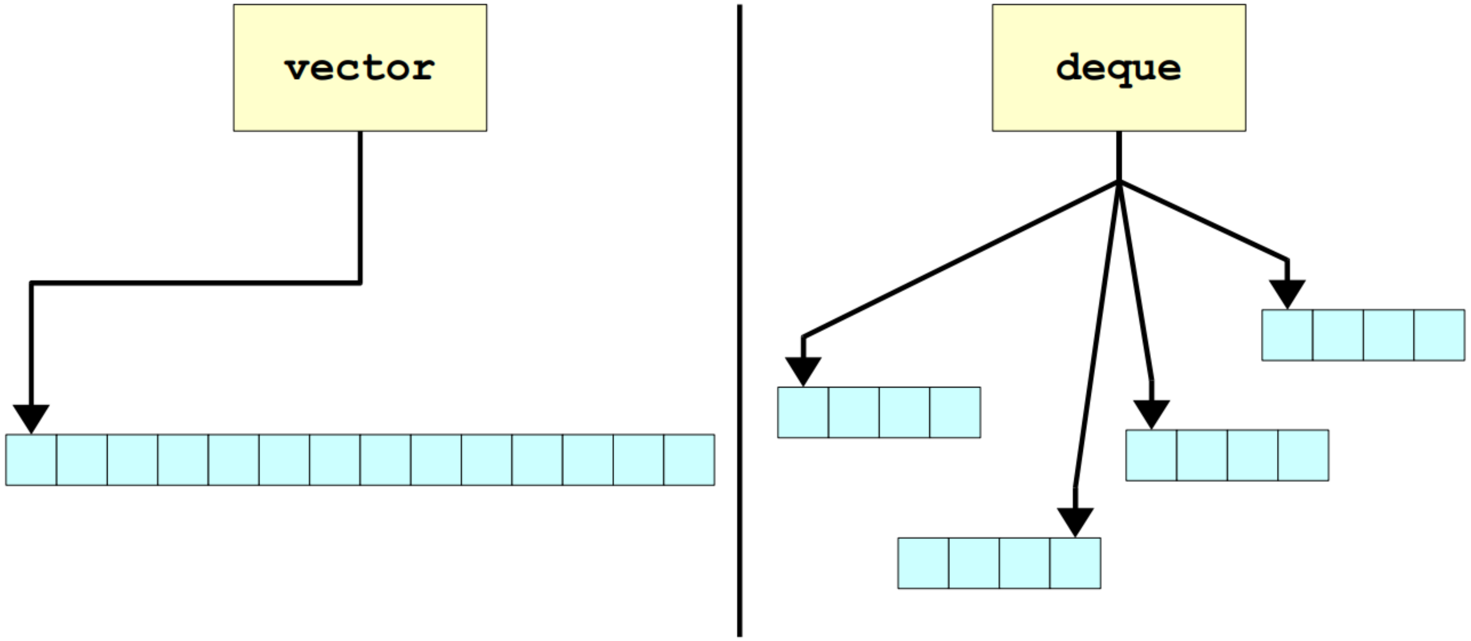
`iterator insert(iterator position,`
`const T& value);`
`void insert(iterator start,`
`size_type numCopies,`
`const T& value);`
	`myVector.insert(myVector.begin() + 4, "Hello");`
	`myVector.insert(myVector.begin(), 2, "Yo!");`
	The first version inserts the specified value into the vector, and the second inserts `numCopies` copies of the value into the vector. Both calls invalidate all outstanding iterators for the vector.

STL : `vector` (Summary)

`iterator erase(iterator position);`
`iterator erase(iterator start,`
`iterator end);`
	`myVector.erase(myVector.begin());`
	`myVector.erase(startItr, endItr);`
	The first version erases the element at the position pointed to by position. The second version erases all elements in the range [`startItr`, `endItr`). Note that this does **not** erase the element pointed to by `endItr`. All iterators after the remove point are invalidated

More STL containers

- **deque** :Supports all vector functionality and **push_front()** and **pop_front()**



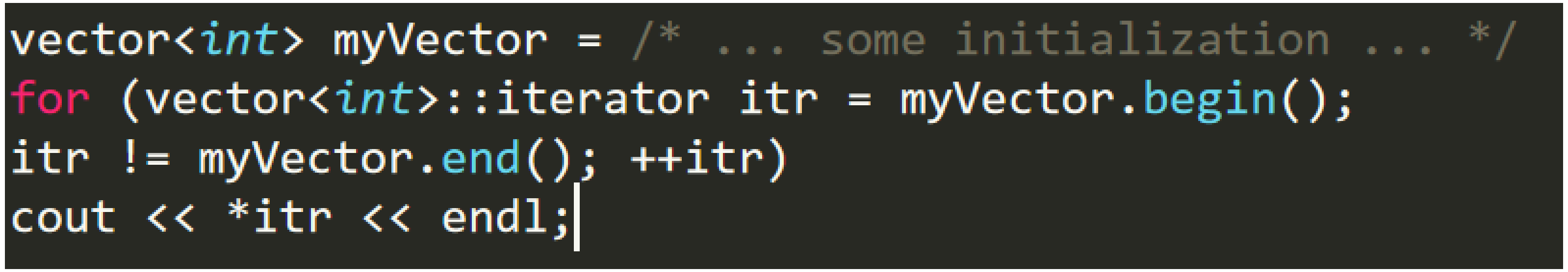
- **stack and queue**

STL: Associative Containers

- **`std::set`**
  - An unordered collection of unique elements that does not permit duplicates. Unlike vector and deque, sets can only store objects for with “<“ operator is defined.
  - Internally, a set is layered on top of a balanced binary search tree. But since a binary search tree orders elements in a certain order, “<“ needs to be defined.

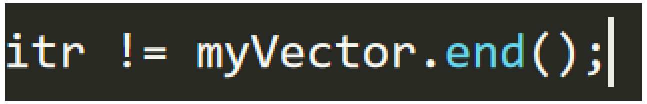
Traversing Containers with Iterators

- What exactly is an iterator? At a high level, it is like a cursor in the text editor.



```
vector<int> myVector = /* ... some initialization ... */
for (vector<int>::iterator itr = myVector.begin();
itr != myVector.end(); ++itr)
cout << *itr << endl;
```

- Every STL container class exports a member function **begin()** which yields an iterator pointing to the first element of that container.
- The strange-looking entity ***itr** is known as an *iterator dereference* and means “the element being iterated over by **itr**.”
- When applied to iterators, the **++** operator means “advance the iterator one step forward.”
- To detect when an iterator has visited all of the elements, we loop on the condition that



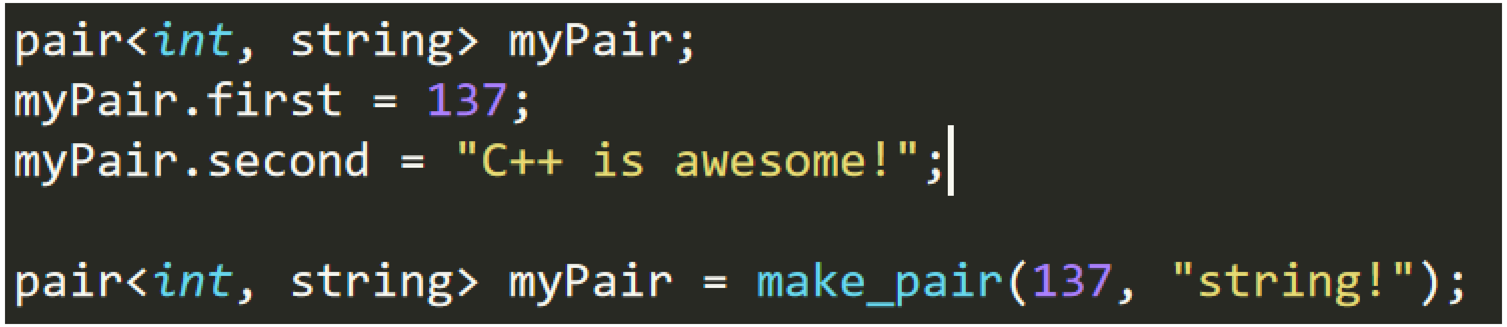
```
itr != myVector.end();
```

Traversing Containers with Iterators

- Notice that the **`begin()`** iterator points to the first element of the vector, while the **`end()`** iterator points to the slot one position past the end of the container.

std::pair<type1, type2>

- Make a **pair**:

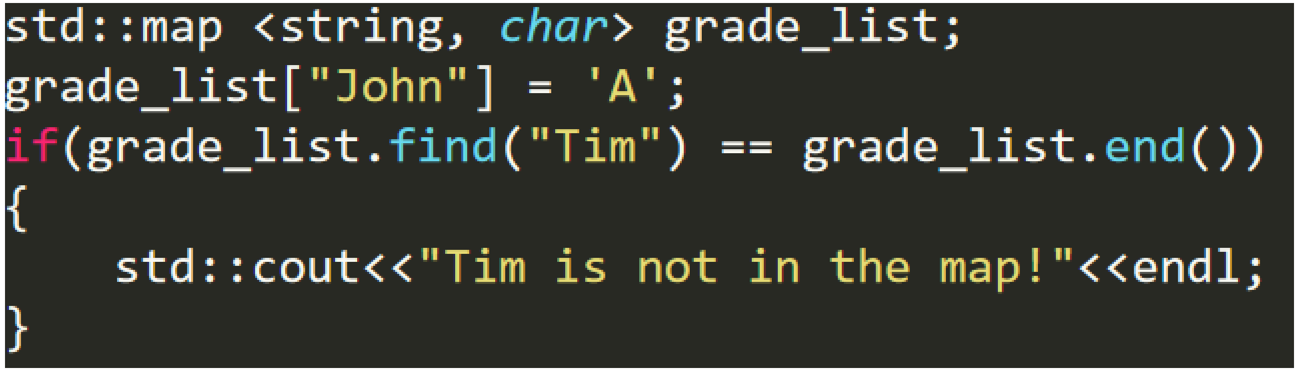


```
pair<int, string> myPair;
myPair.first = 137;
myPair.second = "C++ is awesome!";

pair<int, string> myPair = make_pair(137, "string!");
```

std::map< KeyType, ValueType **>**

- **std:map**
  - Maps are associative containers that allow you to store data indexed by keys.
  - Maps can be accessed using iterators that are essentially pointers to templated objects of base type pair, which has two members, first and second. First corresponds to the key, second to the value associated with the key.
  - Maps are fast, guaranteeing O(log(n)) insertion and lookup time. (Think BST)
  - To check if something is not in the **map**:



```
std::map <string, char> grade_list;
grade_list["John"] = 'A';
if(grade_list.find("Tim") == grade_list.end())
{
    std::cout<<"Tim is not in the map!"<<endl;
}
```

`std::map< KeyType, ValueType >`

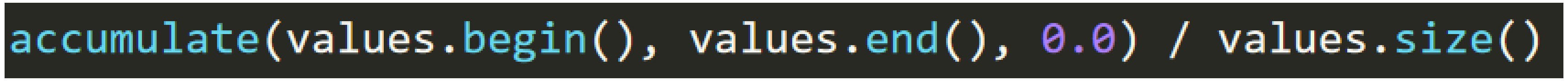
- Because the square brackets both query and create key/value pairs, you should use care when looking values up with square brackets.
- If you want to look up a key/value pair without accidentally adding a new key/value pair to the **`map`**, you can use the **`map`**'s find member function. find takes in a key, then returns an iterator that points to the key/value pair that has the specified key. If the key does not exist, find returns the **`map`**'s end() iterator.
- **`map`** iterators are slightly more complicated because they dereference to a key/value pair. In particular, if you have a **`map<KeyType, ValueType>`**, then the iterator will dereference to a value of type **`pair<const KeyType, ValueType>`**

`std::unordered_map<keyType, valueType>`

- Unordered maps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, and which allows for fast retrieval of individual elements based on their keys. Hash Maps!!!

STL Algorithms

- Let’s say you want to find the average of the values in a **`vector`**, you can loop over, perform the summation and divide by number of elements or simply do:

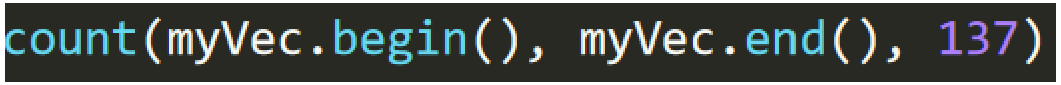


```
accumulate(values.begin(), values.end(), 0.0) / values.size()
```

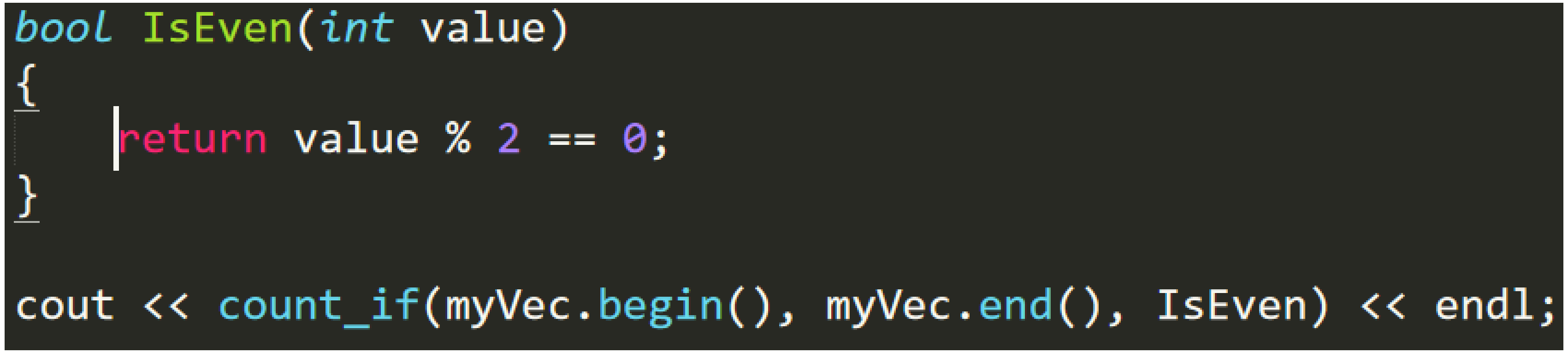
- The power comes in its ability to handle a wide range of STL containers like, **`vector, deque, set`** etc.
- Reasons to do this: simplicity, clarity and correctness

STL Algorithms

- The suffix _if on an algorithm (**`replace_if, count_if`**, etc.) means the algorithm will perform a task on elements only if they meet a certain criterion.



```
count(myVec.begin(), myVec.end(), 137)
```



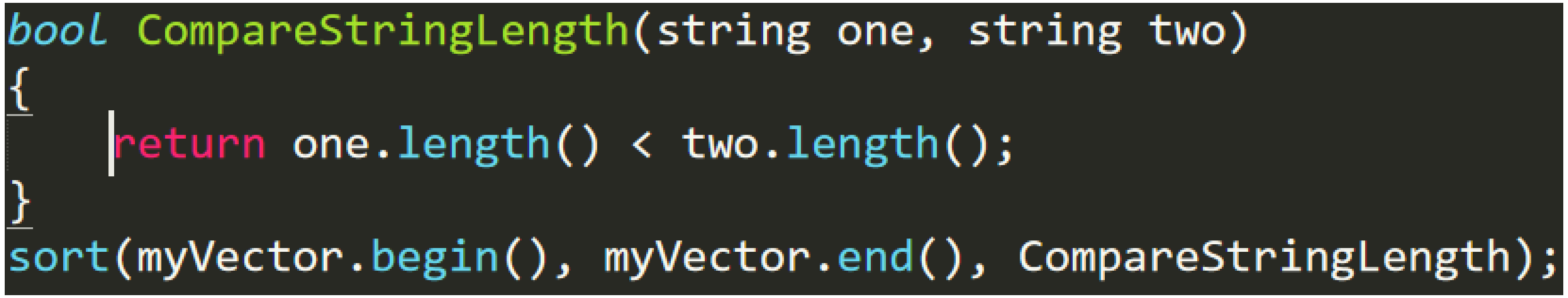
```
bool IsEven(int value)
{
    return value % 2 == 0;
}

cout << count_if(myVec.begin(), myVec.end(), IsEven) << endl;
```

- Copy and fill algorithms can be use to copy a range/initialize (clear) a range.

STL Algorithms

- Reordering Algorithms:



```
bool CompareStringLength(string one, string two)
{
    return one.length() < two.length();
}
sort(myVector.begin(), myVector.end(), CompareStringLength);
```

- See https://en.cppreference.com/w/cpp/algorithm for the full list

References

- The C++ Programming Language, 4th Edition, Bjarne Stroustrup
- Thinking in C++, Bruce Eckel
- Effective Modern C++ , Scott Meyers
- Quick Guide to Modern C++: A Tour of C++, 2nd edition

Course Survey

- When you have 2 minutes, please fill out:
- https://www.surveymonkey.com/r/XKSGTM3