COP3503

Iterators

Iteration Through Contiguous Containers

- You know how this goes…

```
for (int i = 0; i < NumberOThings; i++)
{
    things[i].DoStuff();
    cout << things[i] << endl;
    // Etc
}
```

- What about non-contiguous containers?

Non-Contiguous Data

```
map<int, vector<string>> data;
```

```
data[0]: key might not exist!
```

```
unordered_map<string, int> data;
```

```
data[0]: error, 0 isn't a string!
```

```
LinkedList<Foo> data;
```

```
data[0]: LinkedList might not have operator[] overloaded!
```

```
for (int i = 0; i < data.size(); i++)
{
    cout << data[i];
}
```

This "normal", **index-based** loop won't work in many situations.

Iterators are the solution.

Iterators

A standard interface for iteration

- Iterators provide a **standard interface** for iterating over the elements of a container.

```
// iter == an already created iterator variable (more on this next!)
// someContainer == the container "iter" will work with

for (iter = someContainer.begin(); iter != someContainer.end(); ++iter)
{
    // Use the iterator
    // This is the only unique code when using iterators
    // (Assuming you want to visit each element in order!)
}
```

- The only thing that has to change is how you actually **use** elements of the container (i.e., the code inside the loop).

Iterator Creation Is Easier than it Looks!

```
// Essentially just this:
classToIterateOver<matchTheType(s)>::iterator variableName;

map<int, int> aMap;
map<int, int>::iterator aMapIterator;

map<string, double> anotherMap;
map<string, double>::iterator anotherMapIterator;

unordered_map<string, string> uoMap;
unordered_map<string, string>::iterator uoMapIterator;

// Vectors too! (Though not strictly necessary)
vector<int> numbers;
vector<int>::iterator numbersIterator;

vector<string> words;
vector<int>::iterator wordsIterator;
```

Iterators are implemented on a **per-class basis** for Standard Template Library (STL) classes.

Internally, a map is different than an unordered_map, and different from a vector.

Iterators handle these differences and provide an interface, so you don't have to worry (much) about the details.

Iterators `begin()` at the Beginning

```
// Set the iterator to the first element of some container
vector<int>::iterator iter = someVector.begin();

// Basically the same thing... in this case
unsigned int firstIndex = 0;
```

- `begin()` is an **abstraction**.
- The outside world doesn't need to know or care how or where the first element is stored.
- Trust the function works, and should the class ever change internally, your code won't have to.

end()

- An iterator that is **one past the last** element.
- Take an array with 10 elements □

```
int someArray[10];
```

- Valid indices? 0 thru 9, or 0 thru (sizeOfArray – 1)
- 10 would be **one past the last** element.
- You wouldn't **use** index 10, but… it can be helpful as a boundary value.
- It's like a pointer set to **nullptr**—just **knowing** it's **nullptr** is useful

```
for (unsigned int i = 0; i < 10; i++) // As long as you haven't reached 10…

while (iter != data.end()) // As long as we aren't out of range
```

(AKA while we are **in** range)

Accessing the Iterator's Contents

- Iterators are **like pointers**—but actually class objects.
- You must **dereference** them in order to access whatever they're "pointing" to.
- Dereferencing an iterator returns a **reference to the data element**.

```
vector<int> data;
data.push_back(5);
vector<int>::iterator iter;
for (iter = data.begin(); iter != data.end(); ++iter)
{
    cout << *iter << endl;       // 5
    cout << *iter + 3 << endl;   // 8
    *iter *= 10;     // Change to 50 (*iter returns a reference)

    // You COULD do this, but it's not recommended...
    // 1. (*iter) dereference the iterator (get an int&)
    // 2. ++ increment that integer
    (*iter)++;  // Change 50 to 51 in a not-so-clear way
}
```

Writing unclear code like this is generally something to avoid if possible (and sometimes we can't!)

What About Non-Primitives (i.e., Classes)?

- The same concept applies—dereference the iterator, the use the resulting object.

```
vector<Hero> data;
data.push_back(Hero("Batman", 10, 200));
vector<Hero>::iterator iter;

for (iter = data.begin(); iter != data.end(); ++iter) // Same interface
{
    // *iter is a Hero object
    // Use it however you would use a Hero
    // (*iter). is a bit clumsy though...
    string name = (*iter).GetName();

    /*=== A better approach ===*/
    // Use the indirect-membership operator, just like a pointer
    name = iter->GetName();
}
```

What About Iterators for Non-Vectors?

```
unordered_map<string, int> data;

// class<template type(s)>::iterator
unordered_map<string, int>::iterator iter;

for (iter = data.begin(); iter != data.end(); ++iter) // Seems familiar...
{
    // What does *iter give you here? It depends on the class

    // map (and unordered_map) use pairs, with keys and values
    // *iter is a pair<key, value>
    // iter->first == key
    // iter->second == value
    cout << "Key: " << iter->first << endl;
    cout << "Value: " << iter->second << endl;
}
```

This is just how you retrieve the key and value. What you choose to **do** with those is up to you!

Iterator Arithmetic and Other Operators

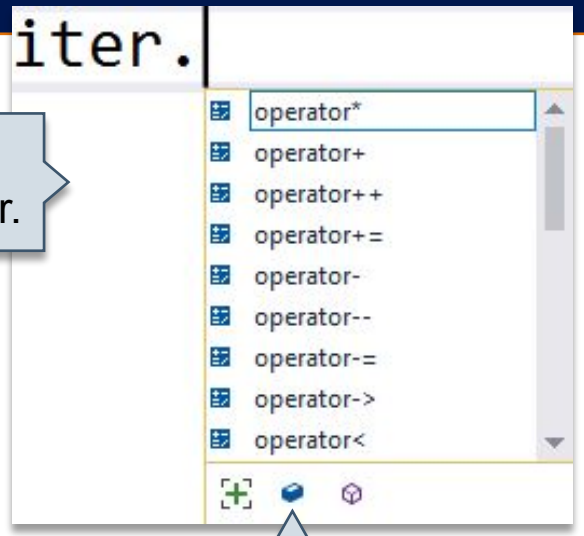
```
vector<int> data;
data.push_back(5);
data.push_back(15);
data.push_back(25);
data.push_back(35);

vector<int>::iterator iter = data.begin();
iter += 3;       // Move forward 3 elements (like "i += 3")
cout << *iter;   // Print 35 (the fourth element)

iter++  (or ++iter) // move the iterator forward one element
iter--              // move the iterator backward one element
iter + 1            // One past the current location
iter - 2            // Two elements before the current location
data.begin() + 2    // Third element in the list (2 past the first)
data.end() - 1      // Last element in the list (like "size()-1")

cout << *(data.end() - 1); // One before "one past the end", or the last element
```

List of overloaded operators in vector::iterator.



NOTE: Not all iterators will overload all of these operators

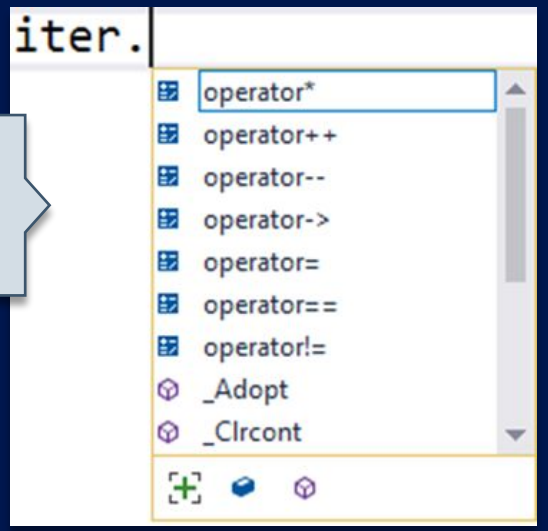
Iterator Arithmetic and Other Operators

```
unordered_map<string, int> data;
data.emplace("Batman", 50);
data.emplace("Robin", 20);

unordered_map<string, int>::iterator iter = data.begin();

// Try to print "Robin"
cout << (iter + 1)->first;   // Won't compile. operator+ not defined
                             // Not all iterators have all operators
```

List of overloaded operators in unordered_map::iterator.



Functions like begin() and end() are "easy" abstractions—data starts and ends **somewhere**.

The concept of "1 past the current" element may not make sense for some containers.

It depends on how the container is implemented—every container is potentially different.

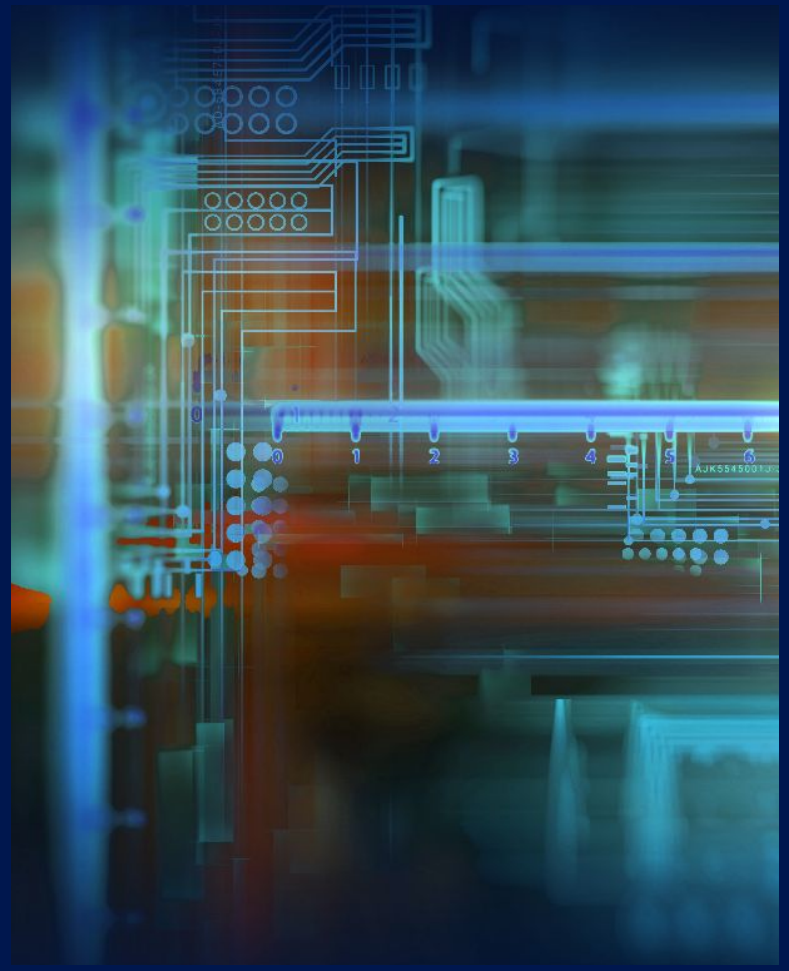
Recap

```
DATA_TYPE_HERE::someIterator = someSTLObject.begin();

for (; someIterator != someSTLObject.end(); someIterator++)
{
    // *iter, (*iter). or iter-> to use the element
}
```

- Iterators are a standardized way of iterating over elements of a container.
- `begin()` at the beginning.
- **Increment** your iterator with **operator++** (just like `i++` or `++i`).
- Keep going until you reach the `end()`, one past the last element.
- They function very similarly to pointers and must be **dereferenced** to access the current element.
- Lots of advanced functionality in C++ (and other languages) requires the use of iterators in ways that go beyond basic usage.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.