Standard Template Library

CIS-3012, C++ Programming

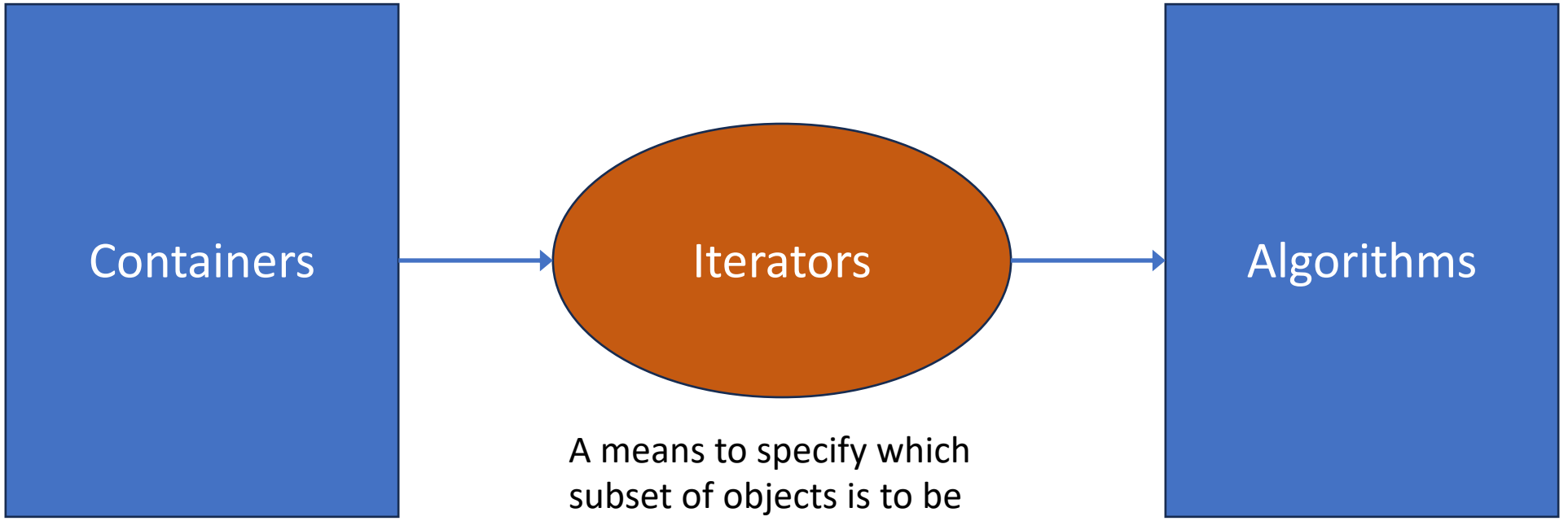
Vermont State University

Peter Chapin

C++ Standard Library

- Every compiler is required by the standard to include a library
  - The standard library includes features for doing I/O, math, string manipulation, regular expressions, and many other things.
- In C++ *most* of the standard library is templates
  - That portion is called the Standard Template Library (STL).
- The STL is maybe 80% of the standard library?
  - What is and is not part of the STL is informal. The standard doesn't talk about the STL, per se. However, people do.

Three Main Parts of the STL (pre-2020)



Class templates for class that hold other objects

A means to specify which subset of objects is to be operated on by an algorithm

Function templates for functions that implement various algorithms

Sequence Containers

Name	Description
`std::vector<T>`	An array-like collection of T that has a fully dynamic size. It provides high speed random access but O(n) insertion and erasure.
`std::deque<T>`	Like vector except with high-speed access to both ends (deque stands for double-ended queue and is pronounced “deck”).
`std::list<T>`	A doubly-linked list with highly efficient insertion and erasure, but O(n) random access. There are also high-speed splicing methods.
`std::forward_list<T>`	A singly-linked list which is more limited than list, but also uses less memory per item. This can be important in constrained systems.

Associative Containers

Name	Description
`std::set<K>`	A collection of keys where the keys are stored in sorted order. Normally sets are implemented as Red-Black trees, although that is not formally required.
`std::multiset<K>`	A collection of keys where the keys are stored in sorted order. Multisets differ from ordinary sets in that they allow multiple, equivalent keys.
`std::map<K, V>`	A collection of (key, value) pairs stored in key-sorted order. Normally maps are implemented as Red-Black trees of pairs, although that is not formally required.
`std::multimap<K, V>`	A collection of (key, value) pairs stored in key-sorted order. Multimaps differ from ordinary maps in that they allow multiple, equivalent keys (with possibly different corresponding values).

Unordered Associative Containers

Name	Description
`std::unordered_set<K>`	A collection of keys where the keys are typically stored in a hash table. Hashing can be faster in some situations, but not others.
`std::unordered_multiset<K>`	Similar in concept to multiset, except using hash tables.
`std::unordered_map<K, V>`	A collection of (key, value) pairs where the keys are typically stored in a hash table.
`std::unordered_multimap<K, V>`	Similar in concept to multimap, except using hash tables.

Container Adaptors

Name	Description
`std::queue<T>`	A container for storing items in FIFO order.
`std::priority_queue<T>`	Like a queue except items are retrieved in priority order.
`std::stack<T>`	A container for storing items in LIFO order.

- Container adaptors are not containers themselves
  - Instead, they wrap an existing container
- However, they have defaults so they can be used easily
  - For example, `stack<T>` wraps a `deque<T>` by default.
  - You can wrap a different kind of container if you have the need.

Container Adapters in Action

```
#include <list>
#include <stack>


stack<int> my_stack1;                  // Uses deque<int> internally (default).
my_stack1.push( 42 );                  // Push onto the stack.
int top_item = my_stack1.top( );       // Get a copy of top item.
my_stack1.pop( );                      // Remove top item.

stack<int, list<int>> my_stack2;       // Uses list<int> internally.
// etc., same as above.
```

Iterators

- An iterator is a pointer-like object…
  - … in the sense that it supports similar operations as do pointers.
- Iterator is not a type!
- Every container has a separate iterator type that can be used to "point into" that container…
  - … and thus access the elements of that container.
- Every container has `begin` and `end` methods
  - `begin( )` returns an iterator that points at the *first element*
  - `end( )` returns an iterator that points *just past the last element*

Example: Vector Iterators

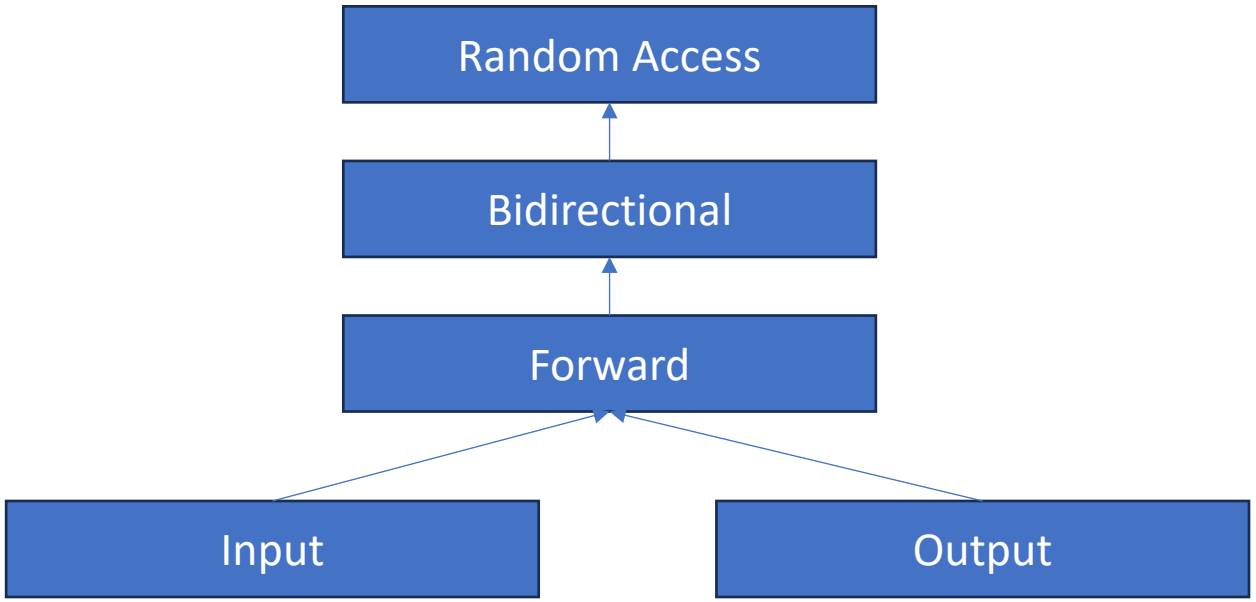
```
#include <vector>

vector<int> vec = { 2, 3, 5, 7, 11, 13, 17, 19 };

vector<int>::iterator it = vec.begin( );
cout << *it << endl;        // Prints 2
cout << *(it + 3) << endl; // Prints 7; vector iterators allow pointer-like arithmetic
++it;                       // Vector iterators allow pointer-like increment
cout << *it << endl;        // Prints 3

it = vec.end( );
cout << *it << endl;        // UNDEFINED! The end iterator points off the end!
--it;
cout << *it << endl;        // Prints 19
```

Iterator Types

```
#include <list>
#include <vector>

vector<int>    vec  = { 2, 3, 5, 7, 11, 13, 17, 19 };
vector<double> dvec = { 3.14, 2.78, 1.62 };
list<int>      lst  = { 2, 3, 5, 7, 11, 13, 17, 19 };

vector<int>::iterator    it_1 = vec.begin( );
vector<double>::iterator it_2 = dvec.begin( );
list<int>::iterator.     it_3 = lst.begin( );

it_1 = it_2;   // Error! Type mismatch!
  // vector<double>::iterator is a different type than vector<int>::iterator.

it_1 = it_3;   // Error! Type mismatch!
  // list<int>::iterator is a different type than vector<int>::iterator.
```

Iterator Categories

- In this diagram, the arrows point in the direction of increasing capability. **This is not a UML class diagram!** Iterator categories are not types!



Random Access Iterators

- Operations:
  - Increment *and* decrement
  - All six relational operators (`it_1 < it_2` is a sensible expression)
  - Pointer arithmetic (`it_1 + 10` is a sensible expression)
  - Multi-pass (can pass over collection multiple times)
- Provided By:
  - Vector
  - Deque

Bidirectional Iterators

- Operations:
  - Increment *and* decrement
  - Only == and != supported
  - No pointer arithmetic
  - Multi-pass (can pass over collection multiple times)
- Provided By:
  - List
  - Set/Multiset
  - Map/Multimap

Forward Iterators

- Operations:
  - Increment *only*
  - Only == and != supported
  - No pointer arithmetic
  - Multi-pass (can pass over collection multiple times), but only one way
- Provided By:
  - Forward List
  - Unordered Set/Multiset
  - Unordered Map/Multimap

Input/Output Iterators

- Operations:
  - Increment *only*
  - Only == and != supported
  - No pointer arithmetic
  - Single-pass (can only pass over collection once)
  - Input Iterators provide read-only access to collection elements
  - Output Iterators provide write-only access to collection elements
- Provided By:
  - Istreams (input)
  - Ostreams (output)

Pointers?

- Ordinary pointers have all the operations of random access iterators
  - Thus, pointers are a kind of iterator
  - This unifies pointers (and therefor arrays) with the other containers in the standard template library.
  - That is, an ordinary C-style array is a kind of container and can be treated largely the same way as the other containers.

```
#include <iterator>

int array[128];

int *p1 = std::begin(array);  // Points at the first element.
int *p2 = std::end(array);    // Points just past the last element.
  // std::begin and std::end can also be used with the STL containers.
```

Iterators and Range-Based For Loops

```
vector<int> my_vector = { … };
for( int x : my_vector ) { … }

list<int> my_list = { … };
for( int x : my_list ) { … }

set<int> my_set = { … };
for( int x : my_set ) { … }

string my_string = … ;
for( char x : my_string) { … }

int my_array[128] = { … };
for( int x : my_array ) { … }

// Any container type that provides appropriate iterators can be used this way.
// Including your own classes!
```

Algorithms!