Advanced Programming Concepts with C++ CSI2372 – Fall 2019

Jochen Lang &
Mohamed Taleb
EECS

uOttawa

L'Université canadienne
Canada's university

Université d'Ottawa | University of Ottawa

uOttawa.ca

Sequential Containers and Iterators

No reinventing the wheel

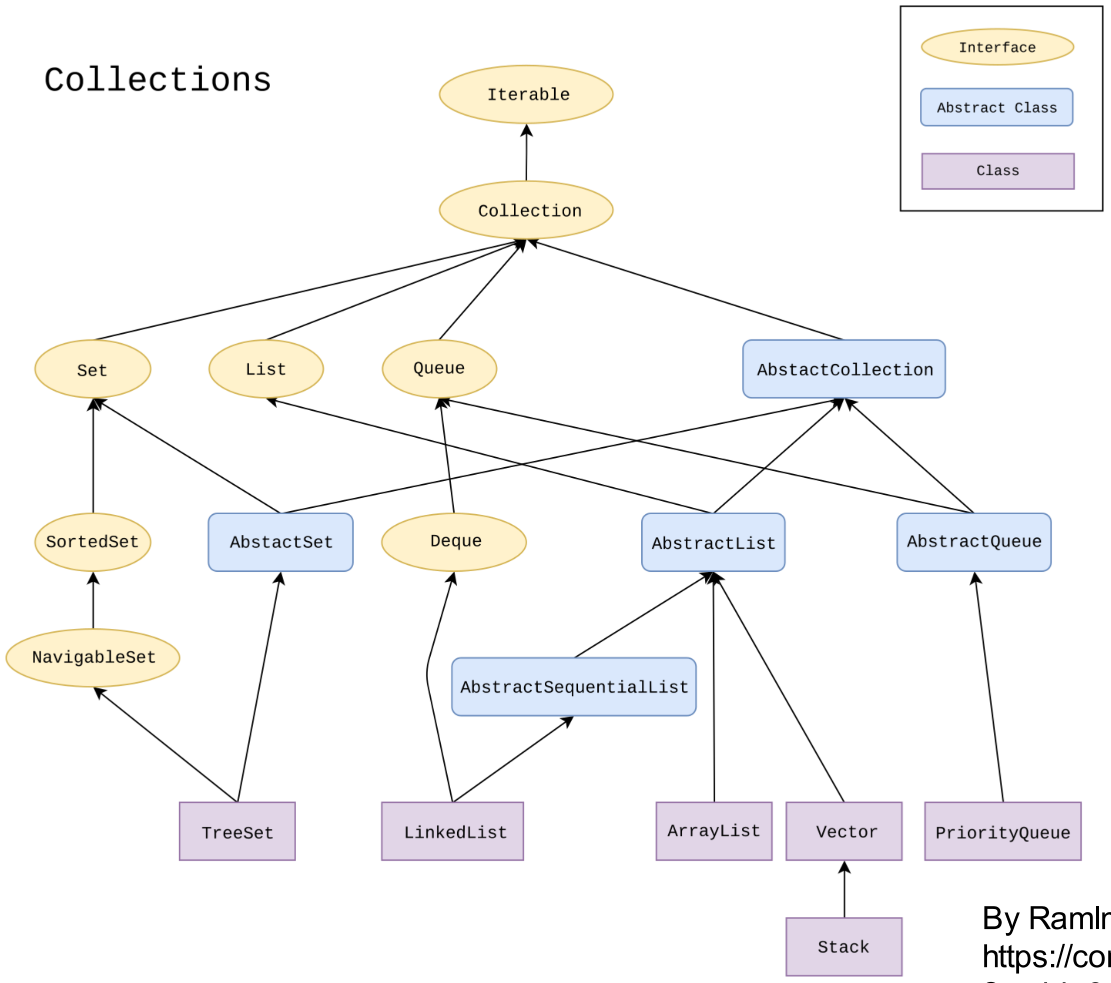
- **STL**
  - Review: Java Collections Framework
  - Sequential containers, Ch. 9.1- 9.4
    - `vector string deque`
    - C++11 `array forward_list`
  - Iterators, Ch. 3.4, 9.2.1

Java Collections Framework

- Aside: Collection is another word for container
- Java Collections Framework is an object-oriented framework with generic types
- Implements most basic data structures
- Including:
  - linked lists, growable arrays, trees, maps and dictionaries, sets

- **Why use collections?**
  - available (no implementation required)
  - reliability and standardization
  - efficient general purpose implementations

Jave Collections



By Ramlmn - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=64043967

uOttawa

Components of the Framework

- **Top-Level Interfaces**
  - `Collection` and its children `Set, List, Queue, Deque` plus related interfaces targeted at concurrency
  - `Map`
- **Implementations**
  - List implementations: `ArrayList`, `LinkedList` and about 6 more.
  - Set implementations: `HashSet`, `TreeSet` and about 4 more.
  - Queue implementations: `LinkedList, PriorityQueue` and about 10 more.
  - Map implementations: `HashMap`, `TreeMap, HashTable` and about 6 more.

Relationships between Abstract Data Type

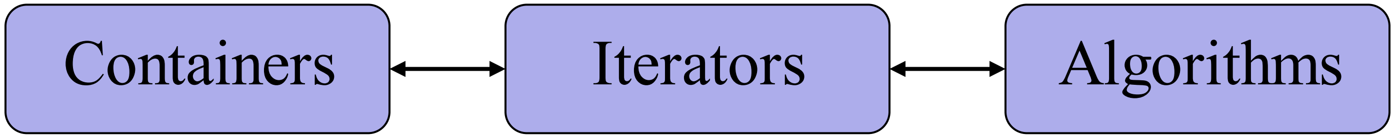
Interfaces	Implementations				
	Hash Table	Resizable Array	Balanced Tree	Linked List	Old
`Set`	`HashSet`		`TreeSet`		
`List`		`ArrayList`		`LinkedList`	`Vector, Stack`
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`HashTables`

Iterators and Algorithms for Collections

- **Iterators**
  - `Iterable` interface allows one to obtain an iterator for the container
  - `Iterator` itself is an interface and is implemented by the container classes
- **Algorithms**
  - In Java, we have polymorphic algorithms (sort of)
    - sort algorithms are implemented in separate class `Sort` and can be used with containers
  - `sort, shuffle, reverse, fill, copy, swap, addAll, binarySearch, frequency, disjoint, min, max` are part of the Collection interface

Abstract Data Types in C++

- **Standard Template Library**
  - Implements standard abstract data types
  - More generic than object-oriented
  - Containers and Adaptors
    - similar to Java collections and interfaces
  - Generic algorithms
    - similar in scope to algorithms in `Collection` but not part of any class



Standard Template Library (STL)

- STL is more than abstract data types

- **STL Content Overview**
  - containers including strings, i.e., abstract data types
  - generic algorithms
  - memory management support
  - runtime environment support
  - streams
  - exceptions

Containers

– Containers store elements of the same type

Sequential Containers

Arrays
Lists

Derived Containers

Defined in terms of other Containers

Associative Containers

Maps

Sequential Containers

- Textbook, Chapter 9
- Store one element after another in sequence
- Order of elements does not depend on element (no key)

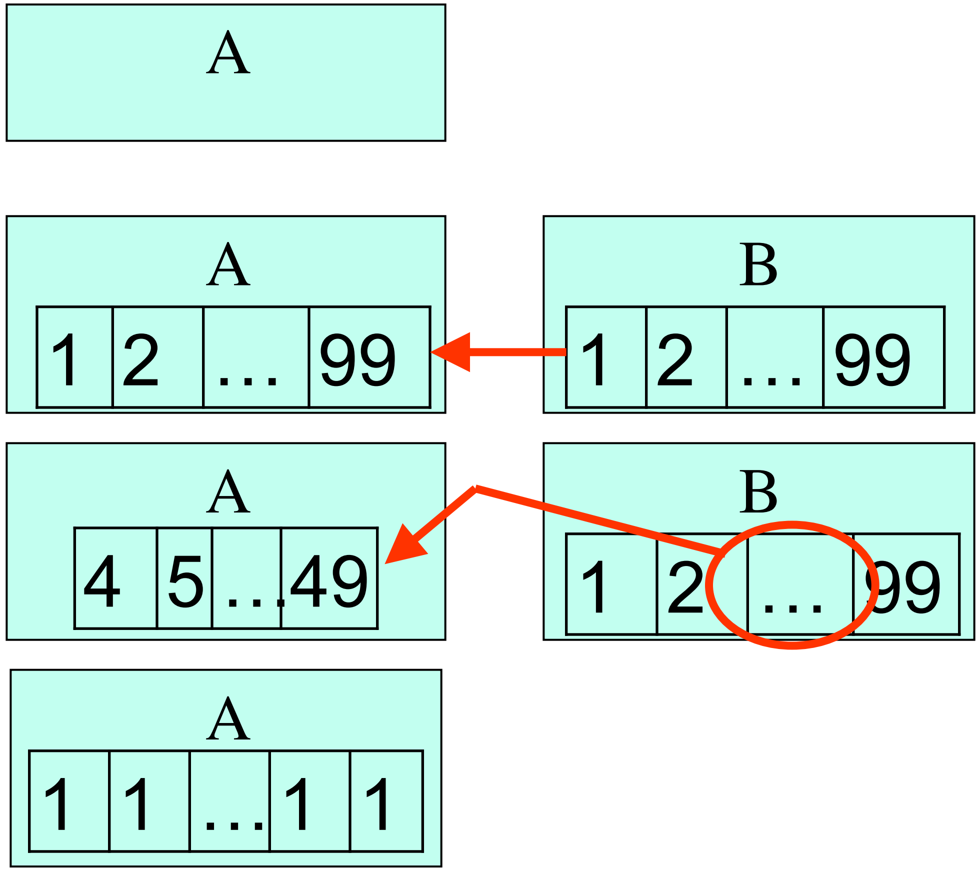
- **Sequential STL Containers**
  - `vector list deque`
    - Defined in headers `<vector> <list> <deque>`
  - C++11: `forward_list array`
    - Defined in headers `<forward_list> <array>`

Review: Why a vector (growable array)?

- **Limitations of built-in arrays**
  - Arrays are fixed size
  - Arrays can not be operated on as a whole
- **Vectors (growable array)**
    - `ArrayList` or `Vector` (deprecated) in Java
  - Adjust their size based on the number of element stored in the vector
  - Vectors can be copied, assigned and compared
  - Offer same random (constant time) access than arrays

Constructing a Container

- Default
  - empty container
- Copy
  - copies the elements
- Copying a portion of another container
- Constructing elements in the container
- C++11: list initialization



Review: Constructing a `vector`

```
#include <vector>
using std::vector;
// Default construction
vector<int> iVecA;
// Construction by constructing 100 elements all = 1
vector<int>::size_type size = 100;
vector<int> iVecB( size, 1 );
// copy construction
vector<int> iVecC( iVecB );
// construction by copying the first 5 elements
vector<int>::iterator iter = iVecB.begin();
vector<int> iVecD( iter, iter + 5 );
// list initialization
vector<int> iVecE{1,2,3,4,5};
```

Containers hold copies

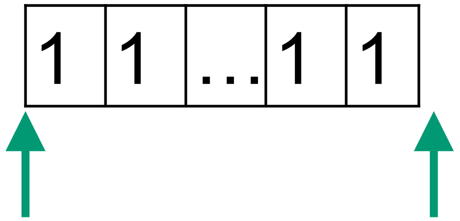
- **Minimum requirements**
  - Container will use assignment operator
  - Container stores copies of elements
    - Only types with a copy constructor
    - Or types with a move constructor (since C++11)
- **Additional requirements for some operations**
  - E.g., container must be able to default construct elements if vector<T>(size_type) is used
    - Only types with a default constructor

Accessing elements in a Container: Iterators

- **Iterators encapsulate the principle of visiting elements in turn**
  - Pointers are iterators for built-in arrays
- **Common Operations with Iterators**
  - Dereference `*iter`
  - Access to methods and attributes of element `iter->foo()`
  - Increment and decrement `iter++ ++iter iter-- --iter`
  - Comparison; equals and not equals `== !=`
  - C++11: `iter=prev(iter) iter=next(iter)` `begin(`*`container`*`) end(`*`container`*`)`

Iterator Range

- **Sequential containers provide an iterator to the beginning and the end**
  - The end iterator points just passed the last element



  - Many methods work with a range, e.g., construction

```
vector<int> iVecA;
// Copy the whole range into a new vector
vector<int> iVecB( iVecA.begin(), iVecA.end());
// Alternatively
vector<int> iVecB( begin(iVec), end(iVecA));
```

Review: Looping over the Elements in a Container – Option A

- **Looping over the elements**
  - using a range loop
  - typically combined with auto type
  - makes a copy of the element in the container
  - Or must use a reference

```
#include <vector>
// 100 elements vector
vector<int>::size_type size = 100;
vector<int> iVec( size );
// loop over the elements
for ( auto element:iVec ) { // read access to element }
for ( auto &refElement:iVec ) { // write access }
```

Looping over the Elements in a Container – Option B

- Looping over the elements with traditional for loop
  - if begin == end the container is empty
  - try to write loop only with operations supported by all containers

```
#include <vector>
// 100 elements vector
vector<int>::size_type size = 100;
vector<int> iVec( size );
// loop over the elements
for ( auto iter = iVec.begin();
      iter != iVec.end();
      ++iter ) {
  // save to access *iter
}
```

```
vector<int>::iterator
```

Aside: Nested Classes

- Containers in STL make use of nested classes (see also Ch. 19.5): Classes defined in other classes

```
template <typename T>
class vector<T>::iterator { …
};
```

- Or in general

```
class Outer {
 private:
  class Inner {
    int d_i;
   public:
    Inner(int _i) : d_i(_i) {};
  };
};
```

More on Iterators

- Erase and insert may invalidate an iterator
  - Be careful when you write a loop
- **`vector`, `deque` (and `array`) iterators support "arithmetic" operations**
  - Addition `iter+n` (same as `iter[n]`)
  - Subtraction `iter-n` and `n=iterA-iterB`
  - Compound assignment `iter+=n` `iter-=n`
  - Extra comparisons `>` `<` `>=` `<=`
  - `vector`, `deque` and `array` are random access containers
    - similar to pointer arithmetic with arrays

Iterator Categories

- **In increasing power**
  - Input/Output iterators
  - Forward iterators
  - Bidirectional iterators
  - Random-Access iterators
- **Iterator also exist in the variations**
  - const
    - Element “pointed to” cannot be changed
  - reverse
    - Directions (++/--) are reversed

Iterators and Containers

	`::iterator`	`::const_iterator`	`::reverse_iterator`	`::const_reverse_iterator`
`vector`, `array`	*random-access*	*const random access*	*reverse random-access iterator*	*const reverse random-access iterator*
`deque`	*random-access*	*const random access*	*reverse random-access iterator*	*const reverse random-access iterator*
`list`	*bidirectional*	*const bidirectional*	*reverse bidirectional iterator*	*const reverse bidirectional iterator*
`forward_list`	*forward*	*const forward*	-	-

Looping backwards over the Elements in a Container

- Looping over the elements
  - if rbegin == rend the container is empty
  - as before: write loop only with operations supported by all containers

```
#include <vector>
// 100 elements vector
vector<int>::size_type size = 100;
vector<int> iVec( size );
// loop over the elements
for (auto rIter = iVec.rbegin();
      rIter != iVec.rend();
      ++rIter ) {
  // go backwards -- save to access *rIter
}
```

```
vector<int>::reverse_iterator
```

Container Methods: Insertion

- **Inserting into a container**
  - Insert at the end push_back(element)
  - Insert at the front push_front(element)
  - Insert anywhere insert(iter, element)
- **C++11**
- **Avoid copy constructor by the container directly calling a constructor for the element**
  - Insert at the end emplace_back(args for element ctor)
  - Insert at the front emplace_front(args for element ctor)
  - Insert anywhere emplace(iter,args for element ctor)

Container Methods: Insertion and Removal

- **Remove element(s)**
    - Do not return the element!
  - Remove at the end `pop_back()`
  - Remove at the front `pop_front()`
  - Remove anywhere `remove(iter)`
  - Remove all `clear()`

Container Methods: Assignment

- **Assignment**
  - Copy all the elements into an existing container (deleting all existing elements in destination) `cA = cB`
  - Copy the elements in the iterator range into an existing container (deleting all existing elements in the destination) `cA.assign(cB.begin(),cB.end())`
  - Swap the elements between the containers `cA.swap(cB)`

Container Methods: Size and Capacity

- **Size (the actual # elements stored)**
  - Number of elements stored in container `size()`
  - Is container empty `empty()`
  - Add or delete elements at the end of the container `resize(`*`#elements`*`)`
- **Vector is a growable array!**
  - Create empty slots in the array `reserve(`*`num`*`)`
  - How many slots in total `capacity()`

Container Methods
Some other useful methods

- **Container comparison**
  - Compare if container has the same size and elements `c1 == c2`
  - Or not `c1 != c2`
- **Reference to elements**
  - Reference to first element `front()`
  - Reference to last element `back()`
  - Reference to any element (vector and deque only) `c.at(n)`
    - Most often iterators are used to access element!

An Example: BubbleSort with a List, Vector or Deque

```
template <class T>
void bubbleSort( T& container ) {
  // loop over the elements
  for ( typename T::iterator iterA = container.begin();
        iterA != container.end();
        ++iterA ) {
    for ( typename T::iterator iterB = iterA;
          iterB != container.end();
          ++iterB ) {
      if ( *iterA > *iterB ) { // swap
        typename T::value_type tmp(*iterA);
        *iterA = *iterB; *iterB = tmp;
      }
    }
  }
  return;
}
```

Next

- **Associative containers**
  - Maps
- **Examples of generic algorithms**
  - Finding
  - Sorting