C++ 11 and the Standard Library: Containers, Iterators, Algorithms

Comp Sci 1575 Data Structures

Missouri S&T Computer Science

C++ versus C

Q: Why did C++ decide not to go out with C?
A: Because C has no class.

Outline

1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr
2. Data Structures Overview
3. STL
   - History
   - Containers
   - Iterators

Outline

1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr
2. Data Structures Overview
3. STL
   - History
   - Containers
   - Iterators

uniform initialization

```
#include <iostream>
 int main()
{
    int value0 = 5; // C++ 98
    int value1(5); // C++ 98

    int value{5}; // C++ 11
    std::string a{‘‘hello”}; // C++ 11

    int value; // C++ 98
    // value() // looks like a function, error
    int value2{}; // default to 0; C++ 11

    int i = 3.99; // i gets 3; no warning/error
    int k{3.99}; // i gets 3; warning

    return 0;
}
```

uniform initialization with arrays

```
#include <iostream>
int main()
{
    // C++ 98
    int arr[] =  {1,2,3,4,5};

    // C++ 11 optional
    int arr[] {1,2,3,4,5};
    return 0;
}
```

This is consistent with non-arrays now.

Outline

1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr
2. Data Structures Overview
3. STL
   - History
   - Containers
   - Iterators

range-based for loops

```
#include <iostream>

int main()
{
    int fibonacci[] = {0, 1, 1, 2, 3, 5, 8};

    for(int number : fibonacci)
        std::cout << number << ‘,’;

    return 0;
}
```

range-based for loops: modify element

```
#include <iostream>

int main()
{
    int fib_plus[] = {0, 1, 1, 2, 3, 5, 8};

    for(int &number : fib_plus)
        number++;

    for(int number : fib_plus)
        std::cout << number << ‘,’;

    return 0;
}
```

Fast and “safe” range-based for loops

```
#include <iostream>

int main()
{
    int fibonacci[] = {0, 1, 1, 2, 3, 5, 8};

    for(const int &number : fibonacci)
        std::cout << number << ‘,’;

    return 0;
}
```

Avoid copy

Outline

1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr
2. Data Structures Overview
3. STL
   - History
   - Containers
   - Iterators

auto type deduction

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    auto d = 5.0; // 5.0 is a double literal
    auto i = 1 + 2; // evaluates to an integer

    auto sum = add(5, 6); // add() returns int

    return 0;
}
```

auto in range-based for loops

```
#include <iostream>

int main()
{
    auto fibonacci[] = {0, 1, 1, 2, 3, 5, 8};

    for(auto number : fibonacci)
        std::cout << number << ‘,’;

    return 0;
}
```

Warning: For-each doesn’t work with pointers to an array, and thus with arrays passed to functions.

Outline

1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr
2. Data Structures Overview
3. STL
   - History
   - Containers
   - Iterators

nullptr should generally be used instead of **NULL**

NULL is a "manifest constant" (a #define of C) that's actually an integer that can be assigned to a pointer because of an implicit conversion.
nullptr is a keyword representing a value of self-defined type, that can convert into a pointer, but not into integers.

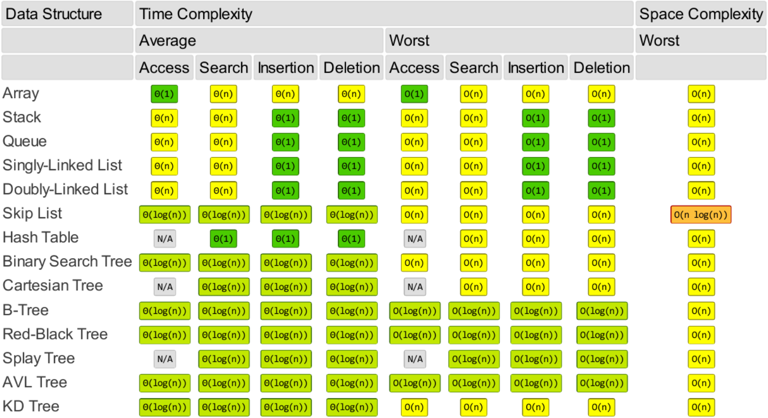
```
#include <iostream>
int main(){
    int i = NULL; // OK
    // int i = nullptr; //error, no convert to int
    int* p = NULL; //ok, int converted into ptr
    int* p = nullptr; // ok
    // suppose you have two functions in overload:
    void func(int x);
    void func(int* x);
}
```

Now, if you call func(NULL), you are actually calling the int variant, NULL being an int. But func(nullptr) will call the pointer variant, nullptr not being an int.

Outline

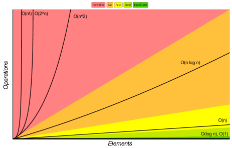
1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr
2. Data Structures Overview
3. STL
   - History
   - Containers
   - Iterators

Data structures

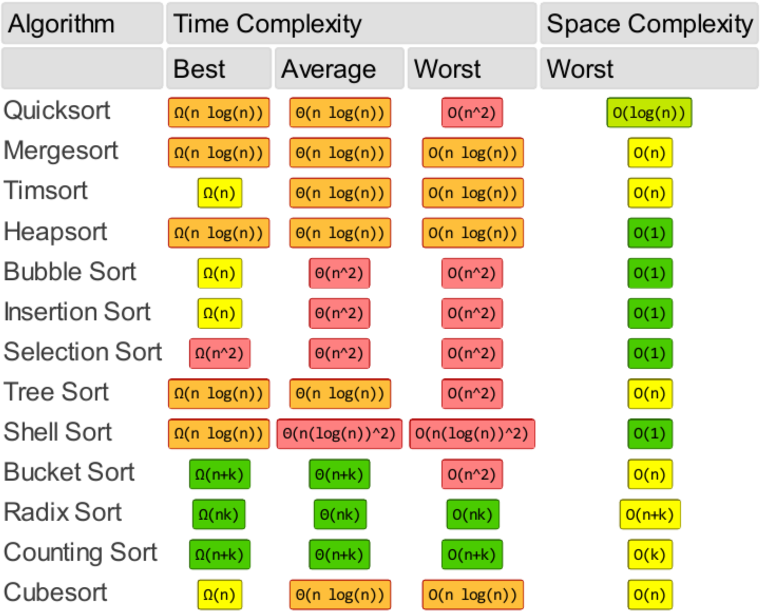


Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Color key:



Sorting algorithms



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

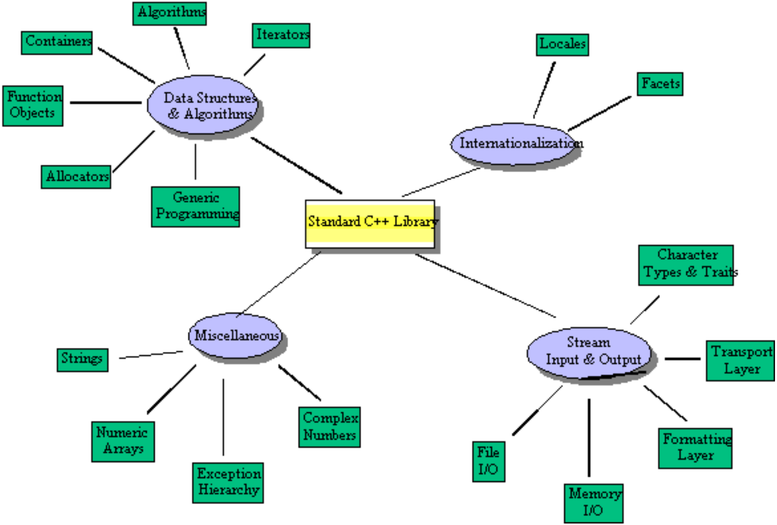
Outline

1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr
2. Data Structures Overview
3. STL
   - History
   - Containers
   - Iterators

Outline

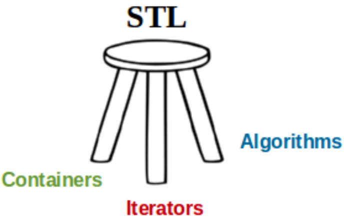
1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr
2. Data Structures Overview
3. STL
   - History
   - Containers
   - Iterators

STL is now just part of standard namespace



- Old diagram of the **S**tandard **T**emplate **L**ibrary (STL)

Main pillars of the STL



- **Containers** manage storage space for elements and provide member functions to access them. Implemented as templated classes, with flexibility of types as elements.
- **Algorithms** act on containers, and perform operations like initialization, sorting, searching, and transforming of the contents of the aforementioned containers.
- **Iterators** step through elements of collections of objects in containers or subsets of containers. Pointer-like iterators can traverse many container classes in modularly.

Example *std* :: *vector*

Check out the code: container with iterators and algorithms

A dynamic C-like array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object.

- Random access is done in constant, O(1) time.
- Insertion or removal of elements at the back takes average time O(1), amortized constant time. Removing the last element takes only constant time, because no re-sizing happens.
- Insertion or removal of elements at the beginning or "middle" is linear in distance to the end of the vector O(n).

See: Intro_vector.cpp

Outline

1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr

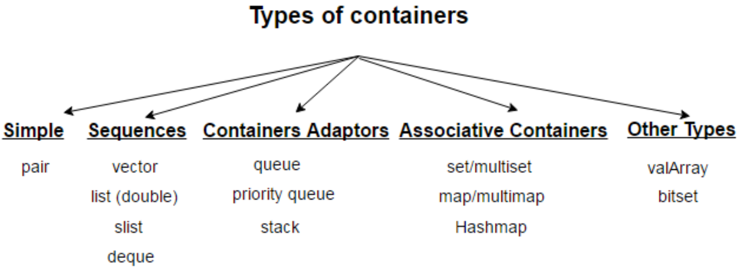
2. Data Structures Overview

3. STL
   - History
   - Containers
   - Iterators

Containers

- Containers library is a generic collection of class templates and algorithms that allow programmers to easily implement common data structures like queues, lists, and stacks.
- Classes of containers, each of which is designed to support a different set of operations:
  1. sequence containers
  2. associative containers
  3. un-ordered associative containers
  4. container adaptors (modify above)
- Containers manage storage space that is allocated for their elements and provides member functions to access them, either directly or through iterators (objects with properties similar to pointers).
- The set of containers have at least several member functions in common with each other, and share functionalities.

Containers



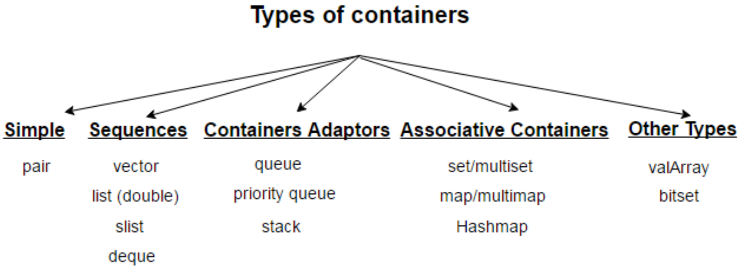
Types of containers

Simple	**Sequences**	**Containers Adaptors**	**Associative Containers**	**Other Types**
pair	vector	queue	set/multiset	valArray
	list (double)	priority queue	map/multimap	bitset
	slist	stack	Hashmap	
	deque			

For a comprehensive list, see:

- `http://en.cppreference.com/w/cpp/container`
- `http://www.cplusplus.com/reference/stl/`
- `https://en.wikipedia.org/wiki/Standard_Template_Library`

Simple utilities: pair, tuple



Types of containers

Simple: pair

Sequences: vector, list (double), slist, deque

Containers Adaptors: queue, priority queue, stack

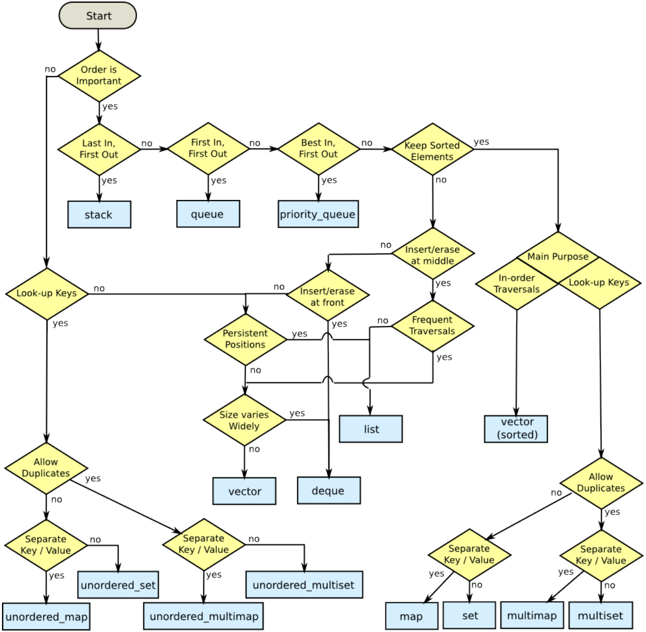
Associative Containers: set/multiset, map/multimap, Hashmap

Other Types: valArray, bitset

Not in the “Containers” but “Utilities”

- `http://en.cppreference.com/w/cpp/utility/pair`
- `http://en.cppreference.com/w/cpp/utility/tuple`

How to choose your container?



Outline

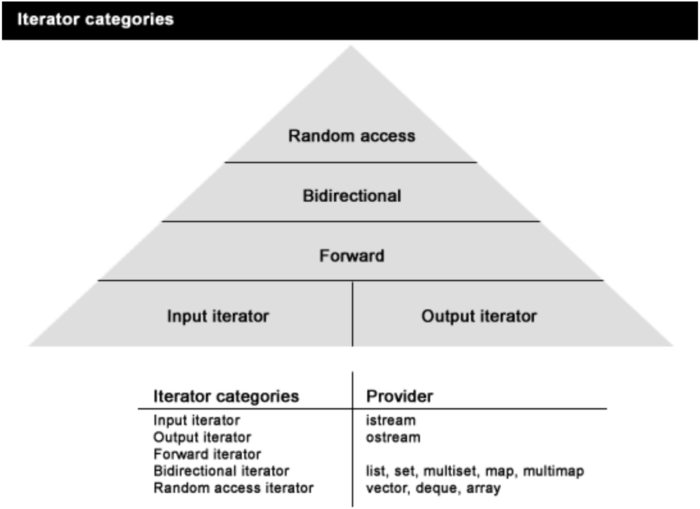
1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr
2. Data Structures Overview
3. STL
   - History
   - Containers
   - Iterators

Iterators

An iterator can be imagined as a pointer to a given element in the container, with overloaded operators to provide a subset of well-defined functions normally provided by pointers:

- $Operator*$ Dereferencing the iterator returns the element that the iterator is currently pointing at.
- $Operator++$ Moves the iterator to the next element in the container. Most iterators also provide $Operator--$ to move to the previous element.
- $Operator ==$ and $Operator! =$ Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, dereference the iterators first, and then use a comparison operator.
- $Operator =$ Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is point at, dereference the iterator first, then use the assign operator.

Iterators

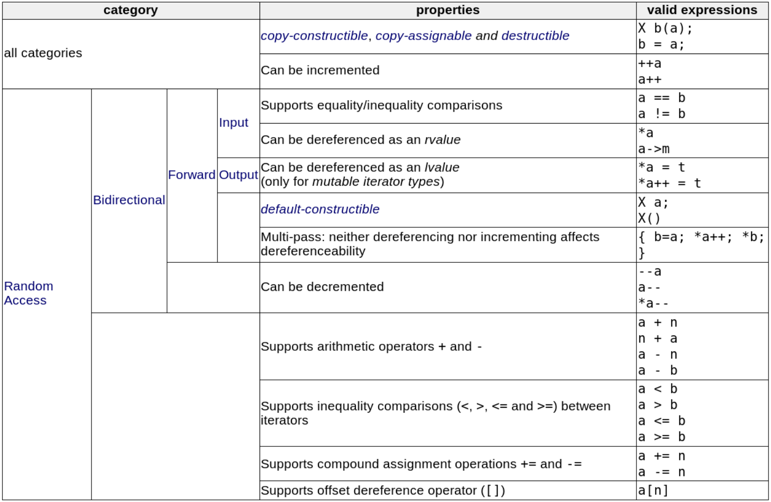


Iterator categories

Iterator categories	Provider
Input iterator	istream
Output iterator	ostream
Forward iterator	
Bidirectional iterator	list, set, multiset, map, multimap
Random access iterator	vector, deque, array

- Where does forward_list go?
- `http://en.cppreference.com/w/cpp/iterator`
- `http://www.cplusplus.com/reference/iterator/`

Iterators



category				properties	valid expressions

all categories				*copy-constructible*, *copy-assignable and destructible*	X b(a);
					b = a;

				Can be incremented	++a a++
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	a == b a != b
				Can be dereferenced as an *rvalue*	*a a->m
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	*a = t *a++ = t

				default-constructible	X a;
					X()

				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	{ b=a; *a++; *b; }
				Can be decremented	--a a-- *a--
				Supports arithmetic operators + and -	a + n n + a a - n a - b
				Supports inequality comparisons (<, >, <= and >=) between iterators	a < b a > b a <= b a >= b
				Supports compound assignment operations += and -=	a += n a -= n
				Supports offset dereference operator ([ ])	a[n]

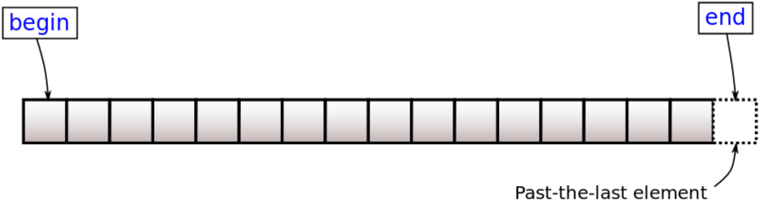
Each category of iterator is defined by the operations that can be performed on it. Any type that supports the necessary operations can be used as an iterator, e.g., pointers support all of the operations required by RandomAccessIterator, so pointers can be used anywhere a RandomAccessIterator is expected.

Iterators

Each container includes at least 4 member functions for the operator= to set the values of named LHS iterators.

- begin() returns an iterator to the first element.
- end() returns an iterator one past the last element.
- cbegin() returns a const (read-only) iterator to the first element.
- cend() returns a const (read-only) iterator one past the last element.

end() doesn't point to the last element in the list. This makes looping easy: iterating over the elements can continue until the iterator reaches end(), and then you know you're done.

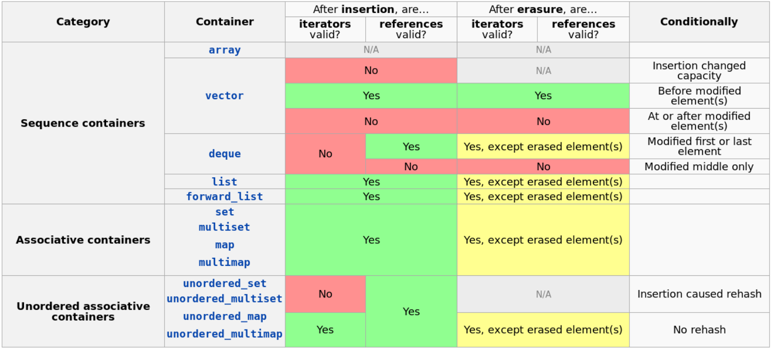


iterator example

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<int> ints {1, 2, 4, 8, 16};
    std::vector<std::string> fruits {"orange", "apple", "raspberry"};
    std::vector<char> empty;

    // Sums all integers in the vector ints (if any), printing only the result.
    int sum = 0;

    for (auto it = ints.cbegin(); it != ints.cend(); it++)
        sum += *it;

    std::cout << "Sum of ints: " << sum << "\n";

    // Prints the first fruit in the vector fruits, without checking
    std::cout << "First fruit: " << *fruits.begin() << "\n";

    // checks
    cout << empty.empty();
    if (empty.begin() == empty.end())
        std::cout << "vector 'empty' is indeed empty.\n";

    // Alternative syntax
    auto it1 = ints.begin();
    auto it2 = std::begin(ints);
}
```

Container have different iterator invalidation rules

Each container has different rules for when an iterator will be invalidated after operations on the container: `http://en.cppreference.com/w/cpp/container`



Category	Container	After **insertion**, are...		After **erasure**, are...		Conditionally
		iterators valid?	**references** valid?	**iterators** valid?	**references** valid?	
Sequence containers	`array`	N/A		N/A		
	`vector`	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s)
		No		No		At or after modified element(s)
	`deque`	No	Yes	Yes, except erased element(s)		Modified first or last element
			No	No		Modified middle only
	`list`	Yes		Yes, except erased element(s)		
	`forward_list`	Yes		Yes, except erased element(s)		

Associative containers	`set`
	`multiset`
	`map`
	`multimap`
		Yes		Yes, except erased element(s)		

Unordered associative containers	`unordered_set`
	`unordered_multiset`
	`unordered_map`
	`unordered_multimap`
		No	Yes	N/A		Insertion caused rehash

		Yes		Yes, except erased element(s)		No rehash