CSE 1325: Object-Oriented Programming

Lecture 24

Standard Template Library (STL)

Containers, Iterators, and Algorithms

Mr. George F. Rice

george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

10 cards == 1 decacards

Overview: Containers, Iterators, and Algorithms

- Containers
  - Sequence Containers
  - Container Adapters
  - Associative Containers
  - Unordered Associative Containers
- Iterators & Const_iterators
- Algorithms

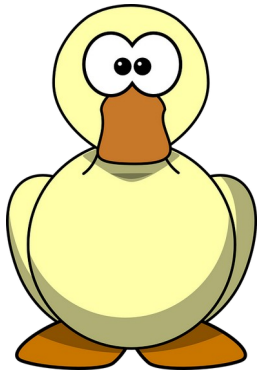
"Containers" is the C++ name for Java's "Collections":
Objects that hold other objects!

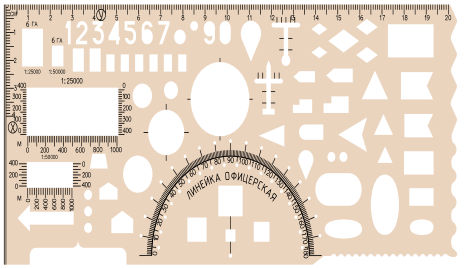
Cargo container ship photo cc0 Public Domain
https://www.publicdomainpictures.net/en/view-image.php?image=104269&picture=cargo-container-ship

Template Summary

Templates are NOT on the Exam

- **Template** – A C++ construct representing a class, method, or function in terms of generic types
  - The algorithm is written independent of data types
  - Any type works with Templates in C++ (even primitives)
  - The “template types” are specified when the class is instanced or the method / function is called
  - Unlike Java generics, templates use “**Duck Typing**”:
    - The type must supply whatever operators and other methods are referenced by the template
- Potentially reusable algorithms that are suitable should generally be defined as templates





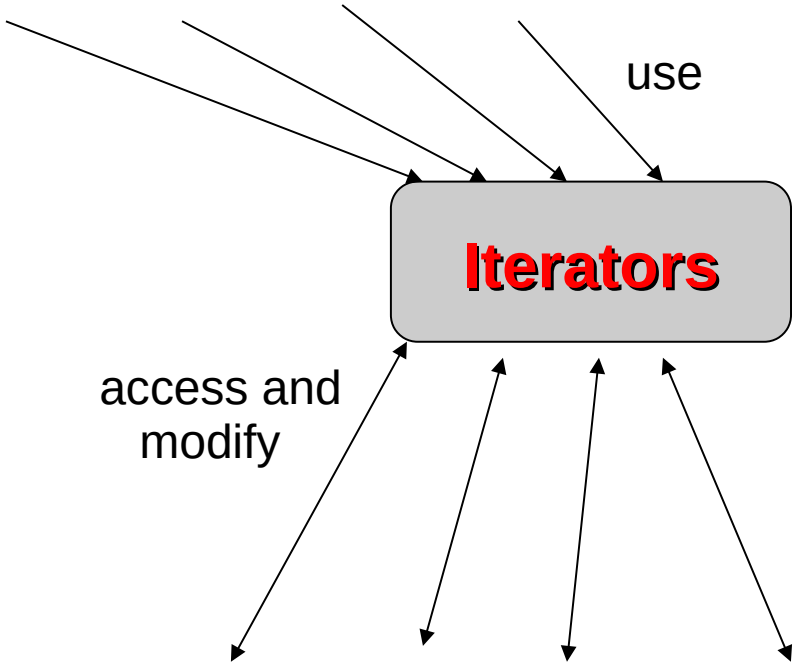
The Standard Template Library (STL)

- The STL provides a good variety of templates
  - Performance is a key goal
  - Our friends **`std::vector`**, **`std::set`** and **`std::map`** are STL *templates*
  - **`std::sort`**, **`std::find`**, and **`std::count`** are STL *functions*
  - **`std::begin`** and **`std::end`** return *iterators*, which are objects that work like pointers
- Originally designed by Alex Stepanov
  - His goal was to make good programming “like math”

Basic STL Model

Algorithms

Sort, find, search, copy, ...



vector, list, map, unordered_map, ...

Containers

- Separation of concerns
  - **Algorithms** manipulate data, but don't know about **containers**
  - **Containers** store data, but don't know about **algorithms**
  - **Iterators** connect the two
    - Each container defines its own iterator types as nested classes

The STL also defines some **Functors**, but we'll ignore those...

Containers

STL Container Overview

- The STL contains the following containers of interest
  - **Sequence Containers** store data linearly
    - **vector**, list and forward_list (linked lists), and deque (string is also a non-STL sequence-like container)
  - **Container Adapters** are façades to Sequence Containers
    - stack, queue, and priority_queue
  - **(Ordered) Associative Containers** provide fast lookup via keys by maintaining the data in sort order (like TreeSet and TreeMap in Java)
    - **set**, **map**, plus multiset (non-deduplicating) and multimap (1+ identical keys)
  - **Unordered Associative Containers** are unsorted, and thus provide faster insertion but slower lookups (like HashSet and TreeSet in Java)
    - unordered_set, unordered_map, unordered_multiset, unordered_multimap
- Let's take a brief look at each of the **green containers**

Sequence Containers
vector and array

- **Vector** provides a resizable, flexible version of the C / C++ standard array (like Java's ArrayList)
  - Very fast lookup (e.g., `v[42]`)
  - Relatively slow insert / delete except at the end
  - We already know vector fairly well
- **Array** is a fixed-size **vector** with virtually identical methods
- **Deque** is a double-ended queue just as in Java, with operator[ ] and both push_back / pop_back AND push_front / pop_front
- **List** is a **double-linked** list that is navigable forward (**++**) or backward (**--**)
- **Forward_list** provides a **single-linked** list that is navigable forward (**++**) only

http://www.cplusplus.com/reference/vector/vector/
http://www.cplusplus.com/reference/array/array
http://www.cplusplus.com/reference/deque/deque
http://www.cplusplus.com/reference/list/list
http://www.cplusplus.com/reference/forward_list/forward_list/

Common Vector Operations

- **v.empty()** is true if the vector contains no values
  - **v.clear()** removes all values from the vector
- **v.size()** returns the number of values in the vector
- **x = v[index]** is random access
  (Warning: undefined if index is out of range)
  - **x = v.at(index)** is like operator[], but instead throws a std::out_of_range exception if the index isn't in the vector
- **v[index] = x** overwrites value
  (Warning: undefined if index out of range – use **at** to throw exception)
  - **v.erase(iterator)** removes the value from the vector
- **s.insert(value, iterator)** adds value to the set
- **for(auto& value : v)** iterates over the vector values
- **v.begin()** and **v.end()** return "iterators" to the first and one past the last value, which behave like pointers

Sequence Containers
string

- OK, **string** is NOT a “real” STL container, but it works very much like a vector of char
  - Relatively fast **`char`** lookup (e.g., **`str[42]`**)
  - Very slow insert / delete
- Actually, **`std::string`** is just a typedef for **`basic_string<char>`**
  - Other string types are not uncommon, e.g., **`std::wstring`** is **`basic_string<wchar_t>`** where **`wchar_t`** *may* be a wider character with more bits (like Unicode, it’s complicated)

http://www.cplusplus.com/reference/string/string/

Common String Operations

- **s.empty()** is true if the string contains no values
  - **s.clear()** removes all chars from the string
- **s.size()** returns the string length (also **s.length()**)
- **c = s[index]** provides random access by index
  (Warning: undefined if out of range)
  - **c = s.at(index)** is like operator[], but instead throws an std::out_of_range exception if the key isn't in the map
  - **s2 = s.substr(first, len)** returns a len-length substring starting at first
- **s[index] = c** silently overwrites the existing char
  (Warning: undefined if out of range – use **at** to throw exception here, too)
  - **s.erase(first, len)** removes length chars from subscript first
- **s.insert(index, str)** inserts the string at index
- **for(auto& c : s)** iterates over the chars in the string
- **s.begin()** and **s.end()** return "iterators" to the first and one past the last char, which behave like pointers

Sequence Adapters
stack and queue

- **Stack** provides a Last-In, First-Out (LIFO) stack
  - Very fast single-ended push and pop only
  - By default, stack is a façade for a **deque**
  - Can be specified to wrap a **vector** or **list** instead, e.g.,
    `stack<int, std::vector<int>> lifo_of_ints;`
- **Queue** provides a First-In, First-Out (FIFO) stack
  - Very fast opposite-ended push and pop only
  - By default, queue is a façade for a **deque**
  - Can be specified to wrap a **list** instead, e.g.,
    `stack<int, std::list<int>> fifo_of_ints;`
- **Priority_queue** provides a stack for which the *highest priority* item is always popped next
  - A comparator method is needed such as `operator<`

http://www.cplusplus.com/reference/stack/stack/
http://www.cplusplus.com/reference/queue/queue/
http://www.cplusplus.com/reference/queue/priority_queue/

Associative Containers
set and map

- **set** is a collection of keys, sorted by keys (like Java's TreeSet)
  - Essentially a non-sequenced vector of objects with duplicates automatically removed, and always sorted
- **map** is a collection of key-value pairs sorted by keys (like Java's TreeMap)
  - Essentially a vector of objects with (almost) any type as the key, which is always sorted
  - Elements are accessed like a vector, e.g., `m["bar"]`
- As with sort and priority_queue, the key type needs a comparator method such as `operator<`

http://www.cplusplus.com/reference/set/set/
http://www.cplusplus.com/reference/map/map/

Common Set Operations

- **s.empty()** is true if the set contains no values
  - **s.clear()** removes all values from the set
- **s.size()** returns the number of values in the set
- Random access (indexing) isn't supported
- **s.insert(value)** adds value to the set
  - **s.count(value)** returns 1 if value exists, 0 otherwise
  - **s.erase(value)** removes value from the set
- **for(auto& value : s)** iterates over the set values
- **s.begin()** and **s.end()** return "iterators" to the first and one past the last value, which behave like pointers

Common Map Operations

- **m.empty()** is true if the map contains no pairs
  - **m.clear()** removes all pairs from the map
- **m.size()** returns the number of pairs in the map
- **val = m[key]** provides random access by key, adding a default constructor value to the map if the key isn't in the map
  - **m.at(key)** is like operator[], but instead throws an std::out_of_range exception if the key isn't in the map
- **m[key] = x** silently overwrites the existing value (if any) for a key
  - **m.count(key)** returns 1 if the key exists, 0 otherwise
  - **m.erase(key)** removes the key and associated value from the map
- **for(auto& [key, val] : m)** iterates over the map, setting variables key and value to the (key, value) pair of each map entry
- **m.begin()** and **m.end()** return "iterators" to the first and one past the last key/value pair, which behave like pointers

Simple Set Example
List Unique Arguments in Sort Order
(Discussed in Lecture 21)

```cpp
#include <set>
#include <iostream>

int main(int argc, char* argv[]) {
    std::set<std::string> words;
    for(int i=1; i<argc; ++i)
        words.insert(std::string{argv[i]});
    std::cout << "Here are the unique arguments in alphabetical order:" << std::endl;
    for(auto word : words)
        std::cout << "  " << word << std::endl;
}
```

```
ricegf@antares:~/dev$ ./a.out This is a good time for a great time just in time
Here are the unique arguments in alphabetical order:
  This
  a
  for
  good
  great
  in
  is
  just
  time
ricegf@antares:~/dev$
```

Simple Set Example

Search for Words in a Set

(Discussed in Lecture 21)

```
#include <set>
#include <iostream>

int main(int argc, char* argv[]) {
    std::set<std::string> words;
    for(int i=1; i<argc; ++i)
        words.insert(std::string{argv[i]});

    std::string word;
    while(true) {
        std::cout << "Search for which word in arguments? ";
        std::cin >> word;
        if(word.empty()) break;
        std::cout << word << ((words.count(word) == 0) ? " is not " : " is ")
                  << "in the argument list" << std::endl;
    }
}
```

```
ricegf@antares:~/dev$ ./a.out when in the course of human events
Search for which word in arguments? course
course is in the argument list
Search for which word in arguments? despot
despot is not in the argument list
Search for which word in arguments? despot
despot is not in the argument list
Search for which word in arguments?
```

Another Set Example

Random Search for Primes

(Discussed in Lecture 21)

```cpp
#include <set>
#include <iostream>
#include <cmath>

// Returns true if "number" is a prime number
bool is_prime (int number) {
    if (number < 2) return false;
    for (int i=2; i <= std::sqrt(number); ++i) {
        if ((number % i) == 0) return false;
    }
    return true;
}
int main() {
    std::set<int> s;
    for(int i=1; i<=100; ++i) {
        int x = rand()%100;
        if(is_prime(x))
            s.insert(x);
    }
    for(auto i : s) std::cout << i << '\n';
}
```

```
ricegf@antares:~/dev$ ./a.out
2
3
5
11
13
19
23
29
37
43
59
67
73
83
ricegf@antares:~/dev$
```

Simple Map Example
Student Grades
(Discussed in Lecture 20)

```cpp
#include <iostream>
#include <vector>
#include <map>

typedef std::vector<int> Grades;
std::ostream& operator<<(std::ostream& ost, const Grades& grades) {
    for (int grade : grades) ost << grade << ' ';
    return ost;
}

typedef std::string Student;
typedef std::map<Student, Grades> Gradebook;
std::ostream& operator<<(std::ostream& ost, const Gradebook& gradebook) {
    for (const auto& [student, grades] : gradebook)
        std::cout << "Student " << student << " grades: " << grades << std::endl;
    return ost;
}

int main() {
    Gradebook gradebook = {
        {"Li",     {100,98}       },
        {"Ajay",   {98,88,92,100}  },
        {"Juan",   {91,73,110,100} },
        {"Sophia", {77,69,75,84,91}},
    };

    std::cout << gradebook << std::endl;
}
```

Map Example
Enum Class to String
(Discussed in Lecture 20)

multimap_vehicle.cpp

```
#include <iostream>
#include <vector>
#include <map>

// This is the enumeration that needs to convert to a string
enum class Car {Civic, Camry, CRV, RAV4, Rogue, Silverado};

// A map is much more natural for to_string conversion of enum classes
std::map<Car, std::string> car_to_string = {
    {Car::Civic,     "Civic"},
    {Car::Camry,     "Camry"},
    {Car::CRV,       "CRV"},
    {Car::RAV4,      "RAV4"},
    {Car::Rogue,     "Rogue"},
    {Car::Silverado, "Silverado"},
};
std::ostream& operator<<(std::ostream& ost, const Car& car) {
    ost << car_to_string[car];
    return ost;
}
```

Associative Containers
multiset and multimap

- **multiset** and **multimap** are similar to set and map, but permit duplicate keys (Java lacks “multi” equivalents)
  - For example, a multiset of chars would allow you to store every character in a document separately, then use `std::count` to determine how many
  - Access is a bit trickier – `mm[“bar”]` is ambiguous – so iterators are needed to access elements (coming shortly)
- As with set and map, you may need to provide a comparator method such as `operator<`

http://www.cplusplus.com/reference/set/multiset/
http://www.cplusplus.com/reference/map/multimap/

Unordered Associative Containers
Unordered_*

- **Unordered_set, unordered_map, unordered_multiset, unordered_multimap,** similar to the ordered versions (**`Unordered_set`** is like Java's **`HashSet`** and **`Unordered_map`** is like Java's **`HashMap`**)
  - Because they are not sorted, insertions are slightly faster but lookups are slightly slower
  - No comparator operator is required
  - I've never actually needed one of these in C++

http://www.cplusplus.com/reference/unordered_set/
http://www.cplusplus.com/reference/unordered_map/

For the Exam

- Be able to *code* with
  - **`std::vector`**
  - **`std::map`**
  - **`std::set`**
  - **`std::string`**

Common Operations
(Know the methods but not the footnotes)

	std::vector	std::string	std::set	std::map
Is it empty?	**v.empty()**	**s.empty()**	**s.empty()**	**m.empty()**
Clear	**v.clear()**	**s.clear()**	**s.clear()**	**m.clear()**
How many?	**v.size()**	**s.size() or s.length()**	**s.size()**	**m.size()**
Random access	**val = v[index]**	**c = s[index]**	--	**val = m[key]**
Throw if bad index	**val = v.at(index)**	**c = s.at(index)**	--	**val = m.at(key)**
Value or key exists?	**std::count**	**std::count**	**s.count(val)**[1]	**m.count(key)**
Overwrite value	**v[index] = val**	**s[index] = c**	**s.insert(val)**	**m[key] = val**
Throw if bad index	**v.at(index) = val**	**s.at(index) = c**	--[2]	**m.at(key) = val**
Erase value	**v.erase(it)**	**s.erase(index, len)**	**s.erase(val)**	**m.erase(key)**
Insert value	**s.insert(val, it)**	**s.insert(index, str)**	**s.insert(val)**	**--** (available but omitted)
Iterate	**for(auto& val : v)**	**for(auto& c : s)**	**for(auto& val : s)**	**for(auto& [key,val] : m)**
Get iterator to first	**it = v.begin()**	**it = s.begin()**	**it = s.begin()**	**it = m.begin()**[3]
Get iterator to last+1	**it = v.end()**	**it = s.end()**	**it = s.end()**	**it = m.end()**[3]

v = vector s = string or set m = map
val = value it = iterator c = char

[1]Or (in C++ 20 or later) **`s.contains(val)`**
[2]**s.insert** succeeded if **`result.second`** is true
[3]**`it->first`** for key, **`it->second`** for value

Reminder
Class Survey

- The class survey is now in progress
  - I see *no* feedback until *after* your final grades are posted
  - I read and consider *every* comment!
  - **Completely anonymous**
- WARNING: Survey closes on April 29!

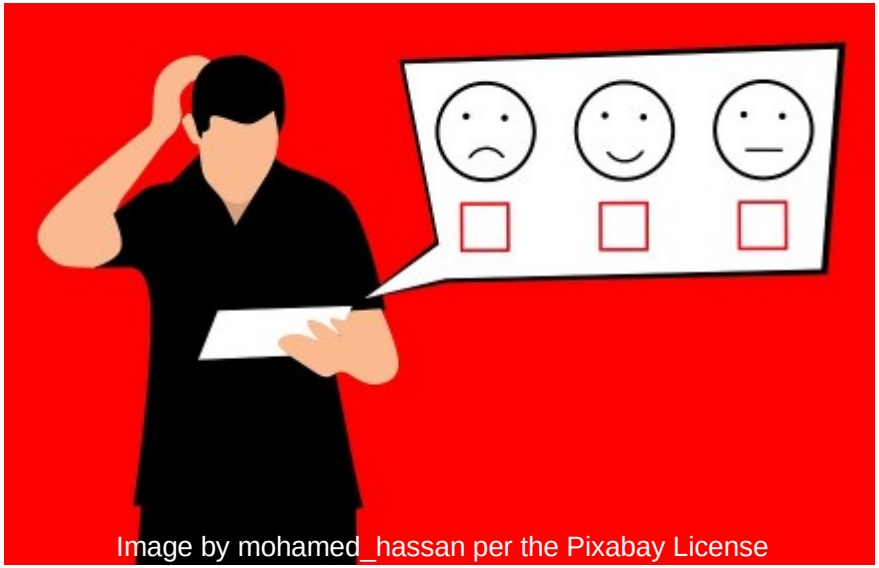
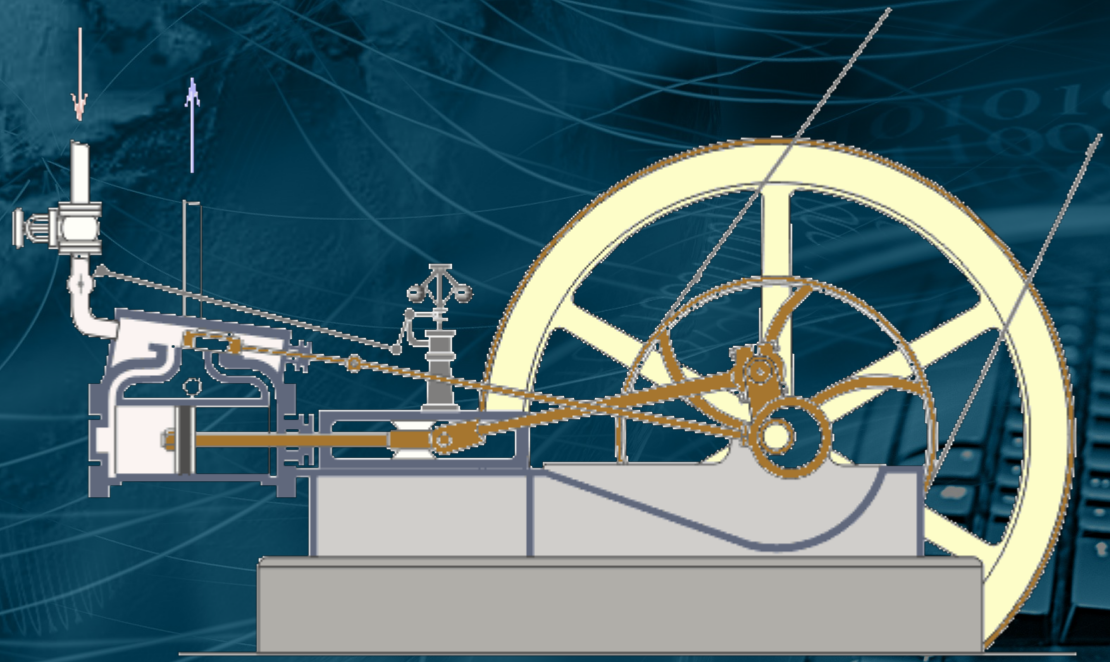


Image by mohamed_hassan per the Pixabay License
https://pixabay.com/en/experience-feedback-survey-customer-3239623/

Benefit for You: The nag screens and reminder emails *may* cease once you take the survey plus it's the right thing to do!

Benefit for Me: Invaluable insight into what worked and what needs to change.

Iterators

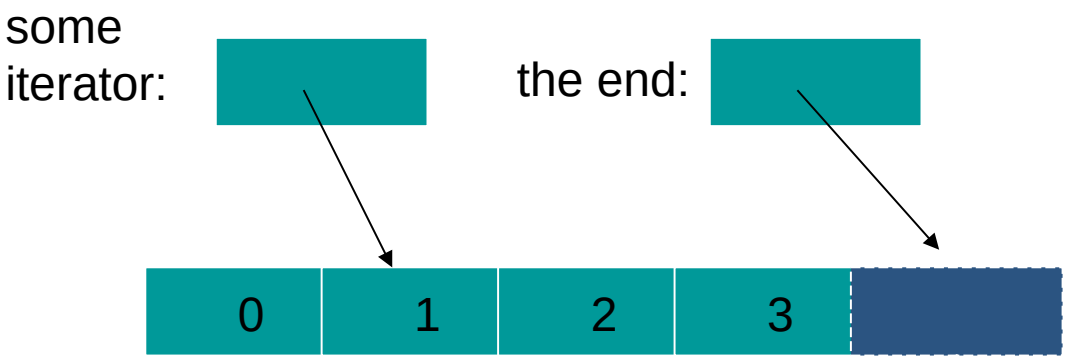


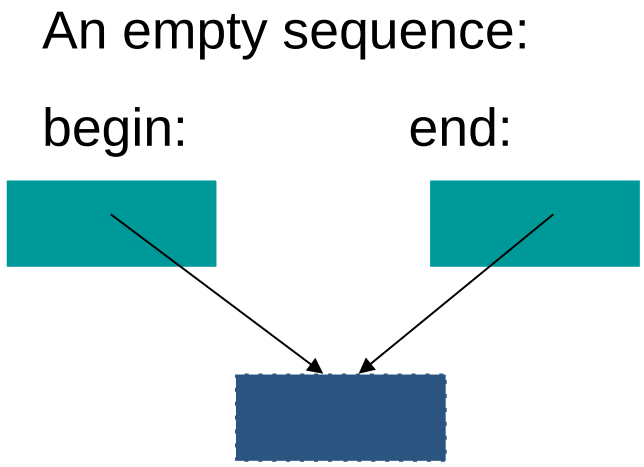
Iterators

- **Iterator**: A pointer-like instance of a nested class used to access items managed by the outer class instance
- Two complementary nested iterator classes are typically provided by a container
  - `iterator` allows the dereferenced item to be modified
  - `const_iterator` prevents modification to the dereferenced item
- Obtain the iterators with container methods
  - Get an `iterator` with `begin()` and `end()`
  - Get a `const_iterator` with `cbegin()` and `cend()`
- Use iterators much like pointers
  - An iterator can always be incremented via `++`, dereferenced with `*`, and compared to other iterators

Algorithms and Iterators

- An iterator object points to one element of a container
- The iterator iterates through the container one element at a time
- The end of the sequence is “one past the last element”
  - *Not* “the last element”
  - That’s necessary to elegantly represent an empty sequence
  - One-past-the-last-element isn’t an element
    - You can compare an iterator pointing to it
    - You can’t dereference it (read its value)
- Returning the end of the sequence is the standard idiom for “not found” or “unsuccessful”





Iterating Through a Vector

With C++ Iterators

iteration.cpp

```cpp
#include <vector>
#include <iostream>

int main() {
  std::vector<int> v = {1, 2, 3, 4, 5};

  std::vector<int>::iterator it = v.begin();

  do {
      std::cout << *it << std::endl;
  } while(++it != v.end());
}
```

The iterator is a nested class inside the container into which it points.

The begin() method returns an iterator pointing to the first element.

You may treat it almost exactly like a pointer!

The end() method returns an iterator pointing one *past* the last element.

```
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/iterators$ make iteration
g++ --std=c++17 -o iteration iteration.cpp
Now type ./iteration to execute the result

ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/iterators$ ./iteration
1
2
3
4
5
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/iterators$
```

NOT Iterating Through a *Constant* Vector

With C++ Iterators

bad_iteration.cpp

```cpp
#include <vector>
#include <iostream>

int main() {
  const std::vector<int> v = {1, 2, 3, 4, 5};

  std::vector<int>::iterator it = v.begin();

  do {
      std::cout << *it << std::endl;
  } while(++it != v.end());
}
```

If the vector is constant, we can't use iterators with it. Iterators may modify the container to which they point.

```
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/iterators$ make bad_iteration
This compilation SHOULD fail - demonstrating const_iterator

g++ --std=c++17 -o bad_iteration bad_iteration.cpp
bad_iteration.cpp: In function 'int main()':
bad_iteration.cpp:8:34: error: conversion from '__normal_iterator<const int*,[...]>' to non-scalar type
'__normal_iterator<int*,[...]>' requested
    8 |   Integers::iterator it = v.begin();
      |                           ~~~~~~~^~
make: [Makefile:11: bad_iteration] Error 1 (ignored)
Now type ./bad_iteration to execute the result
```

Iterating Through a *Constant* Vector

With C++ Const_Iterators

const_iteration.cpp

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

int main() {
  const std::vector<int> v = {1, 2, 3, 4, 5};

  std::vector<int>::const_iterator it = v.cbegin();

  do {
      std::cout << *it << std::endl;
  } while(++it != v.cend());}
```

Using const_iterator, cbegin(), and cend() solves the problem.

```
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/iterators$ make const_iteration
g++ --std=c++17 -o const_iteration const_iteration.cpp
Now type ./const_iteration to execute the result

ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/iterators$ ./const_iteration
1
2
3
4
5
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/iterators$
```

Simple Map::Iterator Example

Student Grades

(Discussed in Lecture 20)

```
typedef std::vector<int> Grades;
std::ostream& operator<<(std::ostream& ost, const Grades& grades) {
    for (int grade : grades) ost << grade << ' ';
    return ost;
}

typedef std::string Student;
typedef std::map<Student, Grades> Gradebook;
std::ostream& operator<<(std::ostream& ost, const Gradebook& gradebook) {
//  for (const auto& [student, grades] : gradebook)
//      std::cout << "Student " << student << " grades: " << grades << std::endl;

    for(auto it = gradebook.cbegin(); it != gradebook.cend(); ++it)
        std::cout << "Student " << it->first << " grades: " << it->second << std::endl;

    return ost;
}

int main() {
    Gradebook gradebook = {
        {"Li",     {100,98}         },
        {"Ajay",   {98,88,92,100}   },
        {"Juan",   {91,73,110,100}  },
        {"Sophia", {77,69,75,84,91}},
    };

    std::cout << gradebook << std::endl;
}
```

For-each rewritten as 3-term for with iterators!
(Using cbegin / cend since gradebook is const)

When using iterators with maps,
the key is (inexplicably) **`first`**
and the value is **`second`**

Common Iterator Methods

The methods in green are the most used!

- All **iterators** must provide:
  - Destructor, copy constructor, and copy assignment operator (it1 = it2)
  - Increment (++it)
  - Deferenced access (x = *it)
- Input iterators add:
  - Comparisons (it1 == it2 and it1 != it2)
- Output iterators add:
  - Dereferenced assignment (e.g., *it = x)
- Forward iterators add:
  - Default constructor
- Bidirectional iterators add:
  - Decrement (--it)
- Random access iterators add:
  - Pointer math (it+3, it-2, it+=5, it[4]), comparison (it1 < it2 and so on)

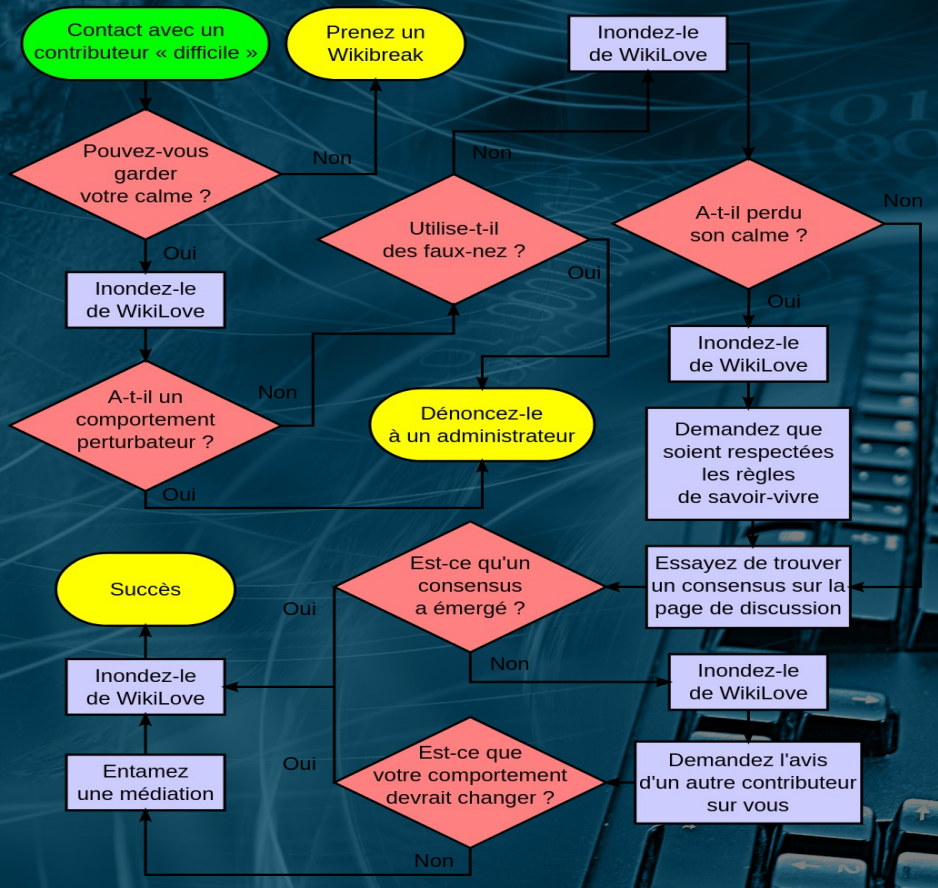
Containers that support iterators usually provide (x is item, p is iterator):

- begin(), cbegin() - returns p to first item
- end(), cend() - returns p to 1 past last item
- size() - number of elements
- empty() - true if no elements
- push_back(x) / push_front(x) – insert x at end / beginning, respectively
- insert(p, x) – insert x immediately before p
- front(), back() - first / last item, respectively
- pop_back() / pop_front() – delete at end / beginning, respectively
- erase(p) – remove item at p

For the Exam

- Be able to *obtain* (using **`begin`** / **`end`**) and *code* using iterators
  - Also know when and be able to use a const_iterator (using **`cbegin`** and **`cend`**)
  - Be able to perform pointer math with iterators and const_iterators (++it, it+25, it->method and such)
  - Know the iterator members / operators and the container members in **green** on the previous slide

Algorithms



Algorithms

- The <algorithm> library hosts 87 algorithm templates for use with STL containers, including
  - **Search**: find, find_if, count, count_if, distance, equal, search…
  - **Modify**: copy, move, swap, transform, replace, fill, generate, reverse, rotate, random_shuffle...
  - **Partition and Sort**: sort, partial_sort, is_sorted, partition, partition_copy, is_partitioned...
  - **Merge**: merge, inplace_merge, includes...
  - **Heap**: push, pop, make, sort, is_heap...
  - **Min/Max**: min, max, minmax, min_element…
- We'll examine just a few to give you the flavor

Count

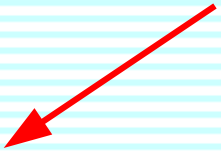
count.cpp

```
#include <algorithm>
#include <iterator>
#include <iostream>

int main() {
    std::vector<int> v;
    for(int i=0; i<100; ++i) {
        v.push_back(std::rand()%10);
        std::cout << v.back() << ' ';
    }
    std::cout << std::endl;

    int target;
    std::cout << "What shall we count today? ";
    while(std::cin >> target) {
        int number = std::count(v.begin(), v.end(), target);
        std::cout << "Found " << number << " elements matching " << target
                  << "\nNext? ";
    }
    std::cout << std::endl;
}
```

Like most algorithms, std::count only looks between the first and second-1 iterators



Count

count.cpp

```
#include <algorithm>
#include <iterator>
#include <iostream>
```

```
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/algorithms$ make count
g++ --std=c++17    count.cpp   -o count
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/algorithms$ ./count
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6 1 8 7 9 2 0 2 3 7 5 9 2 2 8 9 7 3 6 1 2 9 3 1 9 4 7 8 4 5 0 3 6
1 0 6 3 2 0 6 1 5 5 4 7 6 5 6 9 3 7 4 5 2 5 4 7 4 4 3 0 7 8 6 8 8 4 3 1 4 9 2 0 6 8 9 2 6 6 4 9
What shall we count today? 4
Found 10 elements matching 4
Next? 0
Found 8 elements matching 0
Next?
```

```
    while(std::cin >> target) {
        int number = std::count(v.begin(), v.end(), target);
        std::cout << "Found " << number << " elements matching " << target
                  << "\nNext? ";
    }
    std::cout << std::endl;
}
```

Find

std::find and std::distance

find.cpp

```cpp
#include <algorithm>
#include <iterator>
#include <iostream>

int main() {
    std::vector<int> v;
    for(int i=0; i<100; ++i) {
        v.push_back(std::rand()%10);
        std::cout << v.back() << ' ';
    }
    std::cout << std::endl;

    int target;
    std::cout << "What shall we find today? ";
    while(std::cin >> target) {
        auto it = std::find(v.begin(), v.end(), target);
        if(it == v.end()) std::cout << target << " not found!" << std::endl;
        else std::cout << "Found first " << target << " at v["
                       << std::distance(v.begin(), it) << "]\nNext? ";
    }
    std::cout << std::endl;
}
```

Find returns an iterator to the first matching element between the first and second iterators. If not found, the returned iterator is v.end().

std::distance returns the number of elements between the two iterators.
The distance from begin() is just the index!

Find

std::find and std::distance

find.cpp

```
#include <algorithm>
#include <iterator>
#include <iostream>
```

```
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/algorithms$ make find
g++ --std=c++17 find.cpp -o find
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/algorithms$ ./find
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6 1 8 7 9 2 0 2 3 7 5 9 2 2 8 9 7 3 6 1 2 9 3 1 9 4 7 8 4 5 0 3 6
1 0 6 3 2 0 6 1 5 5 4 7 6 5 6 9 3 7 4 5 2 5 4 7 4 4 3 0 7 8 6 8 8 4 3 1 4 9 2 0 6 8 9 2 6 6 4 9
What shall we find today? 1
Found first 1 at v[9]
Next? 4
Found first 4 at v[44]
Next?
```

```
    while(std::cin >> target) {
        auto it = std::find(v.begin(), v.end(), target);
        if(it == v.end()) std::cout << target << " not found!" << std::endl;
        else std::cout << "Found first " << target << " at v["
                       << std::distance(v.begin(), it) << "]\nNext? ";
    }
    std::cout << std::endl;
}
```

std::distance returns the number of elements between the two iterators.
The distance from begin() is just the index!

Find All

std::find and iterator math

findall.cpp

```cpp
#include <algorithm>
#include <iterator>
#include <iostream>

int main() {
    std::vector<int> v;
    for(int i=0; i<100; ++i) {
        v.push_back(std::rand()%10);
        std::cout << v.back() << ' ';
    }
    std::cout << std::endl;

    int target;
    std::cout << "What shall we find today? ";
    while(std::cin >> target) {
        auto it_next = v.begin();
        while(it_next != v.end()) {
            auto it = std::find(it_next, v.end(), target);
            if(it == v.end()) break;
            else std::cout << "Found " << target << " at v["
                           << std::distance(v.begin(), it) << "]\n";
            it_next = it+1;
        }
        std::cout << "Next search? ";
    }
    std::cout << std::endl;
}
```

Iteratively find each matching element by starting the search just after the last element found

Advance past the element that was last found

Find All

std::find and iterator math

findall.cpp

```
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/algorithms$ make findall
g++ --std=c++17 findall.cpp -o findall
ricegf@antares:~/dev/202401/24-c++-std-template-lib/code_from_slides/algorithms$ ./findall
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6 1 8 7 9 2 0 2 3 7 5 9 2 2 8 9 7 3 6 1 2 9 3 1 9 4 7 8 4 5 0 3 6
1 0 6 3 2 0 6 1 5 5 4 7 6 5 6 9 3 7 4 5 2 5 4 7 4 4 3 0 7 8 6 8 8 4 3 1 4 9 2 0 6 8 9 2 6 6 4 9
What shall we find today? 3
Found 3 at v[0]
Found 3 at v[4]
Found 3 at v[14]
Found 3 at v[27]
Found 3 at v[36]
Found 3 at v[41]
Found 3 at v[50]
Found 3 at v[55]
Found 3 at v[68]
Found 3 at v[78]
Found 3 at v[86]
Next search?
```

```
#include <algorithm>
#include <iterator>

            else std::cout << "Found " << target << " at v[
                           << std::distance(v.begin(), it) << "]\n";
            it_next = it+1;
        }
        std::cout << "Next search? ";
    }
    std::cout << std::endl;
}
```

Advance past the element that was last found

Finding a Map Key

- Using **`std::find`** on a **`std::map`** will search for a **`std::pair<key, value>`**, but we often want to search just on a *key* To find the associated *value*.
- So **`std::map`** provides a find *method* that does just that.

```
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <vector>

int main() {
    std::map<Last, First> americans {{"Washington", "George"}, … };
    const int per_line = 3;
    int people_counter = 0;
    for(auto& [last, first] : americans) {
        std::cout << std::left << std::setw(27) << (last + ", " + first)
            << ((++people_counter % per_line == 0) ? "\n" : "");
    }
    Last target;
    std::cout << "\nWho shall we find today (last name)? ";
    while(std::getline(std::cin, target)) {
        auto it = americans.find(target); // get iterator to pair matching the key
        if(it == americans.end())         // returns end() if not found
            std::cout << target << " not found!\n\nNext? ";
        else
            std::cout << it->second << " " << it->first << "\n\nNext? ";
    } //                  the value, First     the key, Last
    std::cout << std::endl;
```

Finding a Map Key

- Using **`std::find`** on a **`std::map`** will search for a **`std::pair<key, value>`**, but we often want to search just on a *key* To find the associated *value*.
- So **`std::map`** provides a find *method* that does just that.

```
#include <algorithm>
#include <iostream>
#incl
#incl

int ma
    st
    co
    in
    fo


    }
    La
    st
    wh
        auto it = americans.find(target); // get iterator to pair matching the key
        if(it == americans.end())         // returns end() if not found
            std::cout << target << " not found!\n\nNext? ";
        else
            std::cout << it->second << " " << it->first << "\n\nNext? ";
    } //                 the value, First     the key, Last
    std::cout << std::endl;
```

```
Anthony, Susan B.          Bull, Sitting              Carver, George Washington
Chavez, Cesar              Curie, Marie               Einstein, Albert
Ginsburg, Ruth Bader       Jackson, Ketanji Brown     Lincoln, Abraham
Muhammad, Ibtihaj          Nadella, Satya             Reagan, Ronald
Ride, Sally                Roosevelt, Eleanor         Su, Lisa
Tubman, Harriet            Turing, Alan               Washington, George

Who shall we find today (last name)? Einstein
 Albert Einstein

Next? Rice
Rice not found!

Next?
```

Shuffle and Sort

std::random_shuffle and std::sort

shuffle_and_sort.cpp

```cpp
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <vector>

int main() {
    // Fill an array
    std::vector<int> v;
    for(int i=0; i<20; ++i) v.push_back(i);
    std::cout << "Original:   ";
    for(auto i : v) std::cout << std::setw(3) << i;
    std::cout << std::endl;

    // Shuffle the ints randomly
    std::random_shuffle(v.begin(), v.end());
    std::cout << "Shuffled:   ";
    for(auto i : v) std::cout << std::setw(3) << i;
    std::cout << std::endl;

    // Sort them back into order
    std::sort(v.begin(), v.end());
    std::cout << "Sorted:     ";
    for(auto i : v) std::cout << std::setw(3) << i;
    std::cout << std::endl;
```

Shuffle (like a card deck) the container contents.

Sort them back into order based on operator<.

```
Original:     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
Shuffled:     4 10 11 15 14 16 17  1  6  9  3  7 19  2  0 12  5 18 13  8
Sorted:       0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
```

For the Exam

- Be able to *code* using **`std::count`**, **`std::find`**, **`std::distance`**, and **`std::sort`**.
- Understand the *concept* of iteratively finding all elements matching the search key with **`std::find`** (“start subsequent searches just after the previously found element”) but you won’t be asked for code

STL Summary

- We briefly explored 3 of the 4 types of STL resources
- **Containers** for managing data
  - vector, array, stack, queue, set / map (multi and unordered variants)
- Each container's **Iterators** for manipulating data
  - iterator and const_iterator, very similar to pointers
- A few of the almost 100 **Algorithms** that can be applied to most iterators with many variations
  - find, find_if, generate, transform, random_shuffle, sort, minmax, search, count, for_each, copy, fill, reverse, merge, hash

For the Exam

- **Containers**
  - Be able to *code* with **`std::vector`**, **`std::map`**, **`std::set`**, **`std::string`**
  - Be able to *iterate* using a for-each, get with **`[]`** and **`at`**, and overwrite with **`[]`**
  - Be able to *code* using **`empty`**, **`clear`**, **`size`**, **`insert`**, **`erase`**, **`count`**, **`push_back / push_front`**, **`front / back`**, and **`pop_front / pop_back`**
- **Iterators**
  - Be able to *obtain* using **`begin / cbegin`**, **`end / cend`** and *code* using **`it1 = it2`**, **`++it`**, **`x = *it`**, **`*it = x`**, **`it1 == it2`**, **`it1 != it2`**
- **Algorithms**
  - Be able to *code* using **`std::find`** (and map's **`find(key)`** *method*), **`std::distance`**, **`std::count`**, **`std::random_shuffle`**, and **`std::sort`**.
  - Understand the *concept* of iteratively finding all elements matching the search key with **`std::find`** ("start subsequent searches just after the previously found element") but you won't be asked to code this on the exam