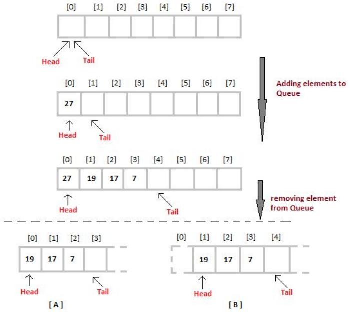
STL - Principles and Practice

Victor Ciura - Technical Lead, Advanced Installer
Gabriel Diaconița - Senior Software Developer, Advanced Installer
http://www.advancedinstaller.com
CAPHYON

Agenda

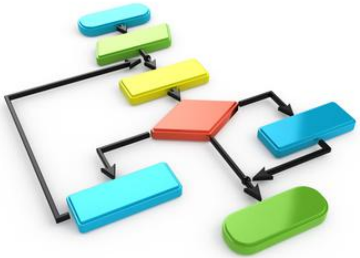
Part 0: STL Intro.

Part 1: Containers and Iterators



Part 2: STL Function Objects and Utilities

Part 3-4: STL Algorithms Principles and Practice



Part 0:

STL Introduction

STL Short History

- Early on (‘70s) Stepanov recognized the full potential for generic programming (first implementations in **Ada**)
- In 1990, **Alex Stepanov** and **Meng Lee** of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the **S**tandard **T**emplate **L**ibrary.
- This brilliant work was recognized by **Andrew Koenig** who led efforts for its introduction to the ISO C++ committee for standardization.
- Documentation and implementation work was completed with the help of **David Musser**.
- In 1994, STL was adopted as part of ANSI/ISO Standard C++.
- STL adoption was helped by HP's decision to make its implementation (Stepanov) freely available on the Internet (1994).

STL and Its Design Principles

Generic Programming

- algorithms are associated with a **set of common properties**

  Eg. op { +, *, min, max } => associative operations => reorder operands

  => parallelize + reduction (std::accumulate)

- find the most general representation of algorithms (**abstraction**)
- exists a **generic algorithm** behind every WHILE or FOR loop

Alexander Stepanov (2002), https://www.youtube.com/watch?v=COuHLky7E2Q

STL and Its Design Principles

Generic Programming

- specify correct and **complete interfaces**
  (eg. binary search should return the insertion point)
- look for interface **symmetry** (eg. stable_sort, stable_partition)
- **Iterators** are good (addresses are real in the machine)
  => ability to refer data through some handle
- Iterators should have fast comparison and dereferencing
- the STL library should be (easily) **extended** with other algorithms & data structures

Alexander Stepanov (2002), https://www.youtube.com/watch?v=COuHLky7E2Q

STL Data Structures

- they implement whole-part semantics (copy is deep - members)
- 2 objects never intersect (they are separate entities)
- 2 objects have separate lifetimes
- STL algorithms work only with ***Regular*** data structures
- **Semiregular** = *Assignable* + *Constructible* (both *Copy* and *Move* operations)
- **Regular** = Semiregular + *EqualityComparable*
- => STL assumes **equality** is always defined (at least, equivalence relation)

Abstraction	***Data type***	***Concept, abstract algorithm***
What it is	Interface (specification, encapsulated implementation)	Semantic properties, algorithms they enable
Focus	Data structures	Algorithms
What's protected	Representation invariant	Generality of algorithm
Who	Parnas, Hoare, Liskov & Zilles, Guttag, Musser, … (870 papers by 1983)	Stepanov and his collaborators: Kapur, Musser, Kershenbaum, Lee; Scheme, Ada, C++

Paul McJones

Generic Programming Drawbacks

- abstraction penalty
- implementation in the interface
- early binding
- horrible error messages (*no formal specification* of interfaces, **yet**)
- duck typing
- algorithm could work on some data types, but fail to work/compile on some other new data structures (different iterator category, no copy semantics, etc)

We need to fully specify requirements on algorithm types => **Concepts**

Part 1:

Containers and Iterators

Containers

- STL offers an assortment of containers (of different types).
- STL publicizes the **time** and **storage complexity** of its containers (Big-O notation).
- STL containers grow and shrink in size automatically.
- STL provides built-in algorithms for processing containers.
- STL is **extensible** which means that users can add new containers and new algorithms such that:
    - STL algorithms can process STL containers as well as user-defined containers
    - User-defined algorithms can process STL containers as well user-defined containers
- STL provides **iterators** that make the containers and algorithms **generic** and **efficient**.

Containers

- The containers are class **templates**.
- When you declare a container, you specify the **type** of the elements that the container will hold.
- Containers can be constructed with ***initializer lists***.
- They have member functions for adding, removing, accessing elements and other common operations.
- The container manages the **storage** space that is allocated for its elements (they even support custom memory allocators).
- Access to elements is always performed via **iterators**.
- Most containers have at least several member functions in **common**, and share functionalities.
- ***Choosing*** the best container for the particular task depends not only on the offered functionality, but also on its efficiency for different workloads.

Container Categories

- Sequence containers
  - array
  - vector
  - list, forward_list
  - deque
- Associative containers
  - set, multiset
  - map, multimap
- Unordered associative containers (hashed key)
  - unordered_set, unordered_multiset
  - unordered_map, unordered_multimap
- Container adapters
  - stack
  - queue
  - priority_queue

Sequence Containers

- Sequence containers maintain the **ordering** of inserted elements that you specify.
- A **vector** container behaves like an array, but can automatically grow as required.
- An **array** container has some of the strengths of vector, but the length is not flexible.
- A **list** container is a doubly linked list that enables bidirectional access, fast insertions, and fast deletions anywhere in the container, but you cannot randomly access an element in the container.
- A **forward_list** container is a singly linked list - the forward-access version of list.
- A **deque** container allows for fast insertions and deletions at the beginning and end of the container.

Associative Containers

- In associative containers, elements are inserted/kept in a **pre-defined order**.
- A **map**, sometimes referred to as a *dictionary*, consists of key/value pairs.
- A **set** is just an ordered container of unique elements - the value is also the key.
- Both map and set only allow one instance of a key or element to be inserted into the container (for multiple instances, use **multiset** and **multimap**).
- Allow for retrieval of values by key in **logarithmic** time.

Unordered Associative Containers

- They use **hash tables** for fast retrieval and insertion.
- Container keys are **hashed** on insertion.

  (a custom hasher must be provided for user-defined key types).
- STL automatically provides predefined hash functions for **builtin types**.

  (integers, chars, std::string, pointers, etc.)
- In hashed containers, elements are inserted/kept in buckets.
- Allow for very fast retrieval of values by key (in **constant** time).

Containers Adapters

- A container adapter is a variation of a sequence or associative container that restricts the interface for simplicity and clarity (very specialized).
- Container adapters do not support **iterators**.
- A **queue** container follows FIFO (first in, first out) semantics. **push()** **pop()** **front()** **back()**
- A **stack** container follows LIFO (last in, first out) semantics. **push()** **pop()** **top()**
- A **priority_queue** container is organized such that the element that has the highest value (according to a specified predicate) is always first in the queue. **push()** **pop()** **top()**
- They are usually implemented (internally) with **deque**, **list** or **vector**.

Containers
vector<T>

- Similar to a C-array
- A back insertion sequence container (elements are arranged in order of insertion)
- Provides random access iterator
- Provides constant amortized complexity for **push_back()**
- Various constructors:
  - empty constructor: **vector<int> v;**
  - with a specific size: **vector<int> v(10);**
  - copy constructor: **vector<int> v(other);**
  - initializer list: **vector<int> v = { 5, 8, 13, 0, 6 };**
- Working with vector size:
  - get current size: **v.size();**
  - resize vector and add new elements: **v.resize(100,42);**
  - erase elements: **v.erase(v.begin(), v.begin()+5);**
  - clear all elements: **v.clear();**

Containers
vector<T>

- Get iterators for start and end positions: **`v.begin(); v.end();`**
- Get reverse iterators for start and end: **`v.rbegin(); v.rend();`**
- Adding at the end: **`v.push_back(42);`**
- Inserting at a specific position: **`v.insert(v.begin()+5, 42);`**

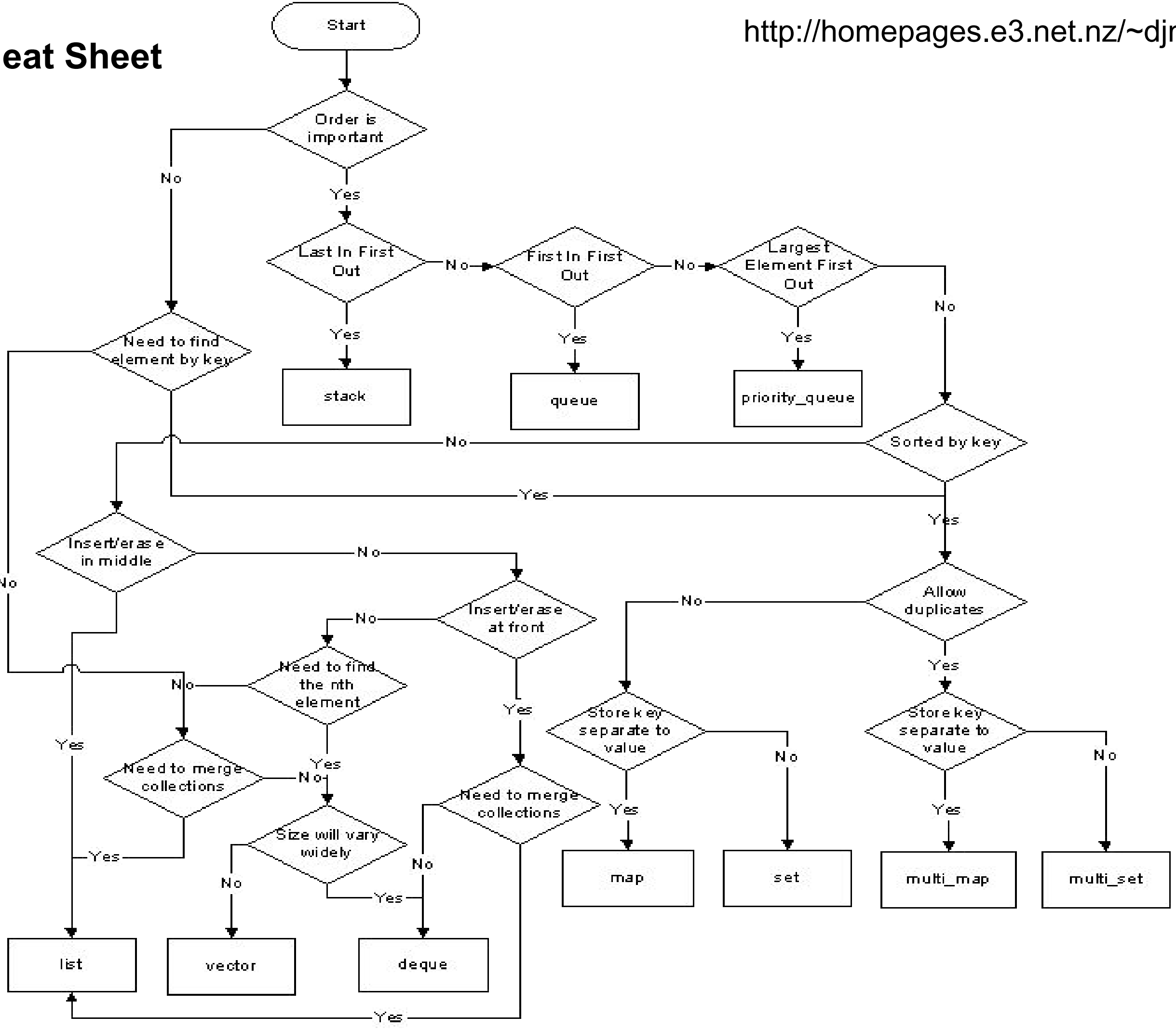
```
for(size_t i = 0; i < v.size(); ++i) { cout << v[i]; }

for(vector<string>::iterator it = v.begin(); it != v.end(); ++it) { cout << *it; }

for(auto it = v.begin(); it != v.end(); ++it) { cout << *it; }

for(auto it = v.begin(), end = v.end(); it != end; ++it) { cout << *it; }

std::for_each(v.begin(), v.end(), [](const auto & val) { cout << val; });

for(const auto & val : v) { cout << val; }
```

Containers
`list<T>`

- A doubly linked list.
- Back insertion sequence (supports both forward and backward operations).
- Various constructors.
- Similar methods like `vector<T>`
- Adding at the beginning: `list.push_front(42);`
- Get reference to the first element: `list.front();`
- Splice the elements of two lists: `list1.splice(list1.end(), list2);`
- Merge elements of two lists: `list1.merge(list2);`
- Sort a list: `list.sort();`
- Make unique elements: `list.unique();`
- Remove all elements matching a specific criteria: `list.remove_if( Predicate() );`

STL Containers Cheat Sheet

http://homepages.e3.net.nz/~djm/cppcontainers.html



SAMPLE: C style array vs std::vector

Scenario: We need to store a non-fixed number of integer values. << Classic approach >>

```
int * numberArray     = new int[currentNumberRoom ];
int currentNumberRoom = 5;  // number of numbers we can store, it will grow as needed
int lastAddedIndex    = -1; // array index of last added number

void addNumber(int number)
{
  if (lastAddedIndex < currentNumberRoom - 1)
  {
    numberArray[++lastAddedIndex] = number; // enough room, just add number
  }
  else // no room, array must grow
  {
    int * moreNumberRoom = new int[currentNumberRoom * 2];              // double the available room
    memcpy(moreNumberRoom, numberArray, currentNumberRoom * sizeof(int)); // copy old numbers in new array
    currentNumberRoom = currentNumberRoom * 2; // we can store twice the numbers now
    numberArray = moreNumberRoom;              // put new numbers in place of old array
    addNumber(nmber);                          // now we can do the insertion
  }
}
.........
int at47 = numberArray[47];
```

Can you spot any issues with this code?

Typo. No harm done, compiler will catch this

Should call delete[] on old array after memcpy, we have a memory leak!

Possible buffer overflow!
Array may have less than 48 elements

SAMPLE: C style array vs std::vector

Scenario: We need to store a non-fixed number of integer values. << C++ STL approach >>

```
std::vector<int> numberVector;
numberVector.reserve(5);

void addNumber(int number)
{
  numberVector.push_back(number);
}

.........

int at47 = numberVector.at(47);
```

Can you sp… s with
thi

- Quicker to write
- Easier to read
- Highly resilient to bugs
- No performance loss
- Code is generic

Will throw `std::out_of_range` exception in case of overflow

Iterators

- **Iterators** are the mechanism that makes it possible to *decouple* **algorithms** from **containers**.
- **Algorithms** are *template functions* parameterized by the **type of iterator**, so they are not restricted to a single type of container.
- An iterator represents an abstraction for a memory address (**pointer**).
- An iterator is an **object** that can iterate over elements in an STL container or range.
- All containers provide iterators so that algorithms can access their elements in a ***standard*** way.

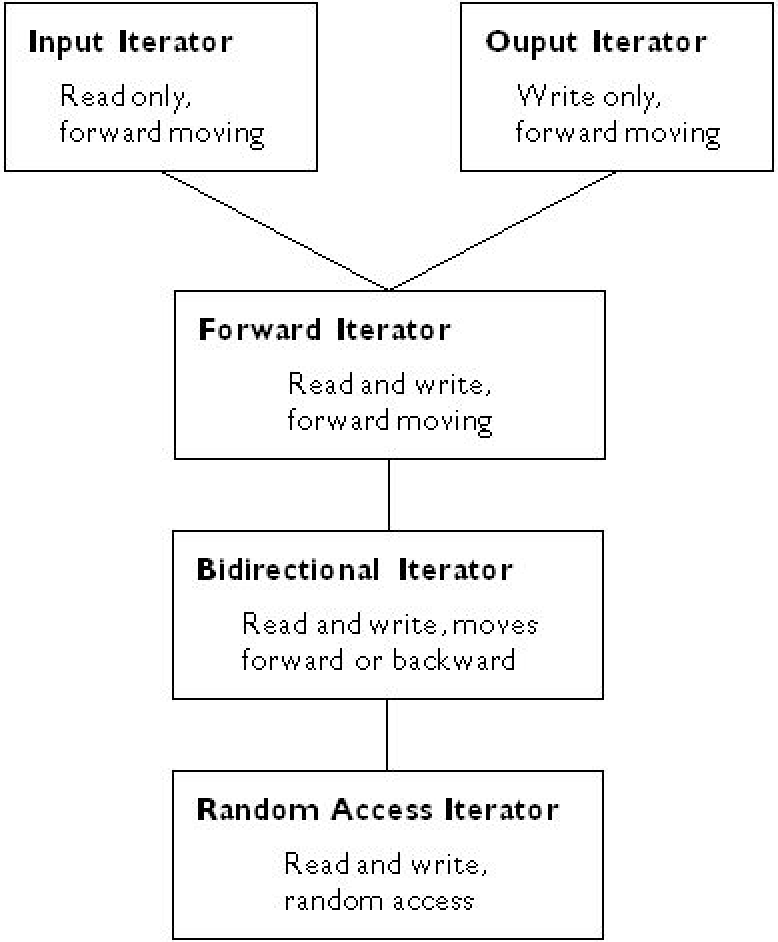
Iterators

- You can use iterator operators such as **++** and **--** to move forward or backward in a range.
- Iterators have different properties and behavior, depending on their **category** (**iterator traits**).
- Instead of being defined by specific ***types***, each category of iterator is defined by the **operations** that can be performed on it.
- There are five kinds of iterators: **`InputIterator, OutputIterator, ForwardIterator, BidirectionalIterator, RandomAccessIterator.`**

Eg.

A **pointer** supports all of the operations required by ***RandomAccessIterator***, so a pointer can be used anywhere a RandomAccessIterator is expected.

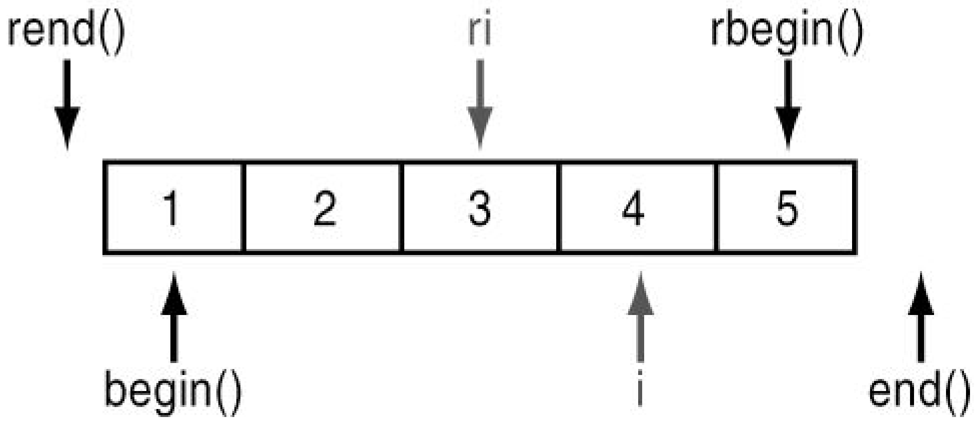
Iterator Categories



Iterators

STL Ranges

- STL ranges are always semi-open intervals: **`[b, e)`**
- Get the beginning of a range/container: **`v.begin();`** or **`begin(v);`**
- You can get a reference to the first element in the range by: **`*v.begin();`**
- You cannot dereference the iterator returned by: **`v.end();`** or **`end(v);`**



SAMPLE: C style iteration vs STL Iterators

Scenario: Refactor existing code so that is prints numbers in reverse order << C approach >>

```
vector<int> numbers = { 1, 549, 3, 52, 6 };
for (unsigned int n = 0; n < numbers.size(); ++n)
  cout << numbers[n] << " ";
```

Output: 1 549 3 52 6

```
vector<int> numbers = { 1, 549, 3, 52, 6 };
for (unsigned int i= numbers.size(); i>= 0; ++i)
  cout << numbers[n] << " ";
```

Can you spot any issues with this code?

Output: ???

Code will execute forever! We just need the decrement operator ...or do we?

Old code forgotten during refactoring. Compiler will catch this

SAMPLE: C style iteration vs STL Iterators

Scenario: Refactor existing code so that is prints numbers in reverse order << STL Iterator approach >>

```
vector<int> numbers = { 1, 549, 3, 52, 6 };
for (auto i = numbers.begin(), endIt = numbers.end(); i != endIt; ++i)
  cout << *it << " ";
```

Output: 1 549 3 52 6

```
vector<int> numbers = { 1, 549, 3, 52, 6 };
for (auto it = numbers.rbegin(), endIt = numbers.rend(); i != endIt; ++it)
  cout << *it << " ";
```

Output: 6 52 3 549 1

Can you spot any issues with this code?

Old code forgotten during refactoring.
Compiler will catch this

SAMPLE: C style iteration vs STL Iterators

Scenario: Refactor existing code so that is prints numbers in reverse order << C++11 range-for approach >>

```
vector<int> numbers = { 1, 549, 3, 52, 6 };
for (auto i : numbers)
  cout << i << " ";
```

Output: 1 549 3 52 6

```
vector<int> numbers = { 1, 549, 3, 52, 6 };
for (auto i : reverse(numbers))
  cout << i << " ";
```

Output: 6 52 3 549 1

Can you spo s with
this

reverse() is an iterator adapter, which will be introduced shortly

Iterator Adaptors

An iterator adapter that helps iterate a collection in reverse order

Eg.

```
  std::vector<int> values;

C style:
  for (int i = values.size() - 1; i >= 0; --i)
    cout << values[i] << endl;

STL+Lambdas:
  for_each( values.rbegin()), values.rend(),
            [](const string & val) { cout << val << endl; } );

Range-for, using adapter:
  for ( auto & val : reverse(values) ) { cout << val << endl; }
```

Iterator Adaptors

An iterator adapter that helps iterate a collection in reverse order

```
namespace detail
{
  template <typename T>
  struct reversion_wrapper
  {
    T & mContainer;
  };
}
/**
 * Helper function that constructs
 * the appropriate iterator type based on ADL.
 */
template <typename T>
detail::reversion_wrapper<T> reverse(T && aContainer)
{
  return { aContainer };
}
```

Iterator Adaptors

An iterator adapter that helps iterate a collection in reverse order

```
namespace std
{
  template <typename T>
  auto begin(detail::reversion_wrapper<T> aRwrapper)
  {
    return rbegin(aRwrapper.mContainer);
  }

  template <typename T>
  auto end(detail::reversion_wrapper<T> aRwrapper)
  {
    return rend(aRwrapper.mContainer);
  }
}
```

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

Eg.

```
std::map<int, string> m;

for_each( MakeSecondIterator(m.begin()), MakeSecondIterator(m.end()),
          [](const string & val) { cout << val << endl; } );

for ( auto & v : IterateSecond(m) ) { cout << val << endl; }
```

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

```
template <typename Iter>
class MapSecondIterator : public std::iterator<std::bidirectional_iterator_tag,
                                               typename Iter::value_type::second_type>
{
public:
  MapSecondIterator() {}
  MapSecondIterator(Iter aOther) : i(aOther) {}

  inline MapSecondIterator & operator++()    {...}
  inline MapSecondIterator   operator++(int) {...}
  inline MapSecondIterator & operator--()    {...}
  inline MapSecondIterator   operator--(int) {...}

  inline bool operator==(MapSecondIterator aOther) const {...}
  inline bool operator!=(MapSecondIterator aOther) const {...}

  inline reference operator*()   { return  i->second; }
  inline pointer   operator->()  { return &i->second; }

private:
  Iter i;
};
```

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

```
/**
 * Helper function that constructs
 * the appropriate iterator type based on ADL.
 */
template <typename Iter>
inline MapSecondIterator<Iter> MakeSecondIterator(Iter aIter)
{
  return MapSecondIterator<Iter>(aIter);
}

Eg.

  std::map<int, string> m;

  for_each( MakeSecondIterator(m.begin()), MakeSecondIterator(m.end()),
            [](const string & val) { cout << val << endl; } );
```

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

```
namespace detail {
  template <typename T>
  struct IterateSecondWrapper
  {
    T & mContainer;
  };
}

namespace std {

  template <typename T>
  auto begin(detail::IterateSecondWrapper<T> aWrapper)
  {
    return MakeSecondIterator( begin(aWrapper.mContainer) );
  }

  template <typename T>
  auto end(detail::IterateSecondWrapper<T> aWrapper)
  {
    return MakeSecondIterator( end(aWrapper.mContainer) );
  }
}
```

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

```
/**
 * Helper function that constructs
 * the appropriate iterator type based on ADL.
 */
template<typename T>
detail::IterateSecondWrapper<T> IterateSecond(T && aContainer)
{
  return { aContainer };
}
```

```
Eg.

  std::map<int, string> m;

  for ( auto & v : IterateSecond(m) ) { cout << val << endl; }
```