STL Iterators and Algorithms

…what?

Iterators

```
Set<int> mySet;

// initialize mySet...

foreach (int value in mySet)
  cout << value << endl;
```

This code creates and initializes a set of integers, and then prints out everything in the set. However, it uses Stanford-specific libraries that aren't readily available outside of CS106B.

Iterators

To implement the same thing using standard C++, we'll use an object called an "iterator." An iterator is intimately tied to a particular data structure; it provides a "view" into that data structure, giving you the means to access and modify different elements.

You can think of an iterator as a remarkably intelligent elf that lives inside the data structure. His job is to report all the different elements that reside in the set. At any point in time, he is standing on top of and looking at one element. In order to do his job, he responds to two commands:

- You can ask the elf what element he's currently looking at. He will look down and report what he sees.
- You can ask the elf to move to the next element. He will hop from where he is to the next spot, where he will now be standing on top of a new element.

Iterator Syntax

```
set<int> mySet;

// initialize mySet...

set<int>::iterator itr = mySet.begin();
while (itr != mySet.end()) {
  cout << *itr << endl;
  itr++;
}
```

This is equivalent to the code with the `foreach`, except it uses standard C++. Note that `Set<int>` has become `set<int>` (lower case ‘s’). Also, we `#include <set>` rather than `#include “set.h”`. The printing loop uses an iterator, just as described before.

Iterator Syntax

```
set<int> mySet;

// initialize mySet...

set<int>::iterator itr = mySet.begin();
while (itr != mySet.end()) {
  cout << *itr << endl;
  itr++;
}
```

The iterator object is a C++ object, so it has a type, highlighted above: `set<int>::iterator`. We specify that it belongs in the scope of `set<int>` because the iterator which understands how to traverse a set is completely from the iterator which understands how to traverse a vector, or a linked list. Each iterator type is specialized to understand one particular data structure.

Iterator Syntax

```
set<int> mySet;

// initialize mySet...

set<int>::iterator itr = mySet.begin();
while (itr != mySet.end()) {
  cout << *itr << endl;
  itr++;
}
```

The set (as well as the other container classes in C++) exports the `.begin()` and `.end()` methods, both of which return special `set` iterators. `.end()` will be explained shortly; `.begin()` simply returns an iterator looking at the very first element in the set.

Iterator Syntax

```
set<int> mySet;

// initialize mySet...

set<int>::iterator itr = mySet.begin();
while (itr != mySet.end()) {
  cout << *itr << endl;
  itr++;
}
```

Finally, to use the two “commands” we can give our iterator, we use the dereference operator (*) and the increment operator (++). Dereferencing the iterator returns a reference to whatever element the iterator is currently looking at, while incrementing it moves it to the next element. The syntax is intentionally similar to pointer syntax; you can almost think of the iterator as a special sort of pointer that describes where things live in a data structure, rather than in memory.

Iterator Syntax

```
set<int> mySet;

// initialize mySet...

set<int>::iterator itr = mySet.begin();
while (itr != mySet.end()) {
  cout << *itr << endl;
  itr++;
}
```

The condition of the `while` loop describes how long we keep looping, alternately reading data and moving to the next element. While it would intuitively seem as though `.end()` returns an iterator looking at the last element in the data structure, this is NOT the case. To see why, consider what happens if we replace the condition of the `while` loop with `true`.

Ending Iterator Loops

```
while (true) {
  cout << *itr << endl;
  itr++;
}
```

Above is the modified loop code. Suppose we are iterating over some container with the five elements listed below. We will animate what exactly happens with the help of our furry friend, Wile E. Coyote.

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

The loop has three main components: a condition check, the data access, and the increment. The corresponding bits of code are shown above. Wile E. Coyote starts off at the beginning of the container.

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

0	1	2	3	4

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

0	1	2	3	4

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

0	1	2	3	4

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

0	1	2	3	4

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

0	1	2	3	4

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

Wile E. Coyote has just read the last element in the container. Danger abounds! He is about to increment and step off the end of the container…

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

…but that's okay! It turns out, Wile E. Coyote never falls immediately when he runs off the edge of a cliff. In fact, he will keep running…

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

…and stay perfectly safe, as long as he keeps running…

Ending Iterator Loops

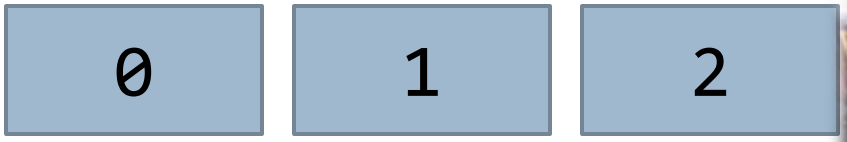
condition check	data access	increment
`true`	`*itr`	`itr++`

…until he looks down. The equivalent action by the iterator is the data access by the dereference operator.

Ending Iterator Loops

condition check	data access	increment
`true`	`*itr`	`itr++`

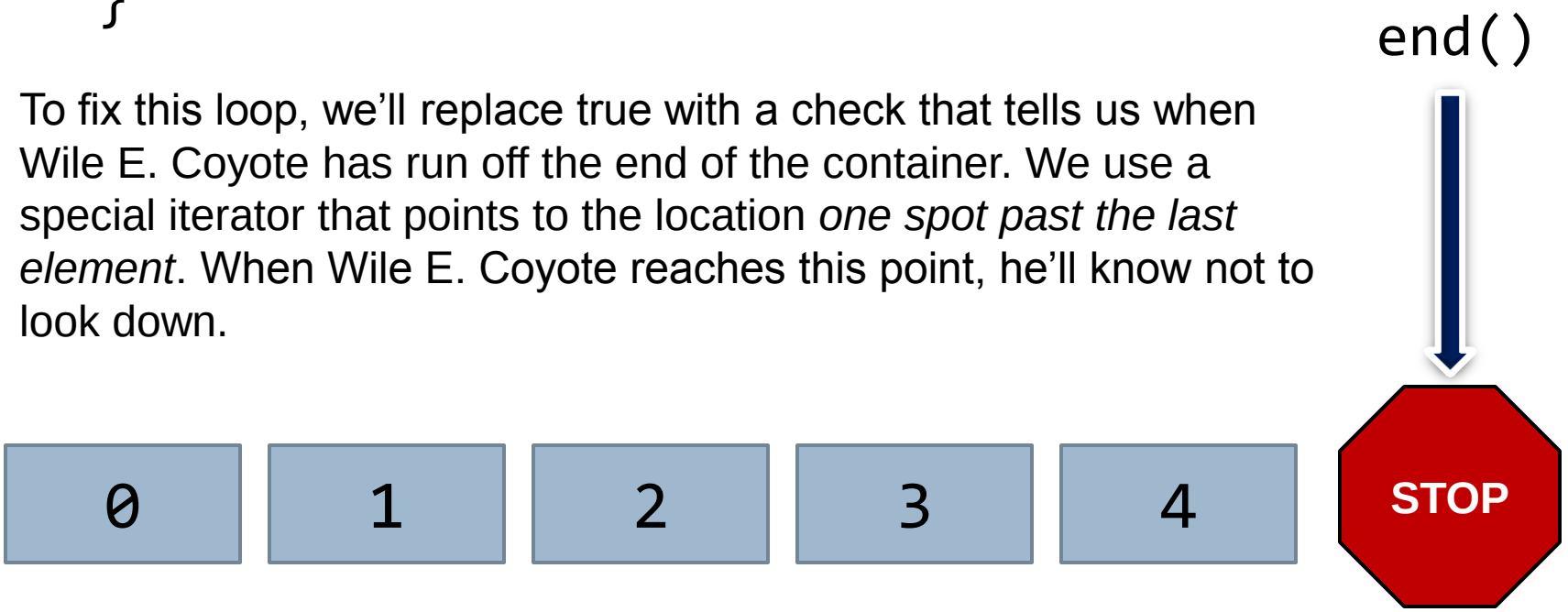
Once he looks down, Wile E. Coyote falls and is mildly injured. When we call `*itr` where `itr` points somewhere bad, the program might crash, or it might return garbage data.



Ending Iterator Loops

```
while (itr != mySet.end()) {
   cout << *itr << endl;
   itr++;
}
```

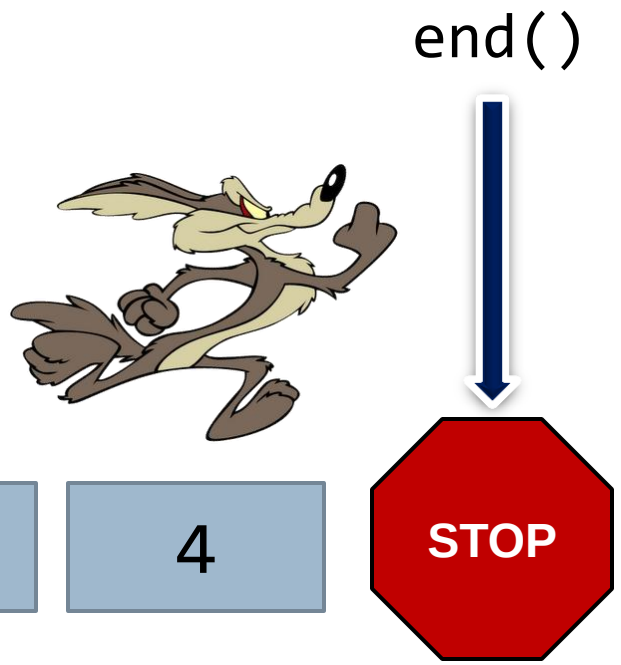
To fix this loop, we'll replace true with a check that tells us when Wile E. Coyote has run off the end of the container. We use a special iterator that points to the location *one spot past the last element*. When Wile E. Coyote reaches this point, he'll know not to look down.



Ending Iterator Loops

condition check	data access	increment
`itr != mySet.end()`	`*itr`	`itr++`

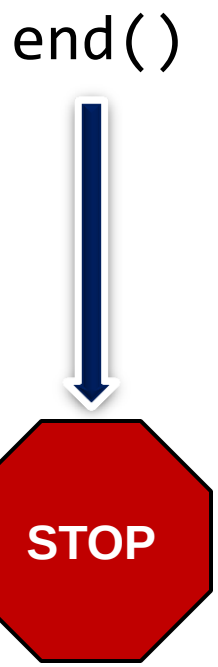
Suppose Wile E. Coyote is looking at the last element in the container. At this point, he hasn't reached the same location where `.end()` points, so he continues iterating.



Ending Iterator Loops

condition check	data access	increment
`itr != mySet.end()`	`*itr`	`itr++`

The data access works fine here. It's the last time that it will in this loop.



Ending Iterator Loops

condition check	data access	increment
`itr != mySet.end()`	`*itr`	`itr++`

The increment takes him past the end of the container, but that's okay since he hasn't yet looked down.

Ending Iterator Loops

condition check	data access	increment
`itr != mySet.end()`	`*itr`	`itr++`

At this point, the condition check evaluates to `false`, because he is at the same location that `.end()` marks. The loop ends, and he never makes the mistake of looking down, so he's completely safe.

Universal Iterator Syntax

Iterators for any STL-compliant container all share the same syntax for reading/writing and increment. For example, the three loops below print the contents of a set, a vector, and a linked list, respectively.

```
set<int>::iterator itr = mySet.begin();
while (itr != mySet.end()) {
  cout << *itr << endl;
  itr++;
}

vector<int>::iterator itr = myVector.begin();
while (itr != myVector.end()) {
  cout << *itr << endl;
  itr++;
}

list<int>::iterator itr = myList.begin();
while (itr != myList.end()) {
  cout << *itr << endl;
  itr++;
}
```

In all three cases, only the iterator type and the name of the container change.

Why iterators?

The reason universal iterator syntax is important because of the role they play in the use of STL algorithms. “Algorithm” is a general math and programming term; in the context of C++, we’re talking about a part of the standard libraries that provides a large variety of useful functionality we can leverage against the data stored in containers.

-ithms!

STL Algorithms

The STL algorithms are a group of functions that perform interesting operations on data that you supply. For example, there are algorithms to:

- sort data
- perform linear or binary searches
- merge two sorted lists
- count the number of appearances of some particular element

…and the list goes on. The algorithms were designed not to need to know how your data is stored, so they operate by accepting *iterator ranges* rather than containers. An iterator range consists of a pair of iterators designating the start and end of the data being supplied as input.

For example, if you wanted to sort everything in `myVector`, you would pass `myVector.begin()` and `myVector.end()` into the `sort` algorithm.

STL Algorithms

Here is some sample code which copies all elements from a set over to a vector, and then randomly shuffles the vector elements.

```
set<int> mySet;

// initialize mySet…

vector<int> myVector(mySet.size()); // myVector has the same size
                                    // as mySet

// The first two arguments to copy are iterators describing the input
// range. The third argument is where to write the results. myVector
// must have enough space!
copy(mySet.begin(), mySet.end(), myVector.begin());
random_shuffle(myVector.begin(), myVector.end());
```

Magic Squares

The following programming example, solving magic squares, illustrates the power of having the STL algorithms at your disposal.

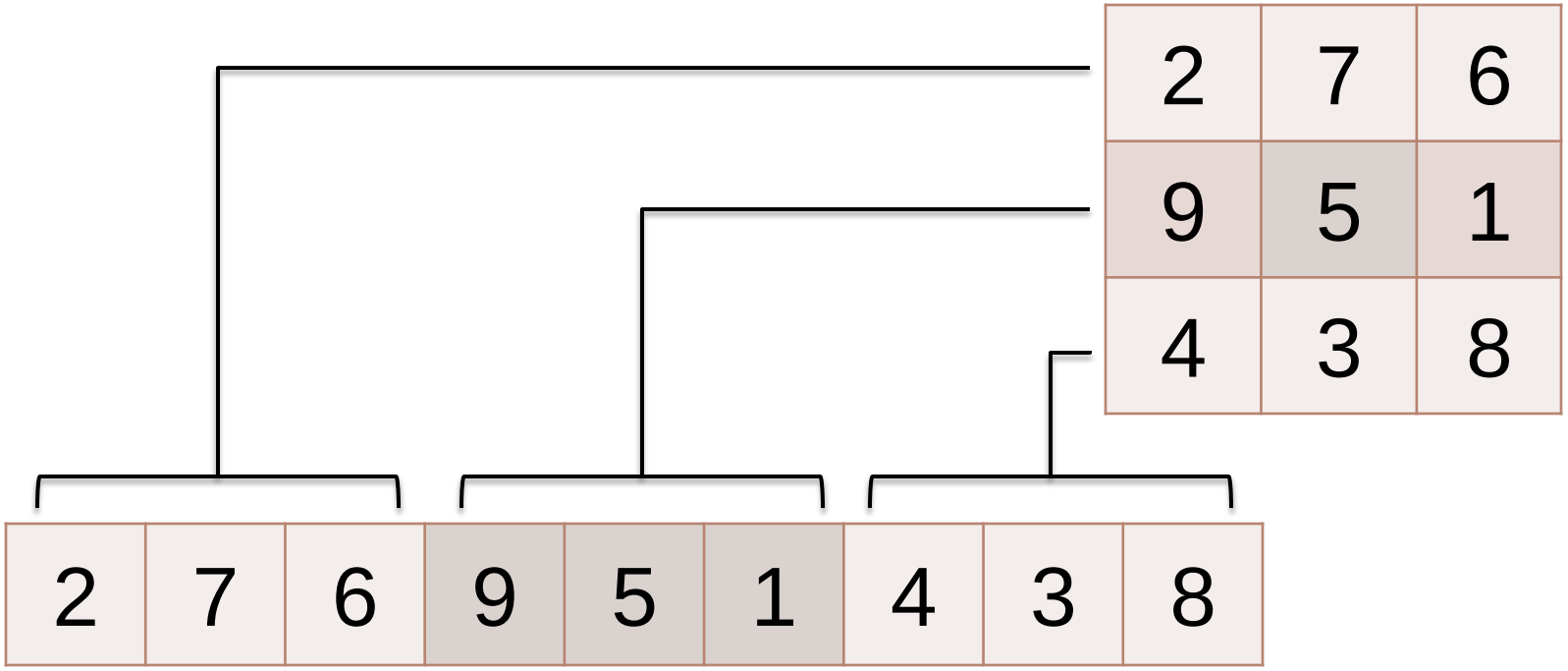
A magic square is a three-by-three grid of numbers in which each of the numbers 1 through 9 appears exactly once, as shown on the right. A key property of valid magic squares is that every row, every column, and each of the two diagonals must sum up to the same value.

In the case of the magic square on the right (and in fact, all three-by-three magic squares), that common sum is 15.

2	7	6
9	5	1
4	3	8

Magic Squares

We are going to find all magic squares by brute-force checking every possible arrangement of the numbers 1 through 9 in the grid. To make this easier programmatically, we'll express the grid as an array in "row-major" form, meaning that the first row comprises the first three indices, the second row comprises the next three, and so on. Then, we'll just generate all possible permutations of that array.



Permutations

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

As it turns out, there is an STL algorithm called `next_permutation` that generates permutations sequentially. It accepts an iterator range consisting of numbers with some ordering, and rearranges them to form the "next" permutation in the list of all possible permutations, where those permutations are sorted in increasing order from left-to-right.

For example, for the list (1, 2, 3), all six permutations are ordered on the left, sorted by the first element first, second element next, and so on. If we were to pass (1, 2, 3) into `next_permutation`, it would produce (1, 3, 2). Passing that into the algorithm would in turn produce (2, 1, 3). Eventually, when (3, 2, 1) is the input, (1, 2, 3) is produced, but the function returns false to indicate that it's finished processing all permutations.

Magic Squares

`next_permutation` can be used in a loop to process all possible permutations of a list of numbers. We'll use this to solve magic squares by representing possible squares as vectors in row-major form.

Assume that `IsMagicSquare` and `PrintSquare` are already implemented (prototypes shown below). The former returns true if the vector passed in represents a grid with the magic square properties, and the latter simply prints the contents of the vector in a nice grid format.

```
bool IsMagicSquare(const vector<int>& candidate);
void PrintSquare(const vector<int>& square);
```

Magic Squares

```
int main() {
  vector<int> candidate;
  for (int i = 1; i <= 9; i++)
    candidate.push_back(i);

  // sort isn't necessary in this case, but it's important when
  // iterating through permutations to make sure to start at
  // the beginning.
  sort(candidate.begin(), candidate.end());
  do {
    if (IsMagicSquare(candidate)) {
      cout << "Solution:" << endl;
      PrintSquare(candidate);
    }
  } while (next_permutation(candidate.begin(), candidate.end());

  return 0;
}
```