STL intro and model

Michael R. Nowak
Texas A&M University

Introduction to the Standard Template Library

Based on slides created by J. Michael Moore, Bjarne Stroustrup, and Jennifer Welch

The Standard Template Library

- **Containers** (16 since C++11 standard)
- **Iterators** (5 types)
- **Algorithms** (80+ since C++11)

- Other organizations provide more containers and algorithms in the style of the STL
  - Boost.org, Microsoft, SGI,...

Common Programming Tasks

- Collect multiple data items into a ***container***
- Organize the data according to some rule
- Retrieve data items
  - by index: e.g., get the i-th item
  - by value: e.g., get the first item with the value "Chocolate"
  - by properties: e.g., get the first item with age < 64
- Add new data items
- Remove existing data items
- Sorting and searching
- Simple numeric operations (e.g., add them all up)

Motivation

- Most of the work to do these programming tasks is independent of the actual type of the data types or even how the data is stored!
- E.g., for sorting, we just need a way to compare the data items
  - numeric types, use $<$
  - strings, use **lexicographic** (alphabetical) orde

Motivation

- For instance, consider the selection sort algorithm, where we repeatedly select the next smallest element in a container and swap it into the correct location in that container:

```
1. Visit each element in the container in order
   (“left-to-right”)
    a. Compare the current element to each element to
       its right, while maintaining the index of the
       smallest element observed so far
    b. Once you’ve found the smallest element, swap
       the element at that index with the current
       element
2. Continue this process until you’ve visited each
   element in the container.
```

Motivation

- We could write a selection sort for a vector<int> v, and easily tailor our solution to sort a vector<char> v:

```
// selection sort algorithm for vector<int>
for ( int i = 0 ; i < v.size() ; ++i ) {
    int smallest = i;
    for ( int j = i + 1 ; j < v.size() ; ++j ) {
        if ( v.at(smallest) > v.at(j) )
            smallest = j;
    }
    int temp = v.at(i);
    v.at(i) = v.at(smallest);
    v.at(smallest) = temp;
}
```

```
// selection sort algorithm for vector<char>
for ( int i = 0 ; i < v.size() ; ++i ) {
    int smallest = i;
    for ( int j = i + 1 ; j < v.size() ; ++j ) {
        if ( v.at(smallest) > v.at(j) )
            smallest = j;
    }
    char temp = v.at(i);
    v.at(i) = v.at(smallest);
    v.at(smallest) = temp;
}
```

(slide intentionally left blank)

Motivation

- While the sort that we've written can be easily applied to different vector<T>, it would require some changes before we could use it to sort, say, a LinkedList<T>
  - Why?
- Ideal: to write code for common programming tasks, such as sorting, that do not have to be rewritten every time we come up with a different way to store the data
  - That is, we want uniform access to the data
  - independent of how it is stored
  - independent of its type
  - And be easy to read, easy to modify, fast,...

Generic Programming

- Generalize ***algorithms***
  - in addition to generalizing data structures
- Advantages:
  - increased correctness
  - greater range of uses (and reuses)
  - better performance (through tuned libraries)

Sum Function for Array

```
double sum(double array[], int n) {
// assume array is of size n
    double s = 0;
    for (int i = 0; i < n; ++i)
        s = s + array[i];
    return s;
}
```

Sum Function for Linked List

```
struct Node {
        Node* next;
        int data;
};

int sum(Node* first) {
        int s = 0;
        while (first) {
                s += first->data;
                first = first->next;
        }
        return s;
}
```

Pseudocode Version of Both Functions

```
int sum(data) {
    int s = 0;
    while (not at end) {        // operation 1
        s = s + get value;      // operation 2
        get next data element; // operation 3
    }
    return s;
}
```

STL-Style Version of Generalized Sum Function

```
// Iter must be an “Input_iterator”
// T must be a type that can be added and assigned
template<class Iter, class T>
// first and last refer to data elements;
// s accumulates the sum
T sum(Iter first, Iter last, T s) {
      while (first != last) {  // operation 1
            s = s + *first;         // operation 2
            ++first;                // operation 3
      }
      return s;
}
```

Using STL-Style Sum Function

```
double a[] = { 1,2,3,4,5,6,7,8 };
double d = 0;
d = sum(a, a + sizeof(a) / sizeof(a[0]), d);
```

- First initialize the array a.
- Then initialize the accumulator d.
- Then call the templated function sum:
  - Iter is replaced with double*
  - T is replaced with double
- First argument is a pointer to the first element of a.
- Second argument is a pointer to just after the last element of a. (why?).

Instantiated Sum Function

```
double sum(double* first, double* last, double s)
{
      while (first != last) {
            s = s + *first;
            ++first;
      }
      return s;
}
```

Sum Function Example

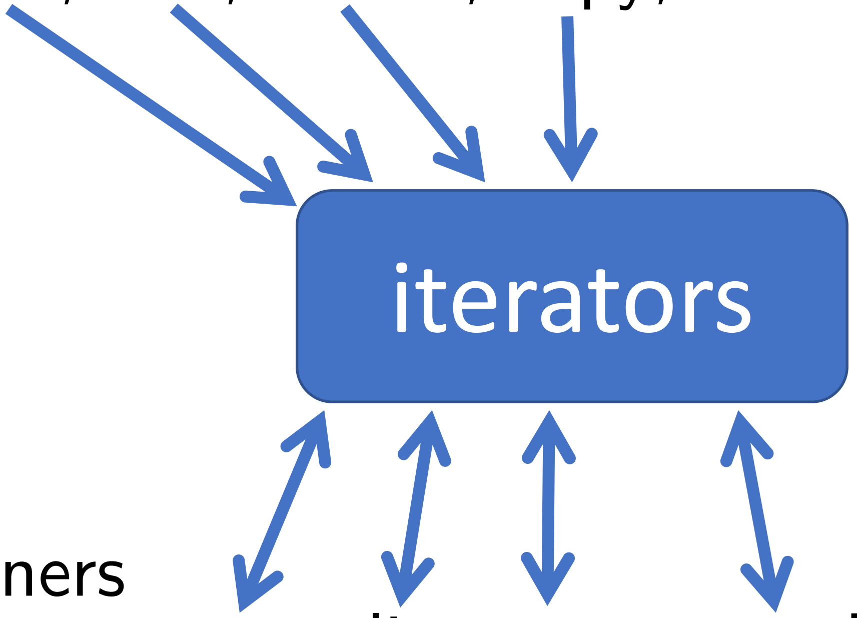
- Almost the standard library accumulate()
- Works for arrays, vectors, lists, istreams,...
- Runs as fast as “hand-crafted” code
- Code’s requirements on the data are made explicit

Standard Template Library Model

Based on slides created by J. Michael Moore, Bjarne Stroustrup, and Jennifer Welch

Basic STL Model

- Algorithms
  sort, find, search, copy, …

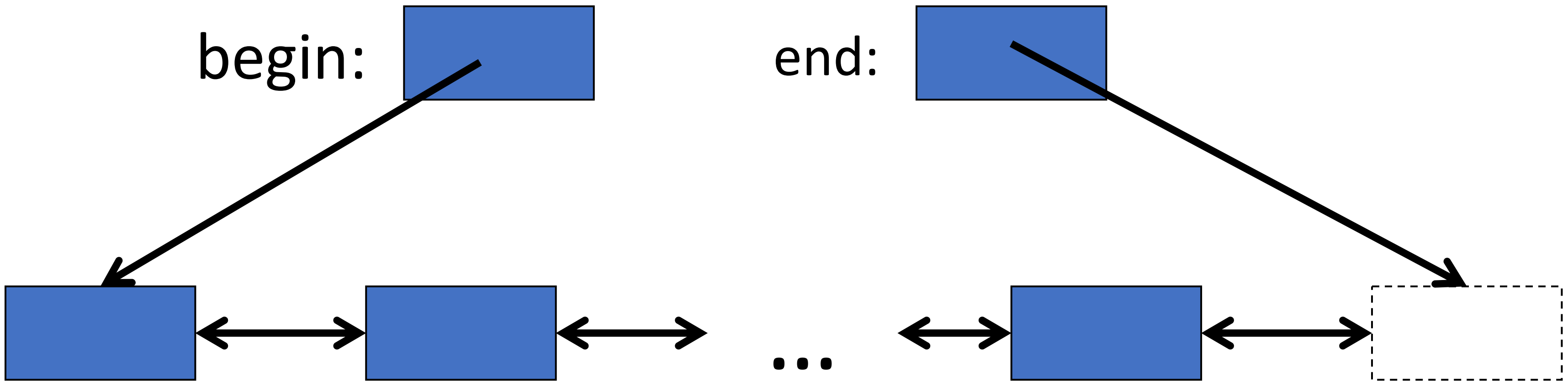


- Containers
  vector, list, map, unordered_map, …

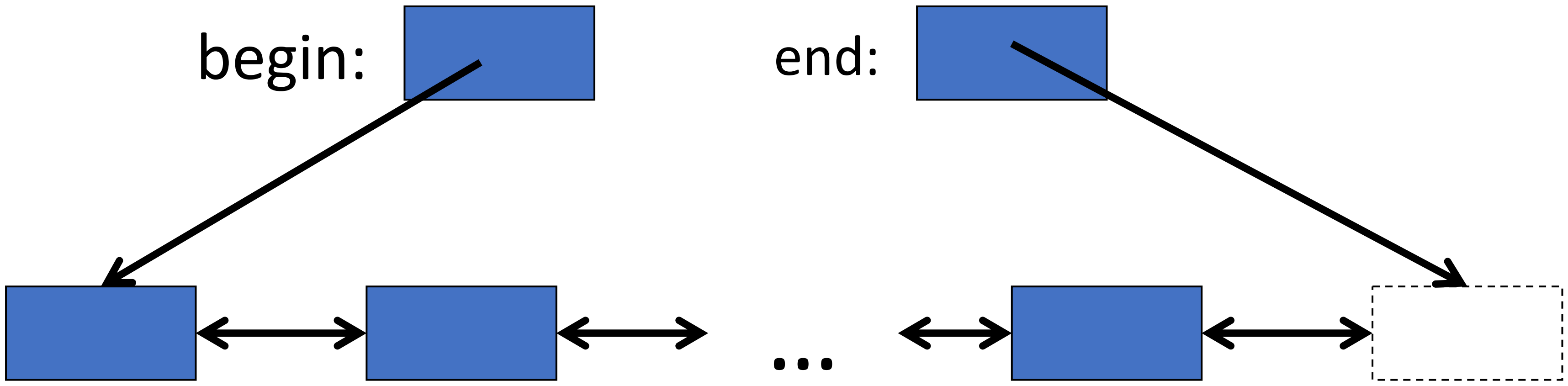
- Separation of concerns
  - Algorithms manipulate data, but don't know about containers
  - Containers store data, but don't know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types

Iterators

- A pair of iterators defines a sequence
  - the beginning (points to first element, if any)
  - the end (points to the one-beyond-the-last element)



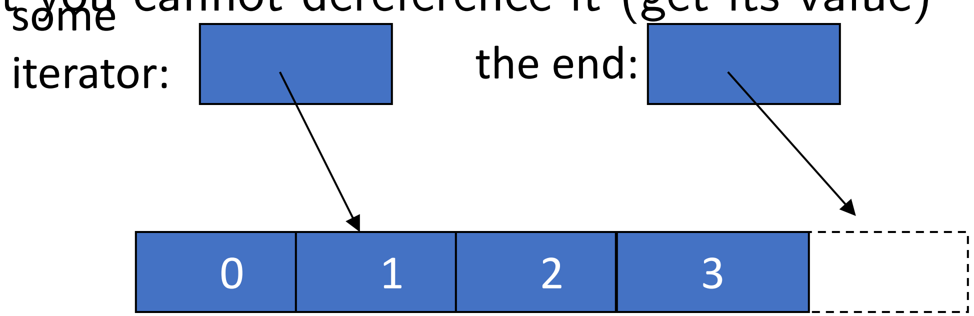
Iterators



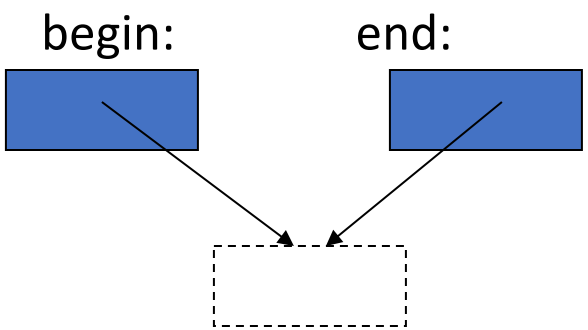
- An ***iterator*** is a type that supports the "iterator operations":
  - Go to next element: ++
  - Get value: *
  - Check if two iterators point to same element: ==
- Frequently a pointer type, but not necessarily
- Some iterators support more operations (--, +, [ ])

One-Past-The-Last

- An iterator point to (refers to, denotes) an element of a sequence
- The end of the sequence is "***one past the last element***", not the last element!
- Reason is to elegantly represent an empty sequence
- One-past-the last element is not an element
  - you can compare an iterator pointing to it
  - but you cannot dereference it (get its value)



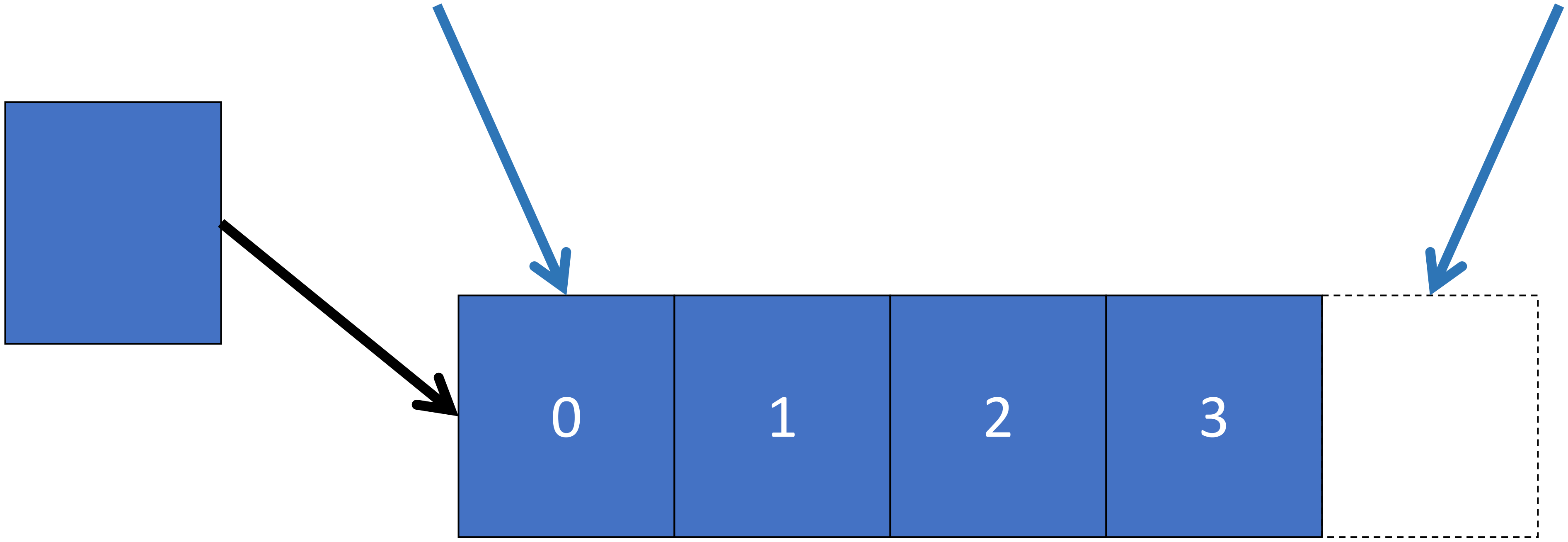
An empty sequence:



Containers

Hold sequences in different ways

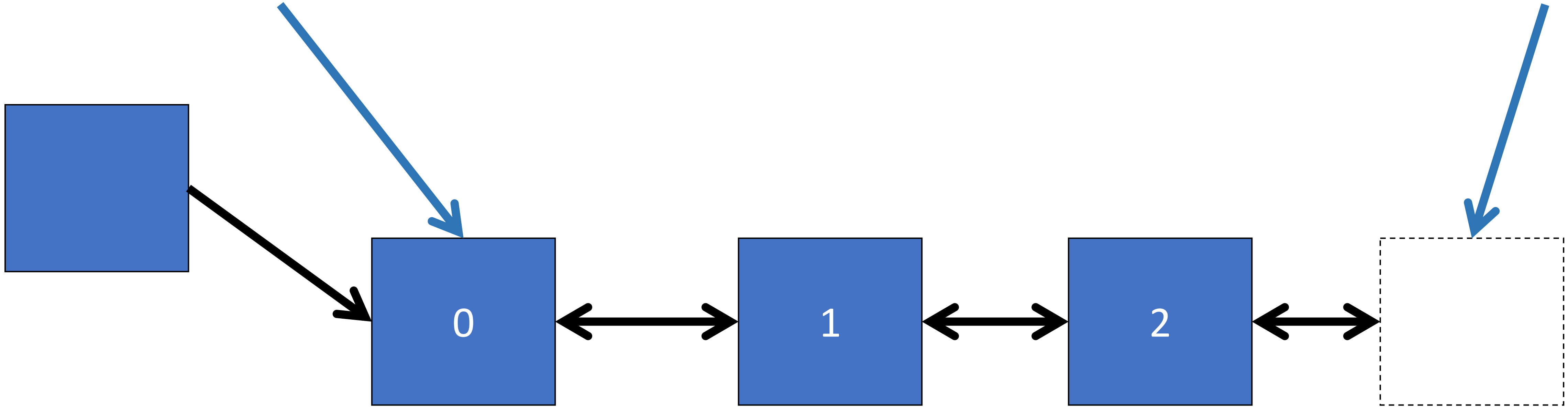
- vector



Containers

Hold sequences in different ways

- list (doubly-linked)



Containers

Hold sequences in different ways

- set (a kind of tree