

# Data Structures in C++

## Chapter 6

Tim Budd

Oregon State University  
Corvallis, Oregon  
USA

# Outline – Chapter 6

## The Standard Library Container Classes

- Variations on Containers
  - Vector
  - List
  - Deque
  - Stack
  - Queue
  - Priority Queue
  - Set
  - Map
- Iterators

# Types of Collections in the Standard Library

Structure	Addition of new element	Removal of first element	Removal of middle element	Inclusion test
<b>Vector</b> indexed, random access to elements, bounded size	$O(1)$ or $O(n)$ †	$O(n)$	$O(1)$ or $O(n)$ †	$O(n)$ or $O(\log n)$ ‡
<b>List</b> sequential access to elements, rapid insertion and removal	$O(1)$	$O(1)$	$O(1)$	$O(n)$
<b>Deque</b> random access, rapid insertion to front and back	$O(1)$	$O(1)$	$O(n)$	$O(n)$ or $O(\log n)$ ‡
<b>Stack</b> insertion and removal only from front	$O(1)$	$O(1)$	NA	NA
<b>Queue</b> insertion only from front, removal only from back	$O(1)$	$O(1)$	NA	NA
<b>Priority Queue</b> rapid removal of largest element	$O(\log n)$	$O(1)$ or $O(\log n)$ §	NA	NA
<b>Set</b> ordered collection, unique values, rapid insertion, removal and test a <code>multiset</code> allows repeated elements	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
<b>Map</b> collection of key-value pairs a <code>multimap</code> allows multiple elements with same key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
<b>Notes</b>	†constant if accessing existing position, linear if inserting/removing ‡logarithmic if ordered, linear if not ordered §constant access time, logarithmic removal			

# Vectors

- Generalization of array
- Efficient, random access to elements
- High Level operations, such as dynamically increasing or decreasing in size

$v[0]$	$v[1]$	$v[2]$	...	$v[n-2]$	$v[n-1]$
--------	--------	--------	-----	----------	----------

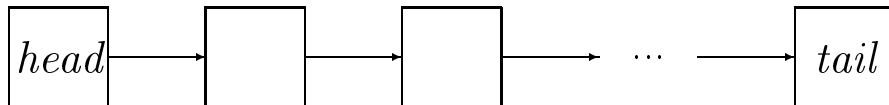
# Strings

- In one sense, a vector of character values
- In another sense, an entirely different high level data type
- Lots of string specific operations

```
string aName = "Benjamin Franklin";
```

# Lists

- Arbitrary size, memory used efficiently as grows and shrinks
- Sequential access only, constant access to first or last element
- Efficient insertion or removal at any location



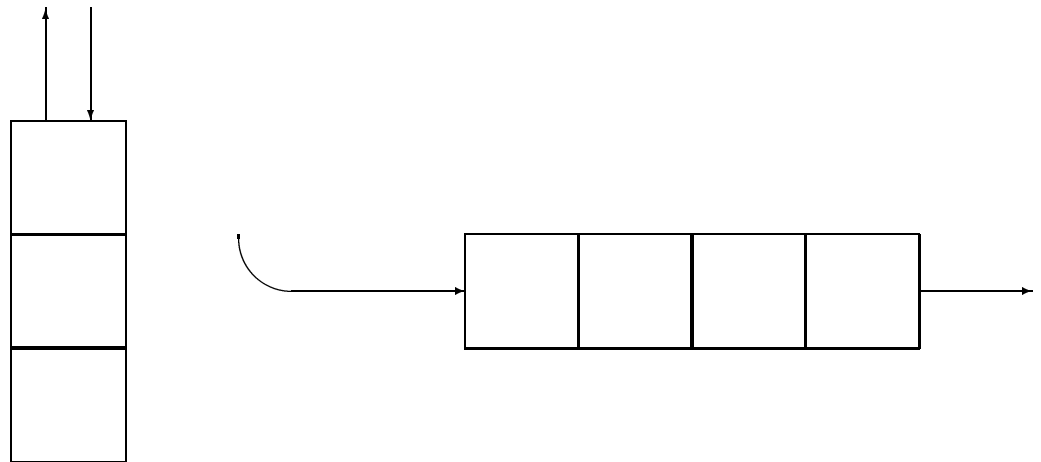
# Deque – Double Ended Queue

- Grows or shrinks as necessary
- Efficient insertion or removal from either end
- Random access to elements



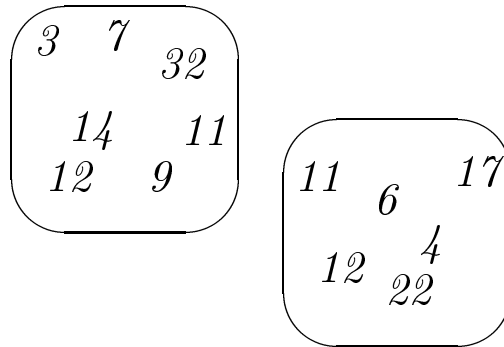
# Stacks and Queues

- Specialized form of deque
- Stack has LIFO, Last In, First Out protocol
- Queue has FIFO, First In, First Out protocol



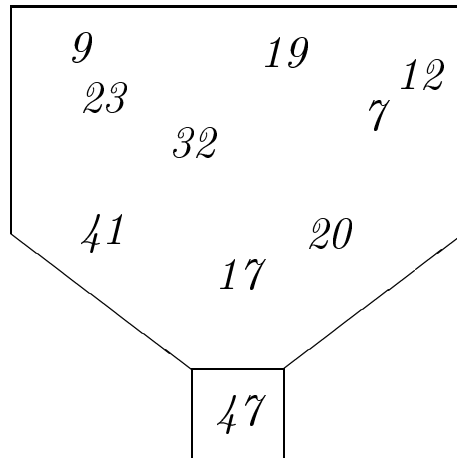
# Sets

- Ordered collection
- Efficient (logarithm) insertion, removal, and test for inclusion
- Efficient merge, union, difference, and other set operations
- Multiset allows more than one entry with same value



# Priority Queue

- Efficient (logarithmic) insertion of new values
- Efficient access to largest (or smallest) value.  
(Constant time access, logarithmic removal)



## Map (Dictionary)

- Collection of key and value pairs
- Keys can be any ordered data type (i.e., string)
- Values can be any data type
- Efficient insertion, removal, test for inclusion

$$\begin{aligned} key_1 &\rightarrow value_1 \\ key_2 &\rightarrow value_2 \\ key_3 &\rightarrow value_3 \\ &\dots \\ key_n &\rightarrow value_n \end{aligned}$$

## Selecting a Container

- *How are values going to be accessed?*

random – vector or deque

ordered – set or map

sequential – list

- *Is the order in which values are maintained in the collection important?*

ordered – set

can be sorted – vector or deque

insertion time dependent – stack or queue

- *Will the size of the structure vary widely over the course of execution?*

Yes – use a list or set

No – use a vector or deque

- *Is it possible to estimate the size of the collection?*

Yes – use a vector

- *Is testing to see whether a value is contained in the collection a frequent operation?*

Yes – use a set

- *Is the collection indexed? That is, can the collection be viewed as a series of key/value pairs?*

Index values are integer – use a vector or deque

Otherwise use a map

- *Can values be related to each other?*

Sets require relational operators

Vectors or Lists do not

- *Is finding and removing the largest value from the collection a frequent operation?*

Yes – use priority queue  
(slightly faster than set)

- *At what positions are values inserted into or removed from the structure?*

Insertions to middle of list are efficient

Insertions into middle of vectors are not

Stacks and Queues can be inserted only at end

- *Is a frequent operation the merging of two or more sequences into one?*

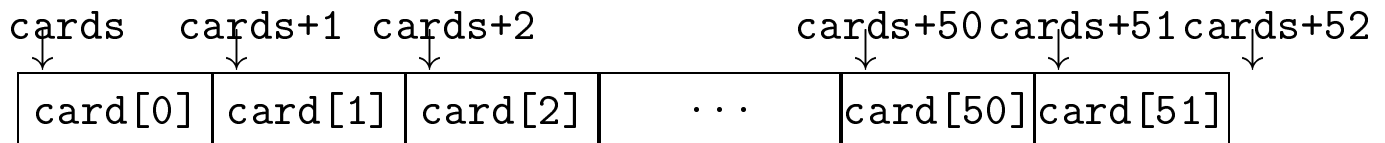
If ordered – use a set

Otherwise – use a list and splice

# Iterators

- Basic problem – how do you allow access to elements of collection, without knowing how the collection is organized?
- Solution, define a new data type specifically for creating loops
- A large number of algorithms are provided by the standard library, all built using iterators.

## How do you describe a range of values



- Notice how a range of values is often described by a starting value and a *past-the-end* value.
- The past the end value is not part of the collection, but just a marker.

## Begin and End

By convention, containers return a starting value in response to `begin()`, and a past-the-end value in response to `end()`.

For example, to shuffle a vector of values:

```
random_shuffle  
    (aVector.begin(), aVector.end(),  
     randomInteger);
```

## What must iterators do

To see what iterators must do, consider a typical algorithm:

```
iterator find
  (iterator first, iterator last, T & value)
{
  while (first != last && *first != value)
    ++first;
  return first;
}
```

Could be used to find values in an array, or in a list:

```
int data[100];
...
int * where = find(data, data+100, 7);

list<int>::iterator where =
  find(aList.begin(), aList.end(), 7);
```

## Iterator Operations

- An iterator can be compared for equality to another iterator. They are equal when they point to the same position, and are otherwise not equal.
- An iterator can be dereferenced using the `*` operator, to obtain the value being denoted by the iterator. Depending upon the type of iterator and variety of underlying container, this value can also sometimes be used as the target of an assignment in order to change the value being held by the container.
- An iterator can be incremented, so that it refers to the next element in sequence, using the operator `++`.

What makes iterators possible is that all of these can be overloaded.

Will make extensive use of iterators in the following chapters.