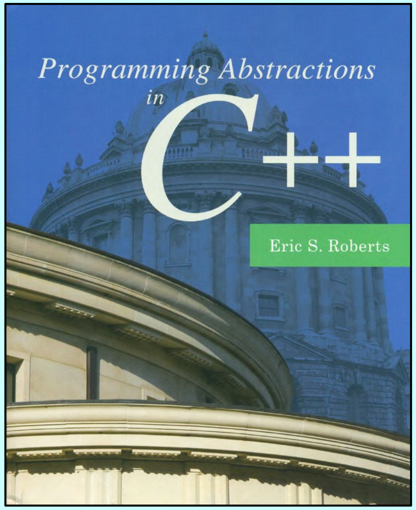
CHAPTER 5

Collections



In this way I have made quite a valuable collection.

—Mark Twain, *A Tramp Abroad,* 1880

Introduction to the C++ Standard Libraries

- A collection of ***classes*** and ***functions***, which are written in the core language and part of the C++ ISO Standard itself. Features of the C++ Standard Library are declared within the **`std`** ***namespace***
  - Containers: vector, queue, stack, map, set, etc.
  - General: algorithm, functional, iterator, memory, etc.
  - Strings
  - Streams and Input/Output: iostream, fstream, sstream, etc.
  - Localization
  - Language support
  - Thread support library
  - Numerics library
  - C standard library: cmath, cctype, cstring, cstdio, cstdlib, etc.

Abstract Data Type (ADT)

- The atomic/primitive data types like **`bool`**, **`char`**, **`int`**, **`double`**, occupy the lowest level in the data structure hierarchy.
- To represent more complex information, you combine the atomic types to form larger structures.
- It is usually far more important to know how to use those structures effectively than to understand their underlying representation, e.g., using strings as abstract values (Ch. 3).
- A type defined in terms of its behavior rather than its representation is called an ***Abstract Data Type (ADT)***.
- ADTs are central to the object-oriented style of programming, which encourages programmers to think about data structures in a holistic way.

ADTs as Software Tools

- Over the relatively short history of software development, one of the clear trends is the increasing power of the tools available to you as a programmer.
- One of the best explanations of the importance of tools is the book *Software Tools* by Brian Kernighan and P. J. Plauger. Even though it was published in 1976, its value and relevance have not diminished over time.
- The primary theme of the book is that the best way to extend your reach in programming is to build on the tools of others.

The Collection Classes

- The classes that contain collections of other objects are called ***containers*** or ***collection*** classes.
- Collection classes specify the type of objects they contain (base/element type) by including the type name in angle brackets following the class name.
- Separating the behavior of a class from its underlying implementation is a fundamental precept of object-oriented programming. As a design strategy, it offers the following advantages:
  - ***Simplicity***: fewer details for the client to understand.
  - ***Flexibility***: free to change underlying implementation as long as the interface remains the same.
  - ***Security***: prevents the client from changing the values in the underlying data structure in unexpected ways.

The *Standard* Collection Classes

- Classes that include a base-type specification are called ***parameterized classes***.
- In C++, parameterized containers are implemented as ***template*** classes, which make it possible for an entire family of classes to share the same code, although their base types are different.
- The Standard Template Library (STL) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library.
- Although the C++ Standard Library and the STL share many features, neither is a strict superset of the other.
- Templates are sometimes called static (or compile-time) ***polymorphism***, as opposed to run-time polymorphism.
- For the moment, you don’t need to understand ~~how these classes are implemented using templates~~, because your primary focus is on learning how to use these classes as a client.

Container class templates C++ Standard Containers

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Other:

Two class templates share certain properties with containers, and are sometimes classified with them: `bitset` and `valarray`.

The *Stanford* Collection Classes

- The Stanford C++ Library includes the simplified version of such containers of the Standard C++ Libraries, such as:

`Vector`	`Grid`	`Stack`	`Queue`	`Map`	`Set`	`Lexicon`

- Here are some general guidelines for using these classes:
  - These classes represent ***ADTs*** whose details are hidden.
  - Each class (except **`Lexicon`**) requires ***type*** parameters.
  - Declaring variables of these types always invokes a class ***constructor***.
  - Any memory for these objects is ***freed (automatically)*** when its declaration scope ends.
  - Assigning one value to another ***copies*** the entire structure.
  - To avoid copying, these structures are usually passed ***by reference***.

The `Vector<`*type*`>` Class

- The **`Vector`** class provides a facility similar to list in Python, and the underlying implementation is based on array in C++.

```
#include "vector.h"
```

- Besides including the appropriate library interface, as a client of the **`Vector`** class, you are concerned with a different set of issues and need to answer the following questions:
  - How is it possible to specify the type of object (base/element type) contained in a **`Vector`**?
  - How does one create an object that is an instance of the **`Vector`** class (declaration)?
  - What methods exist in the **`Vector`** class to implement its abstract behavior?

Declaring a `Vector` object

`Vector<type> vec;`	Initializes an empty vector of the specified element type.
`Vector<type> vec(n);`	Initializes a vector with `n` elements all set to the default value of the type.
`Vector<type> vec(n, value);`	Initializes a vector with `n` elements all set to `value`.

- Instead of using the class name alone, the collection classes require a type parameter that specifies the element type. For example, `Vector<int>` represents a vector of integers.
- The Stanford C++ library implementation of `Vector` includes a shorthand form for initializing an array given a list of values, as illustrated by the following example:

```
Vector<int> digits;
digits += 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
```

Declaring a `Vector` object

- It is possible to create ***nested*** vectors, so that, for example, `Vector< Vector<char> >` represents a two-dimensional vector of characters.
- The spaces around the inner type parameter are necessary for many compilers to ensure that the angle brackets for the type parameters are interpreted correctly. Remember that `>>` is a single extraction operator.
- The type parameter used in the `Vector` class can be any C++ type, so you could use the following definition to represent a list of chess positions (a three-dimensional vector):

```
Vector< Grid<char> > chessPositions;
```

Methods in the **`Vector<`***type***`>`** Class

`vec.size()` Returns the number of elements in the vector.		
`vec.isEmpty()` Returns **`true`** if the vector is empty.		
`vec.get(i)` Returns the **`i`**th element of the vector.	*or*	**`vec[i]`**
`vec.set(i, value)` Sets the **`i`**th element of the vector to **`value`**.	*or*	**`vec[i] = value;`**
`vec.add(value)` Adds a new element to the end of the vector.	*or*	**`vec += value;`**
`vec.insert(index, value)` Inserts the value before the specified index position.		
`vec.remove(index)` Removes the element at the specified index.		
`vec.clear()` Removes all elements from the vector.		

Methods in the `Vector<`*type*`>` Class

Constructor

`Vector()`	O(1)	Initializes a new empty vector.
`Vector(`*n*`,` *value*`)`	O(N)	Initializes a new vector storing *n* copies of the given value.

Methods

`add(`*value*`)`	O(1)	Adds a new value to the end of this vector.
`clear()`	O(1)	Removes all elements from this vector.
`equals(`*v*`)`	O(N)	Returns **true** if the two vectors contain the same elements in the same order.
`get(`*index*`)`	O(1)	Returns the element at the specified index in this vector.
`insert(`*index*`,` *value*`)`	O(N)	Inserts the element into this vector before the specified index.
`isEmpty()`	O(1)	Returns **true** if this vector contains no elements.
`mapAll(`*fn*`)`	O(N)	Calls the specified function on each element of the vector in ascending index order.
`remove(`*index*`)`	O(N)	Removes the element at the specified index from this vector.
`set(`*index*`,` *value*`)`	O(1)	Replaces the element at the specified index in this vector with a new value.
`size()`	O(1)	Returns the number of elements in this vector.
`subList(`*start*`,` *length*`)`	O(N)	Returns a new vector containing elements from a sub-range of this vector.
`toString()`	O(N)	Converts the vector to a printable string representation.

Operators

v`[`*index*`]`	O(1)	Overloads **[ ]** to select elements from this vector.
v1 `+` *v2*	O(N)	Concatenates two vectors.
v1 `+=` *v2*`;`	O(N)	Adds all of the elements from **v2** to **v1**.
v `+=` *value*`;`	O(1)	Adds the single specified value to **v**.
v `+=` *a*`,` *b*`,` *c*`;`	O(1)	Adds multiple individual values to **v**.
v1 `==` *v1*	O(N)	Returns **true** if **v1** and **v2** contain the same elements.
v1 `!=` *v2*	O(N)	Returns **true** if **v1** and **v2** are different.
ostream `<<` *v*	O(N)	Outputs the contents of the vector to the given output stream.
istream `>>` *v*	O(N)	Reads the contents of the given input stream into the vector.

Methods in the STL `vector<type>` Class

fx Member functions

(constructor)	Construct vector (public
(destructor)	Vector destructor (public
operator=	Assign content (public m

Iterators:

begin	Return iterator to begin
end	Return iterator to end (p
rbegin	Return reverse iterator t
rend	Return reverse iterator t
cbegin C++11	Return const_iterator to
cend C++11	Return const_iterator to
crbegin C++11	Return const_reverse_it
crend C++11	Return const_reverse_it

Capacity:

size	Return size (public memb
max_size	Return maximum size (p
resize	Change size (public mem
capacity	Return size of allocated
empty	Test whether vector is e
reserve	Request a change in cap
shrink_to_fit C++11	Shrink to fit (public mem

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

Allocator:

get_allocator	Get allocator (public member function)

fx Non-member function overloads

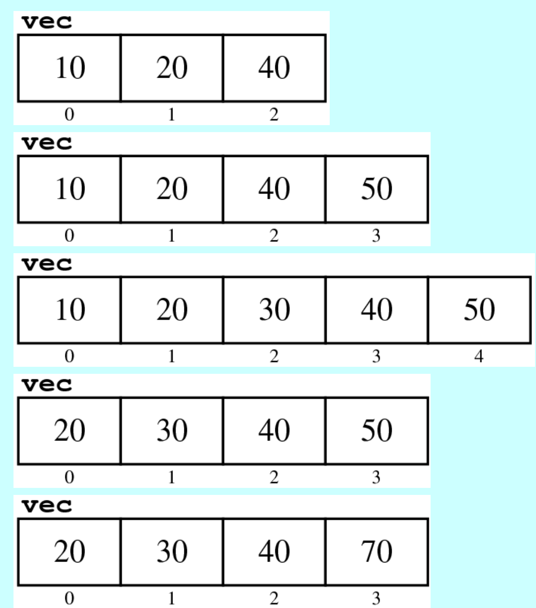
relational operators	Relational operators for vector (function template)
swap	Exchange contents of vectors (function template)

Template specializations

vector<bool>	Vector of bool (class template specialization)

The **Vector<*type*>** Class Operators

```
#include "vector.h"

Vector<int> vec;

vec.add(10);     // vec += 10;

vec.add(20);

vec.add(40);

vec.add(50);

vec.insert(2, 30);

vec.remove(0);

vec.set(3, 70); // vec[3] = 70;
```



The `readEntireFile` Function

```
/*
 * Function: readEntireFile
 * Usage: readEntireFile(is, lines);
 * ---------------------------------
 * Reads the entire contents of the specified input stream
 * into the string vector lines.  The client is responsible
 * for opening and closing the stream
 */

void readEntireFile(istream & is, Vector<string> & lines) {
   lines.clear(); // be careful for a reference
   string line;
   while (true) {
      getline(is, line);
      if (is.fail()) break;
      lines.add(line);
   }
}
```

The **`readEntireFile`** Function

```
/*
 * Function: readEntireFile
 * Usage: readEntireFile(is, lines);
 * ---------------------------------
 * Reads the entire contents of the specified input stream
 * into the string vector lines.  The client is responsible
 * for opening and closing the stream
 */

void readEntireFile(istream & is, Vector<string> & lines) {
   lines.clear();
   string line;
   while (getline(is, line)) {
      lines.add(line);
   }
}
```

Exercise: **`reverseEntireFile`**

```
int main() {
   ifstream infile;
   Vector<string> lines;
   promptUserForFile(infile, "Input file: ");
   readEntireFile(infile, lines);
   infile.close();
   for (int i = lines.size() - 1; i >= 0; i--) {
      cout << lines[i] << endl;
   }
   return 0;
}
```

Methods in the **Grid<*type*>** Class

`Grid<type> grid(nrows, ncols);`	Constructs a grid with the specified dimensions.
`grid.numRows()`	Returns the number of rows in the grid.
`grid.numCols()`	Returns the number of columns in the grid.
`resize(nrows, ncols)`	Changes the dimensions of the grid and clears any previous contents.
`inBounds(row, col)`	Returns **`true`** if the specified row and column position is within the grid.
`get(i, j)` *or* **`grid[i][j]`**	Returns the element at the specified row and column.
`set(i, j, value)` *or* **`grid[i][j] = value`**	Sets the element at the specified row and column to the new value.
`grid[i][j]`	Selects the element in the **`i`**th row and **`j`**th column.

Methods in the `Grid<`*type*`>` Class

Constructor

Grid()	O(1)	Initializes a new empty 0x0 grid.
Grid(*nRows*, *nCols*)	O(N)	Initializes a new grid of the given size.
Grid(*nRows*, *nCols*, *value*)	O(N)	Initializes a new grid of the given size, with every cell set to the given value.

Methods

equals(*grid*)	O(N)	Returns **true** if the two grids contain the same elements.
fill(*value*)	O(N)	Sets every grid element to the given value.
get(*row*, *col*)	O(1)	Returns the element at the specified **row/col** position in this grid.
height()	O(1)	Returns the grid's height, that is, the number of rows in the grid.
inBounds(*row*, *col*)	O(1)	Returns **true** if the specified row and column position is inside the bounds of the grid.
isEmpty()	O(1)	Returns true if the grid has 0 rows and/or 0 columns.
mapAll(*fn*)	O(N)	Calls the specified function on each element of the grid.
numCols()	O(1)	Returns the number of columns in the grid.
numRows()	O(1)	Returns the number of rows in the grid.
resize(*nRows*, *nCols*)	O(N)	Reinitializes the grid to have the specified number of rows and columns.
set(*row*, *col*, *value*)	O(1)	Replaces the element at the specified **row/col** location in this grid with a new value.
size()	O(1)	Returns the total number of elements in the grid.
toString()	O(N)	Converts the grid to a printable single-line string representation.
toString2D()	O(N)	Converts the grid to a printable 2-D string representation.
width()	O(1)	Returns the grid's width, that is, the number of columns in the grid.

Operator

grid[*row*][*col*]	O(1)	Overloads **[]** to select elements from this grid.
grid1 == *grid1*	O(N)	Returns **true** if **grid1** and **grid2** contain the same elements.
grid1 != *grid2*	O(N)	Returns **true** if **grid1** and **grid2** are different.
ostream << *grid*	O(N)	Outputs the contents of the grid to the given output stream.
istream >> *grid*	O(N)	Reads the contents of the given input stream into the grid.

Example: **`TicTacToe`**

ogram that checks to see whether a specific player
game of TicTacToe.

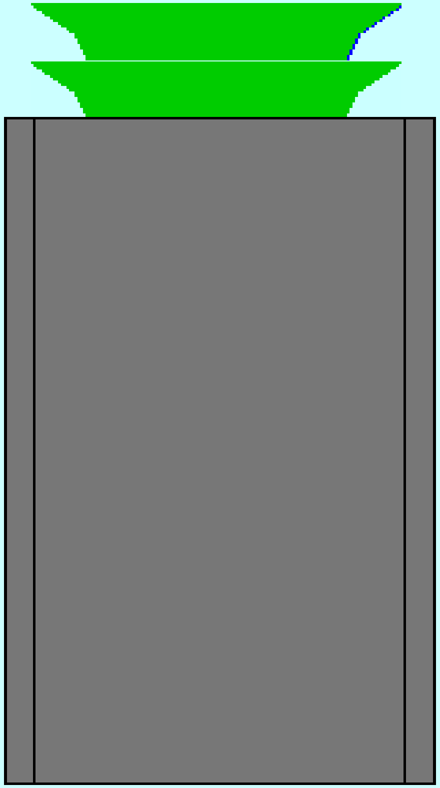
```
Grid<char> board(3, 3);
```

- Exercise: use **`Vector`** to replace **`Grid`**

```
Vector<char> vec(3);
Vector< Vector<char> > board(3, vec);
// or simply
Vector< Vector<char> > board(3, Vector<char>(3));
```

The Stack Metaphor

- A ***stack*** is a data structure in which the elements are accessible only in a ***Last In, First Out (LIFO)*** order.
- The fundamental operations on a stack are **`push`**, which adds a new value to the top of the stack, and **`pop`**, which removes and returns the top value.
- One of the most common metaphors for the stack concept is a spring-loaded storage tray for dishes. Adding a new dish to the stack pushes any previous dishes downward. Taking the top dish away allows the dishes to pop back up.
- In programming, nested function calls behave in a stack-oriented fashion.



Methods in the **`Stack`**<*type*> Class

`stack.size()` Returns the number of values pushed onto the stack.
`stack.isEmpty()` Returns **`true`** if the stack is empty.
`stack.push(value)` Pushes a new value onto the stack.
`stack.pop()` Removes and returns the top value from the stack.
`stack.peek()` Returns the top value from the stack without removing it.
`stack.clear()` Removes all values from the stack.

Methods in the `Stack<type>` Class

Constructor

Stack()	O(1)	Initializes a new empty stack.

Methods

clear()	O(1)	Removes all elements from this stack.
equals(*stack*)	O(N)	Returns **true** if the two stacks contain the same elements in the same order.
isEmpty()	O(1)	Returns **true** if this stack contains no elements.
peek()	O(1)	Returns the value of top element from this stack, without removing it.
pop()	O(1)	Removes the top element from this stack and returns it.
push(*value*)	O(1)	Pushes the specified value onto this stack.
size()	O(1)	Returns the number of values in this stack.
toString()	O(N)	Converts the stack to a printable string representation.

Operators

stack1 == stack1	O(N)	Returns **true** if **stack1** and **stack2** contain the same elements.
stack1 != stack2	O(N)	Returns **true** if **stack1** and **stack2** are different.
ostream << stack	O(N)	Outputs the contents of the stack to the given output stream.
istream >> stack	O(N)	Reads the contents of the given input stream into the stack.

Methods in the STL `stack<`*type*`>` Class

fx Member functions

(constructor)	Construct stack (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access next element (public member function)
push	Insert element (public member function)
emplace C++11	Construct and insert element (public member function)
pop	Remove top element (public member function)
swap C++11	Swap contents (public member function)

fx Non-member function overloads

relational operators	Relational operators for stack (function)
swap (stack) C++11	Exchange contents of stacks (public member function)

fx Non-member class specializations

uses_allocator<stack> C++11	Uses allocator for stack (class template)

Example: Stack Processing

Write a C++ program that checks whether the bracketing operators (parentheses, brackets, and curly braces) in a string are properly matched. As an example of proper matching, consider the string

```
{ s = 2 * (a[2] + 3); x = (1 + (2)); }
```

If you go through the string carefully, you discover that all the bracketing operators are correctly nested, with each open parenthesis matched by a close parenthesis, each open bracket matched by a close bracket, and so on.

IsBalanced

```
Enter string: { s = 2 * (a[2] + 3); x = (1 + (2)); }
Brackets are properly nested
Enter string: (a[2] + b[3)
Brackets are incorrect
Enter string:
```

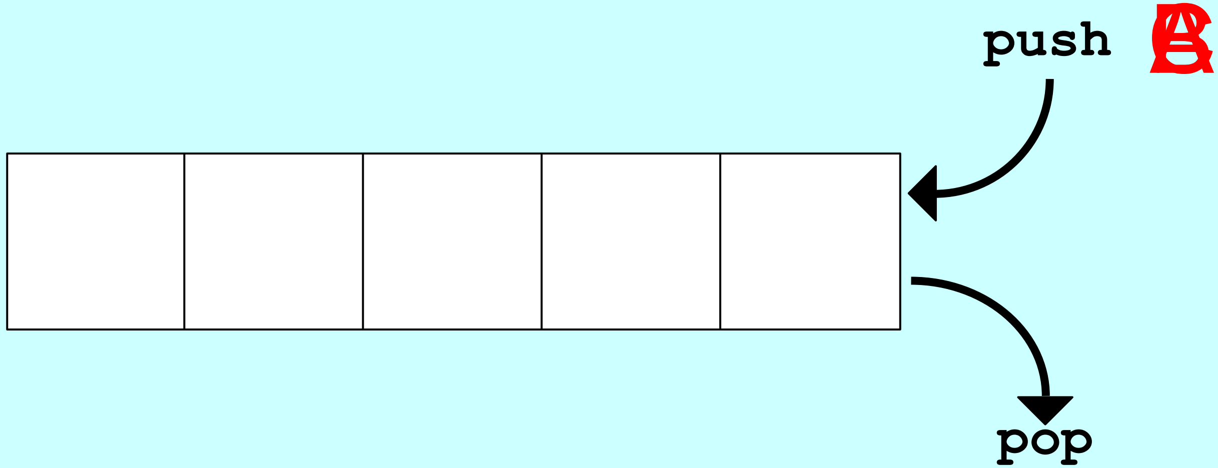

```
ain() {
ile (true) {
 string str = getLine("Enter string: ");
 if (str == "") break;
 if (isBalanced(str)) {
    cout << "Brackets are properly nested" << endl;
 } else {
    cout << "Brackets are incorrect" << endl;
 }

turn 0;
}

bool isBalanced(string str) {
   Stack<char> stack;
   for (int i = 0; i < str.length(); i++) {
      char ch = str[i];
      switch (ch) {
        case '{': case '[': case '(': stack.push(ch); break;
        case '}': case ']': case ')':
          if (stack.isEmpty()) return false;
          if (!operatorMatches(stack.pop(), ch)) return false;
          break;
      }
   }
   return stack.isEmpty();
}

bool operatorMatches(char open, char close) {
   switch (open) {
     case '{': return close == '}';
     case '[': return close == ']';
     case '(': return close == ')';
     default: return false;
   }
}
```

The `Queue`<*type*> Class

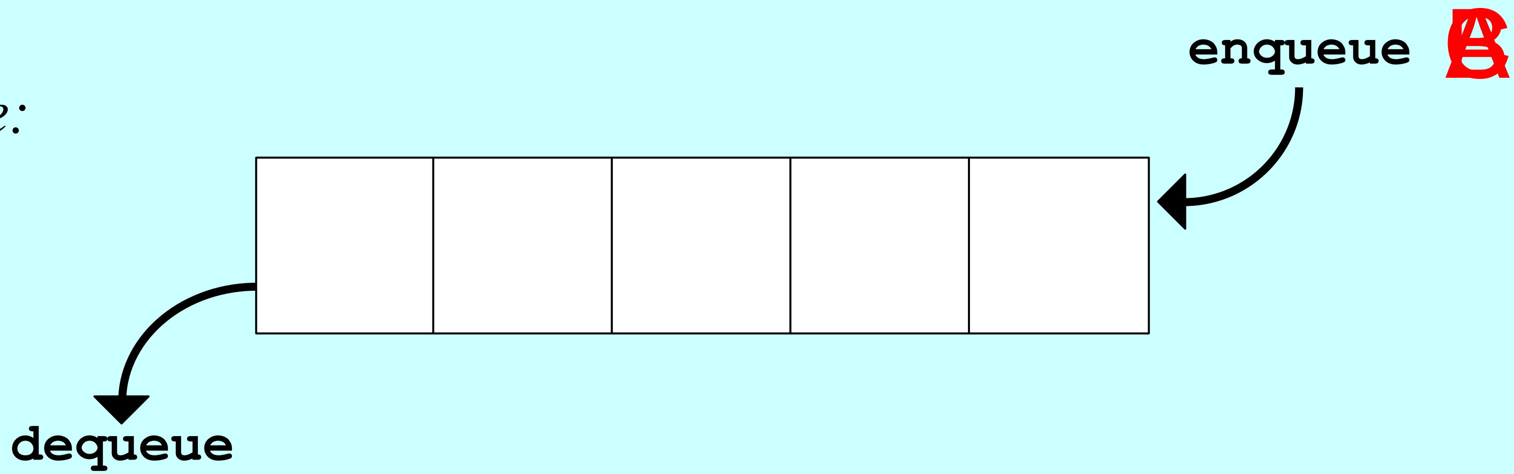
- The ***LIFO*** discipline in a ***stack*** is useful in programming contexts because it reflects the operation of function calls.
- In real-world situations, however, our collective notion of fairness assigns some priority to being first. In programming, the usual phrasing of this ordering strategy is ***First In, First Out (FIFO)***.
- A data structure that stores items using a ***FIFO*** discipline is called a ***queue***. The fundamental operations on a queue, which are analogous to the **`push`** and **`pop`** operations for stacks, are called **`enqueue`** and **`dequeue`**.
- The **`enqueue`** operation adds a new element to the end of the queue, which is traditionally called its ***tail***.
- The **`dequeue`** operation removes the element at the beginning of the queue, which is called its ***head***.

Comparing Stacks and Queues

Stack:



Queue:



Methods in the **`Queue`**<*type*> Class

`queue.size()` Returns the number of values in the queue.
`queue.isEmpty()` Returns **`true`** if the queue is empty.
`queue.enqueue(value)` Adds a new value to the end of the queue (which is called its ***tail***).
`queue.dequeue()` Removes and returns the value at the front of the queue (which is called its ***head***).
`queue.peek()` Returns the value at the head of the queue without removing it.
`queue.clear()` Removes all values from the queue.

Methods in the **`Queue`**<*type*> Class

Constructor

Queue()	O(1)	Initializes a new empty queue.

Methods

back()	O(1)	Returns the last value in the queue by reference.
clear()	O(1)	Removes all elements from the queue.
dequeue()	O(1)	Removes and returns the first item in the queue.
enqueue(*value*)	O(1)	Adds **value** to the end of the queue.
equals(*q*)	O(N)	Returns **true** if the two queues contain the same elements in the same order.
front()	O(1)	Returns the first value in the queue by reference.
isEmpty()	O(1)	Returns **true** if the queue contains no elements.
peek()	O(1)	Returns the first value in the queue, without removing it.
size()	O(1)	Returns the number of values in the queue.
toString()	O(N)	Converts the queue to a printable string representation.

Operators

queue1 == *queue1*	O(N)	Returns **true** if **queue1** and **queue2** contain the same elements.
queue1 != *queue2*	O(N)	Returns **true** if **queue1** and **queue2** are different.
ostream << *queue*	O(N)	Outputs the contents of the queue to the given output stream.
istream >> *queue*	O(N)	Reads the contents of the given input stream into the queue.

Methods in the STL **queue<*type*>** Class

fx Member functions

(constructor)	Construct queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
front	Access next element (public member function)
back	Access last element (public member function)
push	Insert element (public member function)
emplace C++11	Construct and insert element (public member function)
pop	Remove next element (public member function)
swap C++11	Swap contents (public member function)

fx Non-member function overloads

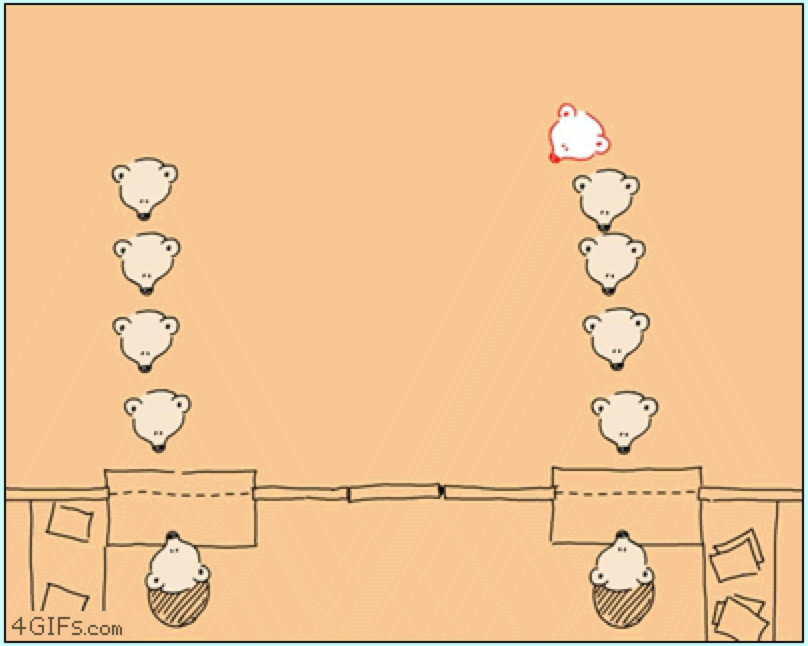
relational operators	Relational operators for queue (function)
swap (queue) C++11	Exchange contents of queues (public member function)

fx Non-member class specializations

uses_allocator<queue> C++11	Uses allocator for queue (class template)

Example: Simulate a checkout line

- One cashier is serving customers from a single queue.
- Customers arrive with a random probability and enter the queue at the end of the line.
- Whenever the cashier is free and someone is waiting in line, the cashier begins to serve that customer.
- After an appropriate service period, the cashier completes the transaction with the current customer, and is free to serve the next customer in the queue.



Example: Simulate a checkout line

- The core of the simulation is a loop that runs for the number of seconds indicated by the parameter SIMULATION_TIME. In each second, the simulation performs the following operations:
  - Determine whether a new customer has arrived according to a certain probability ARRIVAL_PROBABILITY, and, if so, add that person to the queue.
  - If the cashier is busy, note that the cashier has spent another second with the current customer. Eventually, the required service time will be complete, which will free the cashier.
  - If the cashier is free, serve the next customer in the waiting line, for a certain amount of time, randomly chosen between MIN_SERVICE_TIME and MAX_SERVICE_TIME.

ple: Simulate a checkout line

```
& nServed, int & totalWait, int & totalLength) {

0, serveTime = 0, waitTime = 0, totalCustomer = 0;

  totalWait = 0;
  totalLength = 0;
  for (int t = 0; t < SIMULATION_TIME; t++) {
    if (randomChance(ARRIVAL_PROBABILITY)) {
      queue.enqueue(t);
      cout << t << ", Customer No." << ++totalCustomer << " comes in." << endl;
    }
    if (timeRemaining > 0)
      if (--timeRemaining == 0)
        cout << t << ", Customer No." << nServed << " was served for: " << serveTime << endl;
    else
      if (!queue.isEmpty()) {
        waitTime = t - queue.dequeue();
        totalWait += waitTime;
        nServed++;
        cout << t << ", Customer No." << nServed << " waited for: " << waitTime << endl;
        timeRemaining = randomInteger(MIN_SERVICE_TIME, MAX_SERVICE_TIME);
        serveTime = timeRemaining;
      }
    totalLength += queue.size();
  }
}
```

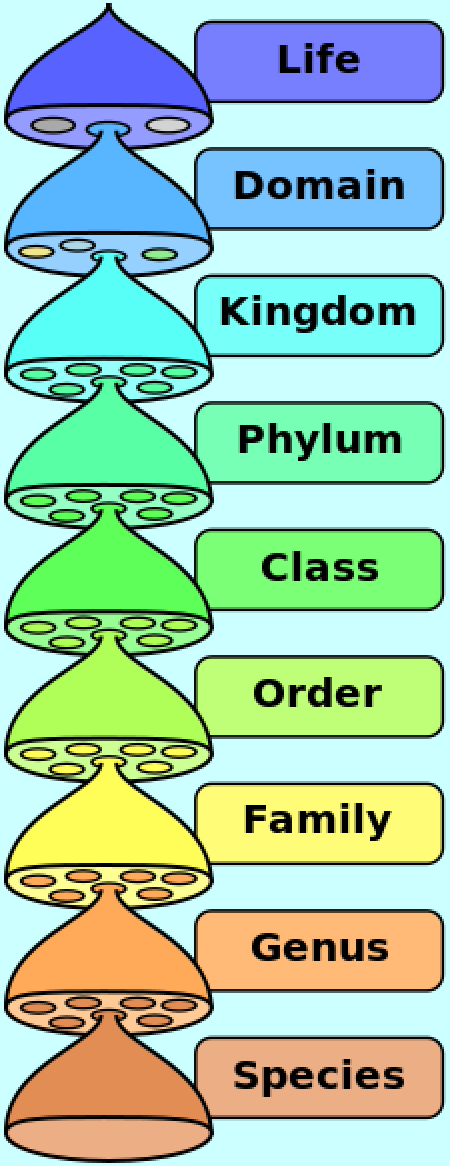
The **`Map`**`<`*type, type*`>` Class

- A ***map*** is conceptually similar to a dictionary in real life (and in Python), which allows you to look up a word to find its meaning.
- The **`Map`** class is a generalization of this idea that provides an association between an identifying tag called a ***key*** (e.g., the word in a dictionary) and an associated ***value*** (e.g., the definition of the word in the dictionary), which may be a much larger and more complicated structure.
- Map declaration: **`Map<`***key type*, *value type***`> map;`**
- The type for the keys stored in a **`Map`** must define a natural ordering, usually through a **`less`** function and/or **`<`** operator so that the keys can be compared and ordered.
- E.g., two most commonly used maps:

```
Map<string, string> dictionary;
Map<string, double> symbolTable;
```

The Map of Biological Taxonomic Rank

Key	Value
Life	Life is the characteristic that distinguishes organisms from inorganic substances and dead objects.
Domain	In biological taxonomy, a domain is the highest taxonomic rank of organisms…
Kingdom	In biology, kingdom is the third highest taxonomic rank, just below domain…
Phylum	In biology, a phylum is a taxonomic rank below kingdom and above class…
Class	…
Order	…
Family	…
Genus	…
Species	…



Methods in the `Map` Classes

- A ***map*** associates ***keys*** and ***values***. The Stanford library offers two flavors of maps, **`Map`** and **`HashMap`** (more about hash later), both of which implement the following methods:

`map.size()` Returns the number of key/value pairs in the map.	
`map.isEmpty()` Returns **`true`** if the map is empty.	
`map.put(key, value)` *or* **`map[key] = value;`** Makes an association between **`key`** and **`value`**, discarding any existing one.	
`map.get(key)` *or* **`map[key]`** Returns the most recent value associated with **`key`**.	
`map.containsKey(key)` Returns **`true`** if there is a value associated with **`key`**.	
`map.remove(key)` Removes **`key`** from the map along with its associated value, if any.	
`map.clear()` Removes all key/value pairs from the map.	

Methods in the `Map` Classes

Constructor

`Map()`	O(1)	Initializes a new empty map that associates keys and values of the specified types.

Methods

`clear()`	O(N)	Removes all entries from this map.
`containsKey(key)`	O(log N)	Returns **`true`** if there is an entry for **`key`** in this map.
`equals(map)`	O(N)	Returns **`true`** if the two maps contain the same elements.
`get(key)`	O(log N)	Returns the value associated with **`key`** in this map.
`isEmpty()`	O(1)	Returns **`true`** if this map contains no entries.
`keys()`	O(N)	Returns a **`Vector`** copy of all keys in this map.
`mapAll(fn)`	O(N)	Iterates through the map entries and calls **`fn(key, value)`** for each one.
`put(key, value)`	O(log N)	Associates **`key`** with **`value`** in this map.
`remove(key)`	O(log N)	Removes any entry for **`key`** from this map.
`size()`	O(1)	Returns the number of entries in this map.
`toString()`	O(N)	Converts the map to a printable string representation.
`values()`	O(N)	Returns a **`Vector`** copy of all values in this map.

Operators

`map[key]`	O(log N)	Selects the value associated with **`key`**.
`map1 == map1`	O(N)	Returns **`true`** if **`map1`** and **`map2`** contain the same elements.
`map1 != map2`	O(N)	Returns **`true`** if **`map1`** and **`map2`** are different.
`ostream << map`	O(N)	Outputs the contents of the map to the given output stream.
`istream >> map`	O(N log N)	Reads the contents of the given input stream into the map.

Methods in the STL **map**<*type*> Class

fx Member functions

(constructor)	Construct map (public member function)
(destructor)	Map destructor (public member function)
operator=	Copy container content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse b
rend	Return reverse iterator to reverse e
cbegin C++11	Return const_iterator to beginning
cend C++11	Return const_iterator to end (public
crbegin C++11	Return const_reverse_iterator to re
crend C++11	Return const_reverse_iterator to re

Capacity:

empty	Test whether container is empty (pu
size	Return container size (public membe
max_size	Return maximum size (public memb

Element access:

operator[]	Access element (public member func
at C++11	Access element (public member func

Modifiers:

insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_hint C++11	Construct and insert element with hint (public member function)

Observers:

key_comp	Return key comparison object (public member function)
value_comp	Return value comparison object (public member function)

Operations:

find	Get iterator to element (public member function)
count	Count elements with a specific key (public member function)
lower_bound	Return iterator to lower bound (public member function)
upper_bound	Return iterator to upper bound (public member function)
equal_range	Get range of equal elements (public member function)

Allocator:

get_allocator	Get allocator (public member function)

Using Maps in an Application

- The **AirportCodes.cpp** program reads a file associating three-letter airport codes with their locations into a **Map<string, string>**, where it can be more easily used.
- The association list is stored in a text file that looks like this:

```
PEK=Beijing, China
PKX=Beijing, China
SHA=Shanghai, China
PVG=Shanghai, China
SZX=Shenzhen, China
SFO=San Francisco, CA, USA
LHR=London, England, United Kingdom
...
```

- The program runs like this:

AirportCodes

```
Airport code: LHR
LHR is in London, England, United Kingdom
Airport code: SFO
SFO is in San Francisco, CA, USA
Airport code: XXX
There is no such airport code
Airport code:
```

```
int main() {
   Map<string,string> airportCodes;
   readCodeFile("AirportCodes.txt", airportCodes);
   while (true) {
      string line;
      cout << "Airport code: ";
      getline(cin, line);
      if (line == "") break;
      string code = toUpperCase(line);
      if (airportCodes.containsKey(code)) {
         cout << code << " is in " << airportCodes.get(code) << endl;
      } else {
         cout << "There is no such airport code" << endl;
      }
   }
   return 0;
}
void readCodeFile(string filename, Map<string,string> & map) {
   ifstream infile;
   infile.open(filename.c_str());
   if (infile.fail()) error("Can't read the data file");
   string line;
   while (getline(infile, line)) {
      if (line.length() < 4 || line[3] != '=') {
         error("Illegal data line: " + line);
      }
      string code = toUpperCase(line.substr(0, 3));
      map.put(code, line.substr(4));
   }
   infile.close();
}
```

`str.substr(pos, len)` *returns the substring of* `str` *starting at* `pos` *and continuing for* `len` *characters.*

Exercise: Symbol Tables

- A map is often called a ***symbol table*** when it is used in the context of a programming language, because it is precisely the structure you need to store variables and their values.
- For example, if you are working in an application in which you need to assign floating-point values to variable names, you could do so using a map declared as follows:

```
Map<string,double> symbolTable;
```

- Write a C++ program that declares such a symbol table and then reads in command lines from the user, which must be in one of the following forms:
  - A simple assignment statement of the form *var* **=** *number*.
  - A variable alone on a line, which is a request to display its value.
  - The command **`quit`**, which exits from the program.
  - The command **`list`**, which lists all the variables.
    - This command relies on an ***iterator*** (more later).

Exercise: Symbol Table Sample Run

SymbolTableTest

```
> pi = 3.14159
> e = 2.71828
> x = 2.00
> pi
3.14159
> x
2
> list
e = 2.71828
pi = 3.14159
x = 2
> x = 42
> a = 1.5
> list
a = 1.5
e = 2.71828
pi = 3.14159
x = 42
> quit
```

Iterating over a collection

- One of the common operations that clients need to perform when using a collection is to iterate through the elements.
- While it is easy to implement iteration for vectors and grids using `for` loops, it is less clear how you would do the same for other collection types. The modern approach to solving this problem is to use a general tool called an ***iterator*** that delivers the elements of the collection, one at a time.
- C++11 uses a ***range-based for statement*** to simplify iterators:

```
for (string key : map) { . . . code to process that key . . . }
```

- The Stanford libraries provide an alternative like this:

```
foreach (string key in map) { . . . code to process that key . . . }
```

- ***Range-based for*** (provided since C++ 11) is a way to access ***iterators*** (provided by C++ and implemented by the implementors of the collections).

Iterator Order

- When you look at the documentation for an iterator, one of the important things to determine is whether the collection class specifies the order in which elements are generated. The Stanford libraries make the following guarantees:
  – Iterators for the **`Vector`** class operate in index order.
  – Iterators for the **`Grid`** class operate in ***row-major order***, which means that the iterator runs through every element in row 0, then every element in row 1, and so on.
  – Iterators for the **`Map`** class deliver the keys in the order imposed by the standard comparison function for the key type; iterators for the **`HashMap`** class return keys in a seemingly random order.
  – Iterators for the **`Set`** class deliver the elements in the order imposed by the standard comparison function for the value type; the **`HashSet`** class is unordered.
  – Iterators for the **`Lexicon`** class always deliver words in alphabetical order.

Container class templates C++ Standard Containers

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Other:

Two class templates share certain properties with containers, and are sometimes classified with them: `bitset` and `valarray`.

The **`Set<`*`type`*`>`** Class

- This class is used to model the mathematical abstraction of a ***set***, which is a collection in which the elements are unordered and in which each value appears only once (no duplicate).
- What does this description of **`Set`** remind you of?

  The ***keys*** in a **`Map`**.
- As is so often the case, the easy way to implement the **`Set`** class is to build it on top of the **`Map`** class (more later).

Methods in the **`Set<`***type***`>`** Class

`set.size()` Returns the number of elements in the set.
`set.isEmpty()` Returns **`true`** if the set is empty.
`set.add(value)` Adds **`value`** to the set.
`set.remove(value)` Removes **`value`** from the set.
`set.contains(value)` Returns **`true`** if the set contains the specified value.
`set.clear()` Removes all words from the set.
`s1.isSubsetOf(s2)` Returns **`true`** if **`s1`** is a subset of **`s2`**.
`set.first()` Returns the first element of the set in the ordering specified by the value type.

The **`Set<`***type***`>`** Class

- **`Set`** also supports several mathematical operations (via operator overloading) based on mathematical *set theory*, in addition to the usual methods exported by the other collection classes.

Operators

s_1 **+** s_2	Returns the ***union*** of s_1 and s_2, which consists of the elements in either or both of the original sets.
s_1 ***** s_2	Returns the ***intersection*** of s_1 and s_2, which consists of the elements common to both of the original sets.
s_1 **-** s_2	Returns the ***set difference*** of s_1 and s_2, which consists of the all elements in s_1 that are not present in s_2.
s_1 **+=** s_2 s_1 **-=** s_2 s_1 ***=** s_2	The **+**, **-**, and ***** operators can be combined with assignment just as they can with numeric values. For **+=** and **-=**, the right hand value can be a set, a single value, or a list of values separated by commas.

Methods in the `Set<`*type*`>` Class

Constructor

Set()	O(1)	Creates an empty set of the specified element type.

Methods

add(*value*)	O(log N)	Adds an element to this set, if it was not already there.
clear()	O(N)	Removes all elements from this set.
contains(*value*)	O(log N)	Returns **true** if the specified value is in this set.
equals(*set*)	O(N)	Returns **true** if the two sets contain the same elements.
first()	O(log N)	Returns the first value in the set in the order established by a for-each loop.
isEmpty()	O(1)	Returns **true** if this set contains no elements.
isSubsetOf(*set2*)	O(N)	Implements the subset relation on sets.
mapAll(*fn*)	O(N)	Iterates through the elements of the set and calls **fn(value)** for each one.
remove(*value*)	O(log N)	Removes an element from this set.
size()	O(1)	Returns the number of elements in this set.
toString()	O(N)	Converts the set to a printable string representation.

Operators

set1 == *set2*	O(N)	Returns **true** if **set1** and **set2** contain the same elements.
set1 != *set2*	O(N)	Returns **true** if **set1** and **set2** are different.
set1 + *set2*	O(N)	Returns the union of sets **set1** and **set2**, which is the set of elements that appear in at least one of the two sets.
set + *value*	O(N)	Returns the union of set **set1** and individual value **value**.
set1 += *set2*;	O(N)	Adds all of the elements from **set2** (or the single specified value) to **set1**.
set += *value*;	O(log N)	Adds the single specified value to the set.
set1 - *set2*	O(N)	Returns the difference of sets **set1** and **set2**, which is all of the elements that appear in **set1** but not **set2**.
set - *value*	O(N)	Returns the set **set** with **value** removed.
set1 -= *set2*;	O(N)	Removes the elements from **set2** (or the single specified value) from **set1**.
set -= *value*;	O(log N)	Removes the single specified value from the set.
set1 * *set2*	O(N)	Returns the intersection of sets **set1** and **set2**, which is the set of all elements that appear in both.
set1 *= *set2*;	O(N)	Removes any elements from **set1** that are not present in **set2**.
ostream << *set*	O(N)	Outputs the contents of the set to the given output stream.
istream >> *set*	O(N log N)	Reads the contents of the given input stream into the set.

Methods in the STL set<*type*> Class

fx Member functions

(constructor)	Construct set (public member function)
(destructor)	Set destructor (public member function)
operator=	Copy container content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse l
rend	Return reverse iterator to reverse e
cbegin C++11	Return const_iterator to beginning
cend C++11	Return const_iterator to end (public
crbegin C++11	Return const_reverse_iterator to re
crend C++11	Return const_reverse_iterator to re

Capacity:

empty	Test whether container is empty (p
size	Return container size (public membe
max_size	Return maximum size (public memb

Modifiers:

insert	Insert element (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_hint C++11	Construct and insert element with hint (public member function)

Observers:

key_comp	Return comparison object (public member function)
value_comp	Return comparison object (public member function)

Operations:

find	Get iterator to element (public member function)
count	Count elements with a specific value (public member function)
lower_bound	Return iterator to lower bound (public member function)
upper_bound	Return iterator to upper bound (public member function)
equal_range	Get range of equal elements (public member function)

Allocator:

get_allocator	Get allocator (public member function)

Flashback

The <cctype> (ctype.h) Interface

`bool isdigit(char ch)` `{return ch >= '0' && ch <= '9';}`   Determines if the specified character is a digit.
`bool isalpha(char ch)` `{return ALPHA_SET.contains(ch);}`   Determines if the specified character is a letter.
`bool isalnum(char ch)`   Determines if the specified character is a letter or a digit.
`bool islower(char ch)`   Determines if the specified character is a lowercase letter.
`bool isupper(char ch)`   Determines if the specified character is an uppercase letter.
`bool isspace(char ch)`   Determines if the specified character is ***whitespace*** (spaces and tabs).
`char tolower(char ch)`   Converts **`ch`** to its lowercase equivalent, if any. If not, **`ch`** is returned unchanged.
`char toupper(char ch)`   Converts **`ch`** to its uppercase equivalent, if any. If not, **`ch`** is returned unchanged.

Implementing the `<cctype>` Library

- `Set`-based implementation of the `<cctype>` (Ch. 3) Library:

```
const Set<char> DIGIT_SET =
         setFromString("0123456789");
const Set<char> LOWER_SET =
         setFromString("abcdefghijklmnopqrstuvwxyz");
const Set<char> UPPER_SET =
         setFromString("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
const Set<char> PUNCT_SET =
         setFromString("!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~");
const Set<char> SPACE_SET =
         setFromString(" \t\v\f\n\r");
const Set<char> XDIGIT_SET =
         setFromString("0123456789ABCDEFabcdef");
const Set<char> ALPHA_SET =
         LOWER_SET + UPPER_SET;
const Set<char> ALNUM_SET =
         ALPHA_SET + DIGIT_SET;
const Set<char> PRINT_SET =
         ALNUM_SET + PUNCT_SET + SPACE_SET;
```

Implementing the `<cctype>` Library

- **`Set`**-based implementation of the `<cctype>` (Ch. 3) Library:

```
Set<char> setFromString(string str)  {
   Set<char> set;
   for (int i = 0; i < str.length(); i++) {
      set.add(str[i]);
   }
   return set;
}

bool isalnum(char ch) { return ALNUM_SET.contains(ch); }
bool isalpha(char ch) { return ALPHA_SET.contains(ch); }
bool isdigit(char ch) { return DIGIT_SET.contains(ch); }
bool islower(char ch) { return LOWER_SET.contains(ch); }
bool isprint(char ch) { return PRINT_SET.contains(ch); }
bool ispunct(char ch) { return PUNCT_SET.contains(ch); }
bool isspace(char ch) { return SPACE_SET.contains(ch); }
bool isupper(char ch) { return UPPER_SET.contains(ch); }
bool isxdigit(char ch) { return XDIGIT_SET.contains(ch); }
```

The **`Lexicon`** Class

- A set of words with no associated definitions is called a ***lexicon***.
- Create an English word list using **`Set`** with the following code:

```
Set<string> english;
ifstream infile;
infile.open("EnglishWords.txt");
if (infile.fail())
   error("Can't open EnglishWords.txt");
string word;
while (getline(infile, word)) {
   english.add(word);
}
infile.close();
```

- Or simply: `Lexicon english("EnglishWords.txt");`
- **`Lexicon`** also supports a space-efficient precompiled binary format (.dat) of text files, defined using a Directed Acyclic Word Graph (DAWG) data structure.

Methods in the `Lexicon` Class

`lexicon.size()` Returns the number of words in the lexicon.
`lexicon.isEmpty()` Returns **`true`** if the lexicon is empty.
`lexicon.add(word)` Adds **`word`** to the lexicon, always in lowercase.
`lexicon.addWordsFromFile(filename)` Adds all the words in the specified file to the lexicon.
`lexicon.contains(word)` Returns **`true`** if the lexicon contains the specified word.
`lexicon.containsPrefix(prefix)` Returns **`true`** if the lexicon contains any word beginning with **`prefix`**.
`lexicon.clear()` Removes all words from the lexicon.

Methods in the `Lexicon` Class

Constructor

`Lexicon()`	O(1)	Initializes a new empty lexicon.
`Lexicon(filename)` `Lexicon(istream)`	O(WN)	Initializes a new lexicon that reads words from the given file or input stream.

Methods

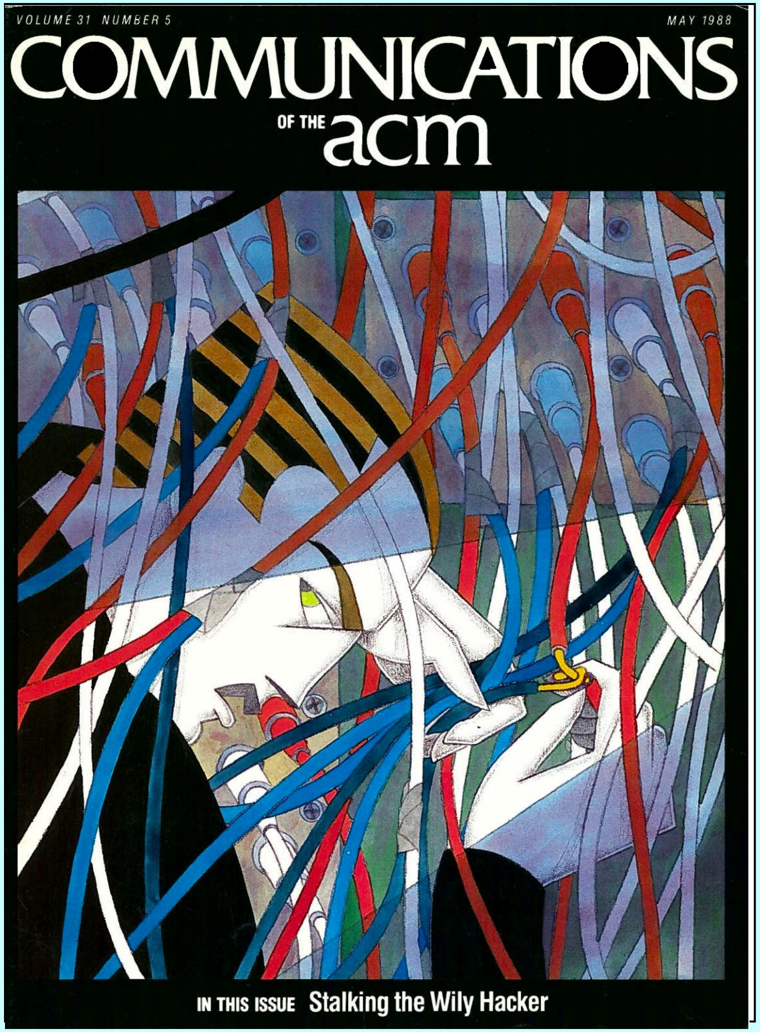
`add(word)`	O(W)	Adds the specified word to the lexicon.
`addWordsFromFile(filename)` `addWordsFromFile(istream)`	O(WN)	Reads the given file/stream and adds all of its words to the lexicon.
`clear()`	O(N)	Removes all words from the lexicon.
`contains(word)`	O(W)	Returns **`true`** if **`word`** is contained in the lexicon.
`containsPrefix(prefix)`	O(W)	Returns true if any words in the lexicon begin with **`prefix`**.
`equals(lex)`	O(WN)	Returns **`true`** if the two lexicons contain the same words.
`isEmpty()`	O(1)	Returns **`true`** if the lexicon contains no words.
`mapAll(fn)`	O(N)	Calls the specified function on each word in the lexicon.
`remove(word)`	O(W)	Removes the specified word from the lexicon.
`removePrefix(prefix)`	O(W)	Removes all words that begin with the specified prefix from the lexicon.
`size()`	O(1)	Returns the number of words contained in the lexicon.
`toString()`	O(1)	Converts the lexicon to a printable string representation.

Operators

`lex1 == lex1`	O(WN)	Returns **`true`** if **`lex1`** and **`lex2`** contain the same elements.
`lex1 != lex2`	O(WN)	Returns **`true`** if **`lex1`** and **`lex2`** are different.
`ostream << lex`	O(N)	Outputs the contents of the lexicon to the given output stream.
`istream >> lex`	O(WN)	Reads the contents of the given input stream into the lexicon.

Why Do Both `Lexicon` and `Set` Exist?

- The `Lexicon` representation is extremely space-efficient. The data structure used in the library implementation stores the full English dictionary in 350,000 bytes, which is shorter than a text file containing those words.
- The underlying representation makes it possible to implement a `containsPrefix` method that is useful in many applications.
- The representation makes it easy for ***iterators*** to process the words in alphabetical order.



Example: TwoLetterWords (1)

```
/* Program to generate all the two-letter English words */
#include <iostream>
#include "lexicon.h"
#include "foreach.h" /* range-based for in Stanford */

using namespace std;
int main() {
   Lexicon english("EnglishWords.txt");
   foreach (string word in english) { /* Stanford */
   // for (string word : english) { /* Standard C++11 */
      if (word.length() == 2) cout << word << endl;
   }
   return 0;
}
```

Example: TwoLetterWords (2)

```
/* Program to generate all the two-letter English words */
#include <iostream>
#include "lexicon.h"
using namespace std;
int main() {
   Lexicon english("EnglishWords.txt");
   string word("aa");
   for (char c0 = 'a'; c0 <= 'z'; c0++) {
      word[0] = c0;
      for (char c1 = 'a'; c1 <= 'z'; c1++) {
         word[1] = c1;
         if (english.contains(word)) cout << word << endl;
      }
   }
   return 0;
}
```

- Question: which strategy is better?
  Depending on the speed of search, i.e. `contains()`.

Exercise: Finding Anagrams

- Write a program that reads in a set of letters and sees whether any anagrams of that set of letters are themselves words:

FindAnagrams

```
Enter tiles: ehprsyz
zephyrs
Enter tiles: aeinstr
anestri
nastier
ratines
retains
retinas
retsina
stainer
stearin
```

- Generating all anagrams of a word is not a simple task. Most solutions require some tricky ***recursion***, but can you think of another way to solve this problem?

  Hint: What if you had a function that sorts the letters in a word? Would that help?

The End