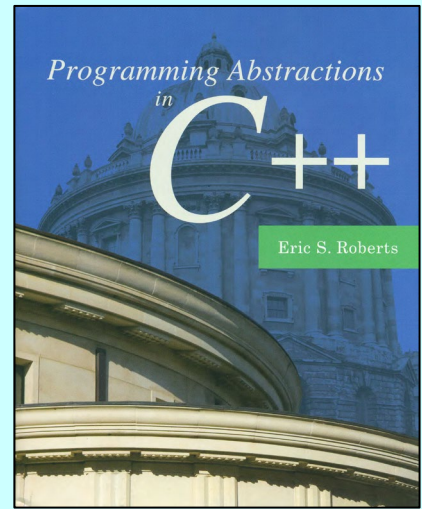


## CHAPTER 20

# Strategies for Iteration

*What needs this iteration.*

—William Shakespeare, *Othello*, ~1603



[20.1 Using iterators](#)

[20.2 Implementing iterators](#)

[20.3 Using functions as data values](#)

[20.4 Encapsulating data with functions](#)

[20.5 The STL algorithm library](#)

[20.6 Functional programming in C++](#)

# Introduction to the C++ 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, ctype, cstring, cstdio, cstdlib, etc.

# 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 the implementors of the collections).

# Using Iterators in C++

- When you work with strings, one of the most important patterns involves iterating through the characters in a string, which requires the following code (index-based loop):

```
for (int i = 0; i < str.length(); i++) {  
    ... Body of loop that manipulates str[i] ...  
}
```

- For collection classes, the **range-based for loop** looks like:

```
for (string word : english) {  
    ... Body of loop involving word ...  
}
```

- The C++ compiler translates the range-based **for** loop using ***iterators***, which is what you would do **before C++11**:

```
for (Lexicon::iterator it = english.begin();  
     it != english.end(); it++) {  
    ... Body of loop involving *it ...  
}
```



# Using Iterators in C++

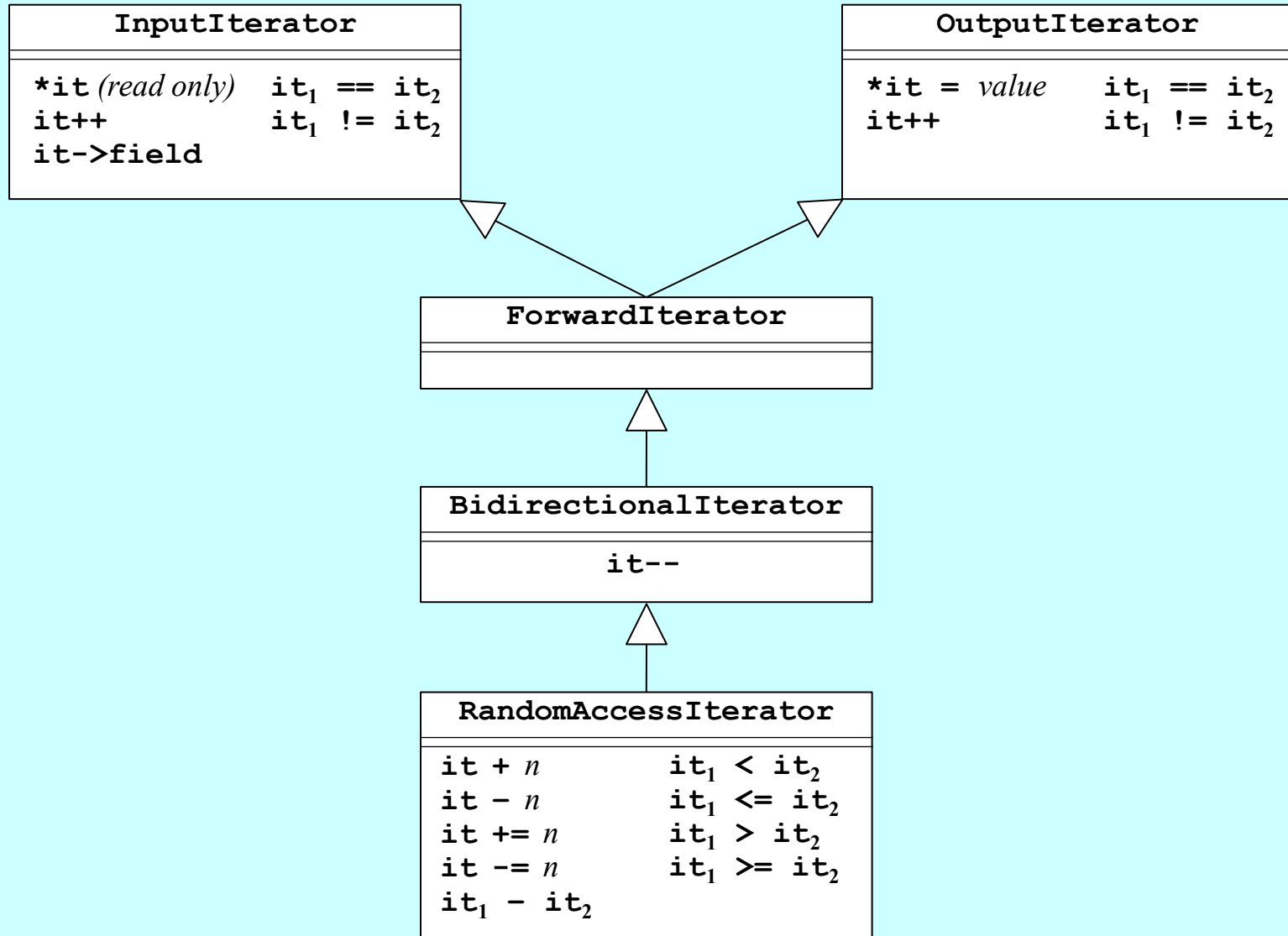
- The C++ Standard Template Library makes extensive use of an abstract data type called an **iterator**, which supports stepping through a collection one element at a time.
- Every collection class in the STL exports an **iterator** type (a **nested type**) along with two standard methods that produce iterators. The **begin** method returns an iterator positioned **at the beginning of the collection**. The **end** method returns an iterator positioned **just past the final element**.
- Iterators in C++ use a syntax derived from pointers. Given an iterator, one reads the corresponding value by dereferencing the iterator variable and increments it using the **++** operator. The pattern for using an iterator to loop over a collection **c** is:

```
for (ctype::iterator it = c.begin(); it != c.end(); it++) {  
    ... Body of loop involving *it ...  
}
```

# Example: TwoLetterWords

```
int main() {
    Lexicon english("EnglishWords.dat");
    for (Lexicon::iterator it = english.begin(); it != english.end(); it++) {
        string word = *it;                // method 1
        if (word.length() == 2) {
            cout << word << endl;
        }
        if (it->length() == 2) {           // method 2
            cout << *it << endl;
        }
    }
    Lexicon::iterator it = english.begin();
    while (it != english.end()) {
        string word = *it;                // method 3
        if (word.length() == 2) {
            cout << word << endl;
        }
        if (it->length() == 2) {           // method 4
            cout << *it << endl;
        }
        it++;
    }
    return 0;
}
```

# The C++ Iterator Hierarchy



# Example: The **Vector** Iterator

- The **Lexicon** class supports only the **InputIterator** level of service, while the iterator for the **Vector** class is a **RandomAccessIterator**.

```
Vector<int>::iterator it = v.end();  
while (it != v.begin()) {  
    cout << *--it << endl;  
}  
  
for (Vector<int>::iterator it = v.begin(); it < v.end(); it += 2) {  
    cout << *it << endl;  
}
```

- Using iterators in such **unusual ways** often ends up making programs **difficult to maintain**, much like code that tries to be too clever with its pointer manipulation. The best way to specify iteration is to use the (range-based) **for** loop whenever you can, because doing so **hides the iterators** altogether.



# Implementation of the **Vector** Iterator

- Iterators are considerably easier to implement for **Vector** (and, e.g., **Grid** and **HashMap**) than they are for most of the other collection classes.
- Implementing **iterator** for the **Vector** class presents a relatively straightforward challenge, because the underlying structure of the vector is defined in terms of a simple dynamic array, and the only state information the iterator needs to maintain is **the current index value**, along with **a pointer back to the **Vector** object itself**.
- Iterators for tree-structured classes like **Map** turn out to be enormously tricky, mostly because the implementation has to translate the **recursive** structure of the data into an **iterative** form.
- As a general rule, it is wise to **leave the implementation of iterators to experts**, in much the same way as random number generators, hash functions, and sorting algorithms.



## TUTORIAL

# Implementation of the Vector Iterator

template<typename ValueType>

methods of class Vector we have studied before  
: { // nested class iterator defined inside class Vector

public:

... // public methods of class iterator

private:

const Vector \*vp;

/\* Pointer to the Vector object \*/

int index;

/\* Index for this iterator \*/

iterator(const Vector \*vp, int index) { // private constructor

this->vp = vp;

this->index = index;

}

friend class Vector;

} // end of class iterator

iterator begin() const { // public method of class Vector

return iterator(this, 0);

}

iterator end() const { // public method of class Vector

return iterator(this, count);

}

private:

ValueType \*array;

/\* A dynamic array of the elements \*/

int capacity;

/\* The allocated size of the array \*/

int count;

/\* The number of elements in use \*/

... // private methods of class Vector

}

# Implementation of the Vector Iterator

```
/*
 * Nested class: iterator
 * -----
 * This nested class implements a standard iterator for the Vector class.
 */

class iterator {

public:

/*
 * Implementation notes: iterator constructor
 * -----
 * The default constructor for the iterator returns an invalid iterator
 * in which the vector pointer vp is set to NULL. Iterators created by
 * the client are initialized by the constructor iterator(vp, k), which
 * appears in the private section.
 */

    iterator() {
        this->vp = NULL;
    }
```

# Implementation of the Vector Iterator

```
/*
 * Implementation notes: dereference operator
 * -----
 * The * dereference operator returns the appropriate index position in
 * the internal array by reference.
 */

ValueType & operator*() {
    if (vp == NULL) error("Iterator is uninitialized");
    if (index < 0 || index >= vp->count) error("Iterator out of range");
    return vp->array[index];
}

/*
 * Implementation notes: -> operator
 * -----
 * Overrides of the -> operator in C++ follow a special idiomatic pattern.
 * The operator takes no arguments and returns a pointer to the value.
 * The compiler then takes care of applying the -> operator to retrieve
 * the desired field.
 */

ValueType* operator->() {
    if (vp == NULL) error("Iterator is uninitialized");
    if (index < 0 || index >= vp->count) error("Iterator out of range");
    return &vp->array[index];
}
```

# Implementation of the Vector Iterator

```
/*
 * Implementation notes: selection operator
 * -----
 * The selection operator returns the appropriate index position in
 * the internal array by reference.
 */

ValueType & operator[](int k) {
    if (vp == NULL) error("Iterator is uninitialized");
    if (index + k < 0 || index + k >= vp->count) {
        error("Iterator out of range");
    }
    return vp->array[index + k];
}

/*
 * Implementation notes: relational operators
 * -----
 * These operators compare the index field of the iterators after making
 * sure that the iterators refer to the same vector.
 */

bool operator==(const iterator & rhs) {
    if (vp != rhs.vp) error("Iterators are in different vectors");
    return vp == rhs.vp && index == rhs.index;
}
```

# Implementation of the Vector Iterator

```
bool operator!=(const iterator & rhs) {
    if (vp != rhs.vp) error("Iterators are in different vectors");
    return !(*this == rhs);
}

bool operator<(const iterator & rhs) {
    if (vp != rhs.vp) error("Iterators are in different vectors");
    return index < rhs.index;
}

bool operator<=(const iterator & rhs) {
    if (vp != rhs.vp) error("Iterators are in different vectors");
    return index <= rhs.index;
}

bool operator>(const iterator & rhs) {
    if (vp != rhs.vp) error("Iterators are in different vectors");
    return index > rhs.index;
}

bool operator>=(const iterator & rhs) {
    if (vp != rhs.vp) error("Iterators are in different vectors");
    return index >= rhs.index;
}
```

# Implementation of the Vector Iterator

```
/*
 * Implementation notes: ++ and -- operators
 * -----
 * These operators increment or decrement the index. The suffix versions
 * of the operators, which are identified by taking a parameter of type
 * int that is never used, are more complicated and must copy the original
 * iterator to return the value prior to changing the count.
 */

iterator & operator++() { // ++iterator
    if (vp == NULL) error("Iterator is uninitialized");
    index++;
    return *this;
}

iterator operator++(int) { // iterator++
    iterator copy(*this);
    operator++();
    return copy;
}
```

# Implementation of the Vector Iterator

```
iterator & operator--() { // --iterator
    if (vp == NULL) error("Iterator is uninitialized");
    index--;
    return *this;
}
```

```
iterator operator--(int) { // iterator--
    iterator copy(*this);
    operator--();
    return copy;
}
```

```
/*
 * Implementation notes: arithmetic operators
 * -----
 * These operators update the index field by the increment value k.
 */
```

```
iterator operator+(const int & k) {
    if (vp == NULL) error("Iterator is uninitialized");
    return iterator(vp, index + k);
}
```

```
iterator operator-(const int & k) {
    if (vp == NULL) error("Iterator is uninitialized");
    return iterator(vp, index - k);
}
```



# Implementation of the Vector Iterator

```
int operator-(const iterator & rhs) {
    if (vp == NULL) error("Iterator is uninitialized");
    if (vp != rhs.vp) error("Iterators are in different vectors");
    return index - rhs.index;
}

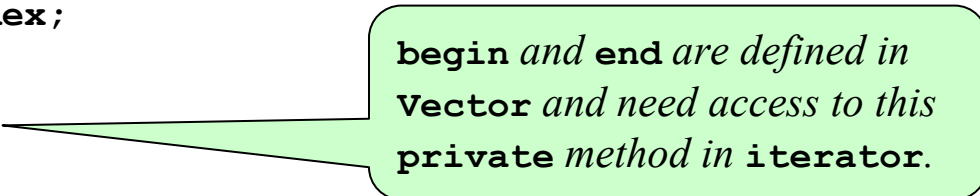
/* Private section */

private:
    const Vector *vp;           /* Pointer to the Vector object */
    int index;                  /* Index for this iterator */

/*
 * Implementation notes: private constructor
 * -----
 * The begin and end methods use the private constructor to create iterators
 * initialized to a particular position. The Vector class must therefore be
 * declared as a friend so that begin and end can call this constructor.
 */

    iterator(const Vector *vp, int index) {
        this->vp = vp;
        this->index = index;
    }

    friend class Vector;
};
```



*begin and end are defined in Vector and need access to this private method in iterator.*

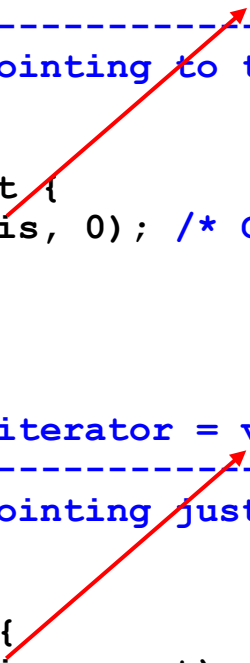
# Implementation of the **Vector** Iterator

```
/*
 * Function: begin
 * Usage: Vector<type>::iterator = vec.begin();
 * -----
 * Returns an iterator pointing to the first element.
 */

iterator begin() const {
    return iterator(this, 0); /* Calling the private constructor */
}

/*
 * Function: end
 * Usage: Vector<type>::iterator = vec.end();
 * -----
 * Returns an iterator pointing just beyond the last element.
 */

iterator end() const {
    return iterator(this, count); /* Calling the private constructor */
}
```

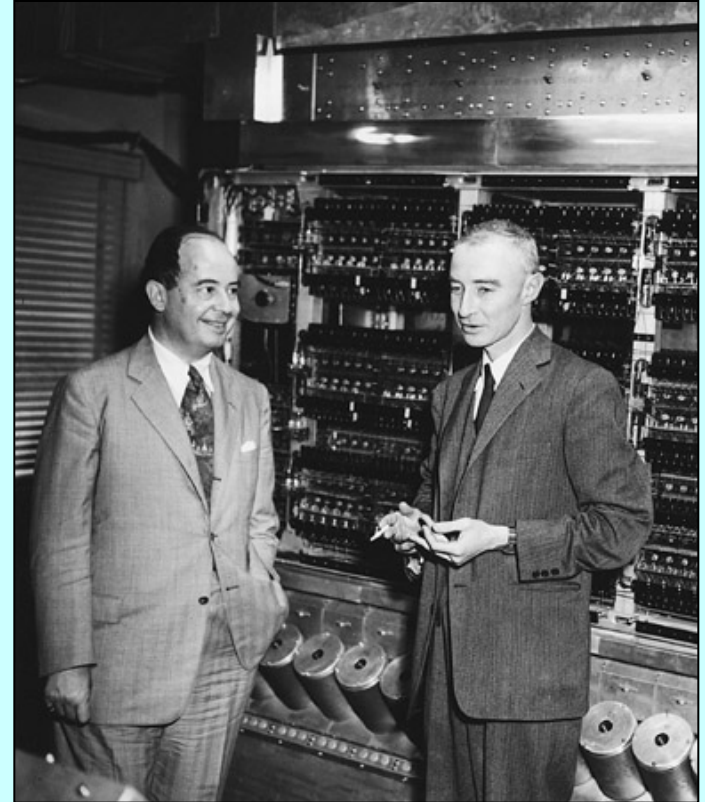


# Yet Another Collection Iterating Strategy

- A different model for applying an operation to every element in a collection is to allow clients to apply a function to each element of the collection in turn.
- Functions that allow you to call a function on every element in a collection are called *mapping functions*.
- Mapping functions are less convenient than iterators and are consequently used less often. They are, however, easier to implement.
- Mapping functions, moreover, are increasingly important to computer science, particularly with the development of massively parallel applications (e.g., **MapReduce**).
- Before introducing the mapping functions, however, it helps to introduce a more general programming concept that is also integral to the design of the STL, which is the ability **to use functions as part of the data structure**.

# The von Neumann Architecture

- One of the foundational ideas of modern computing—traditionally attributed to John von Neumann although others can make valid claims to the idea—is that **code is stored in the same memory as data**. This concept is called the *stored programming model*.
- If you go on to take the Computer Organization course, you will learn more about how code is represented inside the computer. For now, the important idea is that the code for every C++ function is stored somewhere in memory and therefore has an address.



John von Neumann and J. Robert Oppenheimer



# Callback Functions

- The ability to determine the address of a function makes it possible to **pass functions as parameters to other functions**.
- In this example, the function **main** makes two calls to **plot**.
- The first call passes the address of **sin**, which is **0028**.
- The second passes the address of **cos**, which is **0044**.
- The **plot** function can call this function supplied by the caller. Such functions are known as **callback functions**.

```
int main() {  
    GWindow gw;  
    plot(gw, sin, ...);  
    plot(gw, cos, ...);  
    return 0;  
}
```

```
double sin(double x) {  
    ... code to compute sin(x) ...  
}
```

```
double cos(double x) {  
    ... code to compute cos(x) ...  
}
```

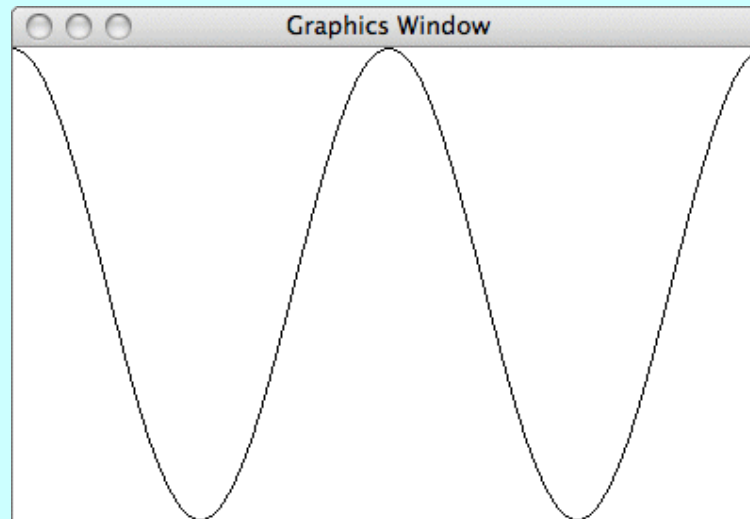
	0000
	0004
	0008
	000C
	0010
	0014
	0018
	001C
	0020
	0024
	0028
	002C
	0030
	0034
	0038
	003C
	0040
	0044
	0048
	004C
	0050
	0054
	0058
	005C
	0060
	0064

# Plotting a Function

- This section defines a **plot** function that draws the graph of a function on the graphics window.
- The arguments to **plot** are the graphics window, the function, and the limits in the  $x$  and  $y$  directions. For example, calling

```
plot(gw, cos, -2 * PI, 2 * PI, -1.0, 1.0);
```

produces the following output:



# Example: Defining the `plot` Function

## 1. What is the prototype for the `plot` function?

*There is no variable name here, why?*

```
void plot(GWindow & gw, double (*fn)(double),
          double minX, double maxX,
          double minY, double maxY);
```

```
double width = gw.getWidth();
double height = gw.getHeight();
double nSteps = int(width);
double dx = (maxX - minX) / nSteps;
double sx0 = 0;
double sy0 = height - (fn(minX) - minY) / (maxY - minY) * height;
for (int i = 1; i < nSteps; i++) {
    double x = minX + i * dx;
    double y = fn(x);
    double sx1 = (x - minX) / (maxX - minX) * width;
    double sy1 = height - (y - minY) / (maxY - minY) * height;
    gw.drawLine(sx0, sy0, sx1, sy1);
    sx0 = sx1;
    sy0 = sy1;
}
```

*Think about the parameterized `Graph` class, where virtual types are defined by the clients with certain required fields, here we have a function which will be defined by the clients with certain signature and return type.*



# Function Pointers in C++

- One of the hardest aspects of function pointers in C++ is writing the type for the function used in its declaration.
- The syntax for declaring function pointers is consistent with the syntax for other pointer declarations, although it takes some getting used to. Consider the following declarations:

<code>double x;</code>	<i>Declares <b>x</b> as a <b>double</b>.</i>
<code>double list[n];</code>	<i>Declares <b>list</b> as an array of <b>n doubles</b>.</i>
<code>double *px;</code>	<i>Declares <b>px</b> as a pointer to a <b>double</b>.</i>
<code>double **ppx;</code>	<i>Declares <b>ppx</b> as a pointer to a pointer to a <b>double</b>.</i>
<code>double f(double);</code>	<i>Declares <b>f</b> as a function taking and returning a <b>double</b>.</i>
<code>double *g(double);</code>	<i>Declares <b>g</b> as a function taking a <b>double</b> and returning a pointer to a <b>double</b>.</i>
<code>double (*fn)(double);</code>	<i>Declares <b>fn</b> as <b>a pointer to a function</b> taking and returning a <b>double</b>.</i>



# The Selection Sort Algorithm

- Of the many sorting algorithms, the easiest one to describe is *selection sort*, which appears in the text like this:

```
void sort(Vector<int> & vec) {  
    int n = vec.size();  
    for (int lh = 0; lh < n; lh++) {  
        int rh = lh;  
        for (int i = lh + 1; i < n; i++) {  
            if (vec[i] < vec[rh]) rh = i;  
        }  
        int temp = vec[lh];  
        vec[lh] = vec[rh];  
        vec[rh] = temp;  
    }  
}
```

# Comparison Functions

```
// Here is a comparison function that sorts in ascending order
bool ascending(int x, int y) {
    return x < y; // true if the first element is smaller than the second
}

// Here is a comparison function that sorts in descending order
bool descending(int x, int y) {
    return x > y; // true if the first element is greater than the second
}

// Note our user-defined comparison is the third parameter
void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int)) {
    ...
    if (comparisonFcn(array[currentIndex], array[bestIndex]))
        bestIndex = currentIndex;
    ...
}

int main() {
    int array[9] { 3, 7, 9, 5, 6, 1, 8, 2, 4 };
    // Sort the array in ascending order using the ascending() function
    selectionSort(array, 9, ascending);
    // Sort the array in descending order using the descending() function
    selectionSort(array, 9, descending);
}
```



# Implementing Sets

- Modern library systems adopt either of two strategies for implementing sets:
  1. ***Hash tables***. Sets implemented as hash tables are extremely efficient, offering average  $O(1)$  performance for adding a new element or testing for membership. The primary disadvantage is that **hash tables do not support ordered iteration**.
  2. ***Balanced binary trees***. Sets implemented using balanced binary trees offer  $O(\log N)$  performance on the fundamental operations, but do make it **possible to write an ordered iterator**.
- The **set** class in standard C++ uses the latter approach.
- One of the implications of using the BST representation is that the underlying value type must support the comparison operators `==` and `<`. In Chapter 20, you'll learn how to relax that restriction by specifying a **comparison function** as an argument to the **Set** constructor.



# Mapping Functions

- The ability to work with pointers to functions offers one solution to the problem of iterating through the elements of a collection.
- To use this approach, the collection must export a *mapping function* that applies a client-specified function to every element of the collection.
- Functions that allow you to call a function on every element in a collection are called *mapping functions*.
- Most collections in the Stanford libraries export the method that calls `fn` on every element of the collection.

```
template <typename ValueType>  
void mapAll(void (*fn) (ValueType)) ;
```

# Implementing Mapping Functions

```
/*
 * Implementation notes: mapAll
 * -----
 * This method uses a for loop to call fn on every element.
 */

template <typename ValueType>
void Vector<ValueType>::mapAll(void (*fn) (ValueType)) const {
    for (int i = 0; i < count; i++) {
        fn(array[i]);
    }
}
```

- The **mapAll** function is at the same level of the **iterator** class, therefore can get access to the low level data structure directly, without relying on the definition of **iterator**.

# Using `mapAll` to Iterate

- As an example, you can print all the elements of an integer vector `Vector<int> v` using an iterator, where `printInt` is defined separately:

```
void printInt(int n) { cout << n << endl; }  
Vector<int> v;  
for (Vector<int>::iterator it = v.begin();  
     it < v.end(); it++) {  
    printInt(*it);  
}
```

- If, on the other hand, a mapping function is defined in `Vector`

```
template <typename ValueType>  
void mapAll(void (*fn) (ValueType)) ;
```

which can perform `fn` on every element of the collection, then you can simply call the following without using an iterator:

```
v.mapAll(printInt);
```

*There is no parameter list here, why?  
Does this work for `Vector<char>`?*

# Using `mapAll` to Iterate

- Another example, the `Lexicon` class exports a mapping function called `mapAll` with the signature

```
void mapAll(void (*fn)(string));
```

- The existence of `mapAll` in the `Lexicon` class makes it possible to recode the `TwoLetterWords` program as

```
void printTwoLetterWords(string word) {  
    if (word.length() == 2) { cout << word << endl; }  
}  
int main() {  
    Lexicon english("EnglishWords.dat");  
    english.mapAll(printTwoLetterWords);  
}
```

as opposed to using an iterator

```
for (Lexicon::iterator it = english.begin();  
     it != english.end(); it++) {  
    printTwoLetterWords(*it);  
}
```

# Implementing Mapping Functions

**FIGURE 20-5** Implementation of the mapAll function for the Map class

```
/*
 * Implementation notes: mapAll
 * -----
 * The exported version of mapAll uses a private helper method that takes
 * the tree as an argument and performs a standard inorder traversal,
 * calling fn(key, value) for every key-value pair.
 */

template <typename KeyType,typename ValueType>
void Map<KeyType,ValueType>::mapAll(void (*fn)(KeyType, ValueType)) const {
    mapAll(root, fn);
}

template <typename KeyType,typename ValueType>
void Map<KeyType,ValueType>::mapAll(BSTNode *t,
                                   void (*fn)(KeyType, ValueType)) const {
    if (t != NULL) {
        mapAll(t->left, fn);
        fn(t->key, t->value);
        mapAll(t->right, fn);
    }
}
```

*What is the traversal order here, why?*



# Implementing Mapping Functions

Exercise: Implement the mapping function

```
void mapAll(void (*fn)(string, string));
```

as part of the hashing based **StringMap** class, for which the private section looks like this:

```
/* Type definition for cells in the bucket chain */

struct Cell {
    std::string key;
    std::string value;
    Cell *link;
};

/* Constant definitions */

static const int INITIAL_BUCKET_COUNT = 13;

/* Instance variables */

Cell **buckets;          /* Dynamic array of pointers to cells */
int nBuckets;            /* The number of buckets in the array */
int count;               /* The number of entries in the map */
```

# Exercise: KLetterWords

- If you use the range-based **for** loop, this function is straightforward to code:

```
void listWordsOfLengthK(const Lexicon & lex, int k) {  
    for (string word : lex) {  
        if (word.length() == k) {  
            cout << word << endl;  
        }  
    }  
}
```

- If you try to use the **mapAll** method instead, the callback function needs access to the value of **k**, the client must somehow pass **k** to the mapping function of the collection class, which must then turn around and pass that same value back to the client's callback function.
- The fundamental problem is that **clients usually need to pass information to the callback function beyond the parameters supplied by the collection class.**


# Exercise: KLetterWords

- Either the client of the collection class has to write many different `print` functions with different `k` values specified and fixed, or the implementer of the collection has to write many different `mapAll` functions to take more parameters.

```
void printKLetterWords(string word, int k) {
    if (word.length() == k) { cout << word << endl; }
}

int main() {
    Lexicon english("EnglishWords.dat");
    english.mapAll(printKLetterWords, k);
}

template <typename ValueType>
void Lexicon::mapAll(void (*fn)(ValueType, int), int k) const {
    ... traverse the DAWG structure to get a string word...
    fn(word, k);
}
}
```



- Neither is a good idea. **Encapsulation** to the rescue!



# Passing Data to Mapping Functions

- The biggest problem with using mapping functions is that it is difficult to **pass client information** from the client back to the callback function. The C++ packages that support callback functions typically support two different strategies for achieving this goal:
  1. Passing an **additional argument** to the mapping function, which is then included in the set of arguments to the callback function (overloading **mapAll** in many forms).
  2. Passing a **function object** or **functors** to the mapping function. A function object is simply any object that overloads the **function-call operator**, which is designated in C++ as **operator()**.
- The text goes through each of these strategies in the context of a program that writes out every word in the English lexicon that has a particular length.

# Passing Data Using Function Object

- Now you can list the words in the lexicon whose length is **k** by passing **k** to the callback function **ListKLetterWords** through the mapping function **mapAll**.

```
class ListKLetterWords {
public:
    ListKLetterWords(int k) {
        this->k = k;
    }
    int operator()(string word) {
        if (word.length() == k) {
            cout << word << endl;
        }
    }
private:
    int k;      /* Length of desired words */
};

void listWordsOfLengthK(const Lexicon & lex, int k) {
    lex.mapAll(ListKLetterWords(k));
}
```

```

/* Method: mapAll
 * Usage: lexicon.mapAll(fn);
 * -----
 * Calls the specified function on each word in the lexicon.
 */
void mapAll(void (*fn)(std::string)) const;
void mapAll(void (*fn)(const std::string&)) const;
template <typename FunctorType>
void mapAll(FunctorType fn) const;

void Lexicon::mapAll(void (*fn)(std::string)) const {
    for (std::string word : *this) {
        fn(word);
    }
}

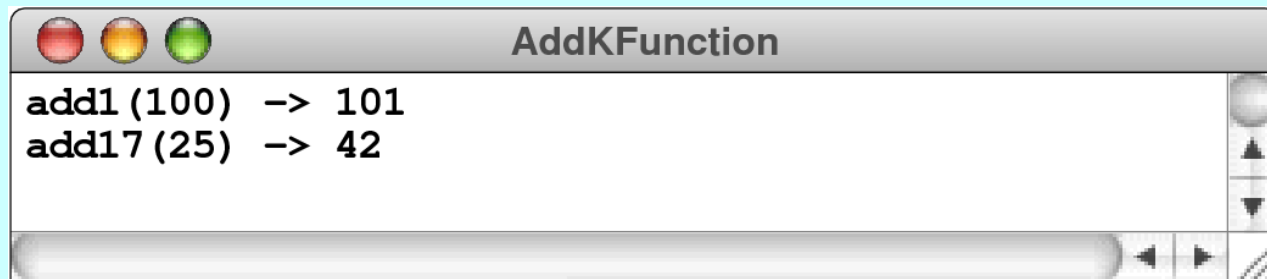
void Lexicon::mapAll(void (*fn)(const std::string&)) const {
    for (std::string word : *this) {
        fn(word);
    }
}

template <typename FunctorType>
void Lexicon::mapAll(FunctorType fn) const {
    for (std::string word : *this) {
        fn(word);
    }
}

```

# Example: Function Objects

```
int main() {  
    AddKFunction add1 = AddKFunction(1);  
    AddKFunction add17 = AddKFunction(17);  
    cout << "add1(100) -> " << add1(100) << endl;  
    cout << "add17(25) -> " << add17(25) << endl;  
    return 0;  
}
```



```

/*
 * Class: AddKFunction
 * -----
 * This class defines a function object that takes a single integer x and
 * computes the value x + k, where k is a constant specified by the client.
 */

class AddKFunction {

public:

    /*
     * Constructor: AddKFunction
     * Usage: AddKFunction addk = AddKFunction(k);
     * -----
     * Creates a function object that adds k to its argument.
     */

    AddKFunction(int k) { this->k = k; }

    /*
     * Operator: ()
     * -----
     * Defines the behavior of an AddKFunction object when it is called
     * as a function.
     */

    int operator()(int x) { return x + k; }

private:

    int k;      /* Instance variable that keeps track of the increment value */

};

```



# Another Use of Iterators in C++

- Besides their original purpose of stepping through the elements of a collection, iterators have even greater importance in C++ because so many of the functions in the Standard Template Library take **iterators as parameters**.
- To sort all the elements of a vector **v**, you call

```
sort(v.begin(), v.end());
```

- If you wanted to sort only the first **k** elements of **v**, you could call

```
sort(v.begin(), v.begin() + k);
```

- The **sort** function in the STL is only one of many useful functions exported by the **<algorithm>** interface.

# Methods in the `algorithm` Library

**`max(x, y)`**

Returns the greater of  $x$  and  $y$ .

**`min(x, y)`**

Returns the lesser of  $x$  and  $y$ .

**`swap(x, y)`**

Swaps the reference parameters  $x$  and  $y$ .

**`iter_swap(i1, i2)`**

Swaps the values addressed by the iterators  $i_1$  and  $i_2$ .

**`binary_search(begin, end, value)`**

Returns **`true`** if the iterator range contains the specified value.

**`copy(begin, end, out)`**

Copies the iterator range to the output iterator.

**`count(begin, end, value)`**

Counts the number of values in the iterator range that are equal to *value*.

**`fill(begin, end, value)`**

Sets every element in the iterator range to *value*.

# Methods in the `algorithm` Library

**`find(begin, end, value)`**

Returns an iterator to the first element in the iterator range that is equal to *value*.

**`merge(begin1, end1, begin2, end2, out)`**

Merges the sorted input sequences into the output iterator.

**`min_element(begin, end)`**

Returns an iterator to the smallest element in the iterator range.

**`max_element(begin, end)`**

Returns an iterator to the largest element in the iterator range.

**`random_shuffle(begin, end)`**

Randomly reorders the elements in the iterator range.

**`replace(begin, end, old, new)`**

Replaces all occurrences of *old* with *new* in the iterator range.

**`reverse(begin, end)`**

Reverses the elements in the iterator range.

**`sort(begin, end)`**

Sorts the elements in the iterator range.

# Methods in the `algorithm` Library

**`for_each`**(*begin*, *end*, *fn*)

Calls *fn* on every value in the iterator range.

**`count_if`**(*begin*, *end*, *pred*)

Returns the number of elements in the iterator range for which *pred* is true.

**`replace_if`**(*begin*, *end*, *pred*, *new*)

Replaces every element in the iterator range for which *pred* is true by *new*.

**`partition`**(*begin*, *end*, *pred*)

Reorders the elements in the iterator range so that the *pred* elements come first.

# Using the `algorithm` Library

- One more way to print two-letter words:

```
void printTwoLetterWords(string word) {
    if (word.length() == 2) {
        cout << word << endl;
    }
}

int main() {
    Lexicon english("EnglishWords.dat");
    for_each(english.begin(), english.end(), printTwoLetterWords);
    return 0;
}
```

- The callback function can also be a function object:

```
for_each(english.begin(), english.end(), ListKLetterWords(k));
```

- Question: what are the differences between `for_each` and `mapAll`?

# Using the **algorithm** Library

- Selection sort in the original form:

```
void sort(Vector<int> & vec) {
    int n = vec.size();
    for (int lh = 0; lh < n; lh++) {
        int rh = lh;
        for (int i = lh + 1; i < n; i++) {
            if (vec[i] < vec[rh]) rh = i;
        }
        int temp = vec[lh];
        vec[lh] = vec[rh];
        vec[rh] = temp;
    }
}
```

- Selection sort using iterators and the **algorithm** library:

```
void sort(Vector<int> & vec) {
    for (Vector<int>::iterator lh = vec.begin(); lh != vec.end(); lh++) {
        Vector<int>::iterator rh = min_element(lh, vec.end());
        iter_swap(lh, rh);
    }
}
```

# Using the `algorithm` Library

- Exercise: selection sort in a more compact form:

```
void sort(Vector<int> & vec) {  
    for (Vector<int>::iterator lh = vec.begin(); lh != vec.end(); lh++) {  
        iter_swap(lh, min_element(lh, vec.end()));  
    }  
}
```

- Can we make it even shorter by using `for_each`?
- If indeed you can convert a sequence of procedural-style statements into a **sequence of embedded calls to a certain set of functions**, you get a new style or paradigm of programming, called ***functional programming***.



# Programming Paradigms

- A *programming paradigm* is a style or “way” of programming.
- One of the characteristics of a programming language is its support for particular paradigms. Some languages make it easy to write in some paradigms but not others.
- A language purposely designed to allow programming in many paradigms is called a *multi-paradigm* programming language, e.g., C++, Python. You can write programs or libraries that are largely *procedural*, *object-oriented*, or *functional* (i.e., some typical paradigms) in these languages.
- In a large program, even different sections can be written in different paradigms.
- C++ supports the *procedural* and *object-oriented* paradigms naturally, supports the *functional* paradigm through the `<functional>` interface, and supports many other paradigms through various external libraries.





# Programming Paradigms

- **Imperative:** An explicit sequence of statements that change a program's state, specifying **how** to achieve the result.
  - **Structured:** Programs have clean, **goto**-free, nested **control structures**.
  - **Procedural:** Imperative programming with **procedures** operating on data.
  - **Object-Oriented:** Objects have/combine **states/data** and **behavior/methods**; Computation is effected by **sending messages to objects (receivers)**.
    - **Class-based:** Objects get their states and behavior based on membership in a class.
    - **Prototype-based:** Objects get their behavior from a prototype object.
- **Declarative:** Programs state the logic of a computation (**what** is the result you want), but not its control flow (**how** to get it).
  - **Functional:** Computation proceeds by (nested) function calls that avoid any global state (e.g., no assignments and loops).
  - **Logic:** The programmer specifies a set of facts and rules, and an engine infers the answers to questions.



# Functional Programming in C++

- Even though *functional programming* was not a goal of the language design, the fact that C++ includes both **templates** and **function objects** make it possible to adopt a programming style that is remarkably close to the functional programming model, which is characterized by the following properties:
  - Programs are expressed in the form of **nested function calls** that perform the necessary computation without performing any operations (such as assignment) that change the program state.
  - Functions are data values and can be manipulated by the programmer just like other data values.
- STL offers **rudimentary support** for the functional programming paradigm through the `<functional>` interface, which exports a variety of classes and methods, which generally fall into two categories represented by the template class `binary_function` that takes two arguments, and `unary_function` that takes a single argument.

# Procedural vs. Functional

- Bake a *procedural* cake:
  - Preheat oven to 175 degrees C. Grease and flour 2 – 8 inch round pans.
  - In a small bowl, whisk together flour, baking soda and salt, set aside.
  - In a large bowl, cream butter, white sugar and brown sugar until light and fluffy. Beat in eggs, one at a time. Mix in the bananas.
  - Add flour mixture alternately with the buttermilk to the creamed mixture. Stir in chopped walnuts.
  - Pour batter into the prepared pans. Bake in the preheated oven for 30 minutes. Remove from oven, and place on a damp tea towel to cool.
- Bake a *functional* cake:
  - A cake is a hot cake that has been cooled on a damp tea towel, where a hot cake is a prepared cake that has been baked in a preheated oven for 30 minutes.
  - A preheated oven is an oven that has been heated to 175 degrees C.
  - A prepared cake is batter that has been poured into prepared pans, where batter is mixture that has chopped walnuts stirred in, and a prepared pan is a greased and floured 2 – 8 inch round pan.
  - Mixture is butter, white sugar and brown sugar that has been creamed in a large bowl until light and fluffy...

# Procedural vs. Functional

- Make a *procedural* dumpling:
  - dough = whisk(flour, water)
  - filling = mix(meat, vegetable)
  - dumpling = stuff(dough, filling)
  - steamed dumpling = steam(dumpling)
- Make a *functional* dumpling:
  - A steamed dumpling is a dumpling that has been steamed
  - A dumpling is a dough stuffed with filling
  - A dough is flour that has been whisked with water
  - Filling is meat mixed with vegetable
  - steam(stuff(whisk(flour, water), mix(meat, vegetable)))

# Procedural vs. Functional

- **State Management:** In procedural programming, state is often managed through variables that are updated step-by-step. In functional programming, state is typically **immutable**, and new states are created through function returns.
- **Side Effects:** Procedural programming may involve **side effects** (e.g., modifying global variables), whereas functional programming aims to minimize side effects by using **pure functions**.
- **Readability and Maintenance:** Functional programming can lead to more concise and potentially more readable code by abstracting away the control flow, while procedural programming's explicit steps can be easier to follow for those familiar with imperative logic.

# Classes in the `functional` Library

**`binary_function`**`<argtype1, argtype2, resulttype>`

Superclass for functions that take the two argument types and return a *resulttype*.

**`unary_function`**`<argtype, resulttype>`

Superclass for functions that take one *argtype* and return a *resulttype*.

**`plus`**`<type>`

Binary function implementing the + operator.

**`minus`**`<type>`

Binary function implementing the - operator.

**`multiplies`**`<type>`

Binary function implementing the \* operator.

**`divides`**`<type>`

Binary function implementing the / operator.

**`modulus`**`<type>`

Binary function implementing the % operator.

**`negate`**`<type>`

Unary function implementing the - operator.

# Classes in the `functional` Library

**`equal_to<type>`**

Function class implementing the `==` operator.

**`not_equal_to<type>`**

Function class implementing the `!=` operator.

**`less<type>`**

Function class implementing the `<` operator.

**`less_equal<type>`**

Function class implementing the `<=` operator.

**`greater<type>`**

Function class implementing the `>` operator.

**`greater_equal<type>`**

Function class implementing the `>=` operator.

**`logical_and<type>`**

Function class implementing the `&&` operator.

**`logical_or<type>`**

Function class implementing the `||` operator.

**`logical_not<type>`**

Function class implementing the `!` operator.

# Methods in the **functional** Library

## **bind1st** (*fn*, *value*)

Returns a unary counterpart to the binary *fn* in which the first argument is *value*.

## **bind2nd** (*fn*, *value*)

Returns a unary counterpart to *fn* in which the second argument is *value*.

## **not1** (*fn*)

Returns a unary predicate function which has the opposite result of *fn*.

## **not2** (*fn*)

Returns a binary predicate function which has the opposite result of *fn*.

## **ptr\_fun** (*fnptr*)

Converts a function pointer to the corresponding function object.

*Replaced by more flexible  
**bind**(*fn*, *value1*, *value2*, ...,  
**placeholders::\_1**,  
**placeholders::\_2**, ...) from  
C++11. Out of the scope of this course.*



# Using the **functional** Library

- **bind2nd** returns a unary function object that calls a binary function object with its second parameter bound to a value. Therefore, the following expression returns a unary function object that adds the constant 1 to its argument:

```
bind2nd(plus<int>(), 1)
```

(remember `AddKFunction add1 = AddKFunction(1);` ?)

# Using the **functional** Library

- The functions in the **algorithm** library that involve sorting, which include **sort**, **merge**, **binary\_search**, **min\_element**, **max\_element**, etc., take an optional functional parameter that defines the ordering. By default, this parameter is generated by calling the **less** constructor appropriate to the value type. You can arrange an integer vector **v** in reverse order:

```
sort(vec.begin(), vec.end(), greater<int>());
```

- The clients can supply their own function pointer or function object instead. That function, which is called a **comparison function**, should take two arguments of the value type and return a Boolean value, which is **true** if the first value should come before the second, e.g., the **ascending()** and **descending()** functions we wrote before.



# Example: programing paradigms

- You can count the number of negative values in an integer vector **v** by the following procedural program:

```
int neg = 0;
for (Vector<int>::iterator it = v.begin();
     it != v.end(); it++) {
    if (*it < 0) neg++;
}
```



# Example: programming paradigms

- Or by a single line of **functional program** using both the `algorithm` and `functional` libraries and iterators:

```
int neg = count_if(v.begin(), v.end(), bind2nd(less<int>(), 0));
```

`count_if` from `<algorithm>` returns the number of values in the iterator range for which calling a functional argument (either a functional object or a function pointer) on that value returns **true**.

- Or by **lambda expressions** introduced in C++11

```
int neg = count_if(v.begin(), v.end(), { return x < 0; });
```

- Lambda expressions are a more flexible and readable way to create function objects that can be used as arguments to higher-order functions, such as those found in the `<algorithm>` and `<numeric>` libraries. This is particularly useful in functional programming paradigms (out of the scope of this course).



# Exercise: programing paradigms

- Question: In **object-oriented programming**, what could we do?
  - We may maintain a public method in the collection class to count the number like the previous procedural program
  - Or even maintain a private data member **neg** and update it in the other member methods whenever necessary, if the efficiency of counting **neg** is crucial.

```
template <typename ValueType>
class Vector {
...
    int numNeg() const;
...
private:
...
    ValueType *array;      /* Dynamic array of the elements */
    int capacity;          /* Allocated size of that array */
    int count;             /* Current count of elements in use */
    int neg;               /* The number of negative elements */
...
}
```

The End