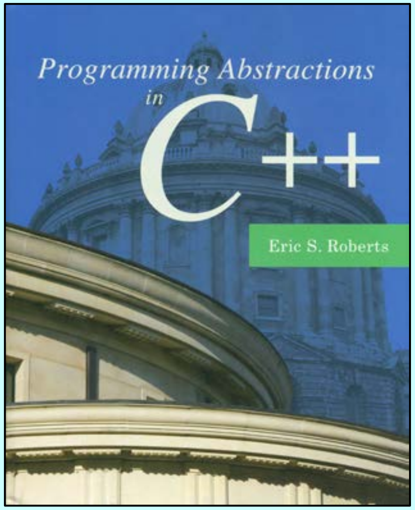
CHAPTER 20

# Strategies for Iteration

*What needs this iteration.*

—William Shakespeare, *Othello*, ~1603

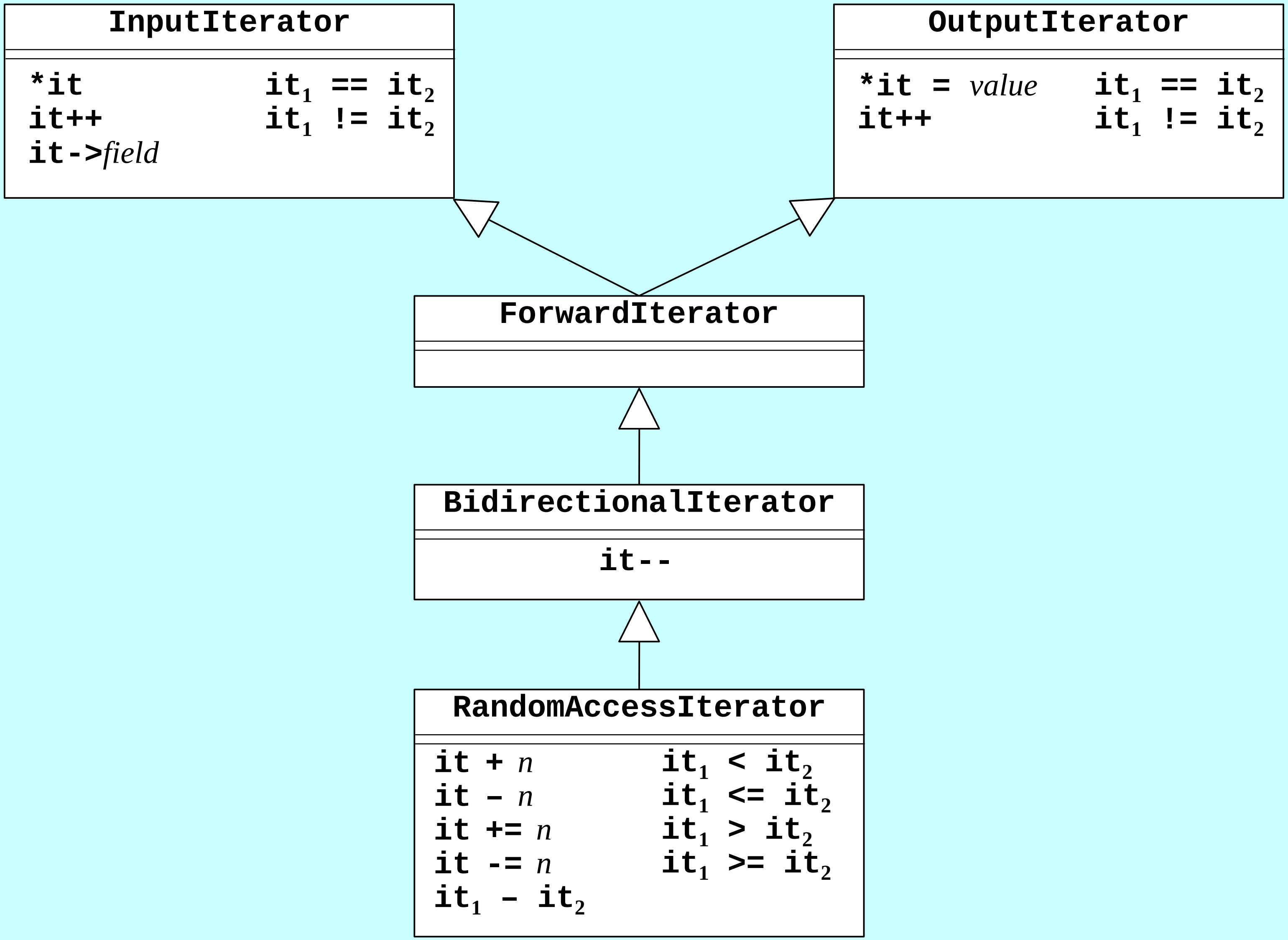


# Using Iterators in C++

- 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 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 (type::iterator it = c.begin(); it < c.end(); it++) {
    . . . Body of loop involving *it . . .
}
```

# The C++ Iterator Hierarchy



# 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:

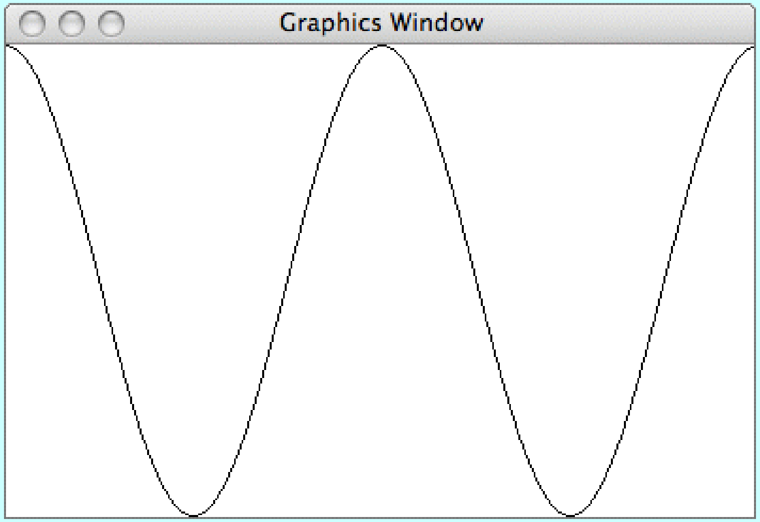
| | |
|---|---|
| **`double x;`** | *Declares* **`x`** *as a* **`double`**. |
| **`double *px;`** | *Declares* **`px`** *as a pointer to a* **`double`**. |
| **`double f();`** | *Declares* **`f`** *as a function returning a* **`double`**. |
| **`double *g();`** | *Declares* **`g`** *as a function returning a pointer to a* **`double`**. |
| **`double (*proc)();`** | *Declares* **`proc`** *as a pointer to a procedure returning a* **`double`**. |
| **`double (*fn)(double);`** | *Declares* **`fn`** *as a pointer to a function taking and returning a* **`double`**. |

# Plotting a Function

- Section 20.2 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?

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

2. How would you convert values **`x`** and **`y`** in the mathematical domain into screen points **`sx`** and **`sy`**?

   Hint: In the time that **`x`** moves from **`minX`** to **`maxX`**, **`sx`** must move from 0 to **`gw.getWidth()`**; **`y`** must move in the opposite direction from **`gw.getHeight()`** to 0.

```
double width = gw.getWidth();
double height = gw.getHeight();
double sx = (x - minX) / (maxX - minX) * width;
double sy = height - (y - minY) / (maxY - minY) * height;
```

# 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.
- Most collections in the Stanford libraries export the method

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

  that calls **`fn`** on every element of the collection.
- As an example, you can print the elements of a **`Set<int> s`**, by calling **`s.mapAll(printInt)`** where **`printInt`** is

  ```
  void printInt(int n) {
      cout << n << endl;
  }
  ```

# Example: Implement `mapAll`

We could use an iterator to implement the function

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

as part of the **`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 */
```

# 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. Pass an additional argument to the mapping function, which is then included in the set of arguments to the callback function.

2. Pass a function object to the mapping function. A ***function object*** is simply any object that overloads the function-call operator, which is designated in C++ as **`operator()`**.

# Methods in the `algorithm` Library

| |
|---|
| **`max(`***x*, *y***`)`**<br>Returns the greater of *x* and *y*. |
| **`min(`***x*, *y***`)`**<br>Returns the lesser of *x* and *y*. |
| **`swap(`***x*, *y***`)`**<br>Swaps the reference parameters *x* and *y*. |
| **`iter_swap(`**$i_1$, $i_2$**`)`**<br>Swaps the values addressed by the iterators $i_1$ and $i_2$. |
| **`binary_search(`***begin*, *end*, *value***`)`**<br>Returns **`true`** if the iterator range contains the specified value. |
| **`copy(`***begin*, *end*, *out***`)`**<br>Copies the iterator range to the output iterator. |
| **`count(`***begin*, *end*, *value***`)`**<br>Counts the number of values in the iterator range that are equal to *value*. |
| **`fill(`***begin*, *end*, *value***`)`**<br>Sets every element in the iterator range to *value*. |

# Methods in the **`algorithm`** Library

| | |
|---|---|
| **`find(`** *begin*, *end*, *value* **`)`**<br>Returns an iterator to the first element in the iterator range that is equal to *value*. | |
| **`merge(`** $begin_1$, $end_1$, $begin_2$, $end_2$, *out* **`)`**<br>Merges the sorted input sequences into the output iterator. | |
| **`min_element(`** *begin*, *end* **`)`**<br>Returns an iterator to the smallest element in the iterator range. | |
| **`max_element(`** *begin*, *end* **`)`**<br>Returns an iterator to the largest element in the iterator range. | |
| **`random_shuffle(`** *begin*, *end* **`)`**<br>Randomly reorders the elements in the iterator range. | |
| **`replace(`** *begin*, *end*, *old*, *new* **`)`**<br>Replaces all occurrences of *old* with *new* in the iterator range. | |
| **`reverse(`** *begin*, *end* **`)`**<br>Reverses the elements in the iterator range. | |
| **`sort(`** *begin*, *end* **`)`**<br>Sorts the elements in the iterator range. | |

# Methods in the **`algorithm`** Library

| |
|---|
| **`for_each(`***begin***`,`** *end***`,`** *fn***`)`**<br>Calls *fn* on every value in the iterator range. |
| **`count_if(`***begin***`,`** *end***`,`** *pred***`)`**<br>Returns the number of elements in the iterator range for which *pred* is true. |
| **`replace_if(`***begin***`,`** *end***`,`** *pred***`,`** *new***`)`**<br>Replaces every element in the iterator range for which *pred* is true by *new*. |
| **`partition(`***begin***`,`** *end***`,`** *pred***`)`**<br>Reorders the elements in the iterator range so that the *pred* elements come first. |

# Classes in the **functional** Library

| |
|---|
| **binary_function<***argtype*$_1$**,** *argtype*$_2$**,** *resulttype***>**<br>Superclass for functions that take the two argument types and return a *resulttype*. |
| **unary_function<***argtype***,** *resulttype***>**<br>Superclass for functions that take one *argtype* and return a *resulttype*. |
| **plus<***type***>**<br>Binary function implementing the **+** operator. |
| **minus<***type***>**<br>Binary function implementing the **-** operator. |
| **multiplies<***type***>**<br>Binary function implementing the ***** operator. |
| **divides<***type***>**<br>Binary function implementing the **/** operator. |
| **modulus<***type***>**<br>Binary function implementing the **%** operator. |
| **negate<***type***>**<br>Unary function implementing the **-** operator. |

# Classes in the **functional** Library

| |
|---|
| **equal_to<*type*>**<br>Function class implementing the **==** operator. |
| **not_equal_to<*type*>**<br>Function class implementing the **!=** operator. |
| **less<*type*>**<br>Function class implementing the **<** operator. |
| **less_equal<*type*>**<br>Function class implementing the **<=** operator. |
| **greater<*type*>**<br>Function class implementing the **>** operator. |
| **greater_equal<*type*>**<br>Function class implementing the **>=** operator. |
| **logical_and<*type*>**<br>Function class implementing the **&&** operator. |
| **logical_or<*type*>**<br>Function class implementing the **\|\|** operator. |
| **logical_not<*type*>**<br>Function class implementing the **!** operator. |

# Methods in the `functional` Library

| |
|---|
| **`bind1st(`*fn*`,` *value*`)`**<br>Returns a unary counterpart to the binary *fn* in which the first argument is *value*. |
| **`bind2nd(`*fn*`,` *value*`)`**<br>Returns a unary counterpart to *fn* in which the second argument is *value*. |
| **`not1(`*fn*`)`**<br>Returns a unary predicate function which has the opposite result of *fn*. |
| **`not2(`*fn*`)`**<br>Returns a binary predicate function which has the opposite result of *fn* |
| **`ptr_fun(`*fnptr*`)`**<br>Converts a function pointer to the corresponding function object. |

# 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++() {            // pre-increment:  ++intvariable
         if (vp == NULL) error("Iterator is uninitialized");
         index++;
         return *this;
      }

      iterator operator++(int) {           // post-increment:  intvariable++
         iterator copy(*this);
         operator++();
         return copy;
      }
```

# Implementation of the **Vector** Iterator

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

      iterator operator--(int) {           // post-decrement:  intvariable--
         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;

   };
```

The End