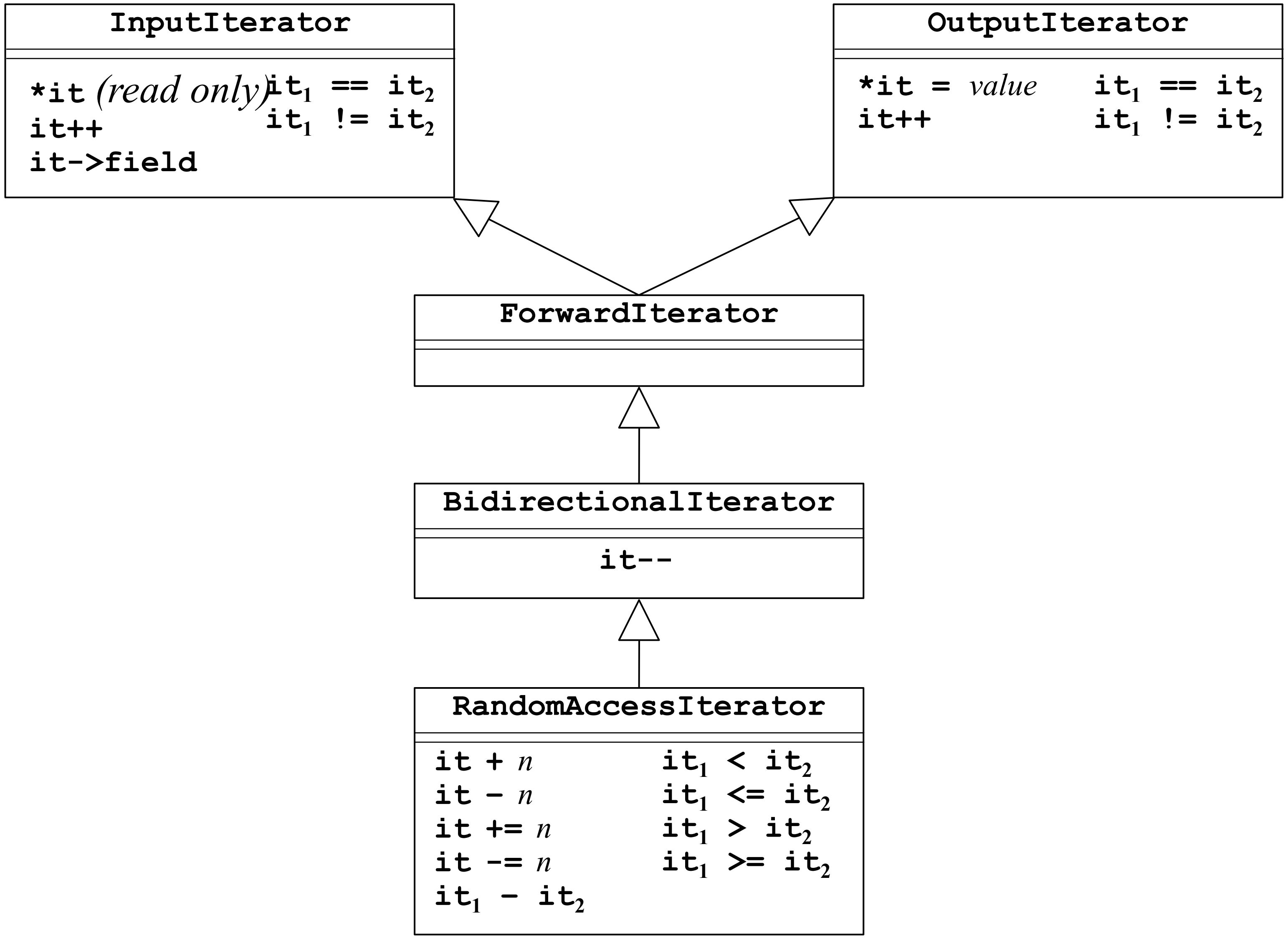
Implementation of the Vector Iterator

The C++ Iterator Hierarchy



InputIterator

***it** *(read only)* **it**$_1$ **== it**$_2$
it++ **it**$_1$ **!= it**$_2$
it->field

OutputIterator

***it =** *value* **it**$_1$ **== it**$_2$
it++ **it**$_1$ **!= it**$_2$

ForwardIterator

BidirectionalIterator

it--

RandomAccessIterator

it + *n* **it**$_1$ **< it**$_2$
it – *n* **it**$_1$ **<= it**$_2$
it += *n* **it**$_1$ **> it**$_2$
it -= *n* **it**$_1$ **>= it**$_2$
it$_1$ **– it**$_2$

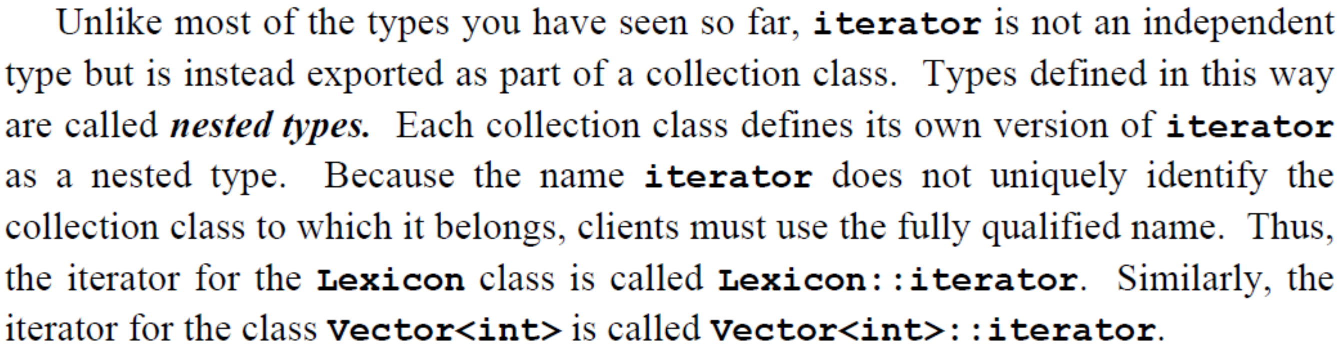
- The most primitive styles of iterator are **InputIterator**, which allows reading values
- **OutputIterator**, which allows writing by assigning a new value to the dereferenced iterator.
- **ForwardIterator** class combines these capabilities and supports both reading and writing.
- **BidirectionIterator** model adds the **--** operator, which makes it possible to move the iterator forward or backward.
- **RandomAccessIterator** is the most general form and includes operators for advancing the iterator by *n* elements as well as the full complement of relational operators.

Example: The Vector Iterator

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

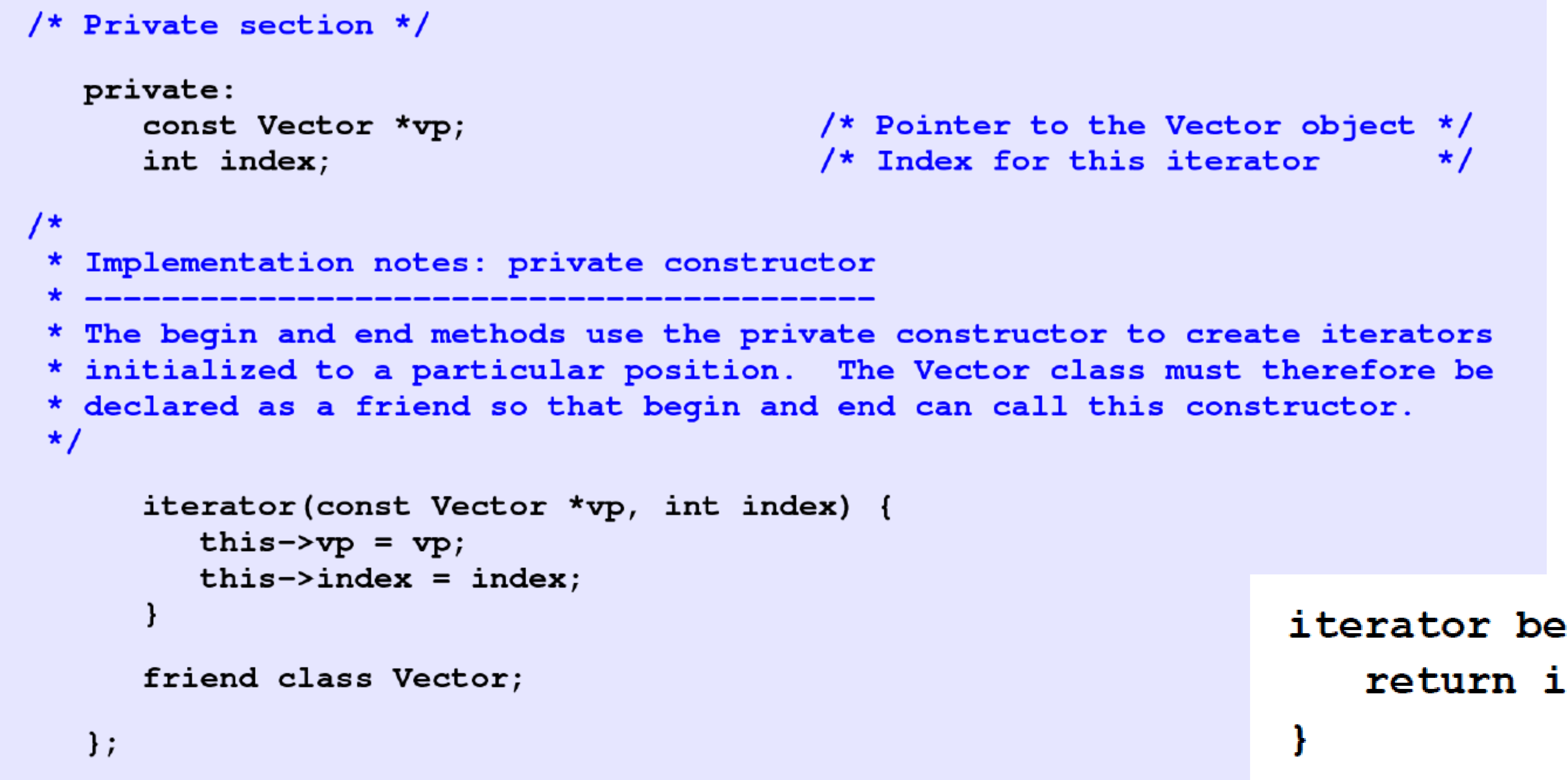
```
Vector<int>::iterator it = v.end();
while (it != v.begin()) {
   cout << *--it << endl;
}
```

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



Unlike most of the types you have seen so far, **`iterator`** is not an independent type but is instead exported as part of a collection class. Types defined in this way are called ***nested types.*** Each collection class defines its own version of **`iterator`** as a nested type. Because the name **`iterator`** does not uniquely identify the collection class to which it belongs, clients must use the fully qualified name. Thus, the iterator for the **`Lexicon`** class is called **`Lexicon::iterator`**. Similarly, the iterator for the class **`Vector<int>`** is called **`Vector<int>::iterator`**.

Implementation of the Vector Iterator



```
/* 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;

   };
```

```
iterator begin() const {
   return iterator(this, 0);
}

iterator end() const {
   return iterator(this, count);
}
```

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];
      }
```

A number of checks to make sure that iterators are used appropriately.

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

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

Implementation of the Vector Iterator

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

      iterator operator--(int) {
         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

- 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.

END