JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 19, 839-856 (2003)

# Iterators: Taxonomy, Strength, Weakness, and Efficiency*

PEI-CHI WU AND FENG-JIAN WANG+
*Department of Computer Science and Information Engineering*
*National Penghu Institute of Technology*
*Penghu, 880 Taiwan*
*E-mail: pcwu@npit.edu.tw*
+*Department of Computer Science and Information Engineering*
*National Chiao Tung University*
*Hsinchu, 300 Taiwan*
*E-mail: fjwang@csie.nctu.edu.tw*

*Iteration* is an operation that traverses the objects of a collection object. *Iterators* are operators/methods widely used for iteration on a collection of objects. However, iterators were considered to be ad hoc and to be a sign of weakness in object-oriented programming languages. There is a need to formalize iterators and to study their strength and weaknesses. In this paper we first propose a taxonomy of iterators based on the number of data states and the number of control points. The taxonomy identifies four categories of iterators. The simplest one can be implemented straightforwardly in an object-oriented language. Iterators of other categories need transformations and are more difficult to design and implement. Finally, we compare the efficiency of iterators with other implementations of iteration. The results show that all iteration techniques tested, including iterators, have been well optimized and are almost as fast as directly coded loops using recent C++ compilers.

***Keywords:*** collection objects, object-oriented programming, control points, efficiency, the C++ language

## 1. INTRODUCTION

A *collection object* is an object that contains a number of elements. Typical examples of collection objects are sets, bags, arrays, strings, etc. A collection object has its own operations, e.g., the union of two sets, and the concatenation on two strings. Besides these operations, a common operation on collection objects is iteration. *Iteration* on a collection object is an operation that traverses the elements of the collection object. Iteration partly opens the encapsulated interface of a collection object to support a series of operations on each element. For example, iteration can be used to compute the sum of elements in an array of numbers.

There are four major issues in the development of iteration:

1. Simplicity: Iteration should be easy to develop. Writing iteration for a collection object should be similar to writing a for-loop for elements of the collection object.

Received January 10, 2000; revised May 15, August 28 and November 19, 2002; accepted January 9, 2003.
Communicated by Claude Kirchner.
* This research was supported in part by National Science Council, Taiwan, R.O.C., under Contract No. NSC 89-2213-E-346-002.

2. Usability: Iteration should be easy to use. The syntax of iteration should be similar to that of for-loops in a programming language.
3. Concurrency: It should be possible to have several iterations working on the same object simultaneously.
4. Efficiency: For a collection object such as an array or a set, the iteration code is usually executed several times. Iteration on these fundamental data structures should be very efficient.

Iterators (e.g., cf. [1]) are technique widely used for collection objects of object-oriented programming languages such as C++ [2], Eiffel [3], and Java [4]. The Eiffel base library includes a generic deferred class called Iterator, which provides the following iteration features: do_all(action), do_if(action, test), do_until(action, test), do_while(action, test), for_all(test), there_exists(test), until_do(action, test), and while_do(action, test). The action is a procedure that takes an item of the collection; the test is a function that takes an item of the collection and returns a Boolean value. The Java utility library includes an interface called Iterator, which provides the following functions: (1) hasNext: whether the iteration has more elements; (2) next: the next element in the iteration; and (3) remove: removes from the underlying collection the last element returned by the iterator.

In the C++ language, an iterator is usually implemented by a friend class of a collection class or more elegantly by a nested class of the collection class. However, iterators were considered to be ad hoc and to be the sign of weakness in object-oriented programming languages (especially C++) [5]. The major disadvantage of iterators addressed is the lack of flexible composition of codes such as higher-order functions [5]. Although iterators are not an elegant technique for the development of all kinds of iteration, they are a practical and effective technique in current object-oriented languages.

There is a need to formalize iterators and to study their strengths and weaknesses. In this paper we examine the iterator technique from the four major issues of iteration. First, we review the iterator idiom by giving two simple examples: string and linked list. The use of iterators is quite elegant in both examples. Second, we propose a taxonomy of iterators based on the number of data states and the number of control points. The taxonomy identifies four categories of iterators. The simplest one can be implemented straightforwardly in an object-oriented language. Iterators of other categories need transformations and are more difficult to design and implement. These iterators explicitly record the control point currently entered and the infinite states in storage such as stacks. Third, we compare the efficiency of iterators on linked lists of integers with other iteration techniques. There are three other implementation techniques for iteration in C++: formal procedure parameter, inheritance, and macro expansion. Our results show that all techniques tested have been well optimized and are almost as fast as directly coded loops in recent C++ compilers.

## 2. STRENGTH OF SIMPLE ITERATORS

An iterator records the state of an iteration on a collection object. It can be used as a control variable in a loop. A simple iterator achieves both simplicity and usability. In this

section, we introduce two simple iterators for string and linked list: StringIter and ListIter. Both examples show the elegance of iterators in handling iteration on simple collection objects.

An iterator creates a temporary record for the state of an iteration when traversing the elements of a collection object. An iterator accesses both the state record and the internal representation of a collection object. Occasionally, an iterator does not have to be a friend class of a collection class if the interface of the collection already provides necessary operations to access its internal structure during iteration. For example, StringIter is not a friend class of any class, but it can traverse each character in a string of char*.

There are many kinds of iterator class interfaces [1, Ch 7.4]. Here we use the following interface: 1) operation operator! (or last) for querying when the iterator runs out of elements, 2) postfix operation operator++ (or next) for stepping into the next element, and 3) operation operator() (or current) for getting the current element.

Fig. 1 shows the code of class StringIter, an iterator for string. It is a simple class containing only ten lines of code. Since a string of char* in C++ ends with a null character ('\0'), the content of variable current (*current) is set to null when the iteration reaches the end of the string.

```
class StringIter
{
public:
   StringIter(char* s) : current(s) { }
   char operator++ (int) { return *current++; }
   bool operator! () const { return *current; }
   char operator() () const { return *current; }
private:
   char* current;
};
```

Fig. 1. Iterator StringIter.

Fig. 2 shows three loops using StringIter and char* in control variables. Using StringIter is in the same way as using char*. The first loop uses an iterator to print out each character in a string. The second compares string p with string q using one iterator for each string. The third computes the sum of characters in a string using an iterator and a local variable sum.

Fig. 3 shows a template class for linked list and the corresponding iterator class ListIter. The template class list provides the insert operation to insert an element in front of the list. The template class ListItem defines a structure that stores an element of a list and a link to the next ListItem in the list. Class ListIter uses the same protocol as StringIter. Fig. 4 shows an example using ListIter<char> to print out a list of characters.

Iterators StringIter and ListIter belong to the simplest category of iteration (see our taxonomy in section 3). The strength of these simple iterators is as follows:

```
char *p, *q;
// Loop by StringIter
   for(StringIter itp(p); !itp; itp++)
      putchar(itp());

   for(StringIter itp(p), itq(q) ;
            !itp && !itq; itp++, itq++)
      if (itp() != itq())
         break;

   int sum = 0;
   for(StringIter itp(p); !itp; itp++)
      sum += itp();
```

```
char *p, *q;
// Loop by char*
   for(char* ip = p; *ip; ip++)
      putchar(*ip);

   for(char* ip = p, *iq = q;
            *ip && *iq; ip++, iq++)
      if (*ip != *iq)
         break;

   int sum = 0;
   for(char* ip = p; *ip; ip++)
      sum += *ip;
```

Fig. 2. Using StringIter and char* in three loops.

```
template<class T> struct ListItem
{
   ListItem(T dat, ListItem<T>* nxt) : data(dat), next(nxt) { }
   ListItem(ListItem<T>* nxt) : next(nxt) { }
   T data;
   ListItem<T>* next;
};
template<class T> class ListIter;
template<class T> class List
{
   friend class ListIter<T>;
public:
   List() : front(0) { }
   insert(T dat) { ListItem<T>* nxt = front; front = new ListItem<T>(dat, nxt); }
private:
   ListItem<T> *front;
};
template<class T>
class ListIter
{
public:
   ListIter(List<T>& list) : current(list.front) { }
   void operator++ (int) { current = current->next; }
   bool operator!() const{ return (int)current; }
   T operator() () const { return current->data; }
private:
   ListItem<T>* current;
};
```

Fig. 3. Class List and class ListIter.

```
List<char> list;
list.insert('3');   list.insert('2');   list.insert('1');

for(ListIter<char> itl(list); !itl; itl++)
   printf("%c ", itl());
```

Fig. 4. An example of using ListIter<char>.

**Simplicity**

- Both StringIter and ListIter are elegantly developed. Each member function of an iterator corresponds to a statement or an expression in a for-loop statement of C++. All these member functions are written in one statement.

**Usability**

- The coding style is similar to using control variables in a for-loop. The constructor corresponds to the declaration of the control variable in a for-loop. The operator! is used in the condition expression; operator++ is used in the stepping expression.

**Concurrency**

- Provides multiple iterators on a collection object.
- Supports accesses to lexical closures. Iteration sometimes involves data in its context (lexical closures). For example, the third loop in StringIter accesses the local variable sum.
- Supports "interleave" operations on two or more collection objects. Iteration on a collection object may be dependent on an iteration of another collection object. Both iterations should be "interleaved." The caller can decide when to step either iteration.

**Efficiency**

- Support function inlining on iterator calls: calls to operator++ can be inlined with bodies of for-loops.
- Support short-circuit computation in a loop. An iterator is stepped by its caller. The caller can decide when to stop an iteration. For example, when searching an element in a list, the searching process is terminated once an element is found.

## 3. A TAXONOMY ON ITERATORS

The weakness of the iterator idiom is that iterators in some collection objects are not as simple as those in section 2. This section introduces a taxonomy of iterators. The taxonomy gives a basis for understanding the difficulties when applying the iterator idiom. Section 3.1 introduces the taxonomy. Section 3.2 gives examples.

### 3.1 The Taxonomy

In this section we classify iterators based on the number of data states and the number of control points. The data state of an iterator provides the information needed to generate the sequence of elements in a collection object. There are two categories:

1. *Finite-state*: The number of states is fixed and the storage for storing the state is constant for any input size. The StringIter is an example that only a pointer is needed to store the state of the iteration.
2. *Infinite-state*: The number of states is dependent on the size of a collection object. For example, an iteration on a tree structure needs $O(h)$ space to traverse all the tree nodes, where $h$ indicates the depth of the tree. The depth of a skewed tree, for example, is the number of nodes in the tree.

A *control point* of an iteration is a control state in the iteration. There are two categories:

1. *Single control point*: Control is transferred to one unique program location for the iteration of each element. For example, there is only one control point in StringIter.
2. *Multiple control points*: There is more than one program location to which the control is transferred. For example, coroutines can be used to implement multiple control points. Each operator++ returns an element and the next call starts to execute in a different program location, which follows the previous return.

A control state can also be maintained as a data state of an iteration. The difference between them is that using a branch instruction according to a stored control state is less efficient and more difficult to implement. A transformation is needed to encode control states in a form of data states.

Iteration may involve infinite states or multiple control points. Writing an iterator for such iteration is not straightforward. The iterator should explicitly record the control point currently entered and the infinite states in storage such as stacks. Since an iterator has only one control point (i.e., function operator++), a transformation is needed in writing an iterator for the iteration with multiple control points.

Table 1 summarizes the classification of iteration. The taxonomy identifies four categories of iteration. Only the simplest one, with finite states and single control point, can be implemented straightforwardly by iterators. The StringIter and ListIter are examples of the simplest iteration. Applying the iterator idiom as an iteration for other categories requires additional transformations. TreeNodeIter_pre, TreeNodeIter_in, and TreeNodeIter_post are the three traversal orders for iteration on tree nodes in a binary tree. TreeLeafIter is an iteration on leave nodes of a tree. PoolIter is an iteration on a data structure for an unbounded array [6].

**Table 1. Classification of iteration based on data states and control points.**

| | single control point | multiple control points |
|---|---|---|
| finite-state | StringIter, ListIter | PoolIter |
| infinite-state | TreeLeafIter, TreeNodeIter_pre | TreeNodeIter_in, TreeNodeIter_post |

### 3.2 Examples

This section gives two examples of iterations in trees. For comparison we present two versions, for-loop and iterator. Iteration in a Pool [6] can be found in the Appendix.

**Iteration on tree-leaves**

The *same-fringe* problem [5, 7] is an example that involves infinite states and a single control point. It checks whether the leaves of two (binary) trees are the same. Fig. 5 shows an iterator TreeLeafIter used for generating the sequence of the leaves of a tree. TreeLeafIter can be easily implemented using a stack to store the path from the current node to the tree root. TreeLeafIter does not maintain any control information. The iterator solution seems acceptable compared with a coroutine implementation [7].

```
template<class T> class Node
{
public:
   Node(Node* l, Node* r, T d) : left(l), right(r), data(d) { }
   T data;    Node* left;    Node* right;
};
template<class T> class BinaryTree
{
   typedef void (*FUN)(Node<T>*, void* args);
public:
   BinaryTree() : root(0), size(0) { }
   Node<T>* node(Node<T>* l, Node<T>* r, T d)
      { size++; return new Node<T>(l, r, d); }
   void forEachLeaf(FUN f, void* args = 0) { forEachLeaf(root, f, args); }
   void forEachLeaf(Node<T>* node, FUN f, void* args = 0) {
      if (node->left==0 && node->right==0) { f(node, args); return; }
      if (node->left) forEachLeaf(node->left, f, args);
      if (node->right) forEachLeaf(node->right, f, args);
   } ...
   Node<T>* root;
   int size;
};
template<class T>
class TreeLeafIter
{
public:
   TreeLeafIter(BinaryTree<T>& t) : tree(&t), stack(), current(0) {
      stack.push(t.root); operator++(0);
   }
   void operator++ (int) {
```

Fig. 5. Iterator TreeLeafIter.

```
    if (stack.empty()) { current = 0; return; }
    for(;;) {
      Node<T>* node = stack.pop();
      if (node->left==0 && node->right==0) { current = node; return; }
      if (node->right) stack.push(node->right);
      if (node->left) stack.push(node->left);
    }
  }
  bool operator!() { return (int) current; }
  Node<T>& operator() () { return *current; }
private:
  Stack<Node<T>*> stack;
  BinaryTree<T>* tree;
  Node<T>* current;
};
```

Fig. 5. (Cont'd) Iterator TreeLeafIter.

**Traversal orders on a binary tree**

There are three traversal orders on a binary tree: preorder, inorder, and postorder. Fig. 6 shows a recursive version of the inorder tree traversal. The other two differ only in the placement of the code that works on the current node. The iteration on the nodes of a tree in preorder is the simplest of the three. It is similar to the code in TreeLeafIter but it does not check whether a node is a leaf node.

```
template<class T> class BinaryTree
{
public:
  ...
  void forEachNode_in(FUN f, void* args = 0) { forEachNode_in(root, f, args); }
  void forEachNode_in(Node<T>* node, FUN f, void* args = 0) {
    if (node->left) forEachNode_in(node->left, f, args);      // control point 1
    f(node, args); // inorder.                                 // control point 2
    if (node->right) forEachNode_in(node->right, f, args);    // control point 3
  }
  void forEachNode_pre(FUN f, void* args = 0);
  void forEachNode_post(FUN f, void* args = 0);    ...
};
```

Fig. 6. A recursive version of the inorder tree traversal.

Iteration in inorder and postorder are more complicated. There are three control points in the recursive version of an inorder traversal. These control points can be transformed into three kinds of control information: INITIAL, LEFT and RIGHT. The data member stack stores the path from the current node to the tree root and the control information that denotes the control point to which the traversal should return. Iterators

TreeNodeIter_in (Fig. 7) and TreeNodeIter_post are similar. They can also be fine-tuned by removing some push and pop operations on the stack by more complicated loops [1, p. 301].

```
template<class T1, class T2> struct tuple
{
   tuple(T1 t1, T2 t2) : _1(t1), _2(t2) { }
   T1 _1; T2 _2;
};
enum Flag {INITIAL, LEFT, RIGHT};
template<class T> class TreeNodeIter_in
{
   typedef tuple< Node<T>*, Flag > Info;
   public:
   TreeNodeIter_in(BinaryTree<T>& t) : tree(&t), stack(), current(0) {
      stack.push( Info(t.root, INITIAL) );    operator++(0);
   }
   void operator++ (int) {
      Node<T>* node;
      enum Flag flag;
      if (stack.empty()) { current = 0; return; }
      for(;;) {
         Info info = stack.pop(); node = info._1; flag = info._2;
         switch(flag) {
            case INITIAL:
               stack.push( Info(node, LEFT) );
               if (node->left) stack.push( Info(node->left, INITIAL) ); break;
            case LEFT:
               stack.push( Info(node, RIGHT) );
               if (node->right) stack.push( Info(node->right, INITIAL) );
               current = node; return;
            case RIGHT:
               if (stack.empty()) { current = 0; return; }
         } /* end switch */
      } /* end for */
   }
   bool operator!() { return (int) current; }
   Node<T>& operator() () { return *current; }
private:
   Stack<Info> stack;
   BinaryTree<T>* tree;
   Node<T>* current;
};
```

Fig. 7. Iterator TreeNodeIter_in.

## 4. EFFICIENCY OF ITERATORS

This section reviews other techniques available to implement iteration and compares some of them with iterators using the linked list example.

### 4.1 Implementations

In imperative languages there are four primary iteration techniques besides iterators; they are:

**Coroutines:** Implementation with coroutines is flexible in handling multiple control points but less efficient. Some techniques have been developed for the implementation of coroutines (e.g., see Mateu [7]).

**Formal procedure parameters:** Iteration is implemented as a loop that applies a passed procedure parameter on each element of a collection object. This is suitable for iteration that involves only one collection object. There are three variations:

1) with lexical closures: The formal procedure parameters in Ada [8] and Pascal [9] contain information on lexical scopes, called *lexical closures*. A procedure can access the data in its lexical scope even if it is passed as a parameter and can be invoked indirectly. Breuel [10] proposes an extension of lexical closures for C++.
2) without lexical closures: A function pointer in C/C++ [2] contains no lexical information. Instead, passing data in lexical scopes to an invocation of a function pointer needs to be done explicitly. Thus, it is efficient when there is no data to pass.
3) block-closure: Block-closures (“embedded” anonymous functions) in Smalltalk [11] and Objective-C [12] are highly readable since functions for iteration are usually short and are used only once. There is no need to explicitly define a name for a block closure.

**Type composition:** Type composition creates a new type for iteration using generic functions (Ada [8]), template/generic classes (C++ [2] and Eiffel [3]), or inheritance (C++, Eiffel, and Java [4]). This is a little difficult to use compared with the previous approaches, since such a type is used just once. On the other hand, type composition makes function inline possible, since compilers can easily track which function is called. For example, templates in C++ are usually implemented by macro expansion and with member functions that can be inlined.

**Macro expansion:** Iteration can also be implemented as a macro that expands into a for-loop. Accessing lexical closures is easy since the macro does not generate function calls. The LEDA library [13], for example, uses the C preprocessor for expanding a for-loop. There is a problem in using macro expansion, as the preprocessor has no knowledge about the language's types and scope rules. The macro names are global to the file scope. Therefore, the macro names for iteration should be carefully selected to avoid name collision. In addition, it is not easy to write iteration code in a single for-loop. Some macros in LEDA [13] just call the corresponding iterators, and their underlying implementations are iterators.

Table 2 summarizes methods of implementation for iteration and their variations. The languages that provide the construct(s) are annotated at the end. Note that for the latter three, formal procedure parameters, type composition, and macro expansion are difficult to handle iterations that involve more than one collection object. One solution to this is to define an iterative loop for each combination of collection objects. As the number of combinations is almost unlimited, the number of resulting iterations may be huge.

**Table 2. Summary of implementation techniques for iteration.**

| Technique | Variations |
|---|---|
| Coroutine | -- |
| Iterator | friend class (C++), nested class (C++)<br>abstract class/interface (C++, Eiffel, Java)<br>package (Ada) |
| Formal procedure parameter | lexical closures (Pascal, Ada)<br>without lexical closures (C/C++)<br>block context (Smalltalk, Objective-C) |
| Type composition | generic function parameters (Ada)<br>template/generic class (C++, Eiffel)<br>inheritance (C++, Eiffel, Java) |
| Macro expansion | preprocessor command (C/C++) |

### 4.2 Performance Comparison for Implementations in C++

This section measures the efficiency of implementations for iteration in C++. The benchmark program performs an iteration on a linked list (section 2). The iteration is used to compute the sum of a linked list of integers. We implement the iteration by using the four techniques available in C++: iterator, formal procedure parameter, inheritance, and macro expansion (the four labeled with C++ in Table 2). A version of a directly coded loop, which breaks encapsulation of a linked list, is also developed for comparison. The following is the code for each version.

**Formal procedure parameter**

```
template<class T>
class List
{ ...
   typedef void (*FUN)(T*, void*);
public:
   void forEachItem(FUN f, void* args = 0) {
      ListItem<T> *ip;
      for(ip = front; ip; ip = ip->next)
         f(&ip->data, args);
   } ...
};
```

```
// use
void summation(int* p, void* args) {
   struct Frame {
      int sum;
   } *frame;
   frame = (Frame*) args;
   frame->sum += *p;
} ...
   list.forEachItem(&summation, &sum);
```

**Inheritance**

```
template<class T>
class Summation {
   T sum;
public:
   Summation() : sum(0) {}
   T result() { return sum; }
   void operator() (T n) { sum += n; }
};
template<class T, class functorT>
class ForEachItem_List : private functorT
{
public:
   ForEachItem_List(List<T>& list) {
      ListItem<T> *ip;
      for(ip = list.front; ip; ip = ip->next)
         operator() (ip->data);
   }
};
// example of summation
   register ForEachItem_List< int, Summation<int> > it(list);
   printf("sum = %d\n", it.result());
```

**Macro expansion**

```
#define FOR_EACH_ITEM_OF_LIST(T, var, list) \
   register T var; \
   for(register ListItem<T> *_ip = list.front; \
                  _ip && (var = _ip->data), _ip; _ip = _ip->next)
// use macro
   register int sum = 0;
   FOR_EACH_ITEM_OF_LIST(int, n, list)
      sum += n;
```

**Directly coded loop**

```
register int sum = 0;
for(ListItem<int>* ip = list.front; ip; ip = ip->next)
   sum += ip->data;
```

Table 3 shows the timing results. The input is a list of 5,000,000 integers. The test program processes the list 10 times. The host machine is a PC with Pentium III 733 MHz with 256 MB of memory. The compilers used are Borland C++ Builder 6.0 (BCB 6) on Windows XP Professional and GNU C++ 3.0 (G++ 3) on Red Hat Linux 7.2. The code is compiled to the "Release" setting in BCB and with flag -O in G++. All the computing times are measured by using the clock function. Each version runs five times and the average is taken. Excepting that the iterator version in BCB runs 16% slower than the other versions, all of the other versions run at roughly the same speed in both compilers and operating systems. This shows that all these iteration techniques have been well optimized and are almost as fast as directly coded loops when using recent C++ compilers. It is advised that application and library developers choose an iteration technique based on simplicity, usability, and concurrency concerns, and leave the efficiency issue mostly to compiler designers.

**Table 3. Execution times for five implementations of iteration on a linked list using BCB 6 and G++ 3 compilers.**

| Implementations of iteration | BCB 6 | G++ 3 |
|---|---|---|
| Formal procedure parameter | 4.07 s. | 3.82 s. |
| Iterator | 4.67 s. | 3.82 s. |
| Inheritance | 4.03 s. | 3.82 s. |
| Macro expansion | 4.06 s. | 3.82 s. |
| Directly coded loop | 4.03 s. | 3.82 s. |

## 5. RELATED WORK

Kim and Kim [14] classify iterators as primitive and recursive depending on whether the number of data states is finite or infinite. Recursive types need additional data structures such as stacks to iterate all elements of a collection object. This category is equivalent to our definition of infinite-state category. Their classification does not address the issue of multiple control points. Our classification of control points is a step in further understanding C++ iterators.

Becker [15] addresses two designs of C++ iterators: one using a pointer of a callback function (formal procedure parameter), and one using an inherited virtual function (inheritance). Becker concludes that the inherited one is much safer to use than the callback one. Our examples in section 4 also show that inheritance is safer than formal procedure parameter in implementing iteration. Since C++ does not support procedure parameter with lexical scopes, the example using procedure parameter needs to access data in lexical scopes via a pointer to arguments (void* args).

Plauger [16] proposes wrapper classes for C++ iterators used during debugging. These classes can be used to validate constraints imposed on various iterators in the C++ Standard Template Library (STL). For example, algorithms using STL input and output iterators should be single pass. They should not pass through the same iterator twice. A wrapper class can be used to validate such a constraint. This is a useful debugging tool for iterators with special constraints.

A CLU iterator is an operation that yields results incrementally [17]. CLU iterators are implemented by coroutines. The iterator and the body of the for-loop pass control back and forth. Sather iterators [18] are derived from CLU iterators but are more flexible and better suited to object-oriented programming. Sather iterators are special member functions (also called methods) of a class and are thought of as structured coroutines. Katrib and Martinez [19] propose a new iterator feature for Eiffel. The proposal is also based on CLU iterators. They also offer an associated control structure called forall loop. Kim [20] proposes a mechanism for writing iterators in C++. His implementation is based on semicoroutines, which are implemented by threads.

Messerschmidt [21] addresses the problem of *iterator integrity*. Does an iterator work reasonably when the element it points to is deleted? He/she gives an example of a list iterator where this problem has been solved. This issue can also be left to dynamic memory management, such as garbage collection, which C++ does not directly support. In practice, the semantics of most iterators are usually undefined when their collection objects are modified [18, p. 13].

## 6. CONCLUSIONS

We have analyzed iterators by proposing a taxonomy based on the number of data states and the number of control points. This taxonomy has provided a systematic view of the complexity of iterators. The strengths and weaknesses in iterators can be analyzed based on this taxonomy. Example iterators, including the leaf node iterator for the same-fringe problem have been presented to validate the usage of the taxonomy. We have also measured the efficiency of various implementations for iteration on a linked list. Our results have shown that all of the tested iteration techniques, including iterators, are well optimized and are almost as fast as directly coded loops when using recent C++ compilers. Iterators with finite states and a single control point are simple, efficient, and easy to use. Iterators of other categories require additional transformations and are best replaced with other iteration techniques such as inheritance.

## ACKNOWLEDGMENT

The author would like to thank the referees whose comments helped improve the overall presentation. Thanks also go to Chris F. J. Hung for editorial help.

## APPENDIX: ITERATION IN A POOL

*Pool* [6] is a collection object that provides an unbounded space. A pool consists of

several segments of contiguous memory space. The following code is an iteration on the elements of a pool. The iteration contains two control points as marked in the code. The number of data states needed is finite; the memory space used contains only the local variables in the code. Since the number of elements in the last segment may not be in a power of 2, iteration on the items of the last segment is treated separately.

```
// Version 1. the code given in [6]
template<class objT>
void Pool<objT>::forEachItem(void (*f)(objT&, void*), void* arg = 0)
{
   int segment_len = 1<<log2_segment_size;
   for(int i = 0; i<nsegments-1; i++, segment_len = segment_len<<1)
      for(int j = 0; j<segment_len; j++)
         f(*(index[i]+j), arg);    // 1st control point

   for(objT *obj = index[i]; obj<item; obj++)      // last segment
      f(*obj, arg); // 2nd control point
}
```

This iteration can be transformed into an equivalent nested loop by adding additional data states. Version 2, shown below, uses an array of indexing items (the variable last) to unify the terminating conditions in each segment. The resulting code contains only one control point.

```
// Version 2. transform control points into data state objT* last[], ...
void Pool<objT>::forEachItem(void (*f)(objT&, void*), void* arg = 0)
{
   objT* last[32]; // maximal number of segments for 32-bit machines
   // initiate last[0..nsegments-1];
   int seg_len = 1<<log2_segment_size;
   for(int I = 0; i<nsegments-1; i++, seg_len = seg_len<<1)
      last[i] = index[i] + seg_len;
   last[nsegments-1] = item;

   for(I = 0; i<nsegments; i++)
      for(objT *obj = index[i]; obj<last[i]; obj++)
         f(*obj, arg);
}
```

The iterator PoolIter can be derived from Version 2. The constructor of PoolIter initializes the data state such as i, obj, and last, as in the for-loop of Version 2. The execution of operator++ is guaranteed to be constant in time.

```
template<class objT>
class PoolIter
{
private:
   Pool<objT>& pool;
   objT* last[32]; // the address beyond last item
   objT* limit; // limit of current segment
   objT* obj; // next item
   int nsegments;
   int i;   // current segment No.
public:
   PoolIter(Pool<objT>& p) : pool(p), nsegments(p.nsegments) {
      // initiate last[0..nsegments-1]
      int seg_len = 1<<pool.log2_segment_size;
      for(int I = 0; i<nsegments-1; i++, seg_len = seg_len<<1)
         last[i] = pool.index[i] + seg_len;
      last[nsegments-1] = pool.item;
      obj = pool.index[0];   limit = last[0];   i = 0;
   }
   void operator++ () {
      if (++obj<limit) return;
      if (++i<nsegments) {
         obj = pool.index[i];   limit = last[i];
      } else
         { obj = 0; return; }
   }
   bool operator!() { return (int) obj; }
   objT& operator() () { return *obj; }
};
```

## REFERENCES


1. T. A. Budd, *Classic Data Structures in C++*, Addison-Wesley, 1994.
2. M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
3. B. Meyer, *Eiffel: The Language*, Prentice Hall PTR, 1991.
4. K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, 3rd Edition, Addison-Wesley, 2000.
5. H. G. Baker, "Iterators: signs of weakness in object-oriented languages," *ACM OOPS Messenger*, Vol. 4, 1993, pp. 18-26.
6. P.-C. Wu and F.-J. Wang, "Pool: an unbounded array," *ACM SIGPLAN Notices*, Vol. 29, 1994, pp.68-71.
7. L. Mateu, "An efficient implementation for coroutines," in *Proceedings of 1992 Int'l Workshop on Memory Management*, LNCS No. 637, Springer-Verlag, pp. 230-247.
8. G. Booch, *Software Engineering with Ada*, The Benjamin/Cummings Publishing

Company, Inc., California, 1983.
9. K. Jensen, N. Wirth, and A. Mickel, *Pascal User Manual and Report: ISO Pascal Standard*, 4th Edition, Springer Verlag, 1991.
10. T. M. Breuel, "Lexical closures for C++," in *Proceeding of 1988 USENIX C++ Conference*, 1988, pp. 293-304.
11. A. Goldberg and D. Robson, *Smalltalk-80: The Language*, Addison-Wesley, 1989.
12. B. J. Cox and A. J. Novobilski, *Object-oriented Programming: An Evolutionary Approach*, 2nd ed., Addison-Wesley, 1991.
13. K. Mehlhorn and S. Näher, "LEDA: a platform for combinatorial and geometric computing," *Communications of the ACM*, Vol. 38, 1995, pp. 96-102.
14. E. J. Kim and M. H. Kim, "Enhancing efficiency of C++ iterators using lightweight ones," *Journal of KISS (B): Software & Applications*, Vol. 24, 1997, pp.104-13.
15. P. Becker, "Iterators, portability, and reuse," *C++ Report*, Vol. 5, 1993, pp. 51-54.
16. P. J. Plauger, "Debugging iterators," *Embedded Systems Programming*, Vol. 10, 1997, pp. 92-94.
17. B. Liskov, "A history of CLU," in *Proceedings of the second ACM SIGPLAN Conference on History of Programming Languages*, 1993, pp. 133-147.
18. S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski, "Iteration abstraction in sather," *ACM Transactions on Programming Languages and Systems*, Vol. 18, 1996, pp. 1-15.
19. M. Katrib and I. Martinez, "Collections and iterators in eiffel," *Journal of Object-Oriented Programming*, Vol. 6, 1993, pp. 45-51.
20. M. H. Kim, "A new iteration mechanism for the C++ programming language," *ACM SIGPLAN Notices*, Vol. 30, 1995, pp. 20-26.
21. H. J. Messerschmidt, "List iterators in C++," *Software-Practice & Experience*, Vol. 26, 1996, pp. 1197-1203.

**Pei-Chi Wu (吳培基)** was born on March 11, 1967, in Hsinchu, Taiwan, the Republic of China. He received B.S., M.S., and Ph.D. from National Chiao Tung University, in 1989, 1991, and 1995, respectively, all in Computer Science and Information Engineering. He became the member of ACM and IEEE computer Society since 1996. He has been with Department of Computer Science and Information Engineering, National Penghu Institute of Technology, Taiwan, since August 1998. His research interests include multilingual systems, extensible markup language, object-oriented programming, compiler design, random number generators, and high-performance distributed computing.

**Feng-Jian Wang (王豐堅)** received his B.S. degree from National Taiwan University, 1980, and M.S. and Ph.D. degree from Northwestern University, U.S.A.,1986 and 1988 respectively. He is currently a professor in the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu. His research interests include software engineering, OO & related techniques, the Internet, workflow and agent applications, and distributed software systems.