Java iterators for C++

Adolfo Di Mare
Universidad de Costa Rica
adolfo.dimare@ecci.ucr.ac.cr

ABSTRACT

We show how to use Java iterators for the C++ standard library containers, which are simpler than those usually used in C++ programs.

Keywords: Iteration abstraction, generator, programming techniques, Java vs C++, software.

1. INTRODUCTION

Abstraction is a concept mentioned when talking about building programs because it is not possible to produce something without having an idea first, although very general, to define what it is to be created. The studious of programming knows which abstractions are used to program: data abstraction, procedural abstraction, iteration abstraction, and data type hierarcal abstraction [LG-2000]. It's easy to ignore iteration abstraction because it is implemented using data abstraction, but that does not diminishes its importance.

The use of iteration abstraction for program construction is not recent, and much has already been discussed on how to incorporate it into a programming language [Parnas-1983]. The ideas that have been implemented, ranging from the proposed generators for Alphard [SW-1977], or iterators [MOSS-1996], to more recent constructions as those proposed for Chapel [JCD-2006]. It has also been argued that it is necessary to specify iterators in order to facilitate reasoning about the correctness of programs [Weide-2006]. If the intelligence to split a task into pieces that can be solved concurrently is in the iterator, the iteration abstraction can be used to increase the concurrency of programs using techniques such as "MapReduce" [DG-2004] (client-server or master-slave), or by using set iterators as it is done in the Galois programming model [KPWRBC-2009].

The current popularity of Java and C++ brings out 2 ways to implement the iteration abstraction. Using operator overloading and templates, C++ iterators act as pointers used to access values from a collection. Java is more in tune with Object Oriented Programming and represents an iterator as an external class that has 2 basic operations: { hasNext() next() }. Java iterators are a concrete realization of ideas well discussed [ZC-1999] or [Eckart-1987]; a great quality of these iterators is that they are very simple which facilitates their specification and usage. Java does not include special syntactic constructions for iteration because the pair { hasNext() next() } is adequate to implement both iterators and generators. For example, unlike CLU iterators [LSAS-1977], Java iterators do allow double iteration over a collection, as it is shown later in the generic implementation of method "isPalindrome()" (see Figure 3).

All these arguments are used to assert that there is agreement that the iteration abstraction is needed to build programs. In the context of the C++ language it is possible to advance in the use of the iteration abstraction by incorporating the Java iteration pattern in the daily use, both to build programs an with the C++ standard library. In this paper it is shown that C++ has enough expressive power for any programmer to use iterators and generators that follow the Java iterator pattern. Additionally, the C++ compiler can verify that an object marked "const" is not modified, reducing the problems that have been identified for iterators implemented in other languages [HPS-2002]. The solution proposed here is simple and compact as the complete implementatio fits in a single header file "iterJava.h", which greatly facilitates the incorporation of the iteration pattern in C++ programs because its enough to include this header file to use the { hasNext() next() } iterators in C++ (this is why it is not necessary to link any library to the C++ program).

2. ITERATOR USAGE

As Java iterators are very simple, it is a good idea to be able to use them in C++. To achieve this goal its is enough to create a C++ wrapper, as a template, to access the more popular C++ containers with Java iterators. The operations of a Java iterator are these [Java-2009]:

hasNext()
Returns "true" until the iteration is finished.

next()
Returns the next iteration value.

remove()
Removes from the container the value recently returned by "next()".

To traverse any C++ standard library container 2 iterators are required [Str-1998]. The first references the next value in the container to be used and the second is a mark outside the iteration range. The C++ iteration cycle is completed when the first iterator reaches up to the second.

```
{   //              [0] [1] [2] [3] [4] [5]
    int V[] = {     0 , 1 , 2 , 3 , 4      };
    int *it    = &V[0];  // it    = V+0;
    int *itEND = &V[5];  // itEND = V+dim(V)
    for ( it = &V[0]; it != itEND ; ++it ) {
        std::cout << (*it);
    }
}
```

Figure 1

C++ standard library iterators work as a pointers into a vector. In Figure 1, pointer "it" is used to access each value in the vector. In each iteration, the value of "it" increases to access the next value in the container until, eventually, "it" denotes a value that is outside the iteration range, which happens when it reaches "itEND" where the cycle is broken.

```
// Java
static void useIterator( java.util.Collection C ) {
    java.util.Iterator iter = C.iterator();
    while ( iter.hasNext() ) {
        System.out.print( iter.next() );
        iter.remove();
    }
}
```

```
// C++
template <class Collection>
void useIterator( Collection& C ) {
    iterJava< typename Collection::iterator > iter( C.begin(), C.end() );
    while ( iter.hasNext() ) {
        std::cout << iter.next();
        C.erase( iter ); // C.erase( iter.current() );
    }
}
```

Figure 2

To get the values stored in a container with a Java iterator it is enough to use its operations, as it is shown in the upper part of Figure 2. Method "hasNext()" indicates whether objects are still accessible, and "next()" gets the next value. Optionally, it is possible to delete from the sequence the value just returned by "next()" invoking method "remove()".

C++ iterators come in different flavors: constant or mutable and forward or backward. So there are 4 types of iterators: "iterator", "const_iterator", "reverse_iterator" and "const_reverse_iterator". The main advantage of these iterators is that they achieve a high degree of independence between algorithms and containers, as the C++ standard library includes many algorithms implemented as templates that use iterators instead of containers. As C++ iterators are very comprehensive and thorough implementations, the task of implementing Java iterators for C++ is simple, to the point that the single class template "iterJava<>" is enough.

3. DESIGN DECISIONS FOR THE "ITERJAVA<>" CLASS

Making the use of C++ iterators similar to Java iterators is the most dominant criteria in deciding among the different design alternatives available. This makes it necessary to follow the java.util.Iterator interface where the 3 operations common to all Java iterators are defined { next() , hasNext() , remove() }. Furthermore, class "iterJava<>" includes the operation "set()" to use the same iterator in another iteration and "current()" to get the current value of iteration, which is what the last invocation of "next()" returned.

It is unusual for a Java iterator to include operation "current()", possibly because the Java objects are references, so if the Java programmer needs to keep the value of the last invocation of "next()" would return it is enough to store it in some temporary variable:

Object tmp = iter.next();

```
template <typename C>
bool isPalindrome( const C& CCC ) {{
    typedef typename C::const_iterator     IterFwd;
    typedef typename std::reverse_iterator<IterFwd> IterBck;

    iterJava< IterFwd > itFwd( CCC.begin(),  CCC.end()  );
    iterJava< IterBck > itBck( CCC.rbegin(), CCC.rend() );

    while ( itFwd.hasNext() && itBck.hasNext() ) {
        if ( itFwd.next() != itBck.next() ) {
            return false;
        }
    }
    return ( !itFwd.hasNext() && !itBck.hasNext() );
}}
```

Figure 3

In Figure 3 it is shown how to traverse the same container object using 2 iterators. As each iterator is an object, it can maintain its state without interfering with other objects. Furthermore, the compiler is responsible for enforcing the constness of these iterators, as they should not change the object being traversed. Unlike Java, C++ has clearly defined the use of "const" to have the compiler prevent a programmer from modifying an object that must not be changed [TE-2005].

```
template <typename C>
bool isPalindrome( const C& CCC ) {{
    typedef typename C::const_iterator Iter;
    iterJava< Iter > itFwd, itBck;
    itFwd.set(CCC);
    itBck.set(CCC); itBck.setReverse();
    while ( itFwd.hasNext() && itBck.hasNext() ) {
        if ( itFwd.next() != itBck.next() ) {
            return false;
        }
    }
    return ( !itFwd.hasNext() && !itBck.hasNext() );
}}
```

Figure 4

As it is shown in Figure 4, methods "setReverse()" and its synonym "setBackward()" in class "iterJava<>" are used for the iteration to be carried out from back to front instead of the natural way, from front to back. There is no "setForward()" operation because it is not valid to use "setReverse()" if the iteration already started, so it is never necessary to invoke "setForward()". In addition to the observer methods "isForward()" and "isBackward()", a copy operations is also included, which is usually present in any class.

```
{{  // test::Tree_BF()                          T = a
    TL::Tree<char> T;     make_a_o(T);             /|\
    Tree_BF<char>  iter;  std::string L;          / / \ \
    iter.set(T);                                 b  c d  e
    while ( iter.hasNext()  ) {                 /|\     /|\
        TL::Tree<char> S = iter.next();        f g h   i j k
        L.push_back( *S );                            / \
    }                                                l m
    assertTrue( L == "abcdefghijklmno" );             / \
}}                                                   n o
```

Figure 5

Perhaps the simplest way to realize the qualities of abstraction "`iterJava<>`" is to examine the non-recursive traversal of a tree, as shown in Figure 5, where it can be seen that the iteration pattern is always the same regardless of where or what is iterated on.

4. "ITERJAVA<>" IMPLEMENTATION

The erase operation "remove()" is named differently from C++ standard library iterators: "erase()". As "erase()" is an operation that does not have a uniform behaviour for all iterators, it is the responsibility of the programmer who uses the class "iterJava<>" determine whether it can be used. For example, when invoking "erase()" in a list its iterators remain valid, but the opposite happens when using "erase()" in a vector.

C++ iterators are constructed based on class "iterator_traits<>" where iterator features are described. Because the parameter for template "iterJava<>" is an iterator, instead of the container traversed with the iterator, to declare the type of the value returned by "iterJava<>::next()" it is necessary to get it from the attributes described in the class "iterator_traits<>" as follows:

```
template <typename Iter>
    typename std::iterator_traits<Iter>::reference
        iterJava<Iter>::next();
```

To implement the delete operation "remove()" for Java iterators, in C++ it is necessary to resort to a trick including in class "iterJava<>" a conversion to an iterator, so the C++ compiler can properly compile the operation invocation for "erase()". As it is shown in Figure 2, to eliminate the value of container recently returned by "next()" a Java programmer writes "iter.remove()" while the C++ programmer should mention the container "C.erase(iter)". The conversion operator "operator Iter() const" returns the iterator that denotes the present value of iteration, and is based on the value of this C++ iterator that the "erase()" method removes the last value of the container returned by operation "next()". An equivalent effect can be obtained using "C.erase(iter.current())", as method "current()" gets, yet again, the value that the last invocation of "next()" returned.

```
{   //              [0] [1] [2] [3] [4] [5]
    int V[] = {     0 , 1 , 2 , 3 , 4      };
    int *it    = &V[0];  // it    = V+0;
    int *itEND = &V[5];  // itEND = V+dim(V)
    iterJava< int* > iter( it, itEND );
    while ( iter.hasNext() ) {
        std::cout << it.next();
    }
}
```

Figure 6

Is it possible for class "iterJava<>" to include a pointer to the container being taversed? The simple answer is yes, but the more complete answer is "better not". As it can be seen at the bottom of Figure 2, the "iterJava<>" class parameter is the type of iterator used to initialize the object used to perform the iteration (in this case, it is variable "iter"). If a pointer to the container is included as a field for class "iterJava<>", it would necessary to use a different class, eg. "iterPtrJava<>", to use pointers directly, and it would not be possible to share the same class "iterJava<>" for values and pointers. In Figure 3 it is shown that traversing a container with an "iterJava<>" object is done in a similar manner as traversing a C++ vector.

```
template <typename Iter>
class iterJava { // Iterate using pointers/iterators
protected:
    Iter itPrev;    // Remember last value returned
    Iter itNext;    // Next value to return
    Iter itEnd;     // Past the last value
    bool isFwd;     // false if it is a reverse_iterator
};
```

Figure 7

The size of an "iterJava<>" object is relatively small and it is usual for programmers to use relatively few iteration variables in their programs. So, instead of saving space, inside an "iterJava<>" object 3 iterators are stored to recall 3 positions in the container: "[itNext]" the next iteration value, "[itPrev]" the value returned on the last invocation of "next()" and "[itEnd]" the indication of the end of the iteration. Furthermore, to allow a "iterJava<>" to go in reverse order, a flag "[isFwd]" is included to indicate whether the iterator moves forward or backward. As it is shown in Figure 7, among the fields for class "iterJava<>" there is no direct reference to the container.

```
// Cannot erase from constant list
void const_compile_error( std::list<long> & L ) {
    typedef iterJava< std::list<long>::const_iterator > Iter;
    Iter iter( L.begin(),L.end() ); //_CONST_
        while ( iter.hasNext() ) {
            L.erase(iter); // forbidden for const_iterator
        }
        assert( L.empty() );
}
```

Figure 8

As only a single class it used for any type of iterator, there arises an important question: if a C++ container is a constant object, will the compiler prevent it to be modified when traversed by an "iterJava<>" iterator? The answer is yes; as this class is a template, upon instantiation the compiler uses the iterator instance included in the declaration. If a non constant iterator is used, the compiler will issue an error when the reference returned by "next()" is used as a left value (l-value) and it will also prevent using the iteration variable for operation "erase()", which requires a non-constant iterator as an argument. For example, in the routine in Figure 8 the compiler will issue the following error:

`... no matching function for call to std::list<...>::erase(...)`

5. USING "ITERJAVA<>" AS AN ITERATION GENERATOR

Iterating over diverse values using the pair { hasNext() next() } is so simple that it is worth using this iteration pattern even if the values are not stored in a container in the C++ standard library. For example, it makes a lot of sense to use this programming style to get a sequence of random numbers or to traverse nonlinear containers, such as trees and graphs.

```
class Random {
     unsigned m_qty;  ///< #.
     unsigned m_last; ///< Max.
     long     m_val;  ///< current().
public:
     Random( int n=1 , long seed=3145159 )
     :   m_qty(0), m_last(n) { srand(seed); }

     bool hasNext() const { return m_qty < m_last; }
     const long& next()
     { m_qty++; return (m_val = rand()) ; }
};
```

```
void generateRandom( ) {
    Random iter( 15, 271828 );
    while ( iter.hasNext()  ) {
        std::cout << iter.next() << std::endl;
    }
}
```

Figure 9

In Figure 9 it is shown how easy it is to encapsulate a random number generator in a Java iterator; at the end of this paper it is included iterator "Tree_BF" to traverse a tree by levels (Breadth-First) which works for the tree

class described in [DiMare-1997]. Class "iterJava<>" can also be used in conjunction with leading C++ libraries such as Boost [Boost-2009].

6. TEACHING AT THE UNIVERSITY

At the Universidad de Costa Rica we have chosen to teach programming with a sequence of 2 courses. In the first we train students in the use and construction of algorithms, and in the second we show them how to reuse modules through abstraction and specification. For the first course we chose the Java language [King-1997] because it has the following qualities:

1. There are very good textbooks for the language [BK-2007], [Ceballos-2006], [Deitel-2007].
2. It includes most of the syntactic constructions of a modern programming language.
3. There are many program development and debugging environments for the language.
4. The compiler does a fairly decent job of detecting and reporting errors based on type checking.
5. It is a popular language both in academia and industry.

From the second programming course and during the whole degree C++ is used as the primary tool for programming. This avoids limiting students to a virtual machine that separates them from the reality of the machine architecture and its operating system. Changing the languages of the first course has several disadvantages which manifest during the second course, among which the following can be highlighted:

1. Both students and teachers have the tendency of focusing on the syntax of second language, to adapt it to the programming practices assimilated by pupils in the first course.
2. As some of the tools for the second language are different, sometimes lot of time get lost trying to make the new tool to have a similar behaviour as the old one.
3. Some programming paradigms for the first language get lost because the programming style for the second language is different.

By changing to the C++ programming paradigm, among other things, Java iterators are also lost, since C++ style iterators are intelligent pointers, implemented as classes that overload the main operations on pointers **{ ++ -- * -> }**, while Java uses generators { hasNext() next() }.

7. CONCLUSIONS

In this work it has been demostrated that there is no need to discard C++ iterators to take advantage of the Java iteration pattern, which is simple, what facilitates iterator specification and usage. The use of class "iterJava<>" helps in taking advantage of this powerful iteration abstraction for C++, increasing the tools available for C++ programmers and helps them to get a better implementation. Moreover, because in C++ it is possible to declare "const" references, the verification that an immutable container is not modified is carried out by the compiler.

If students already know the Java language it is helpful to use class "iterJava<>" in a second course in C++ programming. It is also advantageous to incorporate the Java iterator pattern in the daily practice of any C++ programmer, as it can facilitate incorporating parallelism to an iteration algorithm. The "iterJava.h" implementation presented here fits into a single header file, is efficient because it has no virtual methods and it is implemented using "inline" methods, so using it does not require to give up to the possibility for the compiler to generate better object code. Furthermore, this simple solution can be applied immediately without having to install programs or libraries.

8. ACKNOWLEDGEMENTS

Alejandro Di Mare made several important observations and suggestions for improving this work. Walter Wabe colaborated with the implementation of the iterators { PLR - LPR - LRP } for the "TL::Tree" class.

9. SOURCE CODE

iterJava.zip:

```
http://www.di-mare.com/adolfo/p/iterJava/iterJava.zip
http://www.di-mare.com/adolfo/p/iterJava/es/index.html
http://www.di-mare.com/adolfo/p/iterJava/en/index.html
```

iterJava.h:

```
http://www.di-mare.com/adolfo/p/iterJava/es/iterJava_8h_source.html
http://www.di-mare.com/adolfo/p/iterJava/en/iterJava_8h_source.html
```

test_iterJava.cpp:

```
http://www.di-mare.com/adolfo/p/iterJava/es/classtest__iterJava.html
http://www.di-mare.com/adolfo/p/iterJava/en/classtest__iterJava.html
```

Tree_L.h:

```
http://www.di-mare.com/adolfo/p/iterJava/es/classTL_1_1Tree.html
http://www.di-mare.com/adolfo/p/iterJava/en/classTL_1_1Tree.html
```

Tree_BF.h:

```
http://www.di-mare.com/adolfo/p/iterJava/es/Tree__BF_8h_source.html
http://www.di-mare.com/adolfo/p/iterJava/en/Tree__BF_8h_source.html
```

Tree_PLR.h:

```
http://www.di-mare.com/adolfo/p/iterJava/es/Tree__PLR_8h_source.html
http://www.di-mare.com/adolfo/p/iterJava/en/Tree__PLR_8h_source.html
```

Tree_LPR.h:

```
http://www.di-mare.com/adolfo/p/iterJava/es/Tree__LPR_8h_source.html
http://www.di-mare.com/adolfo/p/iterJava/en/Tree__LPR_8h_source.html
```

Tree_LRP.h:

```
http://www.di-mare.com/adolfo/p/iterJava/es/Tree__LRP_8h_source.html
http://www.di-mare.com/adolfo/p/iterJava/en/Tree__LRP_8h_source.html
```

Doxygen:

```
ftp://ftp.stack.nl/pub/users/dimitri/doxygen-1.5.9-setup.exe
```

REFERENCES

[BK-2007] Barnes, David J. & Kölling Michael: Programación orientada a objetos con Java, ISBN: 978-84-8322-350-5, Pearson Educación, 2007.
http://www.bluej.org/objects-first/

[Boost-2009] Boost: Boost C++ Libraries, 2009.
http://www.boost.org/

[Ceballos-2006] Ceballos, Francisco Javier: Java 2 - Curso de Programación - 3º ed., ISBN 970-15-1164-6, Alfaomega Ra-Ma, 2006.
http://www.fjceballos.es/publicaciones_java.html

[Deitel-2007] Deitel, Harvey M. & Deitel, Paul J.: Java How to Program 7th ed, ISBN: 978-0132222204, Prentice Hall, 2007.
http://www.deitel.com

[DG-2004] Dean, Jeffrey & Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters, Sixth Symposium on Operating System Design and Implementation, OSDI'04, San Francisco, California, Dec 2004.
http://labs.google.com/papers/mapreduce-osdi04.pdf

[DiMare-1997] Di Mare, Adolfo: Una clase C++ completa para conocer y usar árboles, Reporte Técnico ECCI-2005-01, Escuela de Ciencias de la Computación e Informática; Universidad de Costa Rica, 2005.
http://www.di-mare.com/adolfo/p/TreeCpp.htm

[Eckart-1987] Eckart, J. Dana: Iteration and Abstract Data Types, ACM SIGPLAN Notices, Vol.22, No.4, Apr 1987.

[HPS-2002] Hallstrom, Jason O. & Pike, Scott M. & Sridhar, Nigamanth: Iterators Reconsidered, Fifth ICSE Workshop on Component-Based Software Engineering, Orlando Florida, May 2002.

[Java-2009] Sun Micro Systems: Java 2 Platform Standard Edition 5.0 API Specification.
http://java.sun.com/j2se/1.5.0/docs/

[JCD-2006] Joyner, Mackale & Chamberlain, Bradford L. & Deitz, Steven J.: Iterators in Chapel, IPDPS 2006, 20th International Parallel and Distributed Processing Symposium, 2006. pp 25-29, Apr 2006.

[King-1997] King, K. N.: The Case for Java as a First Language, Proceedings of the 35th Annual ACM Southeast Conference, pp 124-131, Apr 1997.
http://knking.com/books/java/java.pdf

[KPWRBC-2009] Kulkarni, Milind & Pingali, Keshav & Walter, Bruce & Ramanarayanan, Ganesh & Bala, Kavita & Chew, Paul: Optimistic parallelism requires abstractions, Communications of the ACM, Vol.52, No.9, Sep 2009.

[LG-2000] Liskov, Barbara & with Guttag, John: Program Development In Java: Abstraction, Specification and Object-Oriented Design, ISBN 0-201-65768-6, Addison-Wesley Professional, 2000.

[LSAS-1977] Liskov, Barbara & Snyder, Alan & Atkinson, Russell & Schaffert, Craig: Abstraction mechanisms in CLU, Communications of the ACM , Vol.20, No.8, Aug 1977.

[MOSS-1996] Murer, Stephan & Omohundro, Stephen & Stoutamire, David & Szyperski, Clemens: Iteration Abstraction in Sather, ACM Transactions on Programming Languages and Systems, Vol. 18, No.1, pp 1-15, Jan 1996.

[Parnas-1983] Parnas, David Lorge: A generalized control structure and its formal definition, Communications of the ACM, Vol.26, No.8 Aug 1983.

[Str-1998] Stroustrup, Bjarne: The C++ Programming Language, 3rd edition, ISBN 0-201-88954-4; Addison-Wesley, 1998.
http://www.research.att.com/~bs/3rd.html

[SW-1977] Shaw, Mary & Wulf, William A.: Abstraction and Verification in Alphard - Defining and Specifying Iteration and Generators, Communications of the ACM, Vol.20, No.8, Aug 1977.

[TE-2005] Tschantz, Matthew S. & Ernst, Michael D.: Javari: Adding reference immutability to Java, OOPSLA'05, pp 211-230, Oct 2005.

[Weide-2006] Weide, Bruce W.: SAVCBS 2006 Challenge: Specification of Iterators, Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), Portland Oregon, Nov 2006.

[ZC-1999] Zendra, Olivier & Colnet, Dominique: Adding external iterators to an existing Eiffel class library, Proceedings on Technology of Object-Oriented Languages and Systems, TOOLS 32, pp 188-199 22-25 Nov 1999.

Authorization and Disclaimer

Authors authorize LACCEI to publish the paper in the conference proceedings. Neither LACCEI nor the editors are responsible either for the content or for the implications of what is expressed in the paper.