**Effective Standard C++ Library: for\_each vs. transform**

**On the surface, the C++ Standard library algorithms for\_each and transform may appear to be quite similar, because they both apply some operation to every element in a sequence. In fact there are significant differences between these two algorithms; ignore them and your code may work okay today but mysteriously crash when you change compilers. The reason for this problem becomes clear as Langer and Kreft detail the differences between these two algorithms. The authors also provide concrete examples of how to se for\_eachand transform appropriately. As you may have guessed, the designers of the Standard C++ library were not simply being redundant. They had two different purposes in mind for for\_each and transform.**

**The Difference between for\_each and transform**

**The algorithms for\_each and transform are often understood as very similar in that they apply an operation (supplied as a function object) to each element in an input range. The difference is that for\_each discards the operation's return value, while transform copies the return value to an element in the output range. This kind of understanding is a fairly common oversimplification. According to the Standard, however, the difference between both algorithms is more fundamental. Our goal in this installment of our column is to explain the conceptual difference between the algorithms and to point out the potential portability trap that the naïve understanding of the algorithms opens.**

**Reading the Standard**

**Before we enter the actual discussion, let us first see what the Standard says**[**[1]**](http://www.ddj.com/cpp/184403769#1)**.**

**FOR\_EACH. The for\_each algorithm is specified in the Standard in the section of non-modifying sequence operations.**

**template<class InputIterator, class Function>**

**Function for\_each(InputIterator first, InputIterator last, Function f);**

* **Effects: Applies f to the result of dereferencing every iterator in the range[first, last), starting from first and proceeding to last - 1.**
* **Returns: f**
* **Complexity: Applies f exactly last - first times.**
* **Notes:If f returns a result, the result is ignored.**

**TRANSFORM. The transform algorithm is specified in the Standard in the section of mutating sequence operations. It comes in two flavors, one version (unary transform) works on one input sequence, the other version (binary transform) takes two input sequences. Since we want to compare for\_each and transform, we only consider the unary transform.**

**template<class InputIterator, class OutputIterator, class UnaryOperation>**

**OutputIterator transform(InputIterator first, InputIterator last,**

**OutputIterator result, UnaryOperation op);**

* **Effects: Assigns through every iterator i in the range [result, result + (last - first)) a new corresponding value equal to op(\*(first + (i - result)).**
* **Requires: op shall not have any side effects**[**[2]**](http://www.ddj.com/cpp/184403769#2)**.**
* **Returns: result + (last - first)**
* **Complexity: Exactly last - first applications of op.**
* **Notes: result may be equal to first.**

**Indeed, the unary transform algorithm and the for\_each algorithm both apply a unary operation to every element in a range of input iterators exactly once. Other than that they have little in common. The differences include:**

* **for\_each is a non-modifying algorithm; transform is a mutating algorithm.**
* **for\_each ignores the operation's return value; transform assigns the return value to successive elements in the output range.**
* **for\_each returns a copy of the function object; transform returns an iterator to the end of the output range.**
* **for\_each applies the operation in a definite order, namely starting at the beginning and proceeding to the end of the input range; no such guarantee is given for transform.**
* **The operation supplied to transform must not have any side effects; no such restriction is imposed on the operation supplied to for\_each.**

**Let us see what these differences mean and why they exist.**

**The Intent**

**When we use an algorithm, we expect that the application of the algorithm has an effect; otherwise the invocation would be pointless. Typical effects include production of a return value and modification of sequence elements.**

**RETURN VALUES. Typical return values produced by algorithms are a Boolean value (see includes), a count (see count\_if), an iterator pointing to a particular element in the input sequence (see find\_if), an iterator pointing to the end of a produced output sequence (see copy), or a pair of iterators denoting an iterator range (see equal\_range). Most algorithms produce a return value, only a few do not (i.e., fill, replace, sort, and swap).**

**Regarding modification of sequence elements, the Standard distinguishes between mutating (or modifying) and inspecting (or non-modifying) algorithms.**

**MUTATORS. Algorithms such as remove, replace, copy, or sort actively produce side effects, namely modification of elements in a sequence. Typically they do so by assigning values through dereferenced iterators. copy, for instance, assigns elements from the input range to elements in the output range. If the modified sequence is the input sequence, then the Standard talks of an *in-place algorithm*; if the modification affects the output range, then it talks of a *copy algorithm*. For instance, replace\_if is an in-place algorithm, while replace\_copy\_if is a copy algorithm.**

**INSPECTORS. The non-modifying algorithms, in contrast, do not assign to any sequence elements. Examples of non-modifying algorithms are find\_if, count\_if, and search. The actual purpose of an inspecting algorithm is not modification of any sequence element, but production of a return value.**

**In this sense, transform is a modifying copy algorithm since it modifies sequence elements by assigning the result of the function object to an element in the output range, whereas for\_each is a non-modifying algorithm because it does not assign to any sequence elements.**

**As stated before, the sole purpose of a non-modifying algorithm is production of a return value. for\_each is a non-modifying algorithm, and it returns the function object that was supplied to it. One might wonder: what is the point in applyingfor\_each to a sequence of elements if it does not modify any elements and returns what it received? Does for\_each have an effect at all? Indeed, for\_each does not actively produce any side effect at all. The actual purpose of invoking for\_eachlies in the effects that the function object that is supplied to for\_each produces when it is invoked for each element in the input sequence. More precisely: the function object may produce an effect by modifying the input sequence, and it can produce a useful result by mutating itself in the course of its invocations.**

**For this reason, the operation supplied to for\_each is not restricted regarding side effects; invocation of for\_each with a side-effect-free function object is utterly pointless. This is radically different for the operation supplied to transform, which according to the Standard must not have any side effects at all. And here we see the fundamental difference between for\_each and transform. for\_each depends on the side effect produced by the function object, while transform produces effects by itself and prohibits any side effects that the function object might produce.**

**In this sense, transform is a modifying copy algorithm since it modifies sequence elements by assigning the result of the function object to an element in the output range, and for\_each is a non-modifying algorithm because it does not assign to any sequence elements.**

**The sole purpose of a non-modifying algorithm is production of a return value.for\_each returns the function object that was supplied to it. Strictly speaking,for\_each does not actively produce any side effect at all. The point of invokingfor\_each is the effects produced by the function object supplied to for\_each. The function object may produce an effect by modify the input sequence, and it can produce a result by mutating itself in the course of its invocations.**

**For this reason, the operation supplied to for\_each is not restricted regarding side effects; this is different from the operation supplied to transform, which according to the Standard must not have any side effects at all. This is the a fundamental difference between for\_each and transform. for\_each depends on the side effect produced by the function object, while transform produces effects by itself and prohibits any side effects that the function object might produce.**

**This difference explains why for\_each gives a guarantee regarding order and number of invocations of the function object. When an operation produces side effects, we need to know how often and in which order the operation is invoked, because it might be sensitive to the number or order of invocations. transform, on the other hand, forbids any side effects of its function object and only guarantees the number of invocations as a sort of a complexity guarantee, but does not say anything about the order of invocation.**

**The Consequences**

**Let us consider the consequences of the specification of for\_each and transformgiven by the Standard. It turns out that the simple notion of "very similar algorithms that only differ in the use of the return value of the function object they invoke" is misleading in many cases.**

**Side Effects**

**A function object with side effects can be supplied to for\_each, but it cannot be supplied to transform. The intent of the Standard is that for\_each is pointless without a side-effect producing function object, while transform does not need any side effects from the function object other than its return value. According to the Standard, the function object supplied to for\_each can have any side effect, and the function object supplied to transform must not have any. Both lead to surprises in practice.**

**Side effects of a function object can be as harmless as writing output to a stream for debugging purposes or modifying its own data members, none of which would interfere with the side effects that the algorithm produces itself. Still, such a function object must not be supplied to transform because it violates the standard requirements.**

**On the other hand, it is common sense that a function object is not free to have any kind of side effect whatsoever. The side effects produced by a function object must not interfere with the activities performed by the algorithm. For instance, invalidating the iterators or the sequences that the algorithm works with is disastrous in any case. Even the function object supplied to for\_each must obey this common-sense rule, even if the Standard does not say so.**

**Order of Invocation**

**A function object that is sensitive to the order of invocation can be supplied tofor\_each, but it is not reasonable to supply it to transform. The Standard does not say anything about the order in which the transform algorithm invokes the function object. For this reason, the result of supplying an order-sensitive operation to transform is unpredictable, while the result is well-defined when supplied to for\_each.**

**Why would this matter in practice? Well, let's study an example.**

**A Concrete Example**

**Say we have the following situation: we have a directory, which contains names and associated information, implemented using a map container. In addition, we have a file that contains a list of names. All entries for the names contained in this file must be removed from the directory. How do we solve such a problem?**

**The first idea might be to use the algorithms remove\_if and remove\_copy\_if: remove an entry from the map if its name is contained in the file (and copy the entry to another map). This of course does not work because remove\_if andremove\_copy\_if are mutating algorithms, which try to assign values to elements in the input sequence through dereferenced iterators. The map container, however, does not allow assignment to its contained elements; its elements are pairs of a const key and an associated value, and the const key cannot be changed. For this reason, an attempted application of remove\_if orremove\_copy\_if to the map container would not compile. Instead of usingremove\_if and remove\_copy\_if, elements in a map are better removed via themap's erase member function.**

**Using for\_each**

**Let us take another approach using for\_each: for each name in the file, apply a function that erases the respective entry from the map. The function object could look like this:**

**template <class MapT>**

**class eraseFct {**

**public:**

**eraseFct(MapT\* m) : theMap(m) {}**

**void operator() (string nam)**

**{ typename MapT::iterator iter = theMap->find(nam);**

**if (iter == theMap->end())**

**throw invalid\_argument(nam);**

**theMap->erase(iter);**

**}**

**private:**

**MapT\* theMap;**

**};**

**template <class MapT>**

**eraseFct<MapT> eraser(MapT\* m)**

**{ return eraseFct<MapT>(m); }**

**The function object would be used like this:**

**map<string,info> directory\_1;**

**// ... populate directory\_1 ...**

**ifstream infile("toBeErased.txt");**

**for\_each(istream\_iterator<string>(infile),istream\_iterator<string>(),**

**eraser(&directory\_1));**

**The use of the function object with for\_each is fine and has the desired effect. The function object's side effect is modification of the map that the function object's data member theMap points to. Note that the side effect is not order-sensitive, so a guarantee regarding the order of invocation of the function object is not needed. In addition, the side effects do not interfere with the activities of the algorithms because the function object does not attempt modification of the input or the output iterators or sequences.**

**So far, so good. Now, imagine the situation changes slightly: instead of removing entries from the directory, we must now split the directory; that is, we must move the entries corresponding to the names in the file from the existing directory into a separate directory. How do we solve the new problem?**

**Using transform**

**An intuitive first idea is to use the transform algorithm with a function very similar to the one that we had used with for\_each: for each name in the file apply a function that erases the respective entry from the map and returns the entry that can then be stored in another map.**

**We slightly modify the original function object so that we can use it withtransform. The main difference compared to the original function object is that the modified function object returns the value of the removed sequence element so that transform can store the value in the output sequence. All necessary modifications are marked in the implementation shown below:**

**template <class MapT>**

**class eraseFct {**

**public:**

**eraseFct(MapT\* m) : theMap(m) {}**

**typename MapT::value\_type operator() (string nam)**

**{ typename MapT::iterator iter = theMap->find(nam);**

**if (iter == theMap->end())**

**throw invalid\_argument(nam);**

**typename MapT::value\_type res = \*iter;**

**theMap->erase(iter);**

**return res;**

**}**

**private:**

**MapT\* theMap;**

**};**

**template <class MapT>**

**eraseFct<MapT> eraser(MapT\* m)**

**{ return eraseFct<MapT>(m); }**

**The function object would be used like this in conjunction with transform for splitting the directory:**

**map<string,info> directory\_2;**

**transform(istream\_iterator<string>(infile),istream\_iterator<string>(),**

**inserter(directory\_2,directory\_2.end()),**

**eraser(&directory\_1));**

**We could also use it in lieu of the original function object with for\_each to solve the initial problem, namely removing the entries:**

**map<string,info> directory\_1;**

**// ... populate directory\_1 ...**

**ifstream infile("toBeErased.txt");**

**for\_each(istream\_iterator<string>(infile),istream\_iterator<string>(),**

**eraser(&directory\_1));**

**The use of the modified function object with for\_each is fine and solves our initial problem as nicely as the original function object did. for\_each simply ignores the function object's return value, and the effect is the same as before with the original function object.**

**The situation is surprisingly different with transform. The effect of supplying the modified function object to transform is neither predictable nor portable, because the Standard only allows side-effect-free function objects in conjunction withtransform, and our function object *has* a side effect, namely removal of an element from the map that its data member points to.**

**Here we see the fundamental difference between for\_each and transform. It's misleading to describe the two algorithms as very similar with the sole difference being that for\_each ignores the return value of the function object whiletransform does not. Instead, the two algorithms work with entirely different categories of function objects: one is side-effect-producing; the other is side-effect-free.**

**Theory vs. Practice**

**The Standard prohibits use of a side-effect-producing function object in conjunction with the transform algorithm. The reason for this is that the Standard wants to give library implementations the latitude for potential optimizations. The requirement that a transformator must not have any side effect whatsoever is a very strong requirement. There is not a lot that a transformator is allowed to do. It cannot modify its own data members; it cannot modify temporaries; it cannot invoke any function that has side effects (for instance writing to a stream); it cannot even retrieve the value of a volatile variable. All it can do is inspect its function argument and other constant, non-volatile fields, calling side-effect-free functions and producing a return value. Under these restrictions a library implementation can safely apply optimizations.**

**One conceivable optimization would be execution of the transformation in parallel threads. A function object that is side-effect free is in particular thread-safe; since it cannot cause any change in the so-called execution environment, there cannot be any potential conflict if the function object is invoked from several threads in parallel. Such an optimized implementation of the transform algorithm would clearly break the code in our example.**

**The transformator in our example might erase an element from a map, which is not an atomic operation. One thread might be in the process of erasing an element while the other checks for the end of the map, which an instant later will be invalidated by the first thread, and as a result the second thread will crash. This is the prototypical race condition, and it stems from the fact that our transformator violates the requirement that it must not have side effects.**

**In practice, you will find that supplying a function object with side effects to thetransform algorithm works just fine and yields predictable and reliable results with most implementations of the standard library. In fact, no library implementation that we know of takes advantage of the latitude that the Standard gives them for optimizing the algorithm. Still, keep in mind: strictly speaking it is not portable to use side-effect-producing transformators.**

**So, what can we do in a portable program instead of using transform? Off-hand we see two potential solutions: implement a relaxed version of transform and use that instead of the standard transform algorithm, or use for\_each instead oftransform.**

**Implement Your Own Version of transform**

**We can implement our own version of the transform algorithm that invokes the function object starting at the beginning and proceeding to the end and allows function objects with arbitrary side effects, except side effects that invalidate the input or output iterators or sequences. Here is a conceivable implementation:**

**template <class InputIterator, class OutputIterator, class Transformator>**

**OutputIterator relaxed\_transform(InputIterator first, InputIterator last,**

**OutputIterator result, Transformator trans) {**

**for ( ; first != last; ++first, ++result)**

**\*result = trans(\*first);**

**return result;**

**}**

**This is the implementation that you find in most implementations of the standard library anyway, but it is safer to use your own version, since it's a portable solution. The algorithm above can be specified as:**

**template<class InputIterator, class OutputIterator, class Transformator >**

**OutputIterator relaxed\_transform(InputIterator first, InputIterator last,**

**OutputIterator result, Transformator trans);**

* **Effects: Applies trans to the result of dereferencing every iterator in the range [first, last) starting from first and proceeding to last - 1 and assigns through every iterator i in the range [result, result + (last - first)) the return value of trans(\*(first + (i - result)).**
* **Requires: trans shall not have any side effects that invalidate any iterator in the range [first, last) and [result, result + (last - first)).**
* **Returns: result + (last - first)**
* **Complexity: Exactly last - first applications of trans and exactly last - first assignments.**
* **Notes: result may be equal to first.**

**In the case that result equals first, that is, input and output sequence are the same, the transform algorithm is used as an in-place transformation. In this case, the user must keep in mind that any modification of the input element performed via the function object will be overwritten by the subsequent assignment to the same element. This introduces a minor pitfall, but a user who supplies a function object that modifies the input elements will probably not use this function object in an in-place transformation anyway.**

**The purpose and benefit of the user-defined relaxed\_transform algorithm is that it eases implementation of portable applications. The downside is that potential performance optimizations that a library implementation might provide for the standard transform algorithm taking advantage of the requirements imposed on the function object of transform are not available in this relaxed, user-defined version of the algorithm.**

**Use for\_each in Case of Doubt**

**Another alternative is to use the for\_each algorithm whenever a side effect must be produced. We could re-implement the function object so that it produces all desired side effects including the one that transform had produced; that is, it removes entries from one directory and inserts them into another directory. Here is a conceivable re-write of the function object:**

**template <class MapT>**

**class moveFct {**

**public:**

**moveFct (MapT\* m1, MapT\* m2) : theMap1(m1), theMap2(m2) {}**

**void operator() (string nam)**

**{ typename MapT::iterator iter = theMap1->find(nam);**

**if (iter == theMap1->end())**

**throw invalid\_argument(nam);**

**theMap2->insert(\*iter);**

**theMap1->erase(iter);**

**}**

**private:**

**MapT\* theMap1;**

**MapT\* theMap2;**

**};**

**template <class MapT>**

**moveFct<MapT> mover(MapT\* m1,MapT\* m2)**

**{ return moveFct<MapT>(m1,m2); }**

**The function object would be used like this:**

**map<string,info> directory\_1;**

**// ... populate directory\_1 ...**

**ifstream infile("toBeErased.txt");**

**for\_each(istream\_iterator<string>(infile),istream\_iterator<string>(),**

**eraser(&directory\_1));**

**This solution culminates in the recommendation that we should in general usefor\_each instead of transform for all transformations that are order-sensitive or have side effects.**

**Summary**

**It is a common misconception that the only difference between the algorithmsfor\_each and transform is the fact that for\_each discards the operation's return value, while transform copies the return value to an element in the output range. A more fundamental difference between the two algorithms is that transform is restricted to side-effect-free function objects, while for\_each is more relaxed regarding the requirements it imposes on its function object.**

**In fact, for\_each is an exception among the algorithms in the standard library: it is the only algorithm that gives exact guarantees regarding order and number of invocations of the function object *and* permits side effects including modification of elements from the input sequence. In detail:**

* **for\_each is the one of the few algorithms in the standard library that gives a guarantee regarding the order in which it invokes the function object**[**[3]**](http://www.ddj.com/cpp/184403769#3)**. This allows application of order-sensitive functionality through a function object. Supplying order-sensitive function objects to any other algorithm is downright nonsensical since the result is unpredictable.**
* **for\_each is the only algorithm that returns the function object. This allows application of function objects that accumulate in their data members' information, which can be retrieved after execution of the algorithm. Supplying such a function object to any other algorithm requires instantiation of those algorithms for a reference to the function object, which is kind of inconvenient since it involves explicit function argument specification, plus you will by struggling with library deficiencies**[**[4]**](http://www.ddj.com/cpp/184403769#4)**.**
* **for\_each is one of the few algorithms that does not restrict the function object regarding side effects**[**[5]**](http://www.ddj.com/cpp/184403769#5)**.This gives users an enormous amount of flexibility regarding the implementation of function objects. All other algorithms impose requirements of varying degrees of severity on the function objects they use.**

**In the next installment of this column, we will discuss unary predicates, which are another category of function objects used in the standard library. We will see which side effects these function objects must or must not have.**

**Notes and References**

**[1] INTERNATIONAL STANDARD.**[**Programming languages**](http://www.ddj.com/cpp/184403769)**&151; C++. ISO/IEC IS 14882:1998(E).**

**[2] The Standard defines a side effect as follows: Accessing an object designated by a volatile lvalue, modifying an object, calling a library I/O function, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment.**

**[3] The other algorithms that give a guarantee regarding the order of invocation are the generalized numeric operations accumulate, inner\_product, partial\_sum, and adjacent\_difference defined in header file <numerics>. These algorithms, different from for\_each, require side-effect-free function objects.**

**[4] Klaus Kreft and Angelika Langer. "Effective Standard C++ Library: Explicit Function Template Argument Specification and the STL," *C/C++ Users Journal C++Experts Forum*, December 1999.**

**[5]The other algorithm that does not restrict the side effects of its function object is generate.**

***Klaus Kreft is Senior Consultant at Siemens Business Services in Germany. He can be reached at klaus.kreft@mch20.sbs.de.***

***Angelika Langer works as an independent instructor and mentor. She can be reached at langer@camelot.de.***

***They are authors of the book*Standard C++ IOStreams and Locales*.***