Sorting Out the Relationships Between Pairs of Iterators, Values, and References

Krister Åhlander

Department of Information Technology, Uppsala University, Uppsala, Sweden
krister.ahlander@it.uu.se

Abstract. Motivated by a wish to sort an array A while simultaneously permuting another array B, iteration over *array pairs* (A, B) is considered.

Traditional solutions to this problem require an adaption of either the algorithm or of the data structure. The generic programming approach described in this paper involves the construction of an iterator adaptor: an iterator pair. The different approaches are implemented in C++ and compared with respect to flexibility and performance.

Our design is also compared with another iterator-based design. When examining our solution, we identify the relationship between a reference type and a value type as an independent abstraction. We find that a valid "reference type" to a value type `T` is not necessarily `T&`. The reference pair developed in this paper serves as an example of a reference type which refers to a standard value pair without being a standard reference.

Our understanding of the relationships between iterator pairs, value pairs, and reference pairs, makes our design simpler than the alternative. It is argued that a recognition of these relationships is useful in many other generic programming contexts as well.

Keywords: C++, iterators, reference types.

1 Introduction

A colleague of ours encountered the following sorting problem while developing the commercial finite element software FemLab 3 [1].

> **Motivating problem:** Consider two huge arrays A and B, where, for each index i, the numbers A_i and B_i are related. The task is to sort A while maintaining the inter-array relationship.

There are three fundamentally different approaches to this problem.

Algorithm-based Adapt the algorithm. Based on any sorting algorithm, it is a simple matter to write a special-purpose sorting algorithm which is dedicated to solving this particular situation.

Data structure-based Reorganize the data structure. Instead of having a pair of arrays, data could be organized as an array of pairs. It is then trivial to accomplish the task by reusing a general-purpose sorting algorithm such as C++ `std::sort`.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 342–356, 2005.

Iterator-based Reuse via generic programming (GP) as pioneered by Stepanov and Lee in the design of STL, the C++ standard template library. Construct an iterator adaptor which makes it possible to reuse `std::sort` without redesigning the data structure.

For the developers of FemLab 3, it was not feasible to change the data structure. It would have required a major redesign of the implementation, and it is also likely that it would have degraded the performance of other parts of the program. Regarding the iterator-based approach, my colleague said that he considered the third alternative for a while, but settled for the more pragmatic first alternative.

The motivation behind the present paper is to discuss the third alternative, the iterator-based GP approach. The key idea is to develop a pair iterator, a concept similar to the `zip_iterator` found in Boost [2]. One such solution has previously been developed by Williams [3]. Independently, we designed an alternative implementation. We compare the iterator-based versions with the other approaches with respect to flexibility and performance. The iterator-based approach is of course superior with respect to flexibility, but when it comes to performance we find that the other approaches are faster.

However, the main issue which we want to discuss concerns the insights drawn from comparing the iterator-based designs with each other. For random access iterators, Williams introduces `OwningRefPair` as a value type in order to properly handle dereferencing. In our approach, we introduce pairs of references as a convenient way to accomplish the same task. When studying the C++ standard in more detail, we realize that our solution is not even standard compliant: the standard requires that a random access iterator with value type `T` *should* have a reference type `T&` [4–Ch. 24.1]. Similar restrictions remain also in the discussions of the new iterator types [5]. However, we argue that the assumption that a value type and a reference type always are related in this way is unnecessarily strict. Our notion of a reference pair which *behaves* as a reference to a pair of values illustrates that a value type `T` may well have different reference types, not only `T&`.

Our presentation is organized as follows. Section 2 reviews the basics of C++ iterators. Section 3 recapitulates Williams' implementation. Our approach is presented in Section 4. Section 5 compares the GP approaches with the traditional approaches, particularly with respect to performance. Section 6 elaborates further on the relationships between iterator types, value types and reference types. Finally, Section 7 summarizes our findings.

2 Review of C++ Iterators

In order to make the paper accessible to a broader audience, we commence with a brief review of C++ iterators, containers and algorithms. STL was proposed by Stepanov and Lee in 1993, and soon thereafter adopted by the C++ standard. See, e.g., the preface of [6] for an account of the historical background. Even

though STL has its limitatations, see, e.g., [5], it has proven to be a very useful tool for C++ programmers today.

STL provides generic containers and algorithms for many common programming tasks. The containers and algorithms communicate via iterators. A key feature of STL is the use of C++ templates. This makes the design type safe and thus more robust, as well as fast, because the template instantiation allows for efficient code optimization in link time.

Iterators have different semantics, depending on their category. The iterator categories defined by STL are input iterators, output iterators, forward iterators, bidirectional iterators and random access iterators. The most basic of these iterator categories are input and output iterators, which provide input and output access, respectively, as well as forward traversal. Forward iterators support both input and output access, but are still limited to forward traversal. Bidirectional iterators also support backward traversal, whereas random access iterators, in addition to the requirements above, support random access.

The different categories allow the algorithms of STL to put different requirements on the iterators they use. The `std::copy` algorithm, for example, requires only input and output iterators, whereas for example the `std::sort` algorithm expects random access iterators.

Different containers provide iterators of different categories. An STL `vector`, for instance, provides random access iterators, whereas an STL `list` only provides bidirectional iterators. But STL containers are not the only C++ mechanisms that can be manipulated via iterators. The design of the iterator's interface has been carefully crafted to allow other data structures to be accessed through iterators. An input stream, for example, can be adapted and used as an input iterator, and a standard pointer does, in fact, qualify as a random access iterator.

As a simple example which illustrates the last point, let us consider an array of hundred time stamps (hours and minutes) which is to be sorted. Assume that the array is declared by

```
typedef std::pair<int,int> TimeStamp; // hour and minute
TimeStamp times[100];                 // array of time stamps
```

After proper initialization, it is then a simple matter to sort the times by a call to the standard sorting routine:

```
std::sort(times,times+100);
```

There are, of course, lots of things going on behind the scene. The call to the generic `sort` is instantiated with pointers of type `TimeStamp*`, but the algorithm needs to know more information about the underlying data types, for example the corresponding value type and reference type. In this case, these types are `TimeStamp` and `TimeStamp&`, respectively, and this is figured out by using the auxiliary class `iterator_traits`. Another detail is that this call to `std::sort` expects `operator<` to be used for comparing two arguments of type `const TimeStamp&`. Since a time stamp is nothing but a standard pair instantiated with two integers, this operator is provided automatically, assuming a lexicographical ordering of the first and second members of the pair.

Table 1. The common category for a pair of iterators is deduced from the categories of the underlying iterators. Note that an input iterator and an output iterator can not be paired.

	input	output	forward	bidirect.	random
input	input	N/A	input	input	input
output	N/A	output	output	output	output
forward	input	output	forward	forward	forward
bidirect.	input	output	forward	bidirect.	bidirect.
random	input	output	forward	bidirect.	random

It is, as already mentioned, not the purpose of the present section to explain STL in detail, but rather to review some of the concepts needed to discuss generic approaches to our motivating problem. We also notice that the data structure-based solution sketched in the introduction is very similar to the example of sorting time stamps above. However, it is not always possible or viable to restructure data in this way, which motivates our search for other approaches.

3 Pairing Off Iterators

In this section, we briefly summarize the solution reported by Williams [3]. Interestingly, his motivation for studying the problem was the same as ours, but beside the required support for random access iterators he also handles the other iterator categories. An auxiliary class, `CommonCategory`, is used to find the appropriate iterator category according to Table 1.

Based on the common category of the iterators, the value type and the reference type of the pair iterator, which he calls `PairIt`, are chosen. Consider two iterators `I1` and `I2` whose value types are `T1` and `T2`, respectively. Using various auxiliary classes, Williams creates the value types and the reference types of a pair iterator `PairIt<I1,I2>` as follows.

- If the pair iterator is an input iterator, the value type is a plain pair, `std::pair<T1,T2>`, and the reference type is `pair<T1,T2>&`. On dereferencing, the underlying iterators are dereferenced and an internal pair is created which holds the result. A reference to the internal value is returned.
- If the pair iterator is an output iterator, Williams follows the standard which prescribes that the value type should be `void`[1] The reference type, however, needs to be something different, since assignments to an object of this type should forward the assignment to the underlying iterators. Williams solves this task by introducing an `OutputPair<I1,I2>` class, which holds references to the underlying iterators.
- Williams recognizes that the "biggest headache" is to support forward, bidirectional, and random access iterators. Here, the standard prescribes that

[1] This requirement is unnecessarily strict and a relaxation has consequently been proposed [7–Article 445].

if the value type is `T`, the reference type should be `T&`. Williams resolves these requirements by introducing `OwningRefPair<T1,T2>` as the value type, and consequently `OwningRefPair<T1,T2>&` as the reference type. The class `OwningRefPair<T1,T2>` contains `T1& first` and `T2& second` as public members. These members may refer to external data, or to data kept in an internal buffer. The handling of the internal buffer requires a utility class `RawMem`, in order to avoid the overhead of dynamic memory allocation.

In addition to figuring out the appropriate types to export by the iterator and the handling of dereferencing, the iterator should of course also provide means of traversal etc. These operations are all fairly straight-forward in the literal sense of the word, since they just forward the appropriate call to the underlying iterators. Therefore, we will not discuss the remaining operations further here.

In order to address the motivating problem, it is not necessary to know all the details behind the scene. Listing 1 illustrates how the pair iterator could be used to solve the problem at hand.

After the first call to `std::sort`, `A` is sorted and the elements of `B` are in the order 2, 3, 4, 1, since the inter-array relationship is maintained. Notice that,

```
// A comparison function
bool compFirst(const std::pair<int,int> &l,
               const std::pair<int,int> &r) {
  return l.first < r.first;
}

// A comparison functor
struct CompSecond {
  template<typename T1, typename T2>
  bool operator()(const T1 &l, const T2 &r) {
    return l.second < r.second;
  }
};

int main() {
  // Two "huge" arrays
  int A[] = { 8,5,6,7 }, B[] = { 1,2,3,4 };

  // An iterator pair for the pair of arrays
  IteratorPair<int*, int*> p(A,B);

  // Reuse standard C++ sort
  std::sort(p,p+4,compFirst);
  std::sort(p,p+4,CompSecond());
}
```

Listing 1: The iterator pair concept offers a convenient way to sort a pair of arrays while maintaining an inter-array relationship

since B originally was sorted, B now contains an encoding of the permutation used to sort A. After the second call to std::sort, both arrays are restored into their original state, since the functor used in this call compares with respect to values in the second array.

Williams also observes that the example's usage of a comparison function, as in the first call to sort, requires unnecessary conversions between the value type of the pair iterator, which is OwningRefPair<int,int>, and the standard pair. Therefore, a functor such as Comp with a parameterized binary predicate should be used instead, as in the second call. In Section 4.2, we present another way of handling the comparisons.

4 An Alternative Design

When solving the motivating problem, we arrived at a design where the key idea of an iterator pair is similar to Williams' solution. However, we focused only on random access iterators, and the comparisons between our designs therefore apply to this category only.

The biggest difference between Williams' design and ours concerns the value type and the reference type: We argue that the natural value type for an iterator pair should be a standard pair, and the reference type should be a pair of references:

```
template <class Iterator1,  class Iterator2>
class IteratorPair
{
public:
  typedef typename iterator_traits<Iterator1>::value_type T1;
  typedef typename iterator_traits<Iterator2>::value_type T2;
  typedef typename iterator_traits<Iterator1>::reference R1;
  typedef typename iterator_traits<Iterator2>::reference R2;
  typedef pair<T1,T2>                          value_type;
  typedef const ReferencePair<R1,R2>           reference;
  // ...
};
```

It is not possible to use std::pair<T1&,T2&>, since it would call for the instantiation of the types T1&& and T2&&, and references to references are not allowed. Therefore, we designed the ReferencePair<R1,R2>.

4.1 Reference Pairs and Reference Traits

Reference pairs are developed in parallel with iterator pairs. To obtain appropriate types related to a reference, we find it convenient to also introduce reference traits, see Listing 2. The Reference traits class is similar to the standard iterator traits, and exports a value type and a reference type. In addition, inspired by [8], reference traits also export a parameter type. Listing 3 shows how the reference pair uses the reference traits. Note the usage of Parameter in the constructor. For instance, consider dereferencing of an iterator of type

```
IteratorPair<int*,IteratorPair<double*,char*> >
```

which leads to a reference of type

```
const ReferencePair<int&, const ReferencePair<double&,char&> >
```

The constructor of this type would have `int&` as its first parameter, which is the same as its `R1` type. Its second parameter, however, would be

```
const ReferencePair<double&,char&>&
```

which is not the same as `R2`, but a reference to `R2` instead.

We also point out that the default constructor is private, since a reference pair always should refer to something. Moreover, the assignment is marked as `const`, since the references are not altered, only what they refer to. The importance of these details are emphasized in Section 6.

```
// ReferenceTrait provide typedefs for a reference.
template<typename R>
struct ReferenceTrait {
  typedef typename R::Value     Value;
  typedef typename R::Reference Reference;
  typedef typename R::Parameter Parameter;
};

// For standard reference to type T, default typedefs are provided
template<typename T>
struct ReferenceTrait<T&>
{
  typedef T  Value;
  typedef T& Reference;
  typedef T& Parameter;
};
```

Listing 2: The reference trait is similar to the iterator trait

4.2 Sorting with `operator<`

The classes sketched above can be used for the motivating problem, exactly as in Listing 1. The only change necessary is to use our `IteratorPair` instead of the class `PairIt`, developed by Williams. It is natural to ask, though, if `operator<` which is defined on `std::pair` can be used when sorting a pair of arrays, cf. the example in Section 2. The answer is no, at least not without some extra machinery. The reason is that the sorting algorithm typically makes comparisons between `pivot` which is of value type, and the object `*it` returned upon dereferencing, and this is of reference type. It would require an implicit conversion from `RefererencePair<T1,T2>` to `pair<T1,T2>`, but this is not resolved by the template overloading mechanism [4–Section 14.8.3]. Therefore, we provide parameterized comparison operators between pairs and reference pairs:

```
template<typename R1, typename R2>
struct ReferencePair {
  typedef typename ReferenceTrait<R1>::Value Value1;
  typedef typename ReferenceTrait<R2>::Value Value2;
  typedef typename ReferenceTrait<R1>::Parameter Parameter1;
  typedef typename ReferenceTrait<R2>::Parameter Parameter2;
  typedef pair<Value1,Value2>                          Value;
  typedef ReferencePair<R1,R2>                         Reference;
  typedef const ReferencePair<Parameter1,Parameter2>& Parameter;

  R1 first; R2 second;

  ReferencePair(Parameter1 a,  Parameter2 b)
    : first(a), second(b) { }

  ReferencePair(const ReferencePair& x)
    : first(x.first), second(x.second) { }

  const ReferencePair & operator=(const ReferencePair & x) const {
    first = x.first; second = x.second;
  }

  const ReferencePair & operator=(const Value & x) const {
    first = x.first; second = x.second;
  }

  operator Value () const {
    return  Value (first, second);
  }
  //...
};
```

Listing 3: The reference pair uses the reference trait

```
template<typename T1, typename T2, typename R1, typename R2>
inline bool operator<(const pair<T1, T2>&, const ReferencePair<R1, R2>&);
```

In addition, we provide overloaded comparisons between reference pairs. The extra operators introduced offers a working solution to our problem, without the unnecessary construction of temporary `pair` objects which implicit conversions requires. However, the usage of a C++ functor as in Listing 1 is probably more elegant, since it does not clutter the name space with several overloaded comparison operators.

5 Comparing the Approaches

We have summarized two different iterator-based versions which solve the motivating problem, but how do these compare with the other approaches discussed in Section 1 and with each other?

Below, we first discuss a few obvious advantages with the GP approach. Next, we present performance measurements.

5.1 Advantages of the Iterator-Based Approach

The GP paradigm allows new data structures to reuse algorithms. The following code snippet illustrates how three arrays are permuted simultaneously. The call to `std::sort` will sort the vector `v1` and permute the others accordingly.

```
int arr1[] = { 2,4,1,3 };   vector<int> v1(arr1,arr1+4);
double arr2[] = { 3.1, 2.2, 5.5, 0.1 }; char arr3[] = "RAND";

typedef vector<int>::iterator iterator;
typedef IteratorPair<iterator,IteratorPair<double*,char*> > MyIterator;
MyIterator p(v1.begin(),IteratorPair<double*,char*>(arr2,arr3));

sort(p,p+4);
```

Traditional approaches would have required either a new sorting routine, or a restructuring of data. In either case, we find that the GP approach implies less programming work.

The GP paradigm allows new algorithms to reuse data structures. In our example, the iterator pair and the auxiliary classes were developed in order to reuse the standard `std::sort` for the data structure of the motivating problem. Thanks to the GP paradigm, we do not only solve this task, but we also get the possibility to use the pairs of arrays with other algorithms as a bonus.

As a trivial example, the following code snippet illustrates how the `std::copy` algorithm may be used in order to copy the values in the pair of arrays, A and B, into an array of pairs, C.

```
int A[] = { 2,4,1,3 }; int B[] = { 20,40,10,30 };

typedef IteratorPair<int*, int*> MyIteratorPair;
typedef pair<int, int> MyValue;

MyIteratorPair AB(A,B);
MyValue C[4];

copy(AB,AB+4,C);
```

This additional power of the GP approach is not achieved with traditional approaches.

5.2 Performance

The iterator-based approach clearly has major advantages, but can it compete with respect to performance? To investigate this, we carried out several experiments. Our performance experiments were carried out on a SUN UltraSPARC-IIIi using the GNU compiler version 3.4.3 for Solaris 2.9, with optimization

flag `-O`. For all our experiments, we present minimum time measurements of 5 consecutive iterations, in order to decrease random effects such as irregular work load etc.

To begin with, we had of course to implement the traditional approaches as well. The data-based approach was trivial to implement, see Section 2. We needed to provide comparison functors though, since we do not want the second member of the pair to affect the comparison. The algorithmic-based approach required more work, since we had to be careful to use the same underlying sorting algorithm in all experiments. Based on the library `std::sort` routine, we developed a special version according to the algorithm-based approach. This routine takes three random access iterators as parameters, indicating the beginning and the end of the first array, which holds the keys, and the beginning of the second array, which holds the related data. The g++ "introspective" `sort` routine is a variation of quick sort, but it uses a limit on the recursion depth to avoid the possibility of $\mathcal{O}(N^2)$ complexity [9]. If the recursion limit is reached, the algorithm switches to heap sort, thereby guaranteeing a worst case complexity of $\mathcal{O}(N \log N)$. In our experiments, we use arrays with random data, and it is therefore very unlikely that the the recursion limit is encountered. To simplify the development of our special sorting routine, we decided not to implement the full introspective sorting algorithm, but only the quick sort recursion. In order to make the comparisons fair, we discarded the very few experiments where the recursion depth was reached.

Our first experiment investigates the performance when addressing the original, motivating problem. For different array sizes, we create a pair of arrays with random data. We measure the time to permute both arrays such that the first array becomes sorted, while maintaing the inter-array relationship. Table 2 shows the time measurements in milliseconds for the special sorting routine (Algo), for standard sort of an array of pairs (Data), for our own GP implementation (GP 1), and for Williams GP implementation (GP 2). The results are quite discouraging, since the traditional sorting routines outperform the iterator-based versions. We note though, that our GP version seems more efficient than Williams', probably because of the extra memory handling present in the `OwningRefPair` class.

Since the iterator-based approaches do incur some overhead, we had expected somewhat worse performance for the GP versions, but we were surprised that the degradation was as large as it was. In order to estimate an acceptable—or at least a not easily avoidable—level of performance degradation, we studied the performance drop when using the standard `std::reverse_iterator`, see Table 3. The iterators used are `double*` (Forward), `reverse_iterator<double*>` (Reverse), and finally a reverse iterator of a reverse iterator (Reverse^2). Since the performance degradation is considerable also in this case, we find that we, at least with this compiler, may have to accept the performance degradation of the pair iterators too. A plausible explanation for the performance difference is that the optimizer may keep plain pointers in registers, when they are passed as parameters to functions, whereas this is not possible for more complex types

Table 2. Time measurements (milliseconds) when sorting pairs of arrays of different sizes (N), using the algorithmic and data-based approaches as well as the iterator-based approaches, where GP 1 is our implementation

N	Algo	Data	GP 1	GP 2
100000	40	50	110	130
200000	110	120	220	270
300000	170	180	350	480
400000	230	260	520	640
500000	300	320	640	830

Table 3. Time measurements (milliseconds) when sorting arrays of different sizes (N), using forward iterators, reverse iterators, and reverse reverse iterators

N	Forward	Reverse	Reverse^2
100000	30	60	50
200000	70	100	110
300000	110	160	170
400000	150	220	220
500000	200	280	260

Table 4. Time measurements (milliseconds) when sorting pairs of arrays of integer keys and images, using the algorithmic and data-based approaches as well as the iterator-based approaches, where GP 1 is our implementation. The fastest method, Perm, uses the permutation obtained from sorting the keys.

N	Algo	Data	GP 1	GP 2	Perm
1000	890	1080	1130	1080	100
2000	2010	2490	2530	1810	220
3000	3080	2950	3900	3710	320
4000	4180	6310	3580	3250	290
5000	3440	5310	4690	5490	550

Table 5. Time measurements (milliseconds) when sorting pairs of arrays of images, using the algorithmic and data-based approaches as well as the iterator-based approaches, where GP 1 is our implementation

N	Algo	Data	GP 1	GP 2
100	450	480	480	250
200	1040	1060	1150	610
300	1600	1670	1760	1090
400	2440	2450	2490	1380
500	3000	3200	3130	1900

such as pair iterators. This explanation is supported by the fact that Reverse^2 is not much worse than Reverse.

We have found that the special sorting routines are faster than the GP approaches for sorting pairs of simple types, but how do they perform if we sort

arrays of more complex data? As an example, consider the sorting of pairs of integer keys and images, where each image is 256×256 characters. Table 4 shows the results for this case. The special sort routine is—apart from the permutation based sorting algorithm discussed below—still fastest, but the difference is smaller. One reason for this should be the extra overhead of copying images. In this case, we also find that Williams' implementation performs better than ours. We believe that this is due to his handling of temporary storage. For small data types, it incurs some overhead, but this overhead seems to pay off if the underlying data types consume more memory. This trend is even more significant when we sort pairs of arrays of images, see Table 5. In this case we use images both as keys and as data, and the (somewhat artificial) comparison operator compares two images with respect to their total brightness. In this experiment, the GP2 implementation is actually the winner.

Finally, we remark that it is a simple matter, see Listing 1, to use the pair iterator and standard sort in order to obtain the actual permutation when sorting a set of keys. If the objective is to sort an array of keys and a corresponding array of memory consuming images, it is thus easy to first sort only the keys and obtain the permutation, and then use the permutation to sort the images. With this approach we need much fewer calls to image assignment, an operation which is quite time consuming. As seen in the last column of Table 4, this solution is much faster than any of the other approaches.

6 A Discussion on References

When comparing the design of the two GP approaches, we note that our solution is not standard compliant, since the reference type of the iterator pair is not a standard C++ reference to the value type. However, we do not think that it is a mere coincidence that we are able to solve the problem at hand. Our implementation works since the reference pair we use *behaves* as a reference. Our example illustrates that the assumption that a value type `T` only has one valid reference type `T&` is more strict than it has to be. This is similar to the more well-known situation that there may be different pointer types—also known as random access iterators—which refer to the same value type.

Sections 8.3.2 and 8.5.3 of [4] discuss references in detail. To keep the discussion simple, we ignore `const` and `volatile` qualifiers here and suggest that the key properties of a reference to type `T` are the following.

1. A reference must be initialized by an object of type `T`.
2. Changes to the reference affect only the object being referenced.
3. The reference cannot be changed to refer to another object.
4. The reference can be converted to the value type.
5. The value type can be deduced from the reference.

Thus, it is possible to define a *valid reference type* to type `T` as any type whose objects meet these criteria. We then find that the value type `pair<T1,T2>` admits not only the usual `pair<T1,T2>&` but also `const ReferencePair<T1&,T2&>` as a valid reference type.

As demonstrated by the time measurements in the previous section, our usage of `ReferencePair` as the reference type to a pair iterator seems to perform better, at least for simple value types, than the alternative implemented by Williams. We also think that our design is simpler, since it is not necessary to introduce an auxiliary class `RawMem` for handling an internal buffer. In addition to these arguments, we would like to motivate the soundness of our design by the following argument.

Let `I1` and `I2` be two iterator types, whose value types and reference types are `T1`, `R1` and `T2`, `R2`, respectively, and assume that the value type of `R1` is `T1` and the value type of `R2` is `T2`. Let us now consider the pairing `Pair`[2] of two types to construct a new type. Thus, we may construct `Pair<I1,I2>`, `Pair<T1,T2>`, and `Pair<R1,R2>` as new types. We now argue that relationships between types should be preserved, in such a way that `Pair<I1,I2>` should have `Pair<T1,T2>` as value type and `Pair<R1,R2>` should be its reference type. Moreover, `Pair<R1,R2>` should have `Pair<T1,T2>` as its value type. This argument is similar to the formal definition of a functor, which maps objects (in our case types) to objects, and morphisms (relationships between types) to morphisms. The point we make is that `IteratorPair<I1,I2>`, the standard `std::pair<T1,T2>`, and `ReferencePair<R1,R2>` preserve these relationships.

We therefore find the recognition of the reference relation as an entity in its own right well motivated. This extra level of indirection may also be useful in other contexts. As a simple example (cf. the discussion in [10–Ch. 2.4]), we may use the reference concept in order to write a generic `Swap` (or, perhaps a `ref_swap`) routine, as shown in Listing 4. The template meta function `ValueOfType` is here used to deduce the appropriate value type corresponding to a valid reference. It corresponds to the type transformation `remove_reference` found in the Boost metaprogramming library. Note that this version of `Swap` also handles swapping of two objects of the same value type referred to by different reference types. This situation could also be resolved by parameter overloading. This approach is however not very practical, since the number of overloaded `swap` functions grow exponentially with the number of valid reference types to a given value type.

7 Conclusions

We have investigated different approaches to the problem of sorting a pair of arrays. We find that the algorithm-based version performs best, but it requires the development of a dedicated sorting routine for this particular situation. Running time is of course only one software metric, and we stress that flexibility, robustness, and programmer time in many situations are more important. The data-based version also performs well and it is much easier to implement, if standard generic tools are used. The disadvantage is that the data-structures have to

[2] We deliberately use capital P here, in order to distinguish this concept from `std::pair`.

be restructured, which often means that this approach is inadequate. Therefore, we advocate the iterator-based approach.

We present the design of two different iterator-based approaches, ours, and an implementation by Williams [3]. We acknowledge that his implementation is more comprehensive, since he addresses all iterator categories, not only the

```
template<typename V>
struct ValueOfType {
  typedef V type;
};

template<typename V>
struct ValueOfType<V&> {
  typedef V type;
};

template<typename R1, typename R2>
struct ValueOfType<const ReferencePair<R1,R2> > {
  typedef pair< typename ValueOfType<R1>::type,
                typename ValueOfType<R2>::type > type;
};

template<typename R1, typename R2>
inline void
Swap(R1& a, R2& b)
{
  const typename ValueOfType<R1>::type tmp = a;
  a = b;
  b = tmp;
}

int main {
  IteratorPair<int*, int*> ip1, ip2;
  pair<int,int> vp3;

  // ...

  Swap( *ip1, *ip2 );
  Swap( *ip1, vp3 );
  Swap( vp3, *ip1 );
  Swap( (*ip2).first, ip2->second );
}
```

Listing 4: The `Swap` routine is generic with respect to valid references. The main program illustrates four different calls to `Swap`. The first call to `Swap` shows that `ReferencePair` behaves as a reference. The second and third calls illustrate calls where different reference types are used to swap the same value type. The last call simply swaps two integers

random access iterator category required to solve the motivating problem. However, we find that our approach to treating random access iterators is simpler, because we do not need to introduce an auxiliary class for managing an internal memory buffer.

We notice, though, that our implementation is not standard compliant. The reason for this is that the standard tacitly assumes that there is a one-to-one correspondance between value types and reference types. We believe that this is too strict. In our opinion, there may be several different types which can refer to the same value type, and this is the insight which simplifies our design. We think that this notion should be useful also in other situations where the objective is to develop generic solutions.

Acknowledgements

I thank Stefan Engblom for bringing my attention to the motivating problem. I also thank Adis Hodzic for valuable comments on the manuscript.

References

1. Comsol homepage, `http://www.comsol.com/`.
2. Boost MPL homepage, `http://www.boost.org/libs/mpl/doc/index.html`.
3. Williams, A.: Pairing off iterators. Overload (2001) Avaliable at `http://web.onetel.com/~anthony_w/cplusplus/pair_iterators.pdf`, 2005-04-07.
4. The C++ Standard, Incorporating Technical Corrigendum No. 1. 2 edn. John Wiley & Sons, Ltd (2003)
5. Abrahams, D., Siek, J., Witt, T.: New iterator concepts (2003) `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1550.htm`.
6. Musser, D., Saini, A.: STL Tutorial and Reference Guide. Addison-Wesley (1996)
7. C++ standard library active issues list (revision 35) (2005) `http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html` 2005-03-04.
8. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
9. Musser, D.: Introspective sorting and selection algorithms. Software–Practice and Experience **8** (1997) 983–993
10. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and beyond. Addison-Wesley (2005)