# Pairing off iterators

Anthony Williams

8th May 2002

## 1 Introduction

Recently, a colleague approached me with an interesting problem; he had two containers with corresponding elements, so the $n$-th entry of container $A$ was related to the $n$-th entry of container $B$, and he needed to sort these containers so the elements of $A$ were "in order", without losing the correspondence property, as in figure 1.

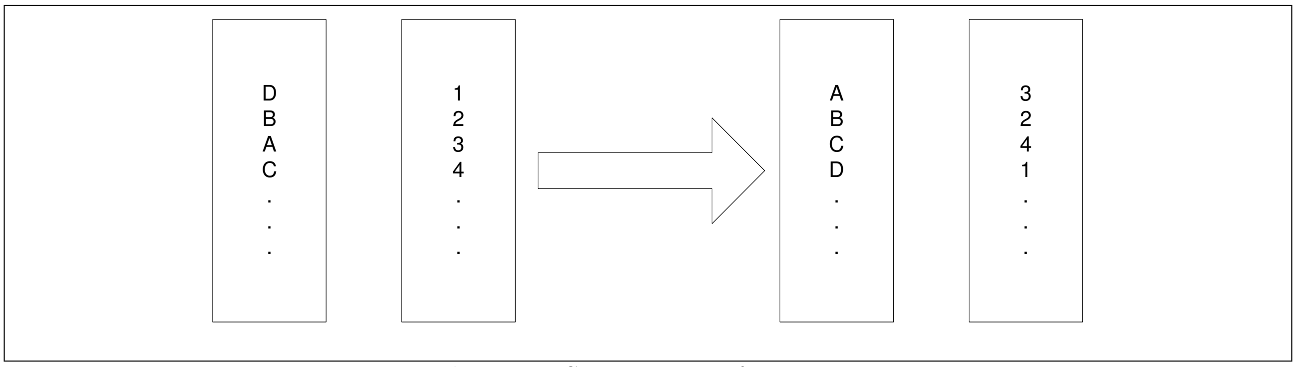

**Figure 1:** Sorting pairs of sequences

There are several solutions to this, each of which has its merits and disadvantages, such as:

1. Create a third container holding either the numeric index, or some form of iterator or pointer into the containers. This container can then be sorted using a special comparison function that references the original containers. Code that processes the data can then either use the new container to index into the originals, or the original containers can be re-shuffled to match the order specified in the new container.
2. Copy the values into a container of pairs and sort that, possibly using the self-sorting property of the Standard Associative Containers, if appropriate

It was a third solution that really interested me — what my colleague conceptually had was a container of pairs of values, even if it was physically stored as a pair of containers of values; why then couldn't we treat the data *as* a container of pairs? This would allow sorting in place, and intuitive access to the data. The answer is: we can — just write an iterator adaptor that iterates through both containers simultaneously, and returns a pair when dereferenced; the pair of containers can thus be viewed as a sequence of pairs of values. The rest of this article covers the complexities hidden in that "just".

## 2 Iterator Categories

To cover the original problem (sorting), we need only worry about random access iterators, since std::sort requires random access. However, the problem had me hooked, and I wanted a general solution, with maximum flexibility, and all the complexities that involved.

For maximum flexibility, we want our adapted iterator to be as capable as possible, but no more — we cannot efficiently provide more facilities than the underlying iterators. Therefore, we must choose the most basic iterator category of the underlying iterators. Since the iterator category tags other than `output_iterator_tag` form an inheritance hierarchy, we can use the implicit conversions applied by the ternary conditional operator `?:` to determine the most basic category — the return type of a conditional expression where the two result expressions are pointers is a pointer to the common base class of the pointed-to classes, so the type of the expression `false ? (std::forward_iterator_tag*)0 : (std::random_access_iterator_tag*)0` is `std::forward_iterator_tag*`, for example. We can then code to handle output iterators separately — if either of the iterators is an output iterator, the result is an output iterator, unless the other is an input iterator, in which case it is an error. This is all handled by the `CommonCategory` class template, shown in listing 1 — the `categoryCheck` functions and `CategoryMap` class templates are shown in listing 2.

```
template<typename Cat1,typename Cat2>
struct CommonCategory
{
private:
    enum {categorySize=sizeof(helper::categoryCheck(false?(Cat1*)0:(Cat2*)0))};
public:
    typedef typename CategoryMap<categorySize>::Type Type;
};
// specializations
template<typename Cat>
struct CommonCategory<std::output_iterator_tag,Cat>
{
    typedef std::output_iterator_tag Type;
};
template<typename Cat>
struct CommonCategory<Cat,std::output_iterator_tag>
{
    typedef std::output_iterator_tag Type;
};
template<>
struct CommonCategory<std::output_iterator_tag,std::output_iterator_tag>
{
    typedef std::output_iterator_tag Type;
};
template<>
struct CommonCategory<std::input_iterator_tag,std::output_iterator_tag>
{
    // no Type, because error
};
template<>
struct CommonCategory<std::output_iterator_tag,std::input_iterator_tag>
{
    // no Type, because error
};
```

**Listing 1:** The `CommonCategory` class template

## 3 Meeting the Requirements

Having chosen our iterator category, we need to implement the appropriate operations to fulfil the Standard Iterator Requirements from Section 24.1 of the C++ Standard [1]. Most operations can easily be implemented by forwarding to the corresponding operations on the underlying iterators — the pre-increment operator can be implemented by incrementing the underlying iterators, for example. It is the dereference operator (`operator*`) and the choice of `value_type` which is complicated, as it depends on the iterator category. Input Iterators

```
// Small, Medium, Large and Huge are types with distinct sizes
Small categoryCheck(std::input_iterator_tag*);
Medium categoryCheck(std::forward_iterator_tag*);
Large categoryCheck(std::bidirectional_iterator_tag*);
Huge categoryCheck(std::random_access_iterator_tag*);

template<>
struct CategoryMap<sizeof(Small)>
{
    typedef std::input_iterator_tag Type;
};

template<>
struct CategoryMap<sizeof(Medium)>
{
    typedef std::forward_iterator_tag Type;
};

// etc.
```

**Listing 2:** The `categoryCheck` overloaded functions and `CategoryMap` specializations

*may* return by value, so if either of the underlying iterators is an Input Iterator, we need to copy the result of dereferencing the underlying iterators. On the other hand, the only operation permitted on the result of dereferencing an Output Iterator is to assign to it, so we cannot store a copy — the dereference operator must return something which, when assigned to, assigns to the result of dereferencing the underlying iterators. Finally, for all other iterators, we need to return a reference that can be used to access and update the elements of the underlying sequences.

The `ValueForCategory` class template assists us with our choice — the `PairIt` iterator template just delegates to `ValueForCategory`, once the appropriate `iterator_category` has been determined, and this is specialized for Input Iterators and Output Iterators, leaving the primary template to handle the other cases.

Implementing the dereference operator for Input Iterators and Output Iterators is actually quite straightforward. For Input Iterators, the `value_type` can be a plain `pair` of values, the elements of which are the `value_type`s of the underlying iterators, and the dereference operator can just copy the values from the underlying iterators into a pair held within the iterator, and return a reference to that pair[1]. For Output Iterators, the `value_type` is `void`, but the result of the dereference operator is something quite different — an `OutputPair` that contains references to the underlying iterators, and which dereferences and writes to the iterators when assigned to. `OutputPair`s cannot be copied, so our iterator's dereference operator should return a reference to an internal instance of `OutputPair`. The definition of `OutputPair` is shown in listing 3.

Supporting Forward Iterators, Bidirectional Iterators and Random Access Iterators is more complicated — the dereference operator must return a reference to the `value_type`, which must hold real references to the elements in the sequences covered by the original iterators. Just to add complexity, we really want the `value_type` to be *Copy-Constructible* and *Assignable*, and to copy the *values* of the elements, not the references, as users wouldn't expect modifying copies of the values to affect the originals; this implies that objects of the same class sometimes contain references to data held elsewhere, and sometimes hold the data directly. For this purpose, we define the `OwningRefPair` class, which has references for its public data members, and contains an internal buffer for the values — the references can either point to external data, in which case the buffer is empty, or they can point to the buffer, in which case the buffer contains instances of the appropriate objects. The objects are stored in an internal buffer, rather than on the heap, to avoid the overhead of dynamic memory allocation; however, this does require care to ensure that the objects are properly destructed, and to ensure that the buffer is correctly aligned.

For alignment, we use a `union` of an appropriately-sized array of `char`, and an instance of `align_t`. `align_t` is itself a `union` of all the fundamental types, and `struct`s containing them. On most platforms, this will

[1]We could return the pair by value, but for uniformity with the other types of iterators, it makes sense to return a reference to an internal object

```
template<typename Iter1,typename Iter2>
struct OutputPair
{
private:
    Iter1& firstIter;
    Iter2& secondIter;

    OutputPair(const OutputPair&); // can't be copied
public:
    OutputPair(Iter1& firstIter_,Iter2& secondIter_):
    firstIter(firstIter_),secondIter(secondIter_)
    {}

    template<typename SomePair>
    OutputPair& operator=(const SomePair& other)
    {
        *firstIter=other.first;
        *secondIter=other.second;
    }
};
```

**Listing 3:** The `OutputPair` class template

have the most rigorous alignment of any type, so (on most platforms[2]) the `union` of `align_t` and the array of `char` is guaranteed to be correctly aligned for any type that has a `sizeof` less than or equal to the size of the array[3]. The `RawMem` template union uses the `sizeof` the template parameter as the size of the `char` array.

We can then cast the address of the `char` array in our `RawMem` union to a pointer of the required type, and use it with placement `new` to construct an instance of the specified type. At the appropriate point, we can also manually invoke the destructor to clean up the object — i.e. in the destructor of our object, we check to see if the buffer contains an object or not, and invoke the destructor if it does, since this is a fixed property of the `OwningRefPair` object — either it contains the referred-to objects in its buffer, or it doesn't, this property doesn't change during its lifetime. In this case, the owned object is an `OwnedPair`; the details are shown in listing 4. Since our `value_types` are distinct, and have different construction syntax, we delegate the actual task of construction and destruction to the `ValueForCategory` template, to give a uniform interface to `PairIt`.

## 4 Putting together the fundamentals

Having pinned down the `value_type` for our iterator, and what we get when we dereference it, we can put together a basic version of our `PairIt`, as in listing 5. This highlights a couple of issues. Firstly, we delegate all the type selection to the `PairItHelper` template, so we can inherit from an appropriate instance of `std::iterator<>` without having to specify all the types explicitly. Secondly, even though `std::iterator<>` defines all the required `typedef`s, we have to repeat them here, so we can use them within the class definition; the base class is a dependent name, so it isn't searched during resolution of unqualified names within the class. This begs the question of whether or not we need to inherit from `std::iterator<>` at all; some existing code expects all iterators that aren't raw pointers to inherit from `std::iterator<>`, and doing so causes no harm. We also can reuse the memory management from `OwningRefPair`, so we don't have to rely on any particular properties of the `value_types` of the underlying iterators, except this time we delegate the construction and destruction to `PairItHelper` as well. Note also that all the members are `mutable` — this is because we don't want to pass on any requirements that the encapsulated iterators be non-`const` for specific operations, and the cache needs to be updated in response to dereferencing the iterator, which is a `const` operation.

[2]Platforms *may* arbitrarily choose to make the alignment of one particular user-defined type distinct from that of any other types

[3]For an implementation that discards types bigger than the type we need the alignment of, to avoid wasting space, see [2]

```
template<typename T,typename U>
struct OwningRefPair
{
public:
    T& first;
    U& second;

    typedef T first_type;
    typedef U second_type;

private:
    struct OwnedPair
    {
        T first;
        U second;

        template<typename Val1,typename Val2>
        OwnedPair(Val1& v1,Val2& v2):
            first(v1),second(v2)
        {}
    };

    utils::RawMem<OwnedPair> pairBuf;
    const bool ownsFlag;

    OwnedPair* getPairPtr()
    {
        return reinterpret_cast<OwnedPair*>(pairBuf.data);
    }
    template<typename Val1,typename Val2>
    void createCopy(Val1& v1,Val2& v2)
    {
        new(getPairPtr()) OwnedPair(v1,v2);
    }

public:
    OwningRefPair(T& first_,U& second_,bool copy):
        first(copy?getPairPtr()->first:first_),
        second(copy?getPairPtr()->second:second_),
        ownsFlag(copy)
    {
        if(ownsFlag)
        {
            createCopy(first_,second_);
        }
    }

    ~OwningRefPair()
    {
        if(ownsFlag)
        {
            getPairPtr()->~OwnedPair();
        }
    }
};
```

**Listing 4:** The `OwningRefPair` class template

```
template<typename Iter1,typename Iter2>
class PairIt:
    public PairItHelper<Iter1,Iter2>::IteratorType
{
private:
    typedef PairItHelper<Iter1,Iter2> PairDefs;
    typedef typename PairDefs::ValueTypeDef ValueTypeDef;
public:
    typedef typename PairDefs::iterator_category iterator_category;
    typedef typename PairDefs::value_type value_type;
    typedef typename PairDefs::difference_type difference_type;
    typedef typename PairDefs::reference reference;
    typedef typename PairDefs::pointer pointer;
private:
    pointer getValuePtr() const
    {
        return reinterpret_cast<pointer>(dataCache.data);
    }
    void emptyCache() const
    {
        if(cacheInitialized)
        {
            ValueTypeDef::destruct(getValuePtr());
            cacheInitialized=false;
        }
    }
    void initCache() const
    {
        emptyCache();
        ValueTypeDef::construct(getValuePtr(),it1,it2);
        cacheInitialized=true;
    }
public:
    PairIt(Iter1 it1_,Iter2 it2_):
        it1(it1_),it2(it2_),cacheInitialized(false)
    {}
    ~PairIt()
    {
        emptyCache();
    }
    reference operator*() const
    {
        initCache();
        return *getValuePtr();
    }
    pointer operator->() const
    {
        initCache();
        return getValuePtr();
    }
private:
    mutable Iter1 it1;
    mutable Iter2 it2;
    mutable utils::RawMem<PairDefs::DeRefType> dataCache;
    mutable bool cacheInitialized;
};
```

**Listing 5:** A basic implementation of `PairIt`

## 5 Beyond the basics

Now that our iterator supports the basic operation of dereferencing, we need to cover the remaining iterator requirements from the C++ Standard. For Input Iterators, the relevant section is 24.1.1, and table 72. This requires that in addition to dereferencing, we also require:

- Copy-construction,
- Assignment,
- Equality and Inequality operators, and
- Pre- and post-increment operators.

These operations are also sufficient for Output Iterators, as they cover all the requirements from section 24.1.2 and table 73.

In all cases, we can just defer to the underlying iterators, and perform the operations on them. However, there is a consequence for exception safety — since we know nothing about the effects of the operations on the underlying iterators, including whether or not they through exceptions, and whether or not the iterator types support a non-throwing `swap` operation, we have to add a disclaimer to the usage of our iterator — if an operation on a `PairIt` throws an exception, then the iterator is to be considered to have become invalid. If we don't add this disclaimer, then it is possible that the state of the iterator may become confused, as (for example) one of the underlying iterators may have advanced, and the other one not.

Another point to make is that copy-construction and assignment should only copy the iterators, not the cache. This is to avoid unnecessary copying of the cached data, which would only provide an additional source of exceptions for no gain — the cache must be regenerated every time the iterators are dereferenced anyway to support Input Iterators that automatically advance when read (which may not actually be allowed anyway). When dealing with non-Input Iterators, the cached value is only a couple of references, so this should add little performance penalty. The alternative is to have every function that modifies the underlying iterators call `emptyCache()`, and only call `initCache()` if the cache is not initialised.

## 6 Moving Forward

For the cases where both the underlying iterators are at least Forward Iterators, we need to meet additional requirements, for `PairIt` to also work as a Forward Iterator. These are given by section 24.1.3 and table 74 of the Standard, and are actually mostly semantic constraints, rather than operational constraints, so these are implemented automatically if the underlying iterators are themselves Forward Iterators. The only additional operation required is that `PairIt` must be *Default-constructible*, which is trivially implemented.

If our underlying iterators are at least Bidirectional Iterators, we should also implement the pre- and post-decrement operators to maintain that level, as detailed in section 24.1.4 and table 75 of the Standard. As for pre- and post-increment, these can be implemented merely by forwarding to the underlying iterators:

```
PairIt& operator--()
{
    --it1;
    --it2;
    return *this;
}
```

## 7 Dereferencing at Random

If our underlying iterators are both Random Access Iterators, then we have a whole swathe of additional requirements to support, as detailed in section 24.1.5 and table 76 of the Standard. These are:

- The arithmetic operators `+` and `-`,
- The arithmetic assignment operators `+=` and `-=`,
- The comparison operators `<`, `>`, `<=` and `>=`, and
- The subscripting operator `[]`.

The arithmetic operators are trivial — just forward to the underlying iterators. The comparison operators require a bit more thought — what do we do if the first iterator is less than its partner, but the second isn't? — but the problems can be defined out of existence; iterators can only be compared if they are in the same range. This implies that one is reachable from the other. If the first and second underlying iterators don't give the same results when compared against their partners in the `PairIt` we are comparing against, then this can't be the case, and the issue can be avoided — in fact, we could get away with just comparing the first iterator in each pair, so the comparison operators are also trivial. The subscripting operator is easy, too — `it[n]` is just syntactic shorthand for `*(it+n)`, so we can implement it that way.

However, a bit more thought reveals that doing things the "simple" way requires that all these operations are either member functions or `friend`s of `PairIt`, yet some could be implemented in terms of others. For example, the idiomatic way of implementing `+` is to use `+=` on a copy, as in listing 6. The same applies to the comparison operators — all the others can be implemented in terms of `operator<`. In fact, `operator<` itself can then be implemented in terms of subtraction, since these are Random Access Iterators, so the list of `friend`s and member functions is now down to:

- `operator+=`,
- `operator-=`,
- `operator-` where both operands are iterators, and
- `operator[]`, which is required to be a member function.

```
template<typename I1,typename I2>
PairIt<I1,I2> operator+(PairIt<I1,I2> temp,std::ptrdiff_t n)
{
    temp+=n;
    return temp;
}
```

**Listing 6:** Implementation of `operator+` for `PairIt`

## 8 Helper functions

Sometimes we don't want to have to specify the precise type of our iterators explicitly, because we're creating a temporary object to pass to an algorithm, and doing so requires excessive typing, if it is possible at all. For this reason, we also provide a helper template function `makePairIterator` that takes two iterators as parameters, and returns a `PairIt` containing them. This simple function is shown in listing 7, and is used like this:

```
std::vector<int> src1;
std::deque<double> src2;

bool myPairComparisonFunc(const std::pair<int,double>&,const std::pair<int,double>&);

std::sort(makePairIterator(src1.begin(),src2.begin()),
          makePairIterator(src1.end(),src2.end()),
          myPairComparisonFunc);
```

```
template<typename Iter1,typename Iter2>
PairIt<Iter1,Iter2> makePairIterator(Iter1 it1,Iter2 it2)
{
    return PairIt<Iter1,Iter2>(it1,it2);
}
```

**Listing 7:** The `makePairIterator` helper function

Of course, as written, this will copy the elements of the containers to do the comparison, as the `value_type` of the pair iterators is a custom pair type, as described above, so a temporary `std::pair` has to be constructed. It is therefore more efficient to write functor class with a template function call operator:

```
struct MyPairComparisonFunctor
{
    template<typename PairType>
    bool operator()(const PairType&,const PairType&) const;
};
```

You could, of course, write a comparison function to take the precise custom pair type in question, but this varies depending on the iterators, so is not as straight-forward as it may seem.

## 9 Conclusion

Implementing an iterator adapter to treat a pair of sequences as a sequence of pairs is not a trivial task, though some of the individual parts are; the biggest headache is deciding the `value_type` and the return type for the dereference operator.

However, it provides a genuinely useful service, and in combination with other iterator adapters and function objects, can be used to access data in intuitive ways, however it is stored.

## References

[1] ISO/IEC 14882, Programming Languages — C++. International Standard, September 1998.

[2] Andrei Alexandrescu. Generic<Programming>: Discriminated unions (II). *C/C++ User's Journal*, 20(6), June 2002. Available online at http://www.cuj.com/experts/2006/alexandr.htm.