make_iterable

(a.k.a. std::range)

Inside-out containers

What we start with

```
class MDTable
{
  MDColumn *m_columns;
  int m_columnCount;
  MDKey *m_keys;
  int m_keyCount;
public:
  MDColumn* GetColumns() const { return m_columns; }
  int GetNumColumns() const { return m_columnCount; }
  MDIndex* GetKeys() const { return m_keys; }
  int GetNumKeys() const { return m_keyCount; }
};
```

So cumbersome

```
void TransformTable(MDTable *tab)
{
  for (int i=0; i < tab->GetNumColumns(); ++i)
  {
    MDColumn& col = tab->GetColumns()[i];
    ... col ...
  }
  for (int i=0; i < tab->GetNumKeys(); ++i)
  {
    MDKey& key = tab->GetKeys()[i];
    ... key ...
  }
}
```

What we'd like to end up with

```
void TransformTable(MDTable *tab)
{
  for (MDColumn& col : Columns(tab))
  {
    ... col ...
  }
  for (MDKey& key : Keys(tab))
  {
    ... key ...
  }
}
```

Our Columns() and Keys() functions

```
#include "iterable.h"

static inline iterable<MDColumn*> Columns(MDTable* tab)
{
  MDColumn* cols = tab->GetColumns();
  return make_iterable(cols, cols + tab->GetNumColumns());
}

static inline iterable<MDKey*> Keys(MDTable* tab)
{
  MDKey* keys = tab->GetColumns();
  return make_iterable(keys, keys + tab->GetNumKeys());
}
```

Our Columns() and Keys() functions

```
#include "iterable.h"

static inline iterable<MDColumn*> Columns(MDTable* tab)
{
  MDColumn* cols = tab->GetColumns();
  return make_iterable(cols, cols + tab->GetNumColumns());
}

static inline iterable<MDKey*> Keys(MDTable* tab)
{
  MDKey* keys = tab->GetColumns();
  return make_iterable(keys, keys + tab->GetNumKeys());
}
```

#include "iterable.h"

```
template<class It>
class iterable
{
  It m_first, m_last;
public:
  iterable() = default;
  iterable(It first, It last) :
    m_first(first), m_last(last) {}
  It begin() const { return m_first; }
  It end() const { return m_last; }
};

template<class It>
inline iterable<It> make_iterable(It a, It b)
{
  return iterable<It>(a, b);
}
```

#include "iterable.h"

```
template<class It>
class iterable
{
  It m_first, m_last;
public:
  iterable() = default;
  iterable(It first, It last) :
    m_first(first), m_last(last) {}
  It begin() const { return m_first; }
  It end() const { return m_last; }
};

template<class It>
inline iterable<It> make_iterable(It a, It b)
{
  return iterable<It>(a, b);
}
```

Alisdair Meredith (N2977) calls it **`std::range`**
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2977.pdf

Marshall Clow calls it **`iterator_pair`**
http://cplusplusmusings.wordpress.com/2013/04/14/range-based-for-loops-and-pairs-of-iterators/

Boost calls it **`iterator_range`**
http://www.boost.org/doc/libs/1_53_0/libs/range/doc/html/range/reference/utilities/iterator_range.html

Original Frankfurt C++11 proposal called it `std::pair`

Original Frankfurt C++11 proposal called it `std::pair`

but this is a bad idea

Original Frankfurt C++11 proposal called it `std::pair`

but this is a bad idea

because there are standard algorithms

that deal in pairs of iterators

that are **not ranges**

Original Frankfurt C++11 proposal called it `std::pair`

but this is a bad idea

because there are standard algorithms

that deal in pairs of iterators

that are **not ranges**

```
template <class InputIt1, class InputIt2>
std::pair<InputIt1,InputIt2> mismatch(InputIt1 first1, InputIt1 last1, InputIt2 first2);

template<class ForwardIt>
std::pair<ForwardIt, ForwardIt> minmax_element(ForwardIt first, ForwardIt last);

template<class InputIt, class OutputIt1, class OutputIt2, class UnaryPredicate>
std::pair<OutputIt1, OutputIt2> partition_copy(InputIt first, InputIt last,
                                               OutputIt1 d_first_true, OutputIt2 d_first_false,
                                               UnaryPredicate p);
```

Inside-out containers

```
template<class It>
class iterable
{
  It m_first, m_last;
public:
  iterable() = default;
  iterable(It first, It last) :
    m_first(first), m_last(last) {}
  It begin() const { return m_first; }
  It end() const { return m_last; }
};

template<class It>
inline iterable<It> make_iterable(It a, It b)
{
  return iterable<It>(a, b);
}
```

Make a “container view” of an object on the fly

One object can have multiple iterable parts, without exposing implementation details

Free functions, as opposed to member functions, can reduce the burden of writing code

Still no word on “ranges” in C++1z (there is a working group)

P.S. – a more complete Columns()

```
static inline iterable<MDColumn*> Columns(MDTable& tab)
{
  MDColumn* cols = tab->GetColumns();
  return make_iterable(cols, cols + tab->GetNumColumns());
}
static inline iterable<const MDColumn*> Columns(const MDTable& tab)
{
  const MDColumn* cols = tab->GetColumns();
  return make_iterable(cols, cols + tab->GetNumColumns());
}

static inline iterable<MDColumn*> Columns(MDTable* tab)
{
  return tab ? Columns(*tab) : {};
}
static inline iterable<const MDColumn*> Columns(const MDTable* tab)
{
  return tab ? Columns(*tab) : {};
}
```