Document Number: N3443=12-0133
Date: 2012-09-21
Project: Programming Language C++, Library Working Group
Reply-to: Gary Powell <gwpowell at gmail dot com>
Tim Blechmann <tim at klingt dot org>

Priority Queue Changes and Additions

Introduction

Priority Queues and heaps are extremely useful data structures suitable for solving many common ordering problems. C++ 2011 provides only a couple of template adaptor classes; priority_queue, stack and queue, which provide limited functionality. To overcome the current limitations, this paper proposes that optional compile time configuration be added to the existing std::priority_queue adaptor. Additionally there are several alternative implementations of heaps having different performance characteristics which should be added to the standard library. Especially, the suggested options for these heaps deal with these additional aspects:

- **Iterators**: Heaps provide iterators to iterate all elements.
- **Mutability**: The priority of heap elements can be modified.
- **Mergeable**: While all heaps can be merged, some can be merged efficiently.
- **Stability**: Heaps can be configured to be stable sorted.
- **Comparison**: Heaps can be compared for equivalence.

Iterators allow a priority queue to be inspected without having to pop every element out of it. This of course is useful for programs which need to store their current state.

Mutability is required to achieve specific algorithmic complexity for some algorithms.
The keys in fibonacci heaps can be changed in O(1) which is better than removing and inserting, which is usually O(log(n)). We propose two different interfaces for mutability: the first family of methods (update, increase, decrease) assigns a new key to a heap node which is referenced by a handle. The increase/decrease methods can be used, if the direction of the key change is known in advance. Apart from this, we propose a more advanced `fixup' interface. This can be used to restore heap order, if the key of a heap node is modified from user code.

Mergability is helpful when efficiently combining two heaps. This is usually more efficient than creating a new heap based on the data from the original heaps. Fibonacci heaps for example can be merged in O(1).

Stability gives the programmer control over the order in which elements which are otherwise equal, are popped from the heap.

Comparison is necessary both for debugging as well as one might wish to compare any other standard containers. Two heaps are equivalent, if their elements are in the same heap order (they do not necessarily have the same internal structure).

The current std::prority_queue is a container adaptor. The authors propose that this interface change to a regular container to allow for better quality of implementation. The adaptor only interface is fragile and in practice not all that useful. Therefore the new heap implementations are all containers vs adaptors.

In addition, a heap_merge function is proposed to merge arbitrary heaps.

Document Number: N3443=12-0133
Date: 2012-09-21
Project: Programming Language C++, Library Working Group
Reply-to: Gary Powell <gwpowell at gmail dot com>
Tim Blechmann <tim at klingt dot org>

The paper uses the boost::mpl syntax to generate some types and instantiate some member functions based on compile time options. Obviously this interface will have to change but at the moment, its not clear to the authors what that interface should be. If the library committee accepts a MPL like library then that would be the interface of choice. It also uses a std::parameter::void_ as a placeholder. However the authors do not want to depend on another library for the acceptance of this one. The full standard text is not provided in this paper, outlining all of the formal constraints on things like constructor complexity etc. If the library is accepted with that as a condition, the authors will be glad to provide the additional work.

The authors fully expect that the reviewing committee will have additional suggestions, and therefore look for guidance even if a full review is not possible due to time constraints. One possible consideration is to whether the library should live in it's own namespace, ie std::heap, or be just reside in the namespace std. Another consideration is whether the individual containers be separated out into unique proposals so that the committee can pick and chose which ones to move forward.

Priority Queue

The template parameter T is the type to be managed by the container.
.
The container supports the following options:

- compare<>, defaults to compare<std::less<T> >
- stable<>, defaults to stable<false>
- stability_counter_type<>, defaults to stability_counter_type<std::uintmax_t>
- allocator<>, defaults to allocator<std::allocator<T> >

Modify the interface from:

template <class T, class Container = vector<T>,
class Compare = less<typename Container::value_type> >
class priority_queue;

to
template <class T, class Container = vector<T>,
class Compare = less<typename Container::value_type>,
class ...Options > class priority_queue

Add the following members:
typedef implementation_defined::allocator_type allocator_type;
typedef implementation_defined::reference reference;
typedef implementation_defined::const_reference const_reference;
typedef implementation_defined::pointer pointer;
typedef implementation_defined::const_pointer const_pointer;
typedef implementation_defined::const_iterator const_iterator;

allocator_type get_allocator(void) const;

```
void reserve(size_type);
value_compare const & value_comp(void) const;
const_iterator cbegin(void) const;
const_iterator cend(void) const;

 // heap equivalence
template<typename HeapType> bool operator==(HeapType const &) const;
template<typename HeapType> bool operator!=(HeapType const &) const;

// heap comparison
template<typename HeapType> bool operator<(HeapType const &) const;
template<typename HeapType> bool operator>(HeapType const &) const;
template<typename HeapType> bool operator>=(HeapType const &) const;
template<typename HeapType> bool operator<=(HeapType const &) const;

// public data members
static const bool constant_time_size;          // size() has constant complexity
static const bool has_ordered_iterators;       // priority queue has ordered iterators
static const bool is_mergable;                 // priority queue is efficiently mergable
static const bool is_stable;                   // priority queue has a stable heap order
static const bool has_reserve;                 // priority queue has a reserve() member
```

d_ary_heap

D-ary heaps are a generalization of binary heap with each non-leaf node having N children. For a low arity, the height of the heap is larger, but the number of comparisons to find the largest child node is bigger. The template parameter T is the type to be managed by the container.

The container supports the following options:

- arity<>, required
- compare<>, defaults to compare<std::less<T> >
- stable<>, defaults to stable<false>
- stability_counter_type<>, defaults to stability_counter_type<std::uintmax_t>
- allocator<>, defaults to allocator<std::allocator<T> >

```
template<class T, class... Options> class d_ary_heap {
public:
  // types
  typedef T                                          value_type;
  typedef implementation_defined::size_type          size_type;
  typedef implementation_defined::difference_type    difference_type;
  typedef implementation_defined::value_compare      value_compare;
  typedef implementation_defined::allocator_type     allocator_type;
  typedef implementation_defined::reference          reference;
  typedef implementation_defined::const_reference    const_reference;
  typedef implementation_defined::pointer            pointer;
  typedef implementation_defined::const_pointer      const_pointer;
  typedef implementation_defined::const_iterator     const_iterator;
```

```
  typedef implementation_defined::const_ordered_iterator const_ordered_iterator;
  typedef implementation_defined::handle_type      handle_type;

  // construct/copy/destruct
  explicit d_ary_heap(value_compare const & = value_compare());
  d_ary_heap(d_ary_heap const &);
  d_ary_heap(d_ary_heap &&);
  d_ary_heap& operator=(d_ary_heap &&);
  d_ary_heap& operator=(d_ary_heap const &);

  // public member functions
  bool empty(void) const;
  size_type size(void) const;
  size_type max_size(void) const;
  void clear(void);
  allocator_type get_allocator(void) const;
  value_type const & top(void) const;
  mpl::if_c< is_mutable, handle_type, void >::type push(value_type const &);
  template<class... Args>
    mpl::if_c< is_mutable, handle_type, void >::type emplace(Args &&...);
  template<typename HeapType> bool operator<(HeapType const &) const;
  template<typename HeapType> bool operator>(HeapType const &) const;
  template<typename HeapType> bool operator>=(HeapType const &) const;
  template<typename HeapType> bool operator<=(HeapType const &) const;
  template<typename HeapType> bool operator==(HeapType const &) const;
  template<typename HeapType> bool operator!=(HeapType const &) const;
  void update(handle_type, const_reference);
  void update(handle_type);
  void increase(handle_type, const_reference);
  void increase(handle_type);
  void decrease(handle_type, const_reference);
  void decrease(handle_type);
  void erase(handle_type);
  void pop(void);
  void swap(d_ary_heap &);
  const_iterator cbegin(void) const;
  const_iterator cend(void) const;
  ordered_iterator ordered_cbegin(void) const;
  ordered_iterator ordered_cend(void) const;
  void reserve(size_type);
  value_compare const & value_comp(void) const;

  // public static functions
  static handle_type s_handle_from_iterator(const_iterator const &);

  // public data members
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
```

```
  static const bool is_mergable;
  static const bool has_reserve;
  static const bool is_stable;
};
```

binomial_heap

Binomial heaps are node-base heaps, that are implemented as a set of binomial trees of piecewise different order. The most important heap operations have a worst-case complexity of O(log n). In contrast to d-ary heaps, binomial heaps can also be merged in O(log n).

```
template<class T, class... Options> class binomial_heap{
public:
 // types
  typedef T                                          value_type;
  typedef implementation_defined::size_type          size_type;
  typedef implementation_defined::difference_type    difference_type;
  typedef implementation_defined::value_compare      value_compare;
  typedef implementation_defined::allocator_type     allocator_type;
  typedef implementation_defined::reference          reference;
  typedef implementation_defined::const_reference    const_reference;
  typedef implementation_defined::pointer            pointer;
  typedef implementation_defined::const_pointer      const_pointer;
  typedef implementation_defined::const_iterator     const_iterator;
  typedef implementation_defined::const_ordered_iterator const_ordered_iterator;
  typedef implementation_defined::handle_type        handle_type;

  // member classes/structs/unions
  template<typename T, typename A0 = std:parameter::void_,
         typename A1 = std:parameter::void_,
         typename A2 = std::parameter::void_,
         typename A3 = std::parameter::void_>
  struct force_inf {

    // public member functions
    template<typename X> bool operator()(X const &, X const &) const;
  };
  template<typename T, typename A0 = std::parameter::void_,
         typename A1 = std::parameter::void_,
         typename A2 = std::parameter::void_,
         typename A3 = std::parameter::void_>
  struct implementation_defined {
    // types
    typedef T                              value_type;
    typedef unspecified                    size_type;
    typedef unspecified                    reference;
    typedef base_maker::compare_argument   value_compare;
    typedef base_maker::allocator_type     allocator_type;
    typedef base_maker::node_type          node;
```

```
    typedef allocator_type::pointer        node_pointer;
    typedef allocator_type::const_pointer  const_node_pointer;
    typedef unspecified                    handle_type;
    typedef base_maker::node_type          node_type;
    typedef unspecified                    node_list_type;
    typedef node_list_type::iterator       node_list_iterator;
    typedef node_list_type::const_iterator node_list_const_iterator;
    typedef unspecified                    value_extractor;
    typedef const_iterator                 const_iterator;
    typedef unspecified                    ordered_iterator;
  };

  // construct/copy/destruct
  explicit binomial_heap(value_compare const & = value_compare());
  binomial_heap(binomial_heap const &);
  binomial_heap(binomial_heap &&);
  explicit binomial_heap(value_compare const &, node_list_type &, size_type);
  binomial_heap& operator=(binomial_heap const &);
  binomial_heap& operator=(binomial_heap &&);
  ~binomial_heap(void);

  // public member functions
  bool empty(void) const;
  size_type size(void) const;
  size_type max_size(void) const;
  void clear(void);
  allocator_type get_allocator(void) const;
  void swap(binomial_heap &);
  const_reference top(void) const;
  handle_type push(value_type const &);
  template<class... Args> handle_type emplace(Args &&...);
  void pop(void);
  void update(handle_type, const_reference);
  void update(handle_type);
  void increase(handle_type, const_reference);
  void increase(handle_type);
  void decrease(handle_type, const_reference);
  void decrease(handle_type);
  void merge(binomial_heap &);
  const_iterator cbegin(void) const;
  const_iterator cend(void) const;
  const_ordered_iterator ordered_cbegin(void) const;
  const_ordered_iterator ordered_cend(void) const;
  void erase(handle_type);
  value_compare const & value_comp(void) const;
  template<typename HeapType> bool operator<(HeapType const &) const;
  template<typename HeapType> bool operator>(HeapType const &) const;
  template<typename HeapType> bool operator>=(HeapType const &) const;
```

```
  template<typename HeapType> bool operator<=(HeapType const &) const;
  template<typename HeapType> bool operator==(HeapType const &) const;
  template<typename HeapType> bool operator!=(HeapType const &) const;

  // public static functions
  static handle_type s_handle_from_iterator(iterator const &);

  // public data members
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
  static const bool is_mergable;
  static const bool is_stable;
  static const bool has_reserve;
}
```

fibonacci_heap

Fibonacci heaps are node-base heaps, that are implemented as a forest of heap-ordered trees. They provide better amortized performance than binomial heaps. Except pop() and erase(), the most important heap operations have constant amortized complexity.

The template parameter T is the type to be managed by the container.

The container supports the following options:

- stable<>, defaults to stable<false>
- compare<>, defaults to compare<std::less<T> >
- allocator<>, defaults to allocator<std::allocator<T> >
- constant_time_size<>, defaults to constant_time_size<true>
- stability_counter_type<>, defaults to stability_counter_type<std:uinmax_t>

Fibonacci heaps have a notion of ‘lazy updates’, which updates the heap structure without forcing a consolidation of the heap. While still having an amortized complexity of O(log(n)), it provides better performance in the average case.

```
template<class T,class... Options> class fibonacci_heap{
public:
  // types
  typedef T                                         value_type;
  typedef implementation_defined::size_type        size_type;
  typedef implementation_defined::difference_type  difference_type;
  typedef implementation_defined::value_compare    value_compare;
  typedef implementation_defined::allocator_type   allocator_type;
  typedef implementation_defined::reference        reference;
  typedef implementation_defined::const_reference  const_reference;
  typedef implementation_defined::pointer          pointer;
  typedef implementation_defined::const_pointer    const_pointer;
  typedef implementation_defined::const_iterator   const_iterator;
  typedef implementation_defined::const_ordered_iterator const_ordered_iterator;
```

```
typedef implementation_defined::handle_type      handle_type;

// construct/copy/destruct
explicit fibonacci_heap(value_compare const & = value_compare());
fibonacci_heap(fibonacci_heap const &);
fibonacci_heap(fibonacci_heap &&);
fibonacci_heap(fibonacci_heap &);
fibonacci_heap& operator=(fibonacci_heap &&);
fibonacci_heap& operator=(fibonacci_heap const &);
~fibonacci_heap(void);

// public member functions
bool empty(void) const;
size_type size(void) const;
size_type max_size(void) const;
void clear(void);
allocator_type get_allocator(void) const;
void swap(fibonacci_heap &);
value_type const & top(void) const;
handle_type push(value_type const &);
template<class... Args> handle_type emplace(Args &&...);
void pop(void);
void update(handle_type, const_reference);
void update_lazy(handle_type, const_reference);
void update(handle_type);
void update_lazy(handle_type);
void increase(handle_type, const_reference);
void increase(handle_type);
void decrease(handle_type, const_reference);
void decrease(handle_type);
void erase(handle_type const &);
const_iterator cbegin(void) const;
const_iterator cend(void) const;
const_ordered_iterator ordered_cbegin(void) const;
const_ordered_iterator ordered_cend(void) const;
void merge(fibonacci_heap &);
value_compare const & value_comp(void) const;
template<typename HeapType> bool operator<(HeapType const &) const;
template<typename HeapType> bool operator>(HeapType const &) const;
template<typename HeapType> bool operator>=(HeapType const &) const;
template<typename HeapType> bool operator<=(HeapType const &) const;
template<typename HeapType> bool operator==(HeapType const &) const;
template<typename HeapType> bool operator!=(HeapType const &) const;

// public static functions
static handle_type s_handle_from_iterator(iterator const &);

// public data members
```

```
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
  static const bool is_mergable;
  static const bool is_stable;
  static const bool has_reserve;
};
```

pairing_heap

Pairing heaps are self-adjusting node-based heaps. Although design and implementation are rather simple, the complexity analysis is yet unsolved. For details, consult:
Pettie, Seth (2005), "Towards a final analysis of pairing heaps", Proc. 46th Annual IEEE Symposium on Foundations of Computer Science, pp. 174–183

The template parameter T is the type to be managed by the container.
The container supports the following options:

- std::heap::compare<>, defaults to compare<std::less<T> >
- std::heap::stable<>, defaults to stable<false>
- std::heap::stability_counter_type<>, defaults to stability_counter_type<std::uintmax_t>
- std::heap::allocator<>, defaults to allocator<std::allocator<T> >
- std::heap::constant_time_size<>, defaults to constant_time_size<true>

```
template<class T, class... Options> class pairing_heap{
public:
  // types
  typedef T                                         value_type;
  typedef implementation_defined::size_type         size_type;
  typedef implementation_defined::difference_type   difference_type;
  typedef implementation_defined::value_compare     value_compare;
  typedef implementation_defined::allocator_type    allocator_type;
  typedef implementation_defined::reference         reference;
  typedef implementation_defined::const_reference   const_reference;
  typedef implementation_defined::pointer           pointer;
  typedef implementation_defined::const_pointer     const_pointer;
  typedef implementation_defined::const_iterator    const_iterator;
  typedef implementation_defined::const_ordered_iterator const_ordered_iterator;
  typedef implementation_defined::handle_type       handle_type;

  // construct/copy/destruct
  explicit pairing_heap(value_compare const & = value_compare());
  pairing_heap(pairing_heap const &);
  pairing_heap(pairing_heap &&);
  pairing_heap& operator=(pairing_heap &&);
  pairing_heap& operator=(pairing_heap const &);
  ~pairing_heap(void);

  // public member functions
```

Reply-to: Gary Powell <gwpowell at gmail dot com>
Tim Blechmann <tim at klingt dot org>

```
  bool empty(void) const;
  size_type size(void) const;
  size_type max_size(void) const;
  void clear(void);
  allocator_type get_allocator(void) const;
  void swap(pairing_heap &);
  const_reference top(void) const;
  handle_type push(value_type const &);
  template<class... Args> handle_type emplace(Args &&...);
  void pop(void);
  void update(handle_type, const_reference);
  void update(handle_type);
  void increase(handle_type, const_reference);
  void increase(handle_type);
  void decrease(handle_type, const_reference);
  void decrease(handle_type);
  void erase(handle_type);
  const_iterator cbegin(void) const;
  const_iterator cend(void) const;
  const_ordered_iterator ordered_cbegin(void) const;
  const_ordered_iterator ordered_cend(void) const;
  void merge(pairing_heap &);
  value_compare const & value_comp(void) const;
  template<typename HeapType> bool operator<(HeapType const &) const;
  template<typename HeapType> bool operator>(HeapType const &) const;
  template<typename HeapType> bool operator>=(HeapType const &) const;
  template<typename HeapType> bool operator<=(HeapType const &) const;
  template<typename HeapType> bool operator==(HeapType const &) const;
  template<typename HeapType> bool operator!=(HeapType const &) const;

  // public static functions
  static handle_type s_handle_from_iterator(iterator const &);

  // public data members
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
  static const bool is_mergable;
  static const bool is_stable;
  static const bool has_reserve;
};
```

skew_heap
Skew heaps are self-adjusting node-based heaps. Although there are no constraints for the tree structure, all heap operations can be performed in O(log n).

The template parameter T is the type to be managed by the container.

The container supports the following options:

- compare<>, defaults to compare<std::less<T> >
- stable<>, defaults to stable<false>
- stability_counter_type<>, defaults to stability_counter_type<std::uintmax_t>
- allocator<>, defaults to allocator<std::allocator<T> >
- constant_time_size<>, defaults to constant_time_size<true>
- store_parent_pointer<>, defaults to store_parent_pointer<true>. Maintaining a parent pointer adds some maintenance and size overhead, but iterating a heap is more efficient.
- mutable<>, defaults to mutable<false>.

```
template<class T, class... Options> class skew_heap{
public:
  // types
  typedef T                                                   value_type;
  typedef implementation_defined::size_type                   size_type;
  typedef implementation_defined::difference_type             difference_type;
  typedef implementation_defined::value_compare               value_compare;
  typedef implementation_defined::allocator_type              allocator_type;
  typedef implementation_defined::reference                   reference;
  typedef implementation_defined::const_reference             const_reference;
  typedef implementation_defined::pointer                     pointer;
  typedef implementation_defined::const_pointer               const_pointer;
  typedef implementation_defined::const_iterator              const_iterator;
  typedef implementation_defined::const_ordered_iterator      const_ordered_iterator;
  typedef mpl::if_c< is_mutable, typename implementation_defined::handle_type, void * >::type
handle_type;

  // member classes/structs/unions

  struct implementation_defined {
    // types
    typedef T                                   value_type;
    typedef base_maker::compare_argument        value_compare;
    typedef base_maker::allocator_type          allocator_type;
    typedef base_maker::node_type               node;
    typedef allocator_type::pointer             node_pointer;
    typedef allocator_type::const_pointer       const_node_pointer;
    typedef unspecified                         value_extractor;
    typedef std::array< node_pointer, 2 > child_list_type;
    typedef child_list_type::iterator           child_list_iterator;
    typedef const_iterator                      const_iterator;
    typedef unspecified                         const_ordered_iterator;
    typedef unspecified                         reference;
    typedef unspecified                         handle_type;
  };
```

Reply-to: Gary Powell <gwpowell at gmail dot com>
Tim Blechmann <tim at klingt dot org>

```
// construct/copy/destruct
explicit skew_heap(value_compare const & = value_compare());
skew_heap(skew_heap const &);
skew_heap(skew_heap &&);
skew_heap& operator=(skew_heap const &);
skew_heap& operator=(skew_heap &&);
~skew_heap(void);

// public member functions
mpl::if_c< is_mutable, handle_type, void >::type push(value_type const &);
template<typename... Args>
  mpl::if_c< is_mutable, handle_type, void >::type emplace(Args &&...);
bool empty(void) const;
size_type size(void) const;
size_type max_size(void) const;
void clear(void);
allocator_type get_allocator(void) const;
void swap(skew_heap &);
const_reference top(void) const;
void pop(void);
const_iterator cbegin(void) const;
const_iterator cend(void) const;
const_ordered_iterator ordered_begin(void) const;
const_ordered_iterator ordered_end(void) const;
void merge(skew_heap &);
value_compare const & value_comp(void) const;
template<typename HeapType> bool operator<(HeapType const &) const;
template<typename HeapType> bool operator>(HeapType const &) const;
template<typename HeapType> bool operator>=(HeapType const &) const;
template<typename HeapType> bool operator<=(HeapType const &) const;
template<typename HeapType> bool operator==(HeapType const &) const;
template<typename HeapType> bool operator!=(HeapType const &) const;
void erase(handle_type);
void update(handle_type, const_reference);
void update(handle_type);
void increase(handle_type, const_reference);
void increase(handle_type);
void decrease(handle_type, const_reference);
void decrease(handle_type);

// public static functions
static handle_type s_handle_from_iterator(iterator const &);

// public data members
static const bool constant_time_size;
static const bool has_ordered_iterators;
static const bool is_mergable;
```

```
  static const bool is_stable;
  static const bool has_reserve;
  static const bool is_mutable;
};
```

Options

These are the current list of options for controlling the compile time aspects of the heaps.

template<typename T> struct compare;
Predicate for defining the heap order, optional (defaults to compare<std::less<T> >)

template<typename T> struct allocator;
Allocator for internal memory management, optional (defaults to allocator<std::allocator<T> >)

template<bool T> struct stable;
Configures the heap to use a stable heap order, optional (defaults to stable<false>).

template<bool T> struct mutable_
Configures the heap to be mutable. d_ary_heap and skew_heap have to be configured with this policy to enable the mutability interface.

template<typename IntType> struct stability_counter_type;
Configures the integer type used for the stability counter, optional (defaults to stability_counter_type<std::uintmax_t>).

template<bool T> struct constant_time_size;
Specifies, whether size() should have linear or constant complexity. This argument is only available for node-based heap data structures and if available, it is optional (defaults to constant_time_size<true>)

template<bool T> struct store_parent_pointer;
Store the parent pointer in the heap nodes. This policy is only applicable to the skew_heap.

template<unsigned int T> struct arity;
Specifies the arity of a d-ary heap. For details, consult the class reference of d_ary_heap

In order to provide the historical interface to std:prority_queue we may also want to provide:
template<typename T, class Allocator> struct container;

Iterators

```
class iteratable_heap_interface
{
public:
   // types
```

```
    typedef unspecified        iterator;
    typedef unspecified        const_iterator;
    typedef unspecified        const_ordered_iterator;

    // public member functions
    const_iterator cbegin(void) const;
    const_iterator cend(void) const;

    const_ordered_iterator ordered_cbegin(void) const;
    const_ordered_iterator ordered_cend(void) const;
};
```

Priority queues provide iterators, that can be used to traverse their elements. All heap iterators are const_iterators, that means they cannot be used to modify the values, because changing the value of a heap node may corrupt the heap order.

Iterators do not visit heap elements in any specific order. Unless otherwise noted, all non-const heap member functions invalidate iterators, while all const member functions preserve the iterator validity.

Heap Merge

template<typename Heap1, typename Heap2> void heap_merge(Heap1 &rhs, Heap2 &lhs);

merge rhs into lhs
Effect: lhs contains all elements that have been part of rhs, rhs is empty.

Acknowledgements
The changes proposed here are based on work done at www.boost.org by Tim Blechmann's Heap library which can be found at http://www.boost.org/doc/libs/1_51_0/doc/html/heap.html