

# EASTL Quick Reference

v1.0

Luc Isaak

EASTL written and

maintained by: Paul Pedriana

Reviewers: Jeremy Paudling  
Paul Pedriana  
Michael Polak  
Russ Trunt

EASTL v1.10.03

## 1. Containers.....2

struct <b>array</b>	2
class <b>basic_string</b>	2
class <b>fixed_string</b>	3
class <b>fixed_substring</b>	3
class <b>bitset</b>	3
class <b>deque</b>	3
class <b>queue</b>	3
class <b>priority_queue</b>	3
class <b>list</b>	4
class <b>fixed_list</b>	4
class <b>intrusive_list</b>	4
class <b>slist</b>	4
class <b>fixed_slist</b>	4
class <b>map</b>	4
class <b>fixed_map</b>	5
class <b>hash_map</b>	5
class <b>fixed_hash_map</b>	5
class <b>vector_map</b>	5
class <b>intrusive_hash_map</b>	5
class <b>multimap</b>	6
class <b>fixed_multimap</b>	6

class <b>hash_multimap</b>	6	not2	12
class <b>vector_multimap</b>	6	class <b>binder1st</b>	12
class <b>intrusive_hash_multimap</b>	7	<b>bind1st</b>	12
class <b>set</b>	7	class <b>binder2nd</b>	12
class <b>fixed_set</b>	7	<b>bind2nd</b>	12
class <b>hash_set</b>	7	class <b>pointer_to_unary_function</b>	12
class <b>fixed_hash_set</b>	7	class <b>pointer_to_binary_function</b>	12
class <b>intrusive_hash_set</b>	7	<b>ptr_fun</b>	12
class <b>vector_set</b>	8	class <b>mem_fun_t</b>	12
class <b>multiset</b>	8	class <b>mem_fun1_t</b>	12
class <b>fixed_multiset</b>	8	class <b>const_mem_fun_t</b>	12
class <b>hash_multiset</b>	8	class <b>const_mem_fun1_t</b>	12
class <b>vector_multiset</b>	8	<b>mem_fun</b>	12
class <b>intrusive_hash_multiset</b>	9	class <b>mem_fun_ref_t</b>	12
struct <b>pair</b>	9	class <b>mem_fun1_ref_t</b>	12
class <b>ring_buffer</b>	9	class <b>const_mem_fun_ref_t</b>	12
class <b>stack</b>	9	class <b>const_mem_fun1_ref_t</b>	12
class <b>vector</b>	9	<b>mem_fun_ref</b>	12
class <b>fixed_vector</b>	10		
		<b>4. Iterators .....</b>	<b>13</b>
		Iterators Categories	13
		struct <b>iterator</b>	13
		struct <b>iterator_traits</b>	13
		class <b>reverse_iterator</b>	13
		class <b>back_insert_iterator</b>	13
		<b>back_inserter</b>	13
		class <b>front_insert_iterator</b>	13
		<b>front_inserter</b>	13
		class <b>insert_iterator</b>	13
		<b>inserter</b>	13
		<b>distance</b>	13
		<b>advance</b>	13
		<b>5. Smart Pointers .....</b>	<b>13</b>
		class <b>intrusive_ptr</b>	13
		class <b>linked_ptr</b>	13
		class <b>linked_array</b>	13
		class <b>safe_object</b>	14
		class <b>safe_ptr</b>	14
		class <b>scoped_ptr</b>	14
		class <b>scoped_array</b>	14
		class <b>shared_ptr</b>	14
		class <b>shared_array</b>	14
		struct <b>smart_ptr_deleter</b>	14
		struct <b>smart_array_deleter</b>	14
		class <b>weak_ptr</b>	14

Template class parameters in *italic*. typename, class dropped.

## 1. Containers

Common Global Functions & Operators

```

bool operator== (const Container& a, const Container& b;
bool operator!= (...) ;
bool operator< (...) ;
bool operator> (...) ;
bool operator<= (...) ;
bool operator>= (...) ;

```

template <T, size\_t N = 1>

**struct array**

Public Types <b>value_type</b> , <b>reference</b> , <b>iterator</b> , <b>reverse_iterator</b> , <b>const_reference</b> , <b>const_iterator</b> , <b>const_reverse_iterator</b>	<b>Public Member Variables</b> <b>value_type</b> mValue[N];
---	--

**Public Member Functions**

```

void swap(this_type& x);
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
bool empty() const;
size_type size();
size_type max_size() const;
T* data();
const T* data() const;
reference operator[](size_type n);
const_reference operator[](size_type n) const;
reference at(size_type n);
const_reference at(size_type n) const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;
bool validate() const;
int validate_iterator(const_iterator i) const;

```

**Global Functions & Operators (+Containers Common)**

```

Global Functions & Operators (+Containers Common)
void swap(array...& a, array...& b);

template <T, Allocator = eastl::allocator>
class basic_string

```

<b>Public Types</b> <b>value_type</b> , <b>pointer</b> , <b>reference</b> , <b>iterator</b> , <b>reverse_iterator</b> , <b>const_pointer</b> , <b>const_reference</b> , <b>const_iterator</b> , <b>const_reverse_iterator</b> ;	<b>Public Member Functions</b> <b>basic_string();</b> <b>basic_string(const allocator_type&amp; allocator);</b>
--	---

const reference front() const;  
**basic\_string**(const this\_type& x, pos, const reference reference back());  
**basic\_string**(const value\_type\* p, size\_type n =npos);  
**basic\_string**(const allocator\_type& allocator = eastl::allocator("EASTL classname"));  
**basic\_string**(const value\_type\* p, const allocator\_type& allocator = eastl::allocator("EASTL classname"));  
**basic\_string**(size\_type n, const allocator\_type& allocator = eastl::allocator("EASTL classname"));  
**basic\_string**(const value\_type\* p, const allocator\_type& allocator = eastl::allocator("EASTL classname"));  
**basic\_string**(const this\_type& x);  
**basic\_string**(const value\_type\* pBegin, const value\_type\* pEnd, const allocator\_type& allocator = eastl::allocator("EASTL classname"));  
**basic\_string**(CtorDoNotInitialize, size\_type n, const allocator\_type& allocator = eastl::allocator("EASTL classname"));  
**basic\_string**(CtorPrintf, const value\_type\* pFormat, ...);  
const allocator\_type& get\_allocator() const;  
allocator\_type& get\_allocator();  
void set\_allocator(const allocator\_type&);  
iterator begin();  
const\_iterator begin() const;  
iterator end();  
const\_iterator end() const;  
reverse\_iterator rbegin();  
const\_reverse\_iterator rbegin() const;  
reverse\_iterator rend();  
const\_reverse\_iterator rend() const;  
bool empty() const;  
size\_type size();  
size\_type max\_size() const;  
T\* data();  
const T\* data() const;  
reference operator[](size\_type n);  
const\_reference operator[](size\_type n) const;  
reference at(size\_type n);  
const\_reference at(size\_type n) const;  
reference front();  
const\_reference front() const;  
reference back();  
const\_reference back() const;  
bool validate() const;  
int validate\_iterator(const\_iterator i) const;

**basic\_string& replace(size\_type pos, size\_type n, const basic\_string& x);**  
**basic\_string& operator+=(const basic\_string& s);**  
**basic\_string& operator+=(const value\_type\* p);**  
**basic\_string& operator+=(value\_type c);**  
**basic\_string& append(const basic\_string& x);**  
**basic\_string& append(const basic\_string& x, size\_type pos, size\_type n);**  
**basic\_string& append(const value\_type\* p, size\_type n);**  
**basic\_string& append(const value\_type\* p);**  
**basic\_string& append(size\_type n, value\_type c);**  
**basic\_string& append(const value\_type\* pBegin, const value\_type\* pEnd);**  
**basic\_string& append\_sprintf\_va\_list(const value\_type\* pFormat, va\_list args);**  
**basic\_string& append\_sprintf(const value\_type\* pFormat, ...);**  
void push\_back(value\_type c);  
void pop\_back();  
**basic\_string& insert(size\_type pos, const basic\_string& x);**  
**basic\_string& insert(size\_type pos, const basic\_string& x, size\_type beg, size\_type n);**  
**basic\_string& insert(size\_type pos, const value\_type\* p, size\_type n);**  
**basic\_string& insert(size\_type pos, const value\_type\* p);**  
**basic\_string& insert(size\_type pos, value\_type c);**  
**basic\_string& insert(iterator p, value\_type c);**  
**basic\_string& insert(iterator p, size\_type n, value\_type c);**  
**basic\_string& insert(iterator p, const value\_type\* pBegin, const value\_type\* pEnd);**  
**basic\_string& erase(size\_type pos = 0, size\_type n =npos);**  
**basic\_string& erase(iterator p);**  
**basic\_string& erase(iterator pBegin, iterator pEnd);**  
bool empty() const;  
size\_type size();  
size\_type length();  
size\_type max\_size() const;  
size\_type capacity() const;  
void resize(size\_type n, const value\_type& c);  
void resize(size\_type n);  
void reserve(size\_type n =0);  
void set\_capacity(size\_type n =npos);  
void force\_size(size\_type n);  
const value\_type\* data() const;  
const value\_type\* c\_str() const;  
reference operator[](size\_type n);  
const\_reference operator[](size\_type n) const;  
reference at(size\_type n);  
const\_reference at(size\_type n) const;  
reference front();

**basic\_string& replace(size\_type pos, size\_type n, const basic\_string& x);**  
**basic\_string& replace(size\_type pos1, size\_type n1, const basic\_string& x, size\_type pos2, size\_type n2);**  
**basic\_string& replace(size\_type pos, size\_type n1, const value\_type\* p, size\_type n2);**  
**basic\_string& replace(size\_type pos, size\_type n1, const value\_type\* p, size\_type n2);**  
**basic\_string& replace(size\_type pos, size\_type n1, const value\_type\* p);**

**size\_type find\_first\_not\_of(const value\_type\* p, size\_type pos = 0) const;**  
**size\_type find\_first\_not\_of(const value\_type\* p, size\_type n) const;**  
**size\_type find\_last\_not\_of(const value\_type\* p, size\_type pos = npos) const;**  
**size\_type find\_last\_not\_of(const value\_type\* p, size\_type n) const;**  
**size\_type find\_last\_not\_of(const value\_type\* p, size\_type pos = npos) const;**  
**size\_type find\_last\_not\_of(const value\_type\* p, size\_type n) const;**  
**size\_type find\_last\_not\_of(const value\_type\* p, size\_type pos = npos) const;**  
**size\_type substr(size\_type pos = 0, size\_type n) const;**  
**int compare(const basic\_string& x, size\_type pos = 0) const;**  
**int compare(const value\_type\* p, size\_type pos = 0) const;**  
**int compare(const value\_type\* p, size\_type pos, size\_type n) const;**  
**int compare(value\_type c, size\_type pos = 0) const;**  
**size\_type rfind(const basic\_string& x, size\_type pos = npos) const;**  
**size\_type rfind(const value\_type\* p, size\_type pos = npos) const;**  
**size\_type rfind(const value\_type\* p, size\_type pos, size\_type n) const;**  
**size\_type rfind(value\_type c, size\_type pos = npos) const;**  
**size\_type find\_first\_of(const basic\_string& x, size\_type pos = 0) const;**  
**size\_type find\_first\_of(const value\_type\* p, size\_type pos = 0) const;**  
**size\_type find\_first\_of(const value\_type\* p, size\_type pos, size\_type n) const;**  
**size\_type find\_first\_of(value\_type c, size\_type pos = 0) const;**  
**size\_type find\_last\_of(const basic\_string& x, size\_type pos = npos) const;**  
**size\_type find\_last\_of(const value\_type\* p, size\_type pos = npos) const;**  
**size\_type find\_last\_of(const value\_type\* p, size\_type pos, size\_type n) const;**  
**size\_type find\_last\_of(value\_type c, size\_type pos = npos) const;**  
**void make\_lower();**  
**void make\_upper();**  
**void ltrim();**  
**void rtrim();**  
**void trim();**  
**basic\_string left(size\_type n) const;**  
**basic\_string right(size\_type n) const;**  
**basic\_string& sprintf\_va\_list(const value\_type\* pFormat, va\_list arguments);**  
**basic\_string& sprintf(const value\_type\* pFormat, ...);**  
**bool validate() const;**  
**int validate\_iterator(const\_iterator i) const;**

**Global Types**

```

typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;

```

```

typedef basic_string<char8_t> string8;
typedef basic_string<char16_t> string16;
typedef basic_string<char32_t> string32;
Global Functions & Operators (+ Containers Common)
void swap(basic_string...& a,
          basic_string...& b);

bool operator==(
    const basic_string...>::value_type* p,
    const basic_string...>::value_type* p);
bool operator!= (...);
bool operator< (...);
bool operator> (...);
bool operator<= (...);
bool operator>= (...);

bool operator==(
    const basic_string...>& a,
    const basic_string...>::value_type* p);
bool operator!= (...);
bool operator< (...);
bool operator> (...);
bool operator<= (...);
bool operator>= (...);

template <T, size_t nodeCount,
         bool bEnableOverflow = true,
         Allocator = eastl::allocator>
class fixed_string
:basic_string<T, fixed_vector_allocator <...> >
    Has all the basic_string functionality.
        Public Types
fixed_vector_allocator <...>::overflow_allocator_type
overflow_allocator_type;
        Public Member Functions
fixed_string();
fixed_string(const base_type& x,
             size_type pos,
             size_type n =npos);
fixed_string(const value_type* p,
             size_type n);
fixed_string(const value_type* p);
fixed_string(size_type n,
             const value_type& value);
fixed_string(const this_type& x);
fixed_string(const base_type& x);
fixed_string(const value_type* pBegin,
             const value_type* pEnd);
fixed_string(CtorDoNotInitialize,
             size_type n);
fixed_string(CtorSprintf,
             const value_type* pFormat, ...);
this_type& operator=(const base_type& x);
overflow_allocator_type& get_overflow_allocator();
void set_overflow_allocator(
    const overflow_allocator_type&);

template <T>
class fixed_substring
:basic_string<T>
    Public Member Functions
fixed_substring();
fixed_substring(const base_type& x);
fixed_substring(const base_type& x,
               size_type pos,
               size_type n =npos);
fixed_substring(const value_type* p,
               size_type n);
fixed_substring(const value_type* p);

```

```

fixed_substring(const value_type* pBegin,
                const value_type* pEnd);
this_type& operator=(const base_type& x)

// The following functions should be used carefully.
// Do not call unsupported resizing functions.

basic_string& operator=(const basic_string& x);
basic_string& operator=(value_type c);
void swap(basic_string& x);
void resize(size_type n,
            value_type c);
void resize(size_type n);
void reserve(size_type n = 0);
void set_capacity(size_type n);
void clear();
basic_string& operator+=(const basic_string& x);
basic_string& operator+=(const value_type* p);
basic_string& operator+=(value_type c);
basic_string& append(const basic_string& x);
basic_string& append(const basic_string& x,
                    size_type pos,
                    size_type n);
basic_string& append(const value_type* p,
                    size_type n);
basic_string& append(const value_type* p);
basic_string& append(size_type n,
                    value_type c);
basic_string& append(const value_type* pBegin,
                    const value_type* pEnd);
basic_string& append_sprintf_va_list(
    const value_type* pFormat,
    va_list arguments);
basic_string& append_sprintf(
    const value_type* pFormat,...);
void push_back(value_type c);
void pop_back();
basic_string& assign(const value_type* p,
                     size_type n);
basic_string& assign(size_type n,
                     value_type c);
basic_string& insert(size_type pos,
                     const basic_string& x);
basic_string& insert(size_type pos,
                     const basic_string& x,
                     size_type beg,
                     size_type n);
basic_string& insert(size_type pos,
                     const value_type* p,
                     size_type n);
basic_string& insert(size_type pos,
                     const value_type* p);
basic_string& insert(size_type pos,
                     const value_type* p);
basic_string& insert(size_type pos,
                     size_type n,
                     value_type c);
iterator insert(iterator p,
                value_type c);
void insert(iterator p,
            size_type n,
            value_type c);
void insert(iterator p,
            const value_type* pBegin,
            const value_type* pEnd);

```

```

basic_string& erase(size_type pos = 0,
                    size_type n = npos);
iterator erase(iterator p);
iterator erase(iterator pBegin,
              iterator pEnd);
basic_string& sprintf_va_list(
    const value_type* pFormat,
    va_list arguments);
basic_string& sprintf(
    const value_type* pFormat, ...);

template <size_t N>
class bitset
    Public Types
word_type
// reference: a helper class for the operator[] to
// manipulate the
// individual bits: x[i] = b; x[i] = y[j]; b = ~x[i]; x[i].flip()
class reference;
    Public Member Variables
word_type mWord[BITSET_WORD_COUNT(N)];
    Public Member Functions
bitset();
bitset(uint32_t value);

this_type& operator+=(const this_type& x);
this_type& operator=(const this_type& x);
this_type& operator^=(const this_type& x);
this_type& operator<<=(size_t n);
this_type& operator>=(size_t n);
this_type& operator<<(size_t n) const;
this_type& operator>=(size_t n) const;
this_type& set();
this_type& set(size_t i, bool value = true);
this_type& reset();
this_type& reset(size_t i);
this_type& flip();
this_type& flip(size_t i);
this_type& operator~() const;
reference operator[](size_t i);
bool operator[](size_t i) const;
const word_type* data() const;
word_type* data();
unsigned long to_ulong() const;
size_t count() const;
size_t size() const;
bool operator==(const this_type& x) const;
bool operator!=(const this_type& x) const;
bool test(size_t i) const;
bool any() const;
bool none() const;
size_t find_first() const;
size_t find_next(size_t last_find) const;
word_type& DoGetWord(size_t i);
word_type& DoGetWord(size_t i) const;
Global Functions & Operators
bitset<N> operator&(const bitset<N>& a,
                      const bitset<N>& b)
bitset<N> operator|...
bitset<N> operator^...

```

```

template <T, Allocator = eastl::allocator,
         unsigned kDequeSubarraySize =
         DEQUE_DEFAULT_SUBARRAY_SIZE(T)>
class deque
    Public Types
value_type,
pointer,
reference,
iterator,
reverse_iterator,
const_pointer,
const_reference,
const_iterator,
const_reverse_iterator
    Public Member Functions
deque();
deque(const allocator_type& allocator);
deque(size_type n,
      const allocator_type& allocator =
      eastl::allocator("EASTL classname"));
deque(size_type n,
      const value_type& value,
      const allocator_type& allocator =
      eastl::allocator("EASTL classname"));
deque(const this_type& x);
deque(InputIterator first,
      InputIterator last);

allocator_type& get_allocator();
void set_allocator(
    const allocator_type&);

deque& operator=(const this_type&);
void swap(this_type&);

void assign(size_type n,
           const value_type&);
void assign(InputIterator first,
           InputIterator last);

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

bool empty() const;
size_type size() const;

void resize(size_type n,
            const value_type&);
void resize(size_type n);
void set_capacity(size_type n = npos);

reference operator[](size_type n);
operator[](size_type n) const;
reference at(size_type n) const;
at(size_type n) const;
front();
front() const;
back();
back() const;
const_reference top() const;

push_front(const value_type&);
push_front();
push_back(const value_type&);
push_back();
pop_front();
pop_back();

insert(iterator pos,
        const value_type&);
insert(iterator pos,
        size_type n,
        const value_type&);
insert(iterator pos,
        InputIterator first,
        InputIterator last);

```

```

iterator erase(iterator pos);
iterator erase(iterator first,
               iterator last);
void clear();

bool validate() const;
int validate_iterator(const iterator i) const;
Global Functions & Operators (+ Containers Common)
void swap(deque...& a,
          deque...& b);

template <T, Container = deque<T>,
         eastl::allocator,DEQUE_DEFAULT_SUBARRAY_SIZE(T)>
class queue
    Public Types
value_type,
container_type, reference, const_reference;
    Public Member Functions
queue();
queue(const Container & x);

bool empty() const;
size_type size() const;

reference front();
const_reference front() const;

reference back();
const_reference back() const;

void push(const value_type&);
void pop();

container_type& get_container();
const container_type& get_container() const;

template <T, Container = vector<T>, Compare =
less<Container::value_type> >
class priority_queue
    Public Types
value_type,
container_type, reference, const_reference;
    Public Member Functions
priority_queue(const Compare& compare = Compare());
priority_queue(const Compare& compare,
               const Container& x);

priority_queue(InputIterator first,
               InputIterator last,
               const Compare& compare = Compare(),
               const Container& x = Container());

bool empty() const;
size_type size() const;

const_reference top() const;

push_front(const value_type& value);
push_front();
push_back(const value_type&);
push_back();
pop_front();
pop_back();

insert(iterator pos,
        const value_type&);
insert(iterator pos,
        size_type n,
        const value_type&);
insert(iterator pos,
        InputIterator first,
        InputIterator last);

```

<b>template &lt;T, Allocator = eastl::allocator&gt;</b>					
<b>class list</b>					
<b>Public Types</b>					
<b>value_type,</b> <b>pointer,</b> <b>reference,</b> <b>iterator,</b> <b>reverse_iterator,</b> <b>const_pointer,</b> <b>const_reference,</b> <b>const_iterator,</b> <b>const_reverse_iterator</b>					
<b>ListNode&lt;T&gt;</b>					
<b>node_type;</b>					
<b>Public Member Functions</b>					
<b>list();</b>					
<b>list(const allocator_type&amp; allocator);</b>					
<b>list(size_type n,</b> <b>const allocator_type&amp; allocator =</b> <b>eastl::allocator("EASTL classname");</b>					
<b>list(size_type n,</b> <b>const value_type&amp; value,</b> <b>const allocator_type&amp; allocator =</b> <b>eastl::allocator("EASTL classname");</b>					
<b>list(const this_type&amp; x);</b>					
<b>list(InputIterator first,</b> <b>InputIterator last);</b>					
<b>allocator_type&amp; get_allocator();</b>					
<b>void set_allocator(</b> <b>const allocator_type&amp;);</b>					
<b>Global Functions &amp; Operators (+ Containers Common)</b>					
<b>void swap(list&lt;...&gt; a,</b> <b>list&lt;...&gt; b);</b>					
<b>template &lt;T, size_t nodeCount,</b> <b>bool bEnableOverflow = true,</b> <b>Allocator = eastl::allocator&gt;</b>					
<b>class fixed_list</b>					
<b>:list&lt;T, fixed_node_pool&lt;...&gt; &gt;</b>					
<b>Has all the list functionality.</b>					
<b>Public Member Functions</b>					
<b>fixed_list();</b>					
<b>fixed_list(size_type n);</b>					
<b>fixed_list(size_type n,</b> <b>const value_type&amp; value);</b>					
<b>fixed_list(const this_type&amp; x);</b>					
<b>fixed_list (InputIterator first,</b> <b>InputIterator last);</b>					
<b>size_type empty() const;</b>					
<b>size_type size() const;</b>					
<b>void resize(size_type n,</b> <b>const value_type&amp;);</b>					
<b>void resize(size_type n);</b>					
<b>reference front();</b>					
<b>const_reference front() const;</b>					
<b>reference back();</b>					
<b>const_reference back() const;</b>					
<b>push_front(const value_type&amp;);</b>					
<b>push_front();</b>					
<b>push_back(const value_type&amp;);</b>					
<b>push_back();</b>					
<b>pop_front();</b>					
<b>pop_back();</b>					
<b>iterator insert(iterator pos);</b>					
<b>iterator insert(iterator pos,</b> <b>const value_type&amp;);</b>					
<b>void insert(iterator pos,</b> <b>this_type&amp; this_type&amp;);</b>					
<b>void swap(this_type&amp; x);</b>					
<b>iterator begin();</b>					
<b>const_iterator begin() const;</b>					
<b>iterator end();</b>					
<b>const_iterator end() const;</b>					
<b>reverse_iterator rbegin();</b>					
<b>const_reverse_iterator rbegin() const;</b>					
<b>reverse_iterator rend();</b>					
<b>const_reverse_iterator rend() const;</b>					
<b>iterator erase(iterator pos);</b>					
<b>iterator erase(iterator first,</b> <b>iterator last);</b>					
<b>void clear();</b>					
<b>Public Member Functions</b>					
<b>reset();</b>					
<b>remove(const T&amp; x);</b>					
<b>remove_if(Predicate);</b>					
<b>reverse();</b>					
<b>splice(iterator pos,</b> <b>this_type&amp; x);</b>					
<b>splice(iterator pos,</b> <b>this_type&amp; x,</b> <b>iterator i);</b>					
<b>splice(iterator pos,</b> <b>this_type&amp; x,</b> <b>iterator first,</b> <b>iterator last);</b>					
<b>merge(this_type&amp; x);</b>					
<b>merge(this_type&amp; x, Compare);</b>					
<b>unique();</b>					
<b>unique(BinaryPredicate);</b>					
<b>sort();</b>					
<b>sort(Compare);</b>					
<b>bool validate() const;</b>					
<b>int validate_iterator(const_iterator i) const;</b>					
<b>Global Functions &amp; Operators (+ Containers Common)</b>					
<b>void swap(list&lt;...&gt; a,</b> <b>list&lt;...&gt; b);</b>					
<b>template &lt;T, size_t nodeCount,</b> <b>bool bEnableOverflow = true,</b> <b>Allocator = eastl::allocator&gt;</b>					
<b>class fixed_list</b>					
<b>:list&lt;T, fixed_node_pool&lt;...&gt; &gt;</b>					
<b>Has all the list functionality.</b>					
<b>Public Member Functions</b>					
<b>fixed_list();</b>					
<b>fixed_list(size_type n);</b>					
<b>fixed_list(size_type n,</b> <b>const value_type&amp; value);</b>					
<b>fixed_list(const this_type&amp; x);</b>					
<b>fixed_list (InputIterator first,</b> <b>InputIterator last);</b>					
<b>size_type empty() const;</b>					
<b>size_type size() const;</b>					
<b>void resize(size_type n,</b> <b>const value_type&amp;);</b>					
<b>void resize(size_type n);</b>					
<b>reference front();</b>					
<b>const_reference front() const;</b>					
<b>reference back();</b>					
<b>const_reference back() const;</b>					
<b>push_front(const value_type&amp;);</b>					
<b>push_front();</b>					
<b>push_back(const value_type&amp;);</b>					
<b>push_back();</b>					
<b>pop_front();</b>					
<b>pop_back();</b>					
<b>iterator insert(iterator pos);</b>					
<b>iterator insert(iterator pos,</b> <b>const value_type&amp;);</b>					
<b>void insert(iterator pos,</b> <b>this_type&amp; this_type&amp;);</b>					
<b>void swap(this_type&amp; x);</b>					
<b>iterator begin();</b>					
<b>const_iterator begin() const;</b>					
<b>iterator end();</b>					
<b>const_iterator end() const;</b>					
<b>reverse_iterator rbegin();</b>					
<b>const_reverse_iterator rbegin() const;</b>					
<b>reverse_iterator rend();</b>					
<b>const_reverse_iterator rend() const;</b>					
<b>iterator erase(iterator pos);</b>					
<b>iterator erase(iterator first,</b> <b>iterator last);</b>					
<b>void clear();</b>					
<b>Public Member Functions</b>					
<b>reset();</b>					
<b>remove(const T&amp; x);</b>					
<b>remove_if(Predicate);</b>					
<b>reverse();</b>					
<b>splice(iterator pos,</b> <b>this_type&amp; x);</b>					
<b>splice(iterator pos,</b> <b>this_type&amp; x,</b> <b>iterator i);</b>					
<b>splice(iterator pos,</b> <b>this_type&amp; x,</b> <b>iterator first,</b> <b>iterator last);</b>					
<b>merge(this_type&amp; x);</b>					
<b>merge(this_type&amp; x, Compare);</b>					
<b>unique();</b>					
<b>unique(BinaryPredicate);</b>					
<b>sort();</b>					
<b>sort(Compare compare);</b>					
<b>bool validate() const;</b>					
<b>int validate_iterator(const_iterator i) const;</b>					
<b>Global Functions &amp; Operators (+ Containers Common)</b>					
<b>void swap(intrusive_list&lt;T&gt; &amp; a,</b> <b>intrusive_list&lt;...&gt; &amp; b);</b>					
<b>template &lt;T, Allocator = eastl::allocator&gt;</b>					
<b>class slist</b>					
<b>Public Types</b>					
<b>value_type,</b> <b>pointer,</b> <b>reference,</b> <b>iterator,</b> <b>reverse_iterator,</b> <b>const_pointer,</b> <b>const_reference,</b> <b>const_iterator,</b> <b>const_reverse_iterator</b>					
<b>ListNode&lt;T&gt;</b>					
<b>node_type;</b>					
<b>Public Member Functions</b>					
<b>intrusive_list();</b>					
<b>intrusive_list(const this_type&amp; x);</b>					
<b>operator=(const</b> <b>this_type&amp; x);</b>					
<b>swap(this_type&amp; x);</b>					
<b>begin();</b>					
<b>const_iterator begin() const;</b>					
<b>end();</b>					
<b>const_iterator end() const;</b>					
<b>reverse_iterator rbegin();</b>					
<b>const_reverse_iterator rbegin() const;</b>					
<b>rend();</b>					
<b>const_reverse_iterator rend() const;</b>					
<b>iterator insert(iterator pos);</b>					
<b>iterator insert(iterator pos,</b> <b>const value_type&amp;);</b>					
<b>void insert(iterator pos,</b> <b>this_type&amp; this_type&amp;);</b>					
<b>void swap(this_type&amp; x);</b>					
<b>iterator begin();</b>					
<b>const_iterator begin() const;</b>					
<b>iterator end();</b>					
<b>const_iterator end() const;</b>					
<b>reverse_iterator rbegin();</b>					
<b>const_reverse_iterator rbegin() const;</b>					
<b>rend();</b>					
<b>const_reverse_iterator rend() const;</b>					
<b>iterator erase(iterator pos);</b>					
<b>iterator erase(iterator first,</b> <b>iterator last);</b>					
<b>void clear();</b>					
<b>Public Member Functions</b>					
<b>reset();</b>					
<b>remove(const T&amp; x);</b>					
<b>remove_if(Predicate);</b>					
<b>reverse();</b>					
<b>splice(iterator pos,</b> <b>this_type&amp; x);</b>					
<b>splice(iterator pos,</b> <b>this_type&amp; x,</b> <b>iterator i);</b>					
<b>splice(iterator pos,</b> <b>this_type&amp; x,</b> <b>iterator first,</b> <b>iterator last);</b>					
<b>merge(this_type&amp; x);</b>					
<b>merge(this_type&amp; x, Compare);</b>					
<b>unique();</b>					
<b>unique(BinaryPredicate);</b>					
<b>sort();</b>					
<b>sort(Compare compare);</b>					
<b>bool validate() const;</b>					
<b>int validate_iterator(const_iterator i) const;</b>					
<b>Global Functions &amp; Operators (+ Containers Common)</b>					
<b>void swap(slist&lt;...&gt; &amp; a,</b> <b>slist&lt;...&gt; &amp; b);</b>					
<b>template &lt;T, size_t nodeCount,</b> <b>bool bEnableOverflow = true,</b> <b>Allocator = eastl::allocator&gt;</b>					
<b>class fixed_slist</b>					
<b>:slist&lt;T, fixed_node_pool&lt;...&gt; &gt;</b>					
<b>Has all the slist functionality.</b>					
<b>Public Member Functions</b>					
<b>fixed_slist();</b>					
<b>fixed_slist(size_type n);</b>					
<b>fixed_slist(size_type n,</b> <b>const value_type&amp; value);</b>					
<b>fixed_slist(const this_type&amp; x);</b>					
<b>fixed_slist (InputIterator first,</b> <b>InputIterator last);</b>					
<b>size_type max_size() const;</b>					
<b>bool full() const;</b>					
<b>template &lt;Key, T, Compare = less&lt;Key&gt;, Allocator = eastl::allocator&gt;</b>					
<b>class map</b>					
<b>: public rbtree&lt;.., bMutableIterators = true,</b> <b>bUniqueKeys = true&gt;</b>					
<b>Public Types</b>					
<b>Key key_type;</b>					
<b>pair&lt;const Key, T&gt; value_type;</b>					
<b>rbtree_node&lt;value_type&gt; node_type;</b>					
<b>value_type&amp; reference;</b>					
<b>const_value_type&amp; const_reference;</b>					
<b>rbtree_iterator&lt;value_type, value_type*, value_type*&gt; iterator;</b>					
<b>rbtree_iterator&lt;value_type, const_value_type*, const_value_type*&gt; const_iterator;</b>					
<b>reverse_iterator, const_reverse_iterator</b>					
<b>Compare key_compare;</b>					
<b>pair&lt;iterator, bool&gt; insert_return_type;</b>					
<b>T mapped_type;</b>					
<b>Public Member Functions</b>					
<b>map(const allocator_type&amp; allocator =</b> <b>eastl::allocator("EASTL classname"));</b>					

**map**(const Compare& compare,  
const allocator\_type& allocator =  
eastl::allocator("EASTL classname");  
**map**(const this\_type& x);  
**map**(Iterator itBegin,  
Iterator itEnd);  
allocator\_type&  
void **get\_allocator**();  
**set\_allocator**(  
const allocator\_type&);  
this\_type&  
void **operator=(const this\_type&);**  
**swap**(this\_type& x);  
iterator  
const\_iterator  
iterator  
const\_iterator  
reverse\_iterator  
const\_reverse\_iterator  
reverse\_iterator  
const\_reverse\_iterator  
bool  
size\_type  
empty() const;  
size() const;  
insert\_return\_type **insert**(const Key& key);  
insert\_return\_type **insert**(const value\_type&);  
iterator **insert**(iterator pos,  
const value\_type&  
value);  
void **insert**(InputIterator first,  
InputIterator last);  
Iterator  
Iterator  
void  
size\_type  
erase(iterator pos);  
erase(iterator first,  
iterator last);  
erase(const key\_type& first,  
const key\_type& last);  
erase(const Key& k);  
void  
void  
iterator  
const\_iterator  
iterator  
const\_iterator  
const\_iterator  
iterator  
const\_iterator  
const\_iterator  
size\_type  
count(const Key& k) const;  
pair<iterator, iterator>  
equal\_range(const Key& k);  
pair<const\_iterator, const\_iterator>  
equal\_range(const Key& k) const;  
T&  
operator[](const Key& key);  
bool validate() const;  
int validate\_iterator(const\_iterator i) const;  
Global Functions & Operators (+ Containers Common)  
void **swap**(rbtree<...> a,  
rbtree<...> b);  
template <Key, T, size\_t nodeCount, bool  
bEnableOverflow = true, Compare = less<Key>,  
Allocator = eastl::allocator  
**class fixed\_map**  
:map<Key, T, Compare, fixed\_node\_pool<...> >

Has all the **map** functionality.  
Public Member Functions

**fixed\_map**();  
**fixed\_map**(const Compare& compare);  
**fixed\_map**(const this\_type& x);  
**fixed\_map**(InputIterator first,  
InputIterator last);  
size\_type **max\_size**() const;  
template <Key, T, Hash = hash<Key>, Predicate =  
equal\_to<Key>, Allocator = eastl::allocator, bool  
bCacheHashCode = false>  
**class hash\_map**  
: public hashtable<..., bMutableIterators=true,  
bUniqueKeys=true>  
Public Types

Key  
pair<const Key, T>  
hash\_node<value\_type, bCacheHashCode=false>  
pair<iterator, bool>  
value\_type&  
const value\_type&  
node\_type;  
insert\_return\_type;  
reference;  
const reference;  
node\_iterator<bConst=false,...>  
local\_iterator;  
node\_iterator<bConst=true,...>  
const\_local\_iterator;  
hashtable\_iterator<bConst=false,...>  
iterator;  
hashtable\_iterator<bConst=true,...>  
const\_iterator;

T  
mapped\_type;

hash\_map(const allocator\_type& allocator =  
eastl::allocator("EASTL classname")  
hash\_map(size\_type nBucketCount,  
const Hash& hashFunction = Hash(),  
const Predicate& predicate =  
Predicate(),  
const allocator\_type& allocator =  
eastl::allocator("EASTL classname")  
hash\_map(ForwardIterator first,  
ForwardIterator last,  
size\_type nBucketCount = 0,  
const Hash& hashFunction = Hash(),  
const Predicate& predicate =  
Predicate(),  
const allocator\_type& allocator =  
eastl::allocator("EASTL classname")  
allocator\_type&  
void **get\_allocator**();  
**set\_allocator**(  
const allocator\_type&);  
this\_type&  
void **operator=(const this\_type&);**  
**swap**(this\_type& x);  
iterator  
const\_iterator  
iterator  
const\_iterator  
const\_iterator  
local\_iterator  
local\_iterator  
const\_local\_iterator  
const\_local\_iterator  
size\_type  
bucket\_count() const  
size\_type bucket\_size(size\_type n) const

float **load\_factor**() const;  
float **get\_max\_load\_factor**() const;  
void **set\_max\_load\_factor**(float fMaxLoadFactor);  
const rehash\_policy\_type&  
rehash\_policy() const  
void **rehash\_policy**(const rehash\_policy\_type&  
rehashPolicy);  
insert\_return\_type **insert**(const key\_type& key);  
insert\_return\_type **insert**(const value\_type&  
value);  
iterator **insert**(const\_iterator,  
const value\_type&  
value);  
void **insert**(InputIterator first,  
InputIterator last);  
iterator  
iterator  
size\_type  
void **erase**(iterator);  
**erase**(iterator, iterator);  
**erase**(const key\_type&);  
clear();  
reset();  
void **rehash**(size\_type nBucketCount);  
find(const key\_type& k);  
find(const key\_type& k) const;  
find\_as(const U8,  
UHash,  
BinaryPredicate);  
find\_as(const U8,  
UHash,  
BinaryPredicate) const;  
find\_as(const U8 u);  
find\_as(const U8 u) const;  
find\_by\_hash(hash\_code\_t);  
find\_by\_hash(hash\_code\_t) const;  
size\_type **count**(const Key& k) const;  
pair<iterator, iterator>  
equal\_range(const Key& k);  
pair<const\_iterator, const\_iterator>  
equal\_range(const Key& k) const;  
T&  
operator[](const Key& key);  
Hash  
Predicate  
hash\_function() const;  
equal\_function();  
bool **validate**() const;  
int validate\_iterator(const\_iterator i) const;  
Global Functions & Operators (+ Containers Common)  
void **swap**(hashtable<...> a,  
hashtable<...> b);  
template <Key, T, size\_t nodeCount, bucketCount =  
nodeCount + 1, bool bEnableOverflow = true, Hash =  
hash<Key>, Predicate = equal\_to<Key>, bool  
bCacheHashCode = false, Allocator =  
eastl::allocator>  
**class fixed\_hash\_map**  
: hash\_map<Key, Value=T, Hash, Predicate,  
fixed\_hashtable\_allocator<...>, bCacheHashCode>  
Has all the **hash\_map** functionality.  
Public Member Functions

fixed\_hash\_map(  
const Hash& hashFunction = Hash(),  
const Predicate& predicate = Predicate());  
fixed\_hash\_map(  
InputIterator first,  
InputIterator last,

const Hash& hashFunction = Hash(),  
const Predicate& predicate = Predicate());  
size\_type **max\_size**() const;  
template <Key, T, Compare = less<Key>, Allocator =  
eastl::allocator, RandomAccessContainer =  
vector<pair<Key, T>, Allocator> >  
**class vector\_map**  
: public RandomAccessContainer  
Public Types

Key  
pair<Key, T>  
Compare  
map\_value\_compare<Key, value\_type, Compare>  
value\_type&  
const value\_type&  
reference;  
const reference;  
T  
mapped\_type;

pointer,  
iterator,  
reverse\_iterator,  
const\_reverse\_iterator;

Public Member Functions

vector\_map();  
vector\_map(const allocator\_type& allocator);  
vector\_map(const key\_compare& comp,  
const allocator\_type& allocator =  
eastl::allocator("EASTL  
classname"));  
vector\_map(const vector\_map& x);  
vector\_map(InputIterator first,  
InputIterator last);  
vector\_map(InputIterator first,  
InputIterator last,  
const key\_compare& compare);  
vector\_map & **operator=(const vector\_map& x);**  
key\_compare **key\_comp**() const;  
value\_compare **value\_comp**() const;  
pair<iterator, bool>  
insert(const value\_type& value);  
iterator **insert**(iterator pos,  
const value\_type& value);  
void **insert**(InputIterator first,  
InputIterator last);  
iterator **erase**(iterator pos);  
iterator **erase**(iterator first,  
iterator last);  
size\_type **erase**(const key\_type& k);  
iterator **find**(const key\_type& k);  
const\_iterator **find**(const key\_type& k) const;  
iterator **find\_as**(const U8,  
UHash,  
BinaryPredicate);  
const\_iterator **find\_as**(const U8,  
UHash,  
BinaryPredicate) const;  
iterator **find\_as**(const U8 u);  
const\_iterator **find\_as**(const U8 u) const;  
iterator **find\_by\_hash**(hash\_code\_t);  
const\_iterator **find\_by\_hash**(hash\_code\_t) const;  
size\_type **count**(const Key& k) const;  
pair<iterator, iterator>  
equal\_range(const Key& k);  
pair<const\_iterator, const\_iterator>  
equal\_range(const Key& k) const;  
T&  
operator[](const Key& key);  
Hash  
Predicate  
hash\_function() const;  
equal\_function();  
bool **validate**() const;  
int validate\_iterator(const\_iterator i) const;  
Global Functions & Operators (+ Containers Common)  
void **swap**(hashtable<...> a,  
hashtable<...> b);  
template <Key, T, size\_type nodeCount, bucketCount =  
nodeCount + 1, bool bEnableOverflow = true, Hash =  
hash<Key>, Predicate = equal\_to<Key>, bool  
bCacheHashCode = false, Allocator =  
eastl::allocator>  
**class intrusive\_hash\_map**  
: intrusive\_hashtable<Key, Value=T, Hash, Equal,  
bucketCount, bConstitutors=false,  
bUniqueKeys=true>  
Public Types

Key  
Value  
Value  
pair<iterator, bool>  
insert\_return\_type;  
value\_type&  
const value\_type&  
reference;  
const reference;  
intrusive\_node\_iterator<value\_type, bConst=false>  
local\_iterator;  
intrusive\_node\_iterator<value\_type, bConst=true>  
const\_local\_iterator;  
intrusive\_hashtable\_iterator<value\_type,  
bConst=false>  
iterator;  
intrusive\_hashtable\_iterator<value\_type,  
bConst=true>  
const\_iterator;  
Value  
use\_intrusive\_key<Value, Key>  
mapped\_type;  
extract\_key();  
Public Member Functions

intrusive\_hash\_map(const Hash& h = Hash(),  
const Equal& eq = Equal());  
void **swap**(this\_type& x);  
iterator  
const\_iterator  
iterator  
const\_iterator  
const\_iterator  
local\_iterator  
local\_iterator  
const\_local\_iterator  
const  
const\_local\_iterator  
size\_type **count**(const key\_type& k);  
iterator **lower\_bound**(const key\_type&);  
const\_iterator **lower\_bound**(const key\_type&) const;  
iterator **upper\_bound**(const key\_type&);  
const\_iterator **upper\_bound**(const key\_type&) const;  
pair<iterator, iterator>  
equal\_range(const key\_type&);  
pair<const\_iterator, const iterator>  
equal\_range(const key\_type&) const;

mapped\_type& **operator[]**(const key\_type&);  
Inherited from base class, RandomAccessContainer  
allocator\_type&  
void **get\_allocator**();  
**set\_allocator**(  
const allocator\_type&);  
begin();  
begin() const;  
end();  
end() const;  
rbegin();  
rbegin() const;  
rend();  
rend() const;  
bool **empty**() const;  
size\_type **size**() const;  
void **clear**();  
Global Functions & Operators (+ Containers Common)  
void **swap**(vector\_map<...> a,  
vector\_map<...> b);  
template <Key, T, size\_t bucketCount, Hash =  
hash<Key>, Equal = equal\_to<Key> >

```

insert_return_type insert(value_type& value);
insert_return_type insert(const_iterator,
    value_type& value);
void insert(InputIterator first,
    InputIterator last);

iterator erase(iterator);
iterator erase(iterator, iterator);
iterator erase(const key_type&);

void clear();

iterator find(const key_type& k);
find(const key_type& k) const;
find(as(const U8,
    UHash,
    BinaryPredicate));
const_iterator find(as(const U8,
    UHash,
    BinaryPredicate) const;
iterator find_as(const U8 u);
const_iterator find_as(const U8 u) const;

size_type count(const key_type& k) const;
pair<iterator, iterator>
equal_range(const key_type&);
pair<const_iterator, const_iterator>
equal_range(const key_type&) const;

bool validate() const;
int validate_iterator(const_iterator i) const;
Global Functions & Operators (+ Containers Common)
void swap(intrusive_hashtable<>& a,
    intrusive_hashtable<>& b);

template <Key, T, Compare = less<Key>,
Allocator = eastl::allocator>
class multimap
: public rbtree<..., bMutableIterators=true,
bUniqueKeys=false>
{
public:
    Key key_type;
    pair<const Key, T> value_type;
    rbtree_node<value_type> node_type;
    value_type& reference;
    const value_type& const_reference;
    rbtree_iterator<value_type,
        value_type*,
        value_type&> iterator;
    rbtree_iterator<value_type,
        const_value_type*,
        const value_type&> const_iterator;
    reverse_iterator, const_reverse_iterator
};

Compare iterator key_compare;
insert_return_type;
T mapped_type;
};

Public Member Functions
multimap(const allocator_type& allocator =
eastl::allocator("EASTL classname"));
multimap(const Compare& compare,
    const allocator_type& allocator =
eastl::allocator("EASTL classname"));
multimap(const this_type& x);
multimap(Iterator itBegin,
    Iterator itEnd);

allocator_type& get_allocator();
void set_allocator(
    const allocator_type&);

this_type& operator=(const this_type&);
swap(this_type& x);

iterator begin();
begin() const;
end();
end() const;

const_iterator begin();
begin() const;
end();
end() const;

reverse_iterator rbegin();
rbegin() const;
rend();
rend() const;

const_reverse_iterator rbegin();
rbegin() const;
rend();
rend() const;

size_type max_size() const;
template <Key, T, Hash = hash<Key>, Predicate =
equal_to<Key>, Allocator = eastl::allocator, bool
bCacheHashCode = false>
class hash_multimap
: public hashtable<..., bMutableIterators=true,
bUniqueKeys=false>
{
public:
    Key key_type;
    pair<const Key, T> value_type;
    hash_node<value_type, bCacheHashCode> node_type;
    iterator insert_return_type;
    value_type& const_value_type& const_reference;
    node_iterator<bConst=false,...> local_iterator;
    node_iterator<bConst=true,...> const_local_iterator;
    hashtable_iterator<bConst=false,...> iterator;
    hashtable_iterator<bConst=true,...> const_iterator;
    mapped_type;
};

Public Member Functions
hash_multimap(const allocator_type& allocator =
eastl::allocator("EASTL classname"));
hash_multimap(
    size_type nBucketCount,
    const Hash& hashFunction = Hash(),
    const Predicate& predicate = Predicate(),
    const allocator_type& allocator =
eastl::allocator("EASTL classname"))
hash_multimap(
    ForwardIterator first,
    ForwardIterator last,
    size_type nBucketCount = 0,
    const Hash& hashFunction = Hash(),
    const Predicate& predicate = Predicate(),
    const allocator_type& allocator =
eastl::allocator("EASTL classname"));

operator[](const Key& key);
Hash hash_function() const;
equal_function();

bool validate() const;
int validate_iterator(const_iterator i) const;
Global Functions & Operators (+ Containers Common)
void swap(hashtable<>& a,
    hashtable<>& b);

template <Key, T, Compare = less<Key>, Allocator =
eastl::allocator, RandomAccessContainer =
vector<pair<Key, T>, Allocator> >
class fixed_multimap
: multimap<Key, T, Compare, fixed_node_pool<...> >
{
public:
    Has all the multimap functionality.
    Public Member Functions
    fixed_multimap();
    fixed_multimap(const Compare& compare);
    fixed_multimap(const this_type& x);
};

fixed_multimap(InputIterator first,
    InputIterator last);
size_type max_size() const;
template <Key, T, Hash = hash<Key>, Predicate =
equal_to<Key>, Allocator = eastl::allocator, bool
bCacheHashCode = false>
class vector_multimap(const vector multimap& x);
vector_multimap(InputIterator first,
    InputIterator last);
vector_multimap(InputIterator first,
    InputIterator last,
    const key_compare& compare);
vector_multimap& operator=(
    const vector multimap&);
void swap(this_type& x);

key_compare key_comp() const;
value_compare value_comp() const;
pair<iterator, bool>
insert(const value_type& value);
iterator insert(iterator pos,
    const value_type& value);
void insert(InputIterator first,
    InputIterator last);

erase(iterator);
erase(iterator, iterator);
erase(const key_type&);

clear();
reset();
rehash(size_type nBucketCount);

find(const key_type& k);
find(const key_type& k) const;
find(as(const U8,
    UHash,
    BinaryPredicate));
find(as(const U8,
    UHash,
    BinaryPredicate) const;
find_as(const U8 u);
find_as(const U8 u) const;

size_type erase(const key_type& k);

find_by_hash(hash_code_t);
find_by_hash(hash_code_t) const;

count(const Key& k) const;
pair<iterator, iterator>
equal_range(const Key& k);
pair<const_iterator, const_iterator>
equal_range(const Key& k) const;
T& operator[](const Key& key);
Hash hash_function() const;
equal_function();

bool validate() const;
int validate_iterator(const_iterator i) const;
Global Functions & Operators (+ Containers Common)
void swap(hashtable<>& a,
    hashtable<>& b);

template <Key, T, Compare = less<Key>, Allocator =
eastl::allocator, RandomAccessContainer =
vector<pair<Key, T>, Allocator> >
class vector_multimap
: public RandomAccessContainer
{
public:
    Key key_type;
    pair<const Key, T> value_type;
    Compare compare;
    multimap_value_compare<Key, value_type, Compare> value_compare;
    value_type& const_value_type& const_reference;
    mapped_type;
};

Public Member Functions
vector_multimap();
vector_multimap(const allocator_type& allocator);
vector_multimap(const key_compare& comp,
    const allocator_type& allocator =
eastl::allocator("EASTL classname"));

```

<b>template &lt;Key, T, size_t bucketCount, Hash = hash&lt;Key&gt;, Equal = equal_to&lt;Key&gt; &gt;</b>	<b>size_type</b> <b>count(const key_type&amp; k) const;</b>	<b>Iterator</b> <b>erase(iterator pos);</b>	<b>Public Member Functions</b>	<b>pair&lt;iterator, iterator&gt;</b> <b>equal_range(const Value&amp;);</b>
<b>class intrusive_hash_multimap</b>	<b>pair&lt;iterator, iterator&gt;</b> <b>equal_range(const key_type&amp;);</b>	<b>Iterator</b> <b>erase(iterator first, iterator last);</b>	<b>hash_set(const allocator_type&amp; allocator = eastl::allocator("EASTL classname")</b>	<b>pair&lt;const_iterator, const_iterator&gt;</b> <b>equal_range(const Value&amp;) const;</b>
<b>: intrusive_hashtable&lt;Key, Value, Hash, Equal, bucketCount, bConstIterators=false, bUniqueKeys=false&gt;</b>	<b>pair&lt;const_iterator, const_iterator&gt;</b> <b>equal_range(const key_type&amp;) const;</b>	<b>void</b> <b>erase(const key_type* first, const key_type* last);</b>	<b>hash_set(size_type nBucketCount, const Hash&amp; hashFunction = Hash(), const Predicate&amp; predicate = Predicate(), const allocator_type&amp; allocator = eastl::allocator("EASTL classname")</b>	<b>Hash</b> <b>hash_function() const;</b>
<b>Public Types</b>	<b>bool validate() const;</b>	<b>size_type</b> <b>erase(const key_type&amp; k);</b>	<b>hash_set(ForwardIterator first, ForwardIterator last, size_type nBucketCount = 0, const Hash&amp; hashFunction = Hash(), const Predicate&amp; predicate = Predicate(), const allocator_type&amp; allocator = eastl::allocator("EASTL classname")</b>	<b>Predicate</b> <b>equal_function();</b>
<b>Key</b> <b>key_type;</b>	<b>int validate_iterator(const_iterator i) const;</b>	<b>void</b> <b>clear();</b>		
<b>Value</b> <b>value_type;</b>	<b>Global Functions &amp; Operators (+ Containers Common)</b>	<b>void</b> <b>reset();</b>		
<b>iterator</b> <b>node_type;</b>	<b>void swap(intrusive_hashtable&lt;...&gt;&amp; a, intrusive_hashtable&lt;...&gt;&amp; b);</b>	<b>iterator</b> <b>find(const key_type&amp; key);</b>		
<b>value_type&amp;</b>		<b>const_iterator</b> <b>find(const key_type&amp; key) const;</b>		
<b>const value_type&amp;</b>		<b>iterator</b> <b>find_as(const U&amp; u, Compare2 compare2);</b>		
<b>intrusive_node_iterator&lt;value_type, bConst=false&gt;</b>		<b>const_iterator</b> <b>find_as(const U&amp; u, Compare2 compare2) const;</b>		
<b>local_iterator;</b>		<b>iterator</b> <b>lower_bound(const key_type&amp;);</b>		
<b>intrusive_node_iterator&lt;value_type, bConst=true&gt;</b>		<b>const_iterator</b> <b>upper_bound(const key_type&amp;);</b>		
<b>const_local_iterator;</b>		<b>size_type</b> <b>count(const Key&amp; k) const;</b>		
<b>intrusive_hashtable_iterator&lt;value_type, bConst=false&gt;</b>		<b>pair&lt;iterator, iterator&gt;</b> <b>operator=(const this_type&amp;);</b>		
<b>intrusive_hashtable_iterator&lt;value_type, bConst=true&gt;</b>		<b>pair&lt;const_iterator, const_iterator&gt;</b> <b>swap(this_type&amp; x);</b>		
<b>Value</b> <b>mapped_type;</b>		<b>iterator</b> <b>begin();</b>		
<b>use_intrusive_key&lt;Value, Key&gt;</b> <b>extract_key();</b>		<b>const_iterator</b> <b>begin() const;</b>		
<b>Public Member Functions</b>		<b>iterator</b> <b>end();</b>		
<b>intrusive_hash_multimap(const Hash&amp; h = Hash(), const Equal&amp; eq = Equal());</b>		<b>const_iterator</b> <b>end() const;</b>		
<b>void</b>	<b>swap(this_type&amp; x);</b>	<b>local_iterator</b> <b>begin(size_type n);</b>		
<b>iterator</b>	<b>begin();</b>	<b>local_iterator</b> <b>end(size_type);</b>		
<b>const_iterator</b>	<b>begin() const;</b>	<b>const_local_iterator</b> <b>begin(size_type n) const;</b>		
<b>iterator</b>	<b>end();</b>	<b>const_local_iterator</b> <b>end(size_type) const;</b>		
<b>const_iterator</b>	<b>end() const;</b>	<b>bool</b> <b>empty() const;</b>		
<b>local_iterator</b>	<b>begin(size_type n)</b>	<b>size_type</b> <b>size() const;</b>		
<b>local_iterator</b>	<b>end(size_type);</b>	<b>size_type</b> <b>bucket_count() const;</b>		
<b>const_local_iterator</b>	<b>begin(size_type n)</b>	<b>size_type</b> <b>bucket_size(size_type n) const;</b>		
<b>const</b>		<b>float</b> <b>load_factor() const;</b>		
<b>const_local_iterator</b>	<b>end(size_type) const</b>	<b>float</b> <b>get_max_load_factor() const;</b>		
<b>bool</b>	<b>empty() const;</b>	<b>void</b> <b>set_max_load_factor(float fMaxLoadFactor);</b>		
<b>size_type</b>	<b>size() const;</b>			
<b>size_type</b>	<b>bucket_count() const;</b>	<b>const rehash_policy_type&amp;</b>		
<b>size_type</b>	<b>bucket_size(size_type n)</b>	<b>fixed_set();</b>		
<b>const;</b>		<b>fixed_set(const Compare&amp; compare);</b>		
<b>size_type</b>	<b>bucket(const key_type&amp; k)</b>	<b>fixed_set(const this_type&amp; x);</b>		
<b>const;</b>		<b>fixed_set(InputIterator first, InputIterator last);</b>		
<b>float</b>	<b>load_factor() const;</b>	<b>size_type</b> <b>max_size() const;</b>		
<b>insert_return_type</b> <b>insert(value_type&amp; value);</b>		<b>template &lt;Value, Hash = hash&lt;Value&gt;, Predicate = equal_to&lt;Value&gt;, Allocator = eastl::allocator, bool bCacheHashCode = false&gt;</b>		
<b>insert_return_type</b> <b>insert(const_iterator, value_type&amp; value);</b>	<b>this_type&amp; void</b> <b>operator=(const this_type&amp;);</b>			
<b>void</b>	<b>insert(InputIterator first, InputIterator last);</b>			
<b>iterator</b>	<b>erase(iterator);</b>	<b>pair&lt;iterator, iterator&gt;</b> <b>insert(const value_type&amp;);</b>		
<b>iterator</b>	<b>erase(iterator, iterator);</b>	<b>iterator</b> <b>insert(const_iterator, const_value_type&amp;);</b>		
<b>size_type</b>	<b>erase(const key_type&amp;);</b>	<b>const_iterator</b> <b>insert(const key_type&amp; k);</b>		
<b>void</b>	<b>clear();</b>	<b>iterator</b> <b>find(const key_type&amp; k) const;</b>		
<b>iterator</b>		<b>const_iterator</b> <b>find_as(const U&amp;, UHash, BinaryPredicate);</b>		
<b>const_iterator</b>	<b>find(const key_type&amp; k);</b>	<b>const_iterator</b> <b>find_as(const U&amp;, UHash, BinaryPredicate) const;</b>		
<b>iterator</b>	<b>find(const key_type&amp; k) const;</b>	<b>iterator</b> <b>find_as(const U&amp; u);</b>		
<b>const_iterator</b>	<b>find_as(const U&amp;, UHash, BinaryPredicate);</b>	<b>const_iterator</b> <b>find_as(const U&amp; u) const;</b>		
<b>const_iterator</b>	<b>find_as(const U&amp;, UHash, BinaryPredicate) const;</b>	<b>iterator</b> <b>find_by_hash(hash_code_t);</b>		
<b>iterator</b>	<b>find_as(const U&amp; u);</b>	<b>const_iterator</b> <b>find_by_hash(hash_code_t) const;</b>		
<b>const_iterator</b>	<b>find_as(const U&amp; u) const;</b>	<b>size_type</b> <b>count(const Value&amp; k) const;</b>		



Public Types		Global Functions & Operators (+ Containers Common)	
Key	key_type;	iterator	find(const key_type& k) const;
Key	value_type;	const_iterator	find_as(const U&, UHash, BinaryPredicate);
Compare	key_compare;	const_iterator	find_as(const U&, UHash, BinaryPredicate) const;
Compare	value_compare;	iterator	find_as(const U& u);
value_type&	reference;	const_iterator	find_as(const U& u) const;
const value_type&	const_reference;	size_type	count(const key_type& k) const;
pointer,	const_pointer,	pair<iterator, iterator>	equal_range(const key_type&);
iterator,	const_iterator,	pair<const_iterator, const_iterator>	equal_range(const key_type&) const;
reverse_iterator,	const_reverse_iterator;	bool	validate() const;
Public Member Functions		int	validate_iterator(const_iterator i) const;
vector_multiset();		Global Functions & Operators (+ Containers Common)	
vector_multiset(const allocator_type& allocator);		void	swap(vector_multiset...& a, vector_multiset...& b);
vector_multiset(const key_compare& comp, const allocator_type& allocator = eastl::allocator("EASTL classname");		template <T, size_t bucketCount, Hash = hash<T>, Equal = equal_to<T> >	void swap(intrusive_hashtable...& a, intrusive_hashtable...& b);
vector_multiset(const vector_multiset& x);		class intrusive_hash_multiset	
vector_multiset(InputIterator first, InputIterator last);		: intrusive_hashtable<Key=T, Value=T, Hash, Equal, bucketCount, bConstIterators=true, bUniqueKeys=false>	Public Types
vector_multiset(InputIterator first, InputIterator last, const key_compare& compare);		T	key_type;
vector_multiset operator=(const vector_multiset&);		Value	value_type;
void swap(this_type& x);		iterator	node_type;
key_compare key_comp() const;		value_type&	insert_return_type;
value_compare value_comp() const;		const value_type&	reference;
pair<iterator, bool> insert(const value_type& value);		intrusive_node_iterator<value_type, bConst=true>	const_reference;
iterator insert(iterator pos, const value_type& value);		local_iterator,	
void insert(InputIterator first, InputIterator last);		const_local_iterator;	
iterator erase(iterator pos);		intrusive_hashtable_iterator<value_type, bConst=true>	iterator;
iterator erase(iterator first, iterator last);		iterator,	const_iterator;
size_type erase(const key_type& k);		Value	mapped_type;
iterator find(const key_type& k);		use_self<Value>	extract_key;
const_iterator find(const key_type& k) const;		Public Member Functions	
iterator find_as(const U& u, BinaryPredicate predicate);		intrusive_hash_set(const Hash& h = Hash(), const Equal& eq = Equal());	
const_iterator find_as(const U& u, BinaryPredicate predicate) const;		void	swap(this_type& x);
size_type count(const key_type& k);		iterator	begin();
iterator lower_bound(const key_type&);		const_iterator	begin() const;
const_iterator lower_bound(const key_type&) const;		iterator	end();
iterator upper_bound(const key_type&);		const_iterator	end() const;
const_iterator upper_bound(const key_type&) const;		local_iterator	begin(size_type n)
pair<iterator, iterator> equal_range(const key_type&);		local_iterator	end(size_type)
pair<const_iterator, const_iterator> equal_range(const key_type&) const;		const_local_iterator	begin(size_type n) const
pair<const_iterator, const_iterator> equal_range_small(const key_type&) const;		const_local_iterator	end(size_type) const
pair<iterator, iterator> equal_range_small(const key_type&)		size_type	empty() const;
Inherited from base class, RandomAccessContainer		size_type	size() const;
allocator_type& get_allocator();		size_type	bucket_count() const;
void set_allocator(const allocator_type&);		size_type	bucket_size(size_type n) const;
iterator begin();		size_type	bucket(const key_type& k) const;
const_iterator begin() const;		float	load_factor() const;
		insert_return_type	insert(value_type& value);
		insert_return_type	insert(const_iterator, value_type& value);
		void	insert(InputIterator first, InputIterator last);
		iterator	erase(iterator);
		iterator	erase(iterator, iterator);
		size_type	erase(const key_type&);
		void	clear();
		iterator	find(const key_type& k);

```

void insert(iterator pos, size_type n, const value_type&); void insert(iterator pos, InputIterator first, InputIterator last);
iterator erase(iterator pos); iterator erase(iterator first, iterator last);
void clear(); void reset();
bool validate() const; int validate_iterator(const_iterator i) const;
Global Functions & Operators (+ Containers Common)
void swap(vector<...>& a, vector<...>& b);

template <T, size_t nodeCount, bool bEnabledOverflow = true, Allocator = eastl::allocator>
class fixed_vector :vector<T, fixed_vector_allocator...> {
    Has all the vector functionality.
    Public Member Functions
fixed_vector();
fixed_vector(size_type n);
fixed_vector(size_type n, const value_type& value);
fixed_vector(const this_type& x);
fixed_vector(InputIterator first, InputIterator last);
void set_capacity(size_type n);
size_type max_size() const;
bool full() const;
Global Functions & Operators (+ Containers Common)
void swap (fixed_vector<...>& a, fixed_vector<...>& b);

```

## 2. Algorithms

	typename T Compare	Meaning
	Predicate	A function which takes two arguments and returns the lesser of the two.
	BinaryPredicate	A function which takes one argument returns true if the argument meets some criteria.
	StrictWeakOrdering	A function which takes two arguments and returns true if some criteria is met (e.g. they are equal).
	Function	A StrictWeakOrdering that compares two objects, returning true if the first precedes the second. Like Compare but has additional requirements. Used for sorting routines.
	SizeGenerator	A function which takes one argument and applies some operation to the target.
	UnaryOperation	A count or size.
	BinaryOperation	A function which takes no arguments and returns a value (which will usually be assigned to an object).
	InputIterator	A function which takes one argument and returns a value (which will usually be assigned to second object).
	ForwardIterator	A function which takes two arguments and returns a value (which will usually be assigned to a third object).
	BidirectionalIterator	An input iterator (iterator you read from) which allows reading each element only once and only in a forward direction.
	RandomAccessIterator	An input iterator which is like InputIterator except it can be reset back to the beginning.
	OutputIterator	An input iterator which is like ForwardIterator except it can be read in a backward direction as well.
		An output iterator (iterator you write to) which allows writing each element only once in only in a forward direction.
		A function which takes an InputIterator will also work with a ForwardIterator, BidirectionalIterator, or RandomAccessIterator.
		The given iterator type is merely the <i>minimum</i> supported functionality the iterator must support.

Query Algorithms	ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate);	const T& median(const T& a, const T& b, const T& c);
	ForwardIterator1 adjacent_find(ForwardIterator first, ForwardIterator last);	const T& median(const T& a, const T& b, const T& c);
	ForwardIterator1 adjacent_find(ForwardIterator first, ForwardIterator last, BinaryPredicate);	pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);
	bool binary_search(ForwardIterator first, ForwardIterator last, const T& value);	pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate);
	bool binary_search(ForwardIterator first, ForwardIterator last, const T& value, Compare compare);	ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);
	ForwardIterator1 binary_search_i(ForwardIterator first, ForwardIterator last, const T& value);	ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate);
	ForwardIterator1 binary_search_i(ForwardIterator first, ForwardIterator last, const T& value, Compare compare);	ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);
	difference_type count(InputIterator first, InputIterator last, const T& value);	ForwardIterator1 find_first_not_of(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);
	difference_type count_if(InputIterator first, InputIterator last, Predicate predicate);	ForwardIterator1 find_first_not_of(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate);
	bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);	BidirectionalIterator1 find_last_of(BidirectionalIterator1 first1, BidirectionalIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);
	bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate);	BidirectionalIterator1 find_last_of(BidirectionalIterator1 first1, BidirectionalIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate);
	pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, const T& value);	BidirectionalIterator1 find_last_not_of(BidirectionalIterator1 first1, BidirectionalIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);
	pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, const T& value, Compare compare);	BidirectionalIterator1 find_last_not_of(BidirectionalIterator1 first1, BidirectionalIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate);
	Function for_each(InputIterator first, InputIterator last, Function function);	bool identical(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);
	InputIterator find(InputIterator first, InputIterator last, const T& value);	bool identical(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, BinaryPredicate predicate);
	InputIterator find_if(InputIterator first, InputIterator last, Predicate predicate);	ForwardIterator1 lower_bound(ForwardIterator first, ForwardIterator last, const T& val);
	ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);	ForwardIterator1 lower_bound(ForwardIterator first, ForwardIterator last, const T& val, Compare);
		ForwardIterator1 upper_bound(ForwardIterator first, ForwardIterator last, const T& val);
		ForwardIterator1 upper_bound(ForwardIterator first, ForwardIterator last, const T& val, Compare);
		void radix_sort(RandomAccessIterator first, RandomAccessIterator last, RandomAccessIterator buffer);
		void comb_sort(ForwardIterator first, ForwardIterator last, StrictWeakOrdering compare);
		void comb_sort(ForwardIterator first, ForwardIterator last, StrictWeakOrdering compare);



### 3. Function Objects

<pre>template &lt;Argument, Result&gt; struct unary_function Argument          argument_type; Result           result_type;</pre>	<pre>template &lt;Predicate&gt; class unary_negate : unary_function&lt;Predicate::argument_type, bool&gt; Public Member Functions unary_negate(const Predicate&amp; a); bool operator()(const Predicate::argument_type&amp; a) const;</pre>	<pre>pointer_to_unary_function() Result (*pFunction)(Arg); Result operator()(Arg x) const;</pre>	<pre>template &lt;Result, T&gt; mem_fun_t... template &lt;Result, T, Argument&gt; mem_fun1_t... template &lt;Result, T&gt; const_mem_fun_t... template &lt;Result, T, Argument&gt; const_mem_fun1_t...</pre>
<pre>template &lt;Argument1, Argument2, Result&gt; struct binary_function Argument1        first_argument_type; Argument2        second_argument_type; Result          result_type;</pre>	<pre>template &lt;Predicate&gt; unary negate&lt;Predicate&gt; not1 (const Predicate&amp; predicate)</pre>	<pre>class pointer_to_binary_function : public binary_function&lt;Arg1, Arg2, Result&gt; pointer_to_binary_function(); pointer_to_binary_function() Result (*pFunction)(Arg1, Arg2);</pre>	<pre>template &lt;Result, T&gt; class mem_fun_ref_t : unary_function&lt;T, Result&gt; Public Types Result (T::*MemberFunction)();</pre>
<pre>template &lt;T&gt; struct plus, minus, multiplies, divides, modulus : public binary_function&lt;T, T&gt; T operator()(const T&amp; a, const T&amp; b) const;</pre>	<pre>template &lt;Predicate&gt; binary negate&lt;Predicate&gt; not2 (const Predicate&amp; predicate)</pre>	<pre>pointer to unary function&lt;Arg, Result&gt; ptr_fun (Result (*pFunction)(Arg))</pre>	<pre>Public Member Functions mem_fun_ref_t(MemberFunction); Result operator()(T&amp; pT) const;</pre>
<pre>template &lt;T&gt; struct equal_to, not_equal_to, less, greater, less_equal, greater_equal : public binary_function&lt;T, T, bool&gt; T operator()(const T&amp; a, const T&amp; b) const;</pre>	<pre>template &lt;Operation&gt; class binder1st : unary_function&lt;Operation::second_argument_type, Operation::result_type&gt; Public Member Functions binder1st( const Operation&amp; x, const Operation::first_argument_type&amp; y); Operation::result_type operator()( const Operation::second_argument_type&amp; x) const; Operation::result_type operator()( const Operation::second_argument_type&amp; x) const;</pre>	<pre>template &lt;Arg1, Arg2, Result&gt; pointer_to_binary_function&lt;Arg1, Arg2, Result&gt; ptr_fun (Result (*pFunction)(Arg1, Arg2))</pre>	<pre>template &lt;Result, T&gt; class mem_fun1_ref_t : binary_function&lt;T, Argument, Result&gt; Public Types Result (T::*MemberFunction)(Argument);</pre>
<pre>template &lt;T, Compare&gt; bool validate_equal_to, validate_not_equal_to, validate_less, validate_greater, validate_less_equal, validate_greater_equal (const T&amp; a, const T&amp; b, Compare compare)</pre>	<pre>template &lt;Operation&gt; binder1st&lt;Operation&gt; bind1st (const Operation&amp; op, const T&amp; x);</pre>	<pre>template &lt;Result, T, Argument&gt; class mem_fun_t : unary_function&lt;T*, Result&gt; Public Types Result (T::*MemberFunction)();</pre>	<pre>Public Member Functions mem_fun_t(MemberFunction); Result operator()(*pT) const;</pre>
<pre>template &lt;T&gt; struct str_equal_to : public binary_function&lt;T, T, bool&gt; T operator()(const T&amp; a, const T&amp; b) const;</pre>	<pre>template &lt;Operation&gt; class binder2nd : unary_function&lt;Operation::first_argument_type, Operation::result_type&gt; Public Member Functions binder2nd( const Operation&amp; x, const Operation::second_argument_type&amp; y); Operation::result_type operator()( const Operation::first_argument_type&amp; x) const; Operation::result_type operator()( const Operation::first_argument_type&amp; x) const;</pre>	<pre>template &lt;Result, T, Argument&gt; class mem_fun1_t : binary_function&lt;T*, Argument, Result&gt; Public Types Result (T::*MemberFunction)(Argument);</pre>	<pre>Public Member Functions mem_fun1_t(MemberFunction); Result operator()(*pT, Argument arg) const;</pre>
<pre>template &lt;T&gt; struct logical_and, logical_or, logical_not : public binary_function&lt;T, T, bool&gt; T operator()(const T&amp; a, const T&amp; b) const;</pre>	<pre>template &lt;Operation, T&gt; binder2nd&lt;Operation&gt; bind2nd (const Operation&amp; op, const T&amp; x)</pre>	<pre>template &lt;Result, T, Argument&gt; class const_mem_fun_t : unary_function&lt;const T*, Result&gt; Public Types Result (T::*MemberFunction)() const;</pre>	<pre>Public Member Functions const_mem_fun_t(MemberFunction); Result operator()(const T* pT) const;</pre>
<pre>template &lt;T, U&gt; struct equal_to_2, not_equal_to_2, less_2 : public binary_function&lt;T, U&gt;, bool&gt; T operator()(const T&amp; a, const U&amp; b) const;</pre>	<pre>template &lt;Arg, Result&gt; class pointer_to Unary function : public unary_function&lt;Arg, Result&gt; Public Member Functions pointer_to Unary function()</pre>	<pre>template &lt;Result, T, Argument&gt; class const_mem_fun1_t : binary_function&lt;const T*, Argument, Result&gt; Public Types Result (T::*MemberFunction)(Argument) const;</pre>	<pre>Public Member Functions const_mem_fun1_t(MemberFunction); Result operator()(const T* pT, Argument arg) const;</pre>
			<pre>mem_fun_ref (Result (T::*MemberFunction)()) (Result (T::*MemberFunction)(Argument)) (Result (T::*MemberFunction)() const) (Result (T::*MemberFunction)(Argument) const)</pre>

## 4. Iterators

Iterators Categories

```
struct input_iterator_tag
{};
struct output_iterator_tag
{};
struct forward_iterator_tag :
    input_iterator_tag
{};
struct bidirectional_iterator_tag :
    forward_iterator_tag
{};
struct random_access_iterator_tag :
    bidirectional_iterator_tag
{};

template <Category, T, Distance = ptrdiff_t,
Pointer = T*, Reference = T&>
struct iterator
{
    Public Types
    iterator_category, value_type, difference_type,
    pointer, reference;
}
```

template <Iterator>
**struct iterator\_traits**

```
    Public Types
    iterator_category, value_type, difference_type,
    pointer, reference;

    template <T> struct iterator_traits<T*>
    template <T> struct iterator_traits<const T*>
        Public Types
        iterator_category, value_type, difference_type,
        pointer, reference;
```

template <It>

**class reverse\_iterator**

```
:iterator<iterator_traits<It>::iterator_category,
iterator_traits<It>::value_type,
iterator_traits<It>::difference_type,
iterator_traits<It>::pointer,
iterator_traits<It>::reference>
    Public Types
    iterator_type, pointer, reference,
difference_type;
    Public Member Functions
```

```
reverse_iterator();
reverse_iterator(iterator_type i);
reverse_iterator(const reverse_iterator& ri);
template <U>
reverse_iterator(const reverse_iterator<U>& ri);
```

```
template <U>
reverse_iterator<Iterator>& operator=(const reverse_iterator<U>& ri);
```

iterator\_type base() const;

```
reference operator*() const;
pointer operator->() const;
reverse_iterator& operator++();
reverse_iterator& operator--(int);
reverse_iterator& operator--();
reverse_iterator& operator--(int);
reverse_iterator operator+(difference_type) const;
reverse_iterator& operator+=(difference_type);
reverse_iterator& operator-(difference_type) const;
reverse_iterator& operator-=(difference_type);
reference operator[](difference_type) const;
```

### Global reverse\_iterator Operators

```
template <Iterator1, Iterator2>
bool operator==(const reverse_iterator<Iterator1>& a,
                  const reverse_iterator<Iterator2>& b);
template <Iterator1, Iterator2>
reverse_iterator<Iterator1>::difference_type
operator-(const reverse_iterator<Iterator1>& a,
            const reverse_iterator<Iterator2>& b);
template <Iterator>
reverse_iterator<Iterator>::difference_type
operator-(const reverse_iterator<Iterator>& a,
            const reverse_iterator<Iterator>& b);
```

template <Container>
**class back\_insert\_iterator**

```
: public iterator<output_iterator_tag, void, void,
void, void>
    Public Member Functions
back_insert_iterator(Container& x);
back_insert_iterator& operator=(const_reference);
back_insert_iterator& operator*();
back_insert_iterator& operator++();
back_insert_iterator& operator++(int);
```

template <Container>
back\_insert\_iterator<Container>

**back\_inserter**

(Container& x)

template <Container>
**class front\_insert\_iterator**

```
: public iterator<output_iterator_tag, void, void,
void, void>
    Public Member Functions
front_insert_iterator(Container& x);
front_insert_iterator& operator=(const_reference);
front_insert_iterator& operator*();
front_insert_iterator& operator++();
front_insert_iterator& operator++(int);
```

template <Container>
front\_insert\_iterator<Container>

**front\_inserter**

(Container& x)

template <Container>
**class insert\_iterator**

```
: public iterator<output_iterator_tag, void, void,
void, void>
    Public Member Functions
insert_iterator(Container& x, iterator_type itNew)
insert_iterator& operator=(const
insert_iterator&);
insert_iterator& operator=(const_reference value);
insert_iterator& operator*();
insert_iterator& operator++();
insert_iterator& operator++(int);
```

### Global reverse\_iterator Operators

```
template <Container, Iterator>
insert_iterator<Container>
 inserter
(Container& x, Iterator i)

template <InputIterator>
iterator_traits<InputIterator>::difference_type
distance
(InputIterator first, InputIterator last)

template <InputIterator, Distance> void
advance
(InputIterator& i, Distance n)
```

## 5. Smart Pointers

template <T>

**class intrusive\_ptr**

Public Types

T

element\_type;

Public Member Functions

intrusive\_ptr();

intrusive\_ptr(T\* p,

bAddRef = true);

intrusive\_ptr(const intrusive\_ptr& ip);

template <U>

intrusive\_ptr(const intrusive\_ptr<U>&);

intrusive\_ptr& operator=(const intrusive\_ptr&);

template <U>

intrusive\_ptr& operator=(const intrusive\_ptr<U>&);

intrusive\_ptr& operator=(T\*);

T& operator \*() const;

T\* operator ->() const;

T\* get() const;

void reset();

void swap(this\_type& ip);

void attach(T\* pobject);

T\* detach();

typedef T\* (this\_type::\*bool\_()) const;

operator bool\_() const;

operator!() const;

Global intrusive\_ptr Functions & Operators

template <T>

T\* get\_pointer()

const intrusive\_ptr<T>&

intrusivePtr)

template <T>

void swap(intrusive\_ptr<T>& iPtr1,

intrusive\_ptr<T>& iPtr2);

template <T, U>

bool operator==(const linked\_array<T>& lArray,

linked\_array<U>& lArray);

linked\_array<U>& operator=(const linked\_array<U>&);

linked\_array<U> operator=(T\* pArray);

void reset(T\* pArray = NULL);

T& operator[](ptrdiff\_t i) const;

T& operator\*() const;

T\* operator ->() const;

T\* get() const;

int use\_count() const;

bool unique() const;

typedef T\* (this\_type::\*bool\_()) const;

operator bool\_() const;

operator!() const;

void force\_delete();

Global linked\_array Functions & Operators

template <T>

T\* get\_pointer(const linked\_array<T>& linkedArray)

template <T, TD, U, UD>

bool operator==(const linked\_array<TD>& linkedArray1,

const linked\_array<U>& linkedArray2);

```

class safe_object
    Public Member Functions
    bool has_references() const;

    template<T>
    class safe_ptr
        Public Member Functions
        safe_ptr();
        safe_ptr(T* pObject);
        safe_ptr(const this_type& safePtr);

        this_type& operator=(const this_type& safePtr);
        this_type& operator=(T* const pObject);

        bool operator==(const this_type& safePtr) const;
        bool empty() const;
        void reset(safe_object* pObject);
        void reset();

        operator T*() const;
        T& operator*() const;
        T* operator->() const;
        get() const;

        bool unique() const;

        typedef T* (this_type::*bool_())() const;
        operator bool_() const;
        bool operator!() const;

        Global safe_ptr Functions & Operators
        template<T>
        bool operator==(const safe_ptr<T>& safePtr,
                          const T* pObject);
        template<T>
        bool operator<(const safe_ptr<T>& safePtrA,
                        const safe_ptr<T>& safePtrB);

        template<T, Deleter = smart_ptr deleter<T>>
        class scoped_ptr
            Public Types
            T element_type;
            Public Member Functions
            scoped_ptr(T* pValue = NULL);

            void reset(T* pValue = NULL);
            swap(this_type& scopedPtr);

            T& operator*() const;
            operator->() const;
            get() const;

            typedef T* (this_type::*bool_())() const;
            operator bool_() const;
            bool operator!() const;

            Global scoped_ptr Functions & Operators
            template<T, D>
            T* get_pointer(const scoped_ptr<T, D>& scopedPtr);

            template<T, D>
            void swap(scoped_ptr<T, D>& scopedPtr1,
                      scoped_ptr<T, D>& scopedPtr2);

            template<T, D>
            bool operator<(const scoped_ptr<T, D>& scopedPtr1,
                            const scoped_ptr<T, D>& scopedPtr2);

template <T, Deleter = smart_ptr deleter<T>>
class shared_ptr
    Public Types
    T element_type;
    Public Member Functions
    shared_ptr(const allocator_type& allocator =
               eastl::allocator("EASTL classname"));
    template <U>
    shared_ptr(U* pValue);
    shared_ptr(const shared_ptr& sharedPtr);
    template <U, A>
    shared_ptr(const shared_ptr<U, A, D>& sharedPtr);
    template <U, A>
    shared_ptr(const weak_ptr<U, A>& weakPtr);
    template <U, A, D>
    shared_ptr(const shared_ptr<U, A, D>& sharedPtr,
              static_cast_tag);
    template <U, A, D>
    shared_ptr(const shared_ptr<U, A, D>& sharedPtr,
              const_cast_tag);

    shared_ptr& operator=(const shared_ptr&);
    template <U, A, D>
    shared_ptr& operator=(const shared_ptr<U, A, D>&);

    template <U>
    shared_ptr& operator=(U* pValue);

    template <U>
    void reset(U* pValue);
    void reset();

    void swap(this_type& sharedPtr);

    reference_type operator*() const;
    T* operator->() const;
    get() const;
    int use_count() const;
    bool unique() const;

    typedef T* (this_type::*bool_())() const;
    operator bool_() const;
    operator!() const;

template <T, A, D>
class smart_ptr
    Public Types
    T element_type;
    Public Member Functions
    smart_ptr(const allocator_type& allocator =
              eastl::allocator("EASTL classname"));
    template <U>
    smart_ptr(U* pValue);
    smart_ptr(const shared_ptr& sharedPtr);
    template <U, A>
    smart_ptr(const weak_ptr<U, A>& weakPtr);
    template <U, A, D>
    smart_ptr(const shared_ptr<U, A, D>& sharedPtr,
              static_cast_tag);
    template <U, A, D>
    smart_ptr(const shared_ptr<U, A, D>& sharedPtr,
              const_cast_tag);

    smart_ptr& operator=(const shared_ptr&);
    template <U, A, D>
    smart_ptr& operator=(const shared_ptr<U, A, D>&);

    template <U>
    shared_ptr& operator=(U* pValue);

    template <U>
    void reset(U* pValue);
    void reset();

    void swap(this_type& sharedArray);

    reference_type operator*() const;
    T* operator->() const;
    get() const;
    int use_count() const;
    bool unique() const;

    typedef T* (this_type::*bool_())() const;
    operator bool_() const;
    operator!() const;

template <T, A, D>
class shared_array
    Public Types
    T element_type;
    Public Member Functions
    shared_array(T* pArray = NULL,
                const allocator_type& allocator =
                  eastl::allocator("EASTL classname"));
    shared_array(const shared_array& sharedArray);
    shared_array& operator=(const shared_array&);

    void reset(T* pArray = NULL);
    void swap(this_type& sharedArray);

    T& operator[](ptrdiff_t i) const;
    operator*() const;
    operator->() const;
    get() const;
    int use_count() const;
    bool unique() const;

    typedef T* (this_type::*bool_())() const;
    operator bool_() const;
    operator!() const;

template <T, A, D>
class weak_ptr
    Public Types
    T element_type;
    Public Member Functions
    weak_ptr(const allocator_type& allocator =
             eastl::allocator("EASTL classname"));
    weak_ptr(const weak_ptr& weakPtr);
    template <U>
    weak_ptr(const weak_ptr<U, Allocator>& weakPtr);
    template <U, A, D>
    weak_ptr(const shared_ptr<U, A, D>& sharedPtr);

    weak_ptr& operator=(const weak_ptr& weakPtr);
    template <U, A>
    weak_ptr& operator=(const weak_ptr<U, A>& weakPtr);
    template <U, A, D>
    weak_ptr& operator=(const shared_ptr<U, A, D>&);

    shared_ptr<T, Allocator> lock() const;

    int use_count() const;
    bool expired() const;

    void reset();
    void swap(this_type& weakPtr);

    template <U, A>
    void assign(const weak_ptr<U, A>& weakPtr);
    void assign(T* pValue,
               ref_count_sp* pRefCnt);

    template <Y>
    bool less_than(weak_ptr<Y> const& weakPtr) const;

Global weak_ptr Functions & Operators
    template <T, TA, U, UA>
    bool operator<(const weak_ptr<T, TA>& weakPtr1,
                    const weak_ptr<U, UA>& weakPtr2);

    template <T, TA>
    void swap(weak_ptr<T, TA>& weakPtr1,
              weak_ptr<T, TA>& weakPtr2);

```