EAST	L Quick Reference		class hash_multimap class vector multimap	6 6	not2 1 1 class binder1st 1
			class intrusive hash multimap	7	bind1st 1
	v1.0		class set	7	class binder2nd 1
	V 1.0		class fixed set	7	bind2nd 1
	Luc Isaak		class hash set	7	<pre>class pointer_to_unary_function 1</pre>
			class fixed_hash_ set	7	class pointer_to_binary_function 1
EASTL written	and		class intrusive_hash_ set	7	ptr_fun 1
maintained by	Davil Dadviana		class vector_set	8	class mem_fun_t
maintained by	r: Paul Pedriana		class multi set	8	class mem_fun1_t 1 class const mem fun t 1
Reviewers:	Jeremy Paulding		class fixed_multi set	8	class const mem fun1 t
neviewers.	•		class hash_multi set	8	mem fun
	Paul Pedriana		class vector_multi set	8	class mem_fun_ref_t
	Michael Polak		class intrusive_hash_multi set	9	class mem fun1 ref t
	MICHAEI POIAK		struct pair	9	class const mem fun ref t
	Russ Trunt		class ring_buffer	9	class const mem fun1 ref t
	nass name		class stack	9	mem fun ref1
			class vector	9	
_	ACTL 4 40 00		class fixed_vector	10	4. Iterators 1
E	ASTL v1.10.03		2. Algorithms	10	Iterators Categories 1 struct iterator 1
			Query Algorithms	10	struct iterator traits
			Sorting	10	class reverse iterator
1 Containers	•••••	2	Set Operations	11	class back_insert_iterator 1
	••••••	••••	Modifying Sequence Operations	11	back_inserter 1
struct array		2	Min and Max	11	class front insert iterator 1
class basic_str		2	Lexicographic Order	11	front inserter
class fixed_str		3	Heap Operations	11	class insert iterator 1
class fixed_su	bstring	3	Numeric Algorithms	11	inserter
class bitset		3	3. Function Objects	12	distance 1
class deque		3	•		advance 1
class queue		3	struct unary_function	12	5. Smart Pointers 1
class priority	_queue	3	struct binary_function	12	
class list	•	4	struct plus, minus, multiplies, divides, modulus	12	class intrusive_ptr 1
class fixed_list		4	struct negate	12	class linked_ptr 1
class intrusiv class s list	e_iist	4	struct negate struct equal_to, not_equal_to, less, greater,	12	class linked_array
class sist	c t	4 4	less_equal, greater_equal	12	class safe_object
class map	31	4	validate_equal_to, validate_not_equal_to,		class safe_ptr 1
class fixed ma	ın	5	validate_less, validate_greater,		class scoped_ptr 1 class scoped array 1
class hash map		5	validate_less_equal, validate_greater_equal	12	class scoped_array 1
class fixed ha		5	struct str equal to	12	class shared array
class vector m		5	struct logical and, logical or, logical not	12	struct smart ptr deleter
class vector_n		5	struct equal to 2, not equal to 2, less 2	12	struct smart array deleter 1
class multimap		6	class unary_negate	12	class weak ptr 1
class fixed mu		6	not1	12	
crass irvca_iiia	TCTIMP	U	class binary_negate	12	

Template class parameters in italic. typename, class dropped	basic_string(const this_type& x, size type pos,	<pre>const_reference front() const; reference back();</pre>	<pre>basic_string& replace(size_type pos,</pre>	<pre>size_type find_first_not_of(</pre>
	size_type n = npos);	const_reference back() const;	size_type n2,	size_type pos = 0) const;
1. Containers	<pre>basic_string(const value_type* p,</pre>		value_type c);	size_type find_first_not_of(
	size_type n, const allocator type& allocator =	<pre>basic_string& operator+=(const basic_string&); basic string& operator+=(const value type* p);</pre>	basic_string& replace(iterator first, iterator last,	const value_type* p, size type pos,
Common Global Functions & Operators bool operator== (const Container & a,	eastl::allocator("EASTL classname");		<pre>const basic_string& x);</pre>	size_type n) const;
const Container a,	<pre>basic_string(const value_type* p,</pre>	basic string& append(const basic string& x);	basic_string& replace(iterator first, iterator last,	<pre>size_type find_first_not_of(</pre>
bool operator!= ();	eastl::allocator("EASTL classname");		const value_type* p,	value_type c, size type pos = 0) const;
bool operator (); bool operator> ();	<pre>basic_string(size_type</pre>	size_type pos,	size_type n);	
bool operator ();	value_type c, const allocator_type& allocator =	size_type n); basic string& append(const value type* p,	<pre>basic_string& replace(iterator first,</pre>	size_type find_last_not_of(const basic string& x,
bool operator>= ();	eastl::allocator("EASTL classname");		const value_type* p);	size_type pos = npos) const;
	<pre>basic_string(const this_type& x);</pre>	<pre>basic_string& append(const value_type* p);</pre>	basic_string& replace(iterator first,	size_type find_last_not_of(
template <t, n="1" size="" t=""></t,>	<pre>basic_string(const value_type* pBegin,</pre>	<pre>basic_string& append(size_type n,</pre>	iterator last, size_type n,	const value_type* p, size type pos = npos) const;
struct array	const allocator type& allocator =	basic string& append(const value type* pBegn,	value type c);	size type find_last_not_of(
Public Types	eastl::allocator("EASTL classname");	<pre>const value_type* pEnd);</pre>	basic_string& replace(iterator first,	const value_type* p,
value_type,	<pre>basic_string(CtorDoNotInitialize,</pre>	basic string& append sprintf va list(iterator last, const value type* pBegin,	size_type pos, size type n) const;
reference, const_reference,	const allocator_type& allocator =	const value_type* pFormat,	const value_type* pEnd);	size_type find_last_not_of(
iterator, const_iterator, reverse_iterator, const_reverse_iterator	eastl::allocator("EASTL classname");		sizo tuno conv(valuo tuno* n	<pre>value_type</pre>
Public Member Variables	<pre>basic_string(CtorSprintf,</pre>	<pre>basic_string& append_sprintf(</pre>	size_type copy(value_type* p, size type n,	size_type pos = npos) const;
value type mValue[N];);	size_type pos = 0) const;	<pre>basic_string substr(</pre>
Public Member Functions	<pre>const allocator_type& get_allocator() const; allocator type& get allocator();</pre>	<pre>void push back(value type c);</pre>	size type find (const basic string& x,	size_type pos = 0, size type pos = npos) const;
void swap(this type& x);	void set allocator(),	<pre>void</pre>	size type pos = 0) const;	312C_type pos = hpos/ const,
10 = 31 //	const allocator_type&);		size_type find(const value_type* p,	<pre>int compare(const basic_string& x) const;</pre>
<pre>iterator</pre>	basic string& operator=(const this type&);	<pre>basic_string& insert(size_type</pre>	size_type pos = 0) const; size type find (const value type* p,	int compare(size_type pos1, size type n1,
iterator end();	basic string& operator=(const value type*);	basic string& insert(size type pos,	size_type pos,	const basic_string& x) const;
<pre>const_iterator end() const;</pre>	<pre>basic_string& operator=(value_type c);</pre>	const basic_string& x,	size_type n) const;	<pre>int compare(size_type pos1,</pre>
<pre>reverse_iterator</pre>	<pre>void swap(this type& x);</pre>	size_type beg, size_type n);	<pre>size_type find(value_type</pre>	size_type n1, const basic string& x,
reverse iterator rend();		basic_string& insert(size_type pos,		size_type pos2,
<pre>const_reverse_iterator rend() const;</pre>	<pre>basic_string& assign(const basic_string& x); basic string& assign(const basic string& x,</pre>	const value_type* p,	size_type rfind(const basic_string& x, size type pos = npos) const;	size_type n2) const; int compare(const value type* p) const;
bool empty() const;	basic_string& assign(const basic_string& x, size_type pos,	size_type n); basic string& insert(size type pos,	size type rfind(const value type* p,	int compare(size type pos1,
size_type size() const;	size_type n);	const value_type* p);	size_type pos = npos) const;	size_type n1,
size_type max_size() const;	<pre>basic_string& assign(const value_type* p,</pre>	basic_string& insert(size_type pos, size type n,	size_type rfind(const value_type* p, size type pos,	const value_type* p) const; int compare(size type pos1,
<i>T*</i> data();	basic_string& assign(const value_type* p);	size_type n, value type c);	size type n) const;	size type n1,
const T* data() const;	<pre>basic_string& assign(size_type</pre>	iterator insert(iterator p,	size_type rfind(value_type c,	const value_type* p,
reference energter[](size type n).	value_type c); basic string& assign(const value type* pBegin,		size_type pos = npos) const;	size_type n2) const; int compare(const value type* pBegin1,
reference <pre>operator[](size_type n); const reference operator[](size type n) const;</pre>	const value type* pEnd);		<pre>size_type find_first_of(</pre>	const value_type* pEnd1,
	1	value_type c);	<pre>const basic_string& x, size type</pre>	const value_type* pBegin2,
reference at(size_type n); const reference at(size_type n) const;	<pre>iterator</pre>	<pre>void insert(iterator p,</pre>	size_type pos = 0) const, size type find_first_of (const value_type* pEnd2);
const_ference at(size_type ii) const,	iterator end();	const value_type* pEnd);	const value_type* p,	<pre>int comparei(const basic_string& x) const;</pre>
reference front ();	const_iterator end() const;	hit-i	<pre>size_type pos = 0) const; size type find_first_of(</pre>	<pre>int comparei(const value_type* p) const; int comparei(const value type* pBegin1,</pre>
<pre>const_reference front() const;</pre>	reverse_iterator	<pre>basic_string& erase(size_type</pre>	const value type* p,	const value_type* pEnd1,
reference back();	reverse_iterator rend();	iterator erase(iterator p);	size_type pos,	const value_type* pBegin2,
<pre>const_reference back() const;</pre>	<pre>const_reverse_iterator rend() const;</pre>	iterator erase(iterator pBegin, iterator pEnd);	size_type n) const; size type find_first_of (const value_type* pEnd2);
bool <pre>validate() const;</pre>	bool empty() const;	iterator pEnd);	value_type c,	<pre>void make_lower();</pre>
	<pre>size_type size() const;</pre>	void clear();	size_type pos = 0) const;	void make_upper();
Global Functions & Operators (+ Containers Common)	<pre>size_type length() const; size_type max_size() const;</pre>	<pre>void reset();</pre>	size type find last of(<pre>void ltrim(); void rtrim();</pre>
void swap (array<>& a,	size type capacity() const;	<pre>basic_string& replace(size_type pos,</pre>	const basic_string& x,	void trim();
array<>& b);		size_type n,	size_type pos = npos) const; size_type find_last_of(basic_string left(size_type n) const;
	<pre>void resize(size_type n,</pre>	const basic_string& x); basic string& replace(size type pos1,	const value_type* p,	<pre>basic_string right(size_type n) const;</pre>
<pre>template <t, allocator="eastl::allocator"></t,></pre>	void resize(size_type n);	size type n1,	size_type pos = npos) const;	basic_string& sprintf_va_list (
class basic_string	void reserve(size type n =0);	const basic_string& x, size type pos2,	size_type find_last_of(const value type* p,	<pre>const value_type* pFormat, va list arguments);</pre>
Public Types	<pre>void</pre>	size_type posz, size_type n2);	size type pos,	basic_string& sprintf(
value_type,		<pre>basic_string& replace(size_type pos,</pre>	size_type n) const;	const`value_type* pFormat,);
pointer, const_pointer, reference, const reference,	<pre>const value_type* data() const; const value type* c str() const;</pre>	size_type n1, const value_type* p,	size_type find_last_of (value type c,	bool validate() const;
iterator, const_iterator,	= ** = ** .	size_type n2);	size_type pos = npos) const;	<pre>int validate_iterator(const_iterator i) const;</pre>
reverse_iterator, const_reverse_iterator;	reference operator[](Size_type ii);	basic_string& replace(size_type pos,	size type find first not of (Global Types
Public Member Functions	<pre>const_reference</pre>	<pre>size_type</pre>	const basic string& x,	<pre>typedef basic_string<char></char></pre>
<pre>basic_string(); basic_string(const allocator type& allocator);</pre>	const reference at(size type n) const;		size_type pos = 0) const;	<pre>typedef basic_string<wchar_t> wstring;</wchar_t></pre>
	reference front();			
2 EASTL 1.10.03 Quick Reference v1.0			Electro	nic Arts Inc Internal Use Only - Confidential

<pre>typedef basic string<char8 t=""> string8;</char8></pre>	fixed_substri	ng(const value type* pBegin,	basic string& erase(size type pos = 0,	template <t, allocator="eastl::al</td"><td>llocator,</td><td></td></t,>	llocator,	
<pre>typedef basic_string<char16_t> string16;</char16_t></pre>	_	const value_type* pEnd);	size_type n = npos);	unsigned kDequeSubarraySize =		<pre>iterator erase(iterator pos);</pre>
<pre>typedef basic_string<char32_t> string32;</char32_t></pre>			<pre>iterator erase(iterator p);</pre>	DEQUE DEFAULT SUBARRAY SIZE(T)>		iterator erase(iterator first,
Global Functions & Operators (+ Containers Common)	this_type& op	erator=(const base_type& x)	iterator erase(iterator pBegin,	class deque		iterator last);
void swap(basic string<>& a,			iterator pEnd);	Public Types		void clear();
basic_string<>& b);		ng functions should be used carefully.	basic string& sprintf va list(value_type,		bool validate() const;
	// Do not call	unsupported resizing functions.	const value type* pFormat,	pointer, const p	oointer,	<pre>int validate iterator(const iterator i) const;</pre>
bool operator==			va_list arguments);		reference,	Global Functions & Operators (+ Containers Common)
<pre>(const basic_string<>::value_type* p, const basic string<>& b);</pre>		<pre>operator=(const basic_string& x); operator=(value type c);</pre>	<pre>basic_string& sprintf(</pre>		iterator,	void swap(deque<>& a,
bool operator!= ();	Dasic_stiiliga	operator=(varue_type t);	<pre>const value_type* pFormat,);</pre>		reverse_iterator	deque<>& b);
bool operator< ();	void	<pre>swap(basic string& x);</pre>		Public Member Function	ons	
bool operator> ();			template <size_t n=""></size_t>	deque();		template <t, container="deque<T,</td"></t,>
bool operator<= ();	void	resize(size_type n,	class bitset	<pre>deque(const_allocator_type& allo</pre>	ocator);	eastl::allocator,DEQUE DEFAULT SUBARRAY SIZE(T)>>
<pre>bool operator>= ();</pre>	void	value_type c);	Public Types	deque(size_type n,		class queue
bool operator==	VOIU	<pre>resize(size_type n);</pre>	word type	<pre>const allocator_type& allo eastl::allocator("EASTL clas</pre>		•
(const basic string<>& a,	void	reserve(size type = 0);	// reference: a helper class for the operator[] to	deque(size type n,	, s. r. a	Public Types
<pre>const basic_string<>::value_type* p);</pre>	void	<pre>set_capacity(size_type n);</pre>	// manipulate the	`const value_type& valu	ue,	value_type,
bool operator!= ();		- 4	// individual bits: x[i] = b; x[i] = y[j]; b = ~x[i]; x[i].flip()	const allocator_type& allo		container_type, reference, const_reference;
bool operator (); bool operator ();	void	<pre>clear();</pre>	// individual bits. x[i] = b, x[i] = y[j], b = x[i], x[i].iiip() class reference;		ssname");	Public Member Functions
bool operator<= ();	hasic string&	<pre>operator+=(const basic string& x);</pre>	Public Member Variables	<pre>deque(const this_type& x); deque(InputIterator first</pre>		queue();
bool operator>= ();	basic_string&			InputIterator last		<pre>queue(const Container& x);</pre>
		<pre>operator+=(value type c);</pre>	<pre>word_type mWord[BITSET_WORD_COUNT(N)];</pre>		-//	bool empty() const;
t1-t- (T -i tt-Ct			Public Member Functions	allocator_type& <pre>get_allocator();</pre>		bool empty() const; size_type size() const;
template <t, nodecount,<br="" size_t="">bool bEnableOverflow = true,</t,>		<pre>append(const basic_string& x);</pre>	bitset();	void set_allocator (5120_type 5120() const,
Allocator = eastl::allocator>	basic_string&	append(const basic_string& x,	<pre>bitset(uint32_t value);</pre>	const a	allocator_type&);	reference front ();
class fixed string		size_type pos, size_type n);	this type& operator&=(const this type& x);	deque& operator =(const th	nis tyne%).	const_reference front () const;
:basic string <pre>:basic string</pre> <pre>, fixed vector allocator <> ></pre>	basic string&	append(const value_type* p,	this type& operator = (const this type& x);	void swap (this typ		117
		size_type n);	this type& operator^=(const this type& x);	= /1	,,	reference back(); const reference back() const;
Has all the basic_string functionality.		<pre>append(const value_type* p);</pre>		void assign (size_type	n,	const_reference back() const,
Public Types		<pre>append(size_type n);</pre>	this_type& operator<<=(size_t n);	const void assign (<i>InputItera</i> t	value_type&);	void push (const value type&);
fixed_vector_allocator<>::overflow_allocator_type	pasic_stringa	<pre>append(size_type</pre>	<pre>this_type& operator>>=(size_t n); this type operator<< (size t n) const;</pre>	InputIterat		void pop();
overflow_allocator_type;	basic string&	append(const value type* pBegin,	this type operator>> (size t n) const;	Impactician	101 1051/,	
Public Member Functions		<pre>const value_type* pEnd);</pre>		iterator begin(<pre>container_type& get_container(); const container type& get container() const;</pre>
	basic_string&	append_sprintf_va_list(this_type& set();		() const;	const container_typea get_tontainer() const,
<pre>fixed_string(); fixed_string(const base_type& x,</pre>		<pre>const value_type* pFormat, va list arguments);</pre>	<pre>this_type& set(size_t i, bool value = true);</pre>	iterator end(); const iterator end()	; const;	to delicate del transcription
size type pos,	basic string&	append sprintf(this type& reset();	reverse iterator rbegin		<pre>template <t, container="vector<T">, Compare = less<container::value type=""> ></container::value></t,></pre>
size_type n = npos);		const value_type* pFormat,);	this type& reset(size t i);		() const·	
<pre>fixed_string(const_value_type* p,</pre>				reverse_iterator rend()	/)	class priority_queue
<pre>size_type n); fixed_string(const value_type* p);</pre>	void void	<pre>push_back(value_type c); pop_back();</pre>	<pre>this_type& flip(); this type& flip(size t i);</pre>	const_reverse_iterator rend()) const;	Public Types
fixed_string(size type n,	VOIU	pop_back();	this type operator () const;	bool empty () const;		value_type,
const value_type& value);	basic_string&	<pre>assign(const value_type* p,</pre>		size_type size() const;		container_type, reference, const_reference;
<pre>fixed_string(const this_type& x);</pre>		size_type	reference operator[](size_t i);			Public Member Functions
<pre>fixed_string(const base_type& x); fixed_string(const value type* pBegin,</pre>	pasic_string&	<pre>assign(size_type</pre>	<pre>bool operator[](size_t i) const;</pre>	void resize(size_type		<pre>priority_queue(const Compare& compare =Compare()); priority_queue(const Compare& compare,</pre>
const value type* pEnd);		varue_type c/,	const word type* data() const;	void resize(size type	n);	const Container& x);
<pre>fixed_string(CtorDoNotInitialize,</pre>	basic_string&	<pre>insert(size_type pos,</pre>	word type* data();	void set_capacity (size		
size_type n);	h	const basic_string& x);				<pre>priority_queue(InputIterator first,</pre>
<pre>fixed_string(CtorSprintf,</pre>	pasic_stringa	<pre>insert(size_type pos,</pre>	unsigned long to_ulong() const;	reference operator[](size const reference operator[](size const reference operator[](size const reference)		InputIterator last,
const ratac_type protingt,),		size_type beg,	size t count() const;	reference at(size type r	ze_type n) const; n):	<pre>const Compare& compare = Compare(), const Container& x = Container());</pre>
<pre>this_type& operator=(const base_type& x);</pre>		size_type n);	size t size() const;	const_reference at(size_type r		const container x container(//;
	basic_string&	<pre>insert(size_type pos,</pre>	= "	reference front ();		bool empty() const;
<pre>overflow_allocator_type& get_overflow_allocator();</pre>		const value_type* p,	<pre>bool operator==(const this_type& x) const;</pre>	const_reference front () const;	;	size_type size () const;
<pre>void set_overflow_allocator(</pre>	hasic string&	size_type n); insert(size type pos,	<pre>bool operator!=(const this_type& x) const;</pre>	reference back(); const reference back() const;		const reference top() const;
		const value_type* p);	bool test(size t i) const;	const_iererence back() const,		const_reference top() const,
template <t></t>	basic_string&	<pre>insert(size_type pos,</pre>	bool any() const;		nst value_type&);	void push (const value_type& value);
		size_type n,	bool none () const;	<pre>void push_front();</pre>		void pop();
class fixed_substring	iterator	value_type c); insert(iterator p,	-i + (i + (i+/)+.	void push_back(cons	st value_type&);	woid shange(cire tune n).
:basic_string< <i>T</i> >	I CCI a COI	<pre>insert(iterator</pre>	<pre>size_t find_first() const; size t find next(size t last find) const;</pre>	<pre>void</pre>		<pre>void change(size_type n); void remove(size type n);</pre>
Public Member Functions	void	<pre>insert(iterator p,</pre>	Size_c iliu_next(Size_c iast_lillu) collst,	void pop_back();		1014 1010 (5126_t)pc 11/)
<pre>fixed_substring();</pre>		size_type n,	<pre>word_type& DoGetWord(size_t i);</pre>			<pre>container_type& get_container();</pre>
fixed_substring(const base_type& x);	waid	value_type c);	word_type DoGetWord (size_t i) const;	iterator insert(iterato		const container_type& get_container() const;
<pre>fixed_substring(const base_type& x,</pre>	void	<pre>insert(iterator</pre>	Global Functions & Operators		/alue_type&);	haal walidata() const.
size_type pos, size type n = npos);		const value_type* pBegin,	bitset <n> operator&(const bitset<n>& a,</n></n>	void insert(iterate	or pos, /pe n,	bool validate() const;
<pre>fixed_substring(const value type* p,</pre>			const bitset <n>& b)</n>		value type&);	
size_type n);			bitset	void insert(iterato	or pos,	
<pre>fixed_substring(const value_type* p);</pre>			bitset <n> operator^()</n>		terator first,	
2 FACTU 4 40 00 0 : 1 D f	<u> </u>			I Inputit	terator last);	

		1			
class list	tor = eastl::allocator>	void reset();	<pre>const_reverse_iterator rend() const;</pre>	<pre>slist(size_type</pre>	<pre>void reverse();</pre>
class list	D. H.C. T	void remove(const 7% x);	bool empty() const;	const allocator_type& allocator =	void splice (iterator pos,
	Public Types	<pre>void remove_if(Predicate);</pre>	size_type size() const;	<pre>eastl::allocator("EASTL classname"); slist(const this type& x);</pre>	this_type& x); void splice (iterator pos,
value_type, pointer,	const_pointer,	void reverse();	reference front ();	<pre>slist(InputIterator first,</pre>	this type& x,
reference,	const_reference,	void splice (iterator pos,	const_reference front () const;	InputIterator last);	iterator i);
iterator,	const_iterator, const reverse iterator	this_type& x); void splice(iterator pos,	reference back(); const reference back() const;	allocator type& get allocator();	void splice (iterator pos, this type& x,
reverse_iterator,	const_reverse_rterator	this_type& x,	const_reference back() const,	void set_allocator(iterator first,
ListNode <t></t>	<pre>node_type;</pre>	iterator i);	<pre>void push_front(value_type& x);</pre>	const allocator_type&);	iterator last);
Publi	c Member Functions	void splice (iterator pos, this type& x,	<pre>void push_back(value_type& x); void pop front();</pre>	this type& operator =(const	void splice_after (iterator pos,
<pre>list();</pre>	. 0	iterator first,	void pop_back();	this type&);	iterator before first,
<pre>list(const allocator list(size type</pre>	r_type& allocator); n,	<pre>iterator last); void merge(this type& x);</pre>	h1	void swap(this_type& x); void assign(size type n,	<pre>iterator before_last); void splice after(iterator pos,</pre>
const allocator		<pre>void</pre>	bool contains (const <i>T</i> & x) const;	<pre>void assign(size_type n,</pre>	iterator previous);
	or("EASTL classname");		iterator locate(7% x);	void assign (InputIterator first,	void splice_after (iterator pos,
<pre>list(size_type const value type)</pre>	n, pe& value,	<pre>void unique(); void unique(BinaryPredicate);</pre>	<pre>const_iterator locate(const T& x) const;</pre>	InputIterator last);	this_type& x);
const allocato	r_type& allocator =	will unique (Bindiyr Tealcate),	iterator insert(iterator pos,	<pre>iterator begin();</pre>	bool validate() const;
	or("EASTL classname");	<pre>void sort();</pre>	7& x);	const_iterator begin() const;	<pre>int validate_iterator(const_iterator i) const;</pre>
<pre>list(const this_type list(InputIterator</pre>	e& x); first,	<pre>void sort(Compare);</pre>	<pre>iterator erase(iterator pos); iterator erase(iterator pos,</pre>	<pre>iterator end(); const iterator end() const;</pre>	Global Functions & Operators (+ Containers Common)
InputIterator	last);	bool validate() const;	iterator pos,	const_iterator end() const,	void swap (const slist<>& a,
allocator tuno	<pre>get allocator();</pre>	<pre>int validate_iterator(const_iterator i) const;</pre>		<pre>iterator before_begin();</pre>	const slist<>& b);
allocator_type& void	set allocator(Global Functions & Operators (+ Containers Common)	<pre>void clear();</pre>	<pre>const_iterator before_begin() const;</pre>	
	<pre>const allocator_type&);</pre>	void swap (list<>& a,	static void remove(T& value);	iterator previous(template <t, nodecount,<br="" size_t="">bool bEnableOverflow = true.</t,>
this type&	<pre>operator=(const this type&);</pre>	list<>& b);		const_iterator pos);	Allocator = eastl::allocator>
void	swap(this type& x);		<pre>void reverse(); void splice(iterator pos,</pre>	const_iterator	class fixed slist
void	<pre>assign(size_type n,</pre>	template <t, nodecount,<br="" size_t="">bool bEnableOverflow = true,</t,>	T& x);	= ' ' ' '	:slist <t, fixed_node_pool<=""> ></t,>
void	<pre>const value_type&); assign(InputIterator first,</pre>	Allocator = eastl::allocator>	void splice (iterator pos,	bool empty() const;	Has all the slist functionality.
VOIU	InputIterator last);	class fixed_list	<pre>intrusive_list& x); void splice(iterator pos,</pre>	size_type size() const;	Public Member Functions
*1	1	:list <t, fixed_node_pool<=""> ></t,>	intrusive_list& x,	void resize (size_type n,	<pre>fixed_slist();</pre>
iterator const iterator	<pre>begin(); begin() const;</pre>	Has all the list functionality.	iterator i); void splice (iterator pos,	const value_type&); void resize(size type n);	<pre>fixed_slist(size_type</pre>
iterator	end();	Public Member Functions	void splice (iterator pos, intrusive list& x,	rotu restze(stze_type n/),	<pre>fixed_slist(size_type</pre>
const_iterator	end() const;	<pre>fixed_list();</pre>	iterator first,	reference front ();	<pre>fixed_slist(const this_type& x);</pre>
reverse_iterator const reverse iterat	<pre>rbegin(); tor rbegin() const:</pre>	<pre>fixed_list(size_type</pre>	iterator last);	const_reference front () const;	<pre>fixed_slist (InputIterator first,</pre>
reverse_iterator	rend();	<pre>fixed_list(size_type</pre>	<pre>void merge(this type& x);</pre>	void <pre>push_front(const value type&);</pre>	inputitefutoi fast),
const_reverse_iterat	tor rend () const;	<pre>fixed_list(const this_type& x);</pre>	void merge(this_type& x,	void <pre>push_front();</pre>	size_type max_size() const;
bool	<pre>empty() const;</pre>	<pre>fixed_list (InputIterator first,</pre>	Compare compare);	<pre>void pop_front();</pre>	bool full() const;
size_type	<pre>size() const;</pre>		<pre>void unique();</pre>	<pre>iterator insert(iterator pos);</pre>	
void	resize(size type n,	size_type max_size() const;	<pre>void unique(BinaryPredicate);</pre>	iterator insert(iterator pos,	<pre>template <key, compare="less<Key" t,="">, Allocator = eastl::allocator></key,></pre>
VOIU	const value type&);	bool full() const;	<pre>void sort();</pre>	const value_type&); void insert(iterator pos,	class map
void	<pre>resize(size_type n);</pre>	to the state of the state of the	<pre>void sort(Compare compare);</pre>	`size_type n, ´	: public rbtree<, bMutableIterators = true,
reference	<pre>front();</pre>	template <t =="" intrusive="" list="" node=""></t>	bool validate () const;	<pre>const value_type&); void insert(iterator pos,</pre>	bUniqueKeys = true>
const_reference	<pre>front() const;</pre>	class intrusive_list	int validate iterator(const iterator i) const;	InputIterator first,	Public Types
reference	back();	Public Types	Global Functions & Operators (+ Containers Common)	<pre>InputIterator last);</pre>	Key key_type;
const_reference	<pre>back() const;</pre>	value_type, pointer, const pointer,	void swap (intrusive_list< <i>T</i> >& a,	<pre>iterator insert_after(iterator pos);</pre>	<pre>pair<const key,="" t=""> value_type; rbtree node<value type=""> node_type;</value></const></pre>
void	<pre>push_front(const value_type&);</pre>	reference, const_reference,	intrusive_list<>& b);	iterator insert_after (iterator pos,	value type& reference;
void void	<pre>push_front(); push_back(const value type&);</pre>	iterator, const_iterator,		const value_type&);	const value_type& const_reference;
void	push_back();	reverse_iterator, const_reverse_iterator	<pre>template <t, allocator="eastl::allocator"></t,></pre>	void <pre>insert_after (iterator pos,</pre>	rbtree_iterator <value_type, value type*,</value_type,
void	<pre>pop_front();</pre>	ListNode <t> node_type;</t>	class slist	const value_type&);	value_type&>
void	pop_back();	Public Member Functions	Public Types	void <pre>insert_after (iterator pos,</pre>	<pre>iterator; rbtree iterator<value pre="" type,<=""></value></pre>
iterator	<pre>insert(iterator pos);</pre>	<pre>intrusive_list();</pre>	value_type,	InputIterator last);	const value_type*,
iterator	<pre>insert(iterator pos,</pre>	<pre>intrusive_list(const this_type& x);</pre>	pointer, const_pointer, reference, const_reference,	,	const value_type&>
void	<pre>const value_type&); insert(iterator pos,</pre>	this_type& operator=(const	iterator, const_iterator	<pre>iterator erase(iterator pos); iterator erase(iterator first,</pre>	const_iterator;
.014	size_type n,	this type&);		` iterator last);	reverse_iterator, const_reverse_iterator
waid	const value_type&);	<pre>void swap(this_type& x);</pre>	SListNode <t> node_type; Dublic Mambar Functions</t>	iterator erase_after(iterator pos);	
void	<pre>insert(iterator pos,</pre>	<pre>iterator begin();</pre>	Public Member Functions	<pre>iterator erase_after(iterator before_first,</pre>	<pre>Compare pair<iterator, bool=""> key_compare; insert_return_type;</iterator,></pre>
	InputIterator last);	<pre>const_iterator begin() const;</pre>	<pre>slist(); slist(const allocator type& allocator);</pre>	void clear();	
iterator	<pre>erase(iterator pos);</pre>	<pre>iterator end(); const iterator end() const;</pre>	slist(size type n,	void reset();	T mapped_type;
iterator	erase(iterator pos); erase(iterator first,	reverse_iterator	<pre>const allocator_type& allocator = eastl::allocator("EASTL classname");</pre>	void remove(const T& x);	Public Member Functions
	iterator last);	<pre>const_reverse_iterator rbegin() const;</pre>		void remove_if(Predicate);	<pre>map(const allocator_type& allocator = eastl::allocator("EASTL classname");</pre>
void	clear();	reverse_iterator rend();			,
/ EACTI 1	10 02 Onick Deference v1 0			Eleatro	nic Arte Inc. Internal Hea Only Confidential

		l u na cara	(C)+ 14 (+/)+	I const Works backFunction Wach()	1
<pre>map(const Compare& const allocato</pre>	compare, r type& allocator =	Has all the map functionality.	<pre>float load_factor() const; float get max load factor() const;</pre>	<pre>const Hash% hashFunction = Hash(), const Predicate% predicate = Predicate());</pre>	mapped type& operator[](const key type&);
eastl::alloca	ator("EASTL classname");	Public Member Functions	<pre>void set_max_load_factor(float fMaxLoadFactor);</pre>		Inherited from base class, RandomAccessContainer
<pre>map(const this_type map(Iterator</pre>	e& x); itBegin,	<pre>fixed_map(); fixed_map(const Compare& compare);</pre>	const rehash policy type&		allocator_type& get_allocator();
Iterator	itEnd);	<pre>fixed_map(const this_type& x);</pre>	<pre>rehash_policy() const</pre>	<pre>template <key, compare="less<Key" t,="">, Allocator = eastl::allocator, RandomAccessContainer =</key,></pre>	<pre>void set_allocator(</pre>
allocator type&	<pre>get_allocator();</pre>	<pre>fixed_map(InputIterator first,</pre>	<pre>void rehash_policy(const rehash_policy_type&</pre>	vector <pair<key, t="">, Allocator> ></pair<key,>	const arrocator_typea/,
void	set_allocator(***	class vector_map	iterator begin();
	<pre>const allocator_type&);</pre>	size_type max_size() const;	<pre>insert_return_type insert(const key_type& key) insert return type insert(const value type&</pre>	: public RandomAccessContainer	<pre>const_iterator begin() const; iterator end();</pre>
this type&	<pre>operator=(const this type&);</pre>		value);	Public Types	const_iterator end() const;
void 7	<pre>swap(this_type& x);</pre>	<pre>template <key, hash="hash<Key" t,="">, Predicate = equal to<key>, Allocator = eastl::allocator, bool</key></key,></pre>	iterator insert(const_iterator,	Key key_type;	reverse_iterator rbegin(); const reverse iterator rbegin() const;
iterator	begin();	bCacheHashCode = false>	const value_type&value);	<pre>pair<key, t=""></key,></pre>	reverse_iterator rend();
const_iterator	<pre>begin() const;</pre>	class hash_map	void insert(InputIterator first,	map_value_compare< <i>Key</i> , value_type, <i>Compare</i> >	<pre>const_reverse_iterator rend() const;</pre>
iterator const iterator	<pre>end(); end() const;</pre>	: public hashtable<, bMutableIterators=true,	InputIterator last);	<pre>value_compare; value type& reference;</pre>	bool empty() const;
reverse_iterator	rbegin();	bUniqueKeys=true>	iterator erase(iterator);	const value_type& const_reference;	size_type size() const;
const_reverse_itera		Public Types	iterator erase(iterator, iterator);	T mapped_type;	void clear();
reverse_iterator const reverse itera	<pre>rend(); ator rend() const;</pre>	<pre>Key pair<const key,="" t=""></const></pre>	size_type erase(const key_type&);	pointer, const pointer,	Global Functions & Operators (+ Containers Common)
		hash_node <value_type, bcachehashcode="false"></value_type,>	void clear();	iterator, const_iterator,	void swap(vector map<>& a,
bool size type	<pre>empty() const; size() const;</pre>	<pre>pair<iterator, bool=""></iterator,></pre>	<pre>void reset(); e; void rehash(size type nBucketCount);</pre>	reverse_iterator, const_reverse_iterator;	vector_map<>& b);
size_type	Size() Const,	value_type& reference;	tenash(Size_type houcketcount),	Public Member Functions	
insert_return_type		const value_type& const_reference;	iterator find(const key_type& k);	<pre>vector_map(); vector_map(const allocator type& allocator);</pre>	template <key, bucketcount,="" hash="</td" size_t="" t,=""></key,>
iterator	<pre>insert(const value_type&); insert(iterator pos,</pre>	<pre>node_iterator<bconst=false,> local_iterator; node iterator<bconst=true,> const_local_iterator</bconst=true,></bconst=false,></pre>	const_iterator find(const key_type& k) const; r; iterator find_as(const U&,	<pre>vector_map(const key_compare& comp,</pre>	hash <key>, Equal = equal_to<key> ></key></key>
	const value_type&	hashtable_iterator <bconst=false,></bconst=false,>	UHash,	<pre>const allocator_type& allocator = eastl::allocator("EASTL</pre>	<pre>class intrusive_hash_map : intrusive_hashtable<key, equal,<="" hash,="" pre="" value="T,"></key,></pre>
value); void	<pre>insert(InputIterator first,</pre>	<pre>iterator; hashtable iterator<bconst=true,></bconst=true,></pre>	BinaryPredicate); const iterator find_as (const U%,	classname");	bucketCount, bConstIterators=false,
VOIU	InputIterator last);	const_iterator;	UHash,	<pre>vector_map(const vector_map& x);</pre>	bUniqueKeys=true>
T++	(:tt	T wanted type.	BinaryPredicate) const; iterator find as (const U& u);	<pre>vector_map(InputIterator first,</pre>	Public Types
	<pre>erase(iterator pos); erase(iterator first,</pre>	<pre>mapped_type;</pre>	iterator find_as(const U& u); const iterator find as(const U& u) const;	<pre>InputIterator last);</pre>	Key key_type; Value value type;
	iterator last);	hash_map(const allocator_type& allocator =		<pre>vector_map(InputIterator first,</pre>	Value node_type;
void	<pre>erase(const key_type* first,</pre>	eastl::allocator("EASTL classname") hash_map(size type nBucketCount,	<pre>iterator</pre>	const key_compare& compare);	<pre>pair<iterator, bool=""> insert_return_type;</iterator,></pre>
size_type	erase(const Key& k);	const Hash& hashFunction = Hash()		waster mank energia (sonst waster mank v)	<pre>value_type& reference; const value type& const_reference;</pre>
	-1()-	<pre>const Predicate& predicate = Predicate(),</pre>	size type	<pre>vector_map& operator=(const vector_map& x); void</pre>	<pre>intrusive_node_iterator<value_type, bconst="false"></value_type,></pre>
	clear(); reset();	const allocator type& allocator =	size_type		<pre>local_iterator; intrusive node iterator<value bconst="true" type,=""></value></pre>
		eastl::allocator("EASTL classname")	pair <iterator, iterator=""></iterator,>	<pre>key_compare key_comp() const; value compare value comp() const;</pre>	const local iterator;
	<pre>ind(const key_type& key); ind(const key type& key) const;</pre>	hash_map(ForwardIterator first, ForwardIterator last,	<pre>equal_range(const Key& k); pair<const const="" iterator="" iterator,=""></const></pre>	varue_compare varue_comp() const,	intrusive_hashtable_iterator <value_type,< td=""></value_type,<>
const_iterator i.	ind(const key_typed key) const,	size_type nBucketCount = 0,	equal_range(const Key& k) const;	pair <iterator, bool=""></iterator,>	<pre>bConst=false> iterator; intrusive hashtable iterator<value pre="" type,<=""></value></pre>
iterator f :	ind_as(const <i>U</i> & u,	const Hash& hashFunction = Hash() const Predicate& predicate =		<pre>insert(const value_type& value);</pre>	bConst=true> const_iterator;
const iterator f :	<pre>Compare2 compare2); ind as(const U& u,</pre>	Predicate(),	7& operator[](const Key& key);	iterator insert(iterator pos,	<pre>Value</pre>
	Compare2 compare2) const;	const allocator_type& allocator =	<pre>Hash hash_function() const;</pre>	<pre>const value_type& value);</pre>	Public Member Functions
iterator low	er_bound(const key type&);	eastl::allocator("EASTL classname");	Predicate equal_function();	<pre>void insert(InputIterator first,</pre>	intrusive hash map(const Hash& h = Hash(),
const iterator low	er_bound(const key_type&) const;	allocator_type& get_allocator();	bool validate () const;	<pre>InputIterator last);</pre>	const Equal& eq = Equal());
iterator upp e	<pre>er_bound(const key_type&);</pre>	<pre>void set_allocator(</pre>	<pre>int validate_iterator(const_iterator i) const;</pre>	<pre>iterator erase(iterator pos);</pre>	void swap (this type& x);
const_fterator uppo	<pre>er_bound(const key_type&) const;</pre>	const director_typed/,	Global Functions & Operators (+ Containers Common)	iterator erase(iterator first,	<pre>void</pre>
size_type	<pre>count(const Key& k) const;</pre>	this_type& operator =(const this_type&) void swap (this type& x):); void swap (hashtable<>& a, hashtable<>& b);	<pre>iterator last); size type erase(const key type& k);</pre>	<pre>iterator</pre>
pair <iterator, ite<="" td=""><td>rator></td><td><pre>void swap(this_type& x);</pre></td><td>masheaste (/ a 5/)</td><td></td><td><pre>const_iterator begin() const; iterator end();</pre></td></iterator,>	rator>	<pre>void swap(this_type& x);</pre>	masheaste (/ a 5/)		<pre>const_iterator begin() const; iterator end();</pre>
	<pre>equal_range(const Key& k);</pre>	<pre>iterator begin();</pre>	template <key, bucketcount="</td" nodecount,="" size="" t="" t,=""><td><pre>iterator find(const key_type& k); const iterator find(const key type& k) const;</pre></td><td>const_iterator end() const;</td></key,>	<pre>iterator find(const key_type& k); const iterator find(const key type& k) const;</pre>	const_iterator end() const;
pair <const_iterato< td=""><td>r, const_iterator> equal range(const Key& k) const;</td><td><pre>const_iterator</pre></td><td>nodeCount + 1, bool bEnableOverflow = true, Hash =</td><td>const_frefator find(const key_typea k) const,</td><td>local iterator begin(size type n)</td></const_iterato<>	r, const_iterator> equal range(const Key& k) const;	<pre>const_iterator</pre>	nodeCount + 1, bool bEnableOverflow = true, Hash =	const_frefator find(const key_typea k) const,	local iterator begin (size type n)
'	equal_lange(const keya k) const,	const_iterator end() const;	hash <key>, Predicate = equal_to<key>, bool bCacheHashCode = false, Allocator =</key></key>	iterator find_as (const U& u,	local iterator end(size type)
T& .	<pre>operator[](const Key& key);</pre>	local iterator begin (size type n)	eastl::allocator>	BinaryPredicate); const iterator find as (const U& u,	const_local_iterator
bool validate ()	const:	local_iterator	class fixed hash map	BinaryPredicate) const;	const const local iterator end(size type) const
	terator(const_iterator i) const;	const_local_iterator	: hash_map <key, hash,="" predicate,<="" td="" value="T,"><td>size_type</td><td></td></key,>	size_type	
Global Functions 8	& Operators (+ Containers Common)	const const local iterator end (size type) const	fixed_hashtable_allocator<>, bCacheHashCode>	size_type count(const key_typea k),	<pre>bool</pre>
void swap (rbtree<:		const_totat_iterator end(size_type) const	rias an the hash_map functionality.	iterator lower_bound(const key_type&);	
rbtree<:	>& b);	bool empty() const;	Public Member Functions	<pre>const_iterator lower_bound(const key_type&) const; iterator upper_bound(const key type&);</pre>	size_type bucket_count() const,
		size_type size() const;	<pre>fixed_hash_map(</pre>	const_iterator upper_bound(const key_type&);	size_type bucket_size(size_type n) const; size_type bucket(const key_type& k) const;
	size_t nodeCount, bool true, Compare = less <key>,</key>	size_type bucket_count() const	<pre>const Predicate& predicate = Predicate());</pre>	pair <iterator, iterator=""></iterator,>	
Allocator = eastl:		size_type bucket_size(size_type n) const	<pre>fixed_hash_map(</pre>	equal_range(const key type&);	float load_factor() const;
class fixed_map			InputIterator first, InputIterator last,	pair <const const="" iterator="" iterator,=""></const>	
:map <key, compa<="" t,="" td=""><td>re, fixed_node_pool<> ></td><td></td><td></td><td>equal_range(const key_type&) const;</td><td></td></key,>	re, fixed_node_pool<> >			equal_range(const key_type&) const;	

<pre>insert_return_type insert(value_type& va insert_return_type insert(const_iterator,</pre>	this_type& operator= (const this_type&	<pre>fixed_multimap(InputIterator first,</pre>	<pre>const rehash_policy_type& rehash_policy() const;</pre>	<pre>vector_multimap(const vector_multimap& x); vector_multimap(InputIterator first,</pre>
value_type& va void insert (InputIterator fi InputIterator la	st,	<pre>size_type max_size() const;</pre>	<pre>void rehash_policy(const rehash_policy_type&) insert return type insert(const key type& key)</pre>	; InputIterator last); vector_multimap(InputIterator first, InputIterator last,
iterator erase(iterator); iterator erase(iterator, iterato	<pre>const_iterator begin() const; iterator end(); const iterator end() const;</pre>	template <key, hash="hash<Key" t,="">, Predicate = equal to<key>, Allocator = eastl::allocator, bool</key></key,>	<pre>insert_return_type insert(const value_type&); iterator insert(const_iterator,</pre>	<pre>const key_compare& compare); vector multimap& operator=(</pre>
size_type erase(const key_type&); void clear();	reverse_iterator rbegin(); const_reverse_iterator rbegin() const;	bCacheHashCode = false> class hash_multimap	void insert(InputIterator first, InputIterator last);	const vector_multimap&); void swap(this_type& x);
iterator find (const key_type& k)	<pre>reverse_iterator rend(); const_reverse_iterator rend() const;</pre>	<pre>: public hashtable<, bMutableIterators=true, bUniqueKeys=false></pre>	<pre>iterator erase(iterator); iterator erase(iterator, iterator);</pre>	<pre>key_compare key_comp() const; value_compare value_comp() const;</pre>
const_iterator find(const key_type& k) iterator find_as(const U&,	size_type size() const;	<pre>Key pair<const key,="" t=""></const></pre>	<pre>size_type erase(const key_type&); void clear();</pre>	<pre>pair<iterator, bool=""> insert(const value_type& value);</iterator,></pre>
BinaryPredicate const_iterator find_as(const_U%,	<pre>insert_return_type insert(const Key% key) insert_return_type insert(const value_type%); iterator insert(iterator pos,</pre>	hash_node <value_type, bcachehashcode=""></value_type,>	<pre>void</pre>	<pre>iterator insert(iterator pos,</pre>
BinaryPredicate iterator find_as(const U& u); const iterator find_as(const U& u) con	<pre>const;</pre>		<pre>iterator const_iterator iterator</pre> find(const key_type& k); find(const key_type& k) const; find as(const U&,	void insert(InputIterator first, InputIterator last);
size_type count(const key_type& k		<pre>node_iterator node_iterator const local_iterator;</pre>	UHash, BinaryPredicate); const iterator find as(const U%,	iterator erase(iterator pos); iterator erase(iterator first,
<pre>pair<iterator, iterator=""></iterator,></pre>	`iterator last);	hashtable_iterator <bconst=false,> iterator; hashtable iterator<bconst=true,></bconst=true,></bconst=false,>	UHash, BinaryPredicate) const; iterator find as(const U& u);	iterator last);
equal_range(const key_type&	const; size_type erase(const Key& k);	const_iterator; mapped_type;	const_iterator find_as (const U& u) const;	<pre>iterator find(const key_type& k); const_iterator find(const key_type& k) const;</pre>
<pre>bool validate() const; int validate_iterator(const_iterator i Global Functions & Operators (+ Containers Con</pre>	non)	Public Member Functions hash_multimap(const allocator_type& allocator = eastl::allocator("EASTL classname");	const_iterator find_by_hash (hash_code_t) const	BinaryPredicate);
<pre>void swap(intrusive_hashtable<>& a,</pre>	iterator find(const key_type& key); const_iterator find(const key_type& key) const iterator find_as(const U8 u,	hash multiman(pair <iterator, iterator=""></iterator,>	const_iterator find_as(const U% u,
<pre>template <key, compare="less<Key" t,="">,</key,></pre>	Compare2 compare2); const_iterator find_as(const U% u, Compare2 compare2) cons:	const Predicate& predicate = Predicate(),	<pre>equal_range(const Key% k); pair<const_iterator, const_iterator=""></const_iterator,></pre>	
<pre>class multimap : public rbtree<, bMutableIterators=true,</pre>	<pre>iterator lower_bound(const key_type&); const iterator lower_bound(const key_type&) cons</pre>	hash_multimap(ForwardIterator first,	7& operator[](const Key& key);	<pre>const_iterator lower_bound(const key_type&) const; iterator upper_bound(const key_type&); const iterator upper_bound(const key type&) const;</pre>
bUniqueKeys=false> Public Types	<pre>iterator upper_bound(const key_type&); const_iterator upper_bound(const key_type&) const</pre>	const hasha hashi unction = hash(),	<pre>Hash hash function() const; Predicate equal_function();</pre>	<pre>pair<iterator, iterator=""></iterator,></pre>
<pre>Key pair<const key,="" t=""> rbtree_node<value_type></value_type></const></pre>	<pre>size_type</pre>	<pre>const Predicate& predicate = Predicate(), const allocator_type& allocator =</pre>	int validate() const; int validate_iterator(const_iterator i) const	pair <const_iterator, const_iterator=""></const_iterator,>
<pre>value_type& reference; const value_type& const_referenc rbtree_iterator<value_type,< td=""><td>equal range(const Key& k).</td><td>allocator_type& <pre>get_allocator(); void</pre></td><td>Global Functions & Operators (+ Containers Common) void swap(hashtable<>& a, hashtable<>& b);</td><td><pre>pair<const_iterator, const_iterator=""></const_iterator,></pre></td></value_type,<></pre>	equal range(const Key& k).	allocator_type& <pre>get_allocator(); void</pre>	Global Functions & Operators (+ Containers Common) void swap(hashtable<>& a, hashtable<>& b);	<pre>pair<const_iterator, const_iterator=""></const_iterator,></pre>
<pre>value_type&> iterator; rbtree_iterator<value_type,< pre=""></value_type,<></pre>	<pre>pair<iterator, iterator=""></iterator,></pre>	this_type& operator=(const this_type&); void swap(this_type& x);	eastl::allocator, RandomAccessContainer =	
const value_type*, const value_type&> const_iterator	equal_range_small(const Key& k) const;	<pre>iterator</pre>	<pre>vector<pair<key, t="">, Allocator> > class vector_multimap : public RandomAccessContainer</pair<key,></pre>	void set_allocator(
reverse_iterator, const_reverse_iterator	<pre>bool validate() const; int validate_iterator(const_iterator i) const</pre>	iterator end(); const_iterator end() const;	Public Types Key key_type;	<pre>iterator</pre>
<pre>Compare key_compare; iterator insert_return_</pre>	Global Functions & Operators (+ Containers Common) void swap(rbtree<>& a,	local_iterator	pair <const key,="" t=""> value_type; Compare key compare;</const>	<pre>const_iterator</pre>
T mapped_type; Public Member Functions	template <key, bool<="" nodecount,="" size_t="" t,="" td=""><td>const_local_iterator end(size_type) const bool empty() const;</td><td>value_compare; value type& reference;</td><td>reverse_iterator rend(); const_reverse_iterator rend() const;</td></key,>	const_local_iterator end(size_type) const bool empty() const;	value_compare; value type& reference;	reverse_iterator rend(); const_reverse_iterator rend() const;
<pre>multimap(const allocator_type& allocator = eastl::allocator("EASTL classname"); multimap(const Compare& compare,</pre>	<pre>bEnableOverflow = true, Compare = less<key>, Allocator = eastl::allocator></key></pre>	size_type size() const;	const_value_type& const_reference; T mapped_type;	bool empty() const; size_type size() const;
const allocator type& allocator = eastl::allocator("FASTL classname"); multimap(const this type& x);	<pre>class fixed_multimap : multimap<key,t,compare,fixed_node_pool<> ></key,t,compare,fixed_node_pool<></pre>	size_type bucket_count() const size_type bucket_size(size_type n) const	pointer, const_pointer, iterator, const_iterator, reverse_iterator, const_reverse_iterator	void clear(); Global Functions & Operators (+ Containers Common)
multimap(tonst this_typeα x), multimap(Iterator itBegin,	Has all the multimap functionality. Public Member Functions fixed multimap();	float load_factor() const float get_max_load_factor() const; void set_max_load_factor(float	Public Member Functions vector_multimap();	<pre>void swap(vector_multimap<>& a, vector_multimap<>& b);</pre>
allocator_type& get_allocator(); void set_allocator(<pre>fixed_multimap(const Compare& compare); fixed_multimap(const this_type& x);</pre>	fMaxLoadFactor);	<pre>vector_multimap(const allocator_type& allocator); vector_multimap(const key compare& comp,</pre>	

EASTL 1.10.03 Quick Reference v1.0

template <key, t,<br="">hash<key>, Equal</key></key,>	size_t bucketCount, Hash =	size_type count(const key_type& k) cons	; Iterator erase(iterator pos); Iterator erase(iterator first,	Public Member Functions	<pre>pair<iterator, iterator=""></iterator,></pre>
class intrusive h		pair <iterator, iterator=""></iterator,>	iterator last);	<pre>hash_set(const allocator_type& allocator =</pre>	pair <const_iterator, const_iterator=""></const_iterator,>
: intrusive_hasht	able <key, equal,<="" hash,="" td="" value="T,"><td><pre>equal_range(const key_type&); pair<const const="" iterator="" iterator,=""></const></pre></td><td><pre>void erase(const key_type* first,</pre></td><td><pre>hash_set(size_type</pre></td><td><pre>equal_range(const Value&) const;</pre></td></key,>	<pre>equal_range(const key_type&); pair<const const="" iterator="" iterator,=""></const></pre>	<pre>void erase(const key_type* first,</pre>	<pre>hash_set(size_type</pre>	<pre>equal_range(const Value&) const;</pre>
bucketCount, bCon bUniqueKeys=false	stIterators=false, >	equal_range(const key_type&) const;	size_type erase(const <i>Key</i> & k);	const Hash& hashFunction = Hash(), const Predicate& predicate = Predicate(),	Hash hash_function() const;
7. 9.		bool validate() const;	void clear();	<pre>const allocator_type& allocator =</pre>	Predicate equal_function();
	Public Types	int validate_iterator(const_iterator i) cons		eastl::allocator("EASTL classname") hash set(ForwardIterator first,	<pre>bool validate() const; int validate iterator(const iterator i) const;</pre>
Key Value	key_type; value_type;	Global Functions & Operators (+ Containers Common)	iterator find(const key type& key);	ForwardIterator last, size type nBucketCount = 0,	int validate_iterator (const_iterator i) const; Global Functions & Operators (+ Containers Common)
Value	node_type;	<pre>void swap(intrusive_hashtable<>& a, intrusive hashtable<>& b);</pre>	<pre>const_iterator find(const key_type& key) const;</pre>	const Hash& hashFunction = Hash(),	void swap(hashtable<>& a,
iterator value type&	<pre>insert_return_type; reference;</pre>		<pre>iterator find_as(const U& u,</pre>	<pre>const Predicate& predicate = Predicate(), const allocator type& allocator =</pre>	hashtable<>& b);
const value type&	<pre>const_reference;</pre>	template < <i>Key</i> , <i>Compare</i> = less< <i>Key</i> >,	<pre>const_iterator find_as(const U& u,</pre>	eastl::allocator("EASTL classname");	
intrusive_node_it	erator <value_type, bconst="false"> local iterator;</value_type,>	Allocator = eastl::allocator>		allocator type& <pre>get_allocator();</pre>	<pre>template <value, +="" 1,="" benableoverflow<="" bool="" bucketcount="nodeCount" nodecount,="" pre="" size_t=""></value,></pre>
intrusive_node_it	erator <value_type, bconst="true"></value_type,>	<pre>class set : public rbtree<, bMutableIterators=false,</pre>	<pre>iterator lower_bound(const key_type&); const iterator lower bound(const key type&) const;</pre>	void set_allocator(= true, Hash = hash <value>, Predicate =</value>
const_local_itera	tor;	bUniqueKeys=true>	<pre>iterator upper_bound(const key_type&);</pre>	<pre>const allocator_type&);</pre>	equal_to <value>, bool bCacheHashCode = false, Allocator = eastl::allocator></value>
	le_iterator <value_type, nst=false> iterator;</value_type, 	Public Types	<pre>const_iterator upper_bound(const key_type&) const;</pre>	this_type& operator =(const this_type&);	class fixed hash set
	le_iterator <value_type,< td=""><td>Key key_type; Key value_type;</td><td>size_type count(const Key& k) const;</td><td><pre>void swap(this_type& x);</pre></td><td>: public hash_set<value, hash,="" predicate,<="" td=""></value,></td></value_type,<>	Key key_type; Key value_type;	size_type count(const Key& k) const;	<pre>void swap(this_type& x);</pre>	: public hash_set <value, hash,="" predicate,<="" td=""></value,>
bCo Value		rbtree_node <value_type> node_type;</value_type>	pair <iterator, iterator=""></iterator,>	<pre>iterator begin();</pre>	fixed_hashtable_allocator<>, bCacheHashCode>
	«Value, Key» extract_key;	<pre>value_type& reference; const value type& const_reference;</pre>	<pre>equal_range(const Key& k);</pre>	<pre>const_iterator begin() const; iterator end();</pre>	Has all the hash_set functionality. fixed hash set(
Pu	blic Member Functions	rbtree_iterator <value_type,< td=""><td><pre>pair<const_iterator, const_iterator=""></const_iterator,></pre></td><td>const_iterator end() const;</td><td>const Hash& hashFunction = Hash(),</td></value_type,<>	<pre>pair<const_iterator, const_iterator=""></const_iterator,></pre>	const_iterator end() const;	const Hash& hashFunction = Hash(),
intrusive_hash_mu	<pre>ltimap(const Hash& h = Hash(),</pre>	const value_type*, const value type&>		local iterator begin (size type n)	<pre>const Predicate& predicate = Predicate()); fixed hash set(</pre>
Equal());	const <i>Equal</i> & eq =	iterator;	<pre>bool validate() const; int validate_iterator(const iterator i) const;</pre>	local_iterator	- InputIterator first,
void	<pre>swap(this type& x);</pre>	<pre>rbtree_iterator<value_type,< td=""><td>Global Functions & Operators (+ Containers Common)</td><td><pre>const_local_iterator const local iterator const_local_iterator cons</pre></td><td><pre>InputIterator last, const Hash& hashFunction = Hash(),</pre></td></value_type,<></pre>	Global Functions & Operators (+ Containers Common)	<pre>const_local_iterator const local iterator const_local_iterator cons</pre>	<pre>InputIterator last, const Hash& hashFunction = Hash(),</pre>
		const value_type&>	void swap (rbtree<>& a,	hool matu() const.	<pre>const Predicate& predicate = Predicate());</pre>
iterator const iterator	<pre>begin(); begin() const;</pre>	const_iterator;	rbtree<>& b);	bool empty() const; size_type size() const;	<pre>size_type max_size() const;</pre>
iterator	end();	reverse_iterator, const_reverse_iterator	template <key, bool<="" nodecount,="" size="" t="" td=""><td>size type bucket count() const;</td><td></td></key,>	size type bucket count() const;	
const_iterator	<pre>end() const;</pre>	Compare key_compare;	bEnableOverflow = true, Compare = less <key>,</key>	size_type bucket_size(size_type n) const;	template <t, bucketcount,="" hash="hash<T" size="" t="">,</t,>
local_iterator	<pre>begin(size type n)</pre>	<pre>pair<iterator, bool=""> insert_return_type;</iterator,></pre>	Allocator = eastl::allocator>		Equal = equal to <t> ></t>
		DIEM I E «	class fixed set	float load factor() const:	class intrusive hach set
local_iterator const local itera	<pre>end(size_type) tor begin(size type n)</pre>	Public Member Functions	<pre>class fixed_set :set<key,compare,fixed node="" pool<=""> ></key,compare,fixed></pre>	<pre>float load_factor() const; float get_max_load_factor() const;</pre>	<pre>class intrusive_hash_set : intrusive hashtable</pre> Key=T, Value=T, Hash, Equal,
<pre>const_local_itera const</pre>	tor begin (size_type n)	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname");</pre>	<pre>class fixed_set :set<key,compare,fixed_node_pool<> ></key,compare,fixed_node_pool<></pre>		: intrusive_hashtablecKey=T, Value=T, Hash, Equal, bucketCount, bConstIterators=true,
const_local_itera	tor begin (size_type n)	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare,</pre>	:set< <i>Key</i> , <i>Compare</i> ,fixed_node_pool<> >	<pre>float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy_type&</pre>	: intrusive_hashtable <key=t, equal,<br="" hash,="" value="T,">bucketCount, bConstIterators=true, bUniqueKeys=true></key=t,>
<pre>const_local_itera const const_local_itera bool</pre>	<pre>tor begin(size_type n) tor end(size_type) const empty() const;</pre>	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = eastl::allocator("EASTL classname");</pre>	:set <key,compare,fixed_node_pool<> > Has all the set functionality. Public Member Functions fixed_set();</key,compare,fixed_node_pool<>	<pre>float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor);</pre>	: intrusive_hashtablecKey=T, Value=T, Hash, Equal, bucketCount, bConstIterators=true,
<pre>const_local_itera const const_local_itera bool size_type</pre>	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const;</pre>	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x);</pre>	:set <key,compare,fixed_node_pool<> Has all the set functionality. Public Member Functions fixed_set(); fixed_set(const Compare& compare);</key,compare,fixed_node_pool<>	<pre>float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy_type&</pre>	: intrusive_hashtable <key=t, bconstiterators="true," bucketcount,="" buniquekeys="true" equal,="" hash,="" value="T,"> Public Types T key_type; Value value_type;</key=t,>
<pre>const_local_itera const const_local_itera bool size_type size_type</pre>	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const;</pre>	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = eastl::allocator("EASTL classname");</pre>	:set <key,compare,fixed_node_nool<> Has all the set functionality. Public Member Functions fixed_set(); fixed_set(const Compare& compare); fixed_set(const this_type& x); fixed_set(InputIterator first,</key,compare,fixed_node_nool<>	<pre>float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy_type&</pre>	: intrusive_hashtable <key=t, bconstiterators="true," bucketcount,="" buniquekeys="true" equal,="" hash,="" value="T,"> Public Types T</key=t,>
const_local_itera const const_local_itera bool size_type size_type size_type const;	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n)</pre>	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itBegin, Iterator itEnd);</pre>	<pre>:set<key,compare,fixed_node_nool<> > Has all the set functionality. Public Member Functions fixed_set(); fixed_set(const Compare& compare); fixed_set(const this_type& x);</key,compare,fixed_node_nool<></pre>	<pre>float</pre>	: intrusive_hashtable <key=t, bconstiterators="true," bucketcount,="" buniquekeys="true" equal,="" hash,="" value="T,"> Public Types T</key=t,>
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const;</pre>	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itBegin, Iterator itEnd); allocator_type& get_allocator(); yoid set allocator()</pre>	<pre>:set<key,compare,fixed_node_nool<> > Has all the set functionality. Public Member Functions fixed_set(); fixed_set(const Compare& compare); fixed_set(const this type& x); fixed_set(InputIterator first, InputIterator last); size_type max_size() const;</key,compare,fixed_node_nool<></pre>	float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy_type&	: intrusive_hashtable <key=t, bconstiterators="true," bucketcount,="" buniquekeys="true" equal,="" hash,="" value="T,"> Public Types T</key=t,>
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type const;	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k)</pre>	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itBegin, Iterator itEnd); allocator_type& get_allocator();</pre>	<pre>:set<key,compare,fixed_node_nool<> > Has all the set functionality. Public Member Functions fixed_set(); fixed_set(const Compare% compare); fixed_set(const this type% x); fixed_set(InputIterator first, InputIterator last); size_type max_size() const;</key,compare,fixed_node_nool<></pre>	float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy_type&	: intrusive_hashtable <key=t, bconstiterators="true," bucketcount,="" buniquekeys="true" equal,="" hash,="" value="T,"> Public Types T</key=t,>
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n)</pre>	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itBegin, Iterator itEnd); allocator_type& get_allocator(); void</pre>	:set <key,compare,fixed_node_nool<> Has all the set functionality. Public Member Functions fixed_set(); fixed_set(const Compare& compare); fixed_set(const this_type& x); fixed_set(InputIterator first,</key,compare,fixed_node_nool<>	float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy_type&	: intrusive_hashtable <key=t, bconstiterators="true," bucketcount,="" buniquekeys="true" equal,="" hash,="" value="T,"> Public Types T</key=t,>
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value);</pre>	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itBegin, Iterator itEnd); allocator_type&</pre>	<pre>:set<key,compare,fixed_node_nool<> ></key,compare,fixed_node_nool<></pre>	float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy type&	: intrusive_hashtable <key=t, bconstiterators="true," bucketcount,="" buniquekeys="true" equal,="" hash,="" value="T,"> Public Types 7</key=t,>
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ insert_return_typ	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const_iterator,</pre>	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itBegin, Iterator itEnd); allocator_type& get_allocator(); void</pre>	<pre>:set<key,compare,fixed_node_nool<> ></key,compare,fixed_node_nool<></pre>	float get_max_load_factor() const; void set_max_load_factor() const; const rehash_policy_type&	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types T Value Value type; Value pair <iterator, bool=""> value type% const value_type& const value_type& intrusive_node_iterator bConst=true> local_iterator, bConst=true> intrusive_node_iterator const_local_iterator; intrusive_hashtable_iterator const_iterator; const_iterator; const_iterator; const_iterator;</iterator,>
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(InputIterator first,	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEnd); allocator_type&</pre>	<pre>:set<key,compare,fixed_node_nool<> ></key,compare,fixed_node_nool<></pre>	float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy type&	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types T Value Value type; value_type; value_type; value_type; value_type& const value_type& intrusive_node_iterator bConst=true> const_local_iterator; intrusive_hashtable_iterator Value Value Value bConst=true> iterator; const_iterator; value_type, bConst=true> iterator; const_iterator; value Walue Value wapped_type;
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ insert_return_typ	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const_iterator,	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEegin, Iterator itEnd); allocator_type& get_allocator(); soid</pre>	<pre>:set<key,compare,fixed_node_pool<></key,compare,fixed_node_pool<></pre>	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 value_type&); iterator insert(const_iterator, const value_type&); void insert(InputIterator first, InputIterator last); iterator erase(iterator); iterator erase(iterator, iterator); size_type erase(const_key_type&); void clear();	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types 7 key_type; value_type; node_type; insert_return_type; reference; const_value_type& const_value_type& intrusive_node_iterator bConst=true> const_reference; intrusive_hashtable_iterator const_iterator; intrusive_hashtable_iterator const_iterator; const_iterator; wapped_type;
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_type insert_return_type void iterator	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const iterator,	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEnd); allocator_type&</pre>	<pre>:set<key,compare,fixed_node_nool<> ></key,compare,fixed_node_nool<></pre>	float get_max_load_factor() const; void set_max_load_factor() const; void rehash_policy() const void rehash_policy() const void rehash_policy() const rehash_policy_type&	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types T Value Value type; value type; value type& const_return_type; intrusive_node_iterator bConst=true> const_local_iterator; intrusive_hashtable_iterator value bConst=true> iterator, const_iterator; intrusive_hashtable_iterator value walue walue bConst=true> intrusive_hashtable_iterator const_iterator; const_iterator; walue wapped_type; extract_key; Public Member Functions intrusive_hash_set(const_Hash& h = Hash(),
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ insert_return_typ	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const_iterator,	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEnd); allocator_type& get_allocator(); soid</pre>	<pre>:set<key,compare,fixed_node_pool<></key,compare,fixed_node_pool<></pre>	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 value_type&); iterator insert(const_iterator, const value_type&); void insert(InputIterator first, InputIterator last); iterator erase(iterator); iterator erase(iterator, iterator); size_type erase(const key_type&); void clear(); void reset(); void reset(); void reset(); void reset(); iterator find(const key_type& k); iterator find(const key_type& k) const; iterator find(const key_type& k) const;	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types T Value Value value_type; value_type; value_type; value_type; value_type const return_type; reference; const value_type& const_intrusive_node_iterator bConst=true> bConst=true> const_intrusive_hashtable_iterator intrusive_hashtable_iterator const_iterator; const_iterator; walue walue wapped_type; extract_key; Public Member Functions
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ insert_return_typ void iterator iterator size_type	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const_iterator,	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEnd); allocator_type&</pre>	<pre>:set<key,compare,fixed_node_nool<> Has all the set functionality. Public Member Functions fixed_set(); fixed_set(const Compare& compare); fixed_set(const this_type& x); fixed_set(InputIterator first,</key,compare,fixed_node_nool<></pre>	float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy() type&	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types T Value Value type; value type; value type& const_return_type; intrusive_node_iterator bConst=true> const_local_iterator; intrusive_hashtable_iterator value bConst=true> iterator, const_iterator; intrusive_hashtable_iterator value walue walue bConst=true> intrusive_hashtable_iterator const_iterator; const_iterator; walue wapped_type; extract_key; Public Member Functions intrusive_hash_set(const_Hash& h = Hash(),
const_local_iteraconst const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_type insert_return_type void iterator iterator size_type void	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const_iterator,</pre>	<pre>set(const allocator_type& allocator = east1::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = east1::allocator("EASTL classname"); set(const this_type& x); set(Iterator itBegin, Iterator itEnd); allocator_type& get_allocator(); set_allocator(const allocator_type&) this_type& operator=(const this_type& x); iterator const_iterator begin(); const_iterator end(); const_iterator end(); const_reverse_iterator reverse; therator reverse_iterator rend(); const_reverse_iterator rend(); const</pre>	<pre>:set<key,compare,fixed_node_pool<></key,compare,fixed_node_pool<></pre>	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 value_type&); iterator insert(const value_type&); void insert(InputIterator first, InputIterator fast); iterator erase(iterator); iterator erase(iterator, iterator); size_type erase(const key_type&); void clear(); void reset(); void reset(); void reset(); void reset(); find(const key_type& k); iterator find(const key_type& k) const; iterator iterator iterator, BinaryPredicate);	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types T
const_local_itera const const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ insert_return_typ void iterator iterator size_type void iterator	<pre>tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(value_type& value); insert(InputIterator first,</pre>	<pre>set(const allocator_type& allocator = east1::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = east1::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEnd); allocator_type&</pre>	<pre>:set<key,compare,fixed_node_nool<></key,compare,fixed_node_nool<></pre>	float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy() type&	: intrusive_hashtable bucketCount, bConstIterators=true, bulic Types T Value Value public Types Value value type; value type; value type value type& const_return_type; intrusive_node_iterator bConst=true> local_iterator, bConst=true> const_local_iterator; intrusive_hashtable_iterator value walue walue pConst=true> intrusive_hashtable_iterator const_iterator; const_iterator; walue wapped_type; extract_key; Public Member Functions intrusive_hash_set(const Hash& h = Hash(), const Equal& eq = Equal()); void swap(this_type& x); iterator begin(); const_iterator begin() const;
const_local_iteraconst const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_type insert_return_type void iterator iterator size_type void	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const_iterator,	<pre>set(const allocator_type& allocator = east1::allocator("EASTL classname"); set(const Compare& compare, const allocator type& allocator = east1::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEnd); allocator_type& get_allocator(); set_allocator(</pre>	:set <key,compare,fixed_node_pool<> Has all the set functionality. Public Member Functions fixed_set(); fixed_set(const Compare& compare); fixed_set(const this type& x); fixed_set(InputIterator first,</key,compare,fixed_node_pool<>	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 value_type&); iterator insert(const value_type&); void insert(InputIterator first, InputIterator fast); iterator erase(iterator); iterator erase(iterator, iterator); size_type erase(const key_type&); void clear(); void reset(); void re	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types T
const_local_iteraconst const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ insert_return_typ void iterator iterator size_type void iterator const_iterator iterator const_iterator iterator	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const_iterator,	<pre>set(const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = eastl::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEnd); allocator_type& get_allocator(); soid set_allocator(); set_allocator(const allocator_type&) this_type& operator=(const this_type& x); iterator semple(); iterator begin(); const_iterator end(); const_iterator end(); const_iterator reverse_iterator red(); const_reverse_iterator red(); const_reverse_iterator rend(); const_reverse_iterator rend() const_reverse_iterator rend(); const_reverse_iterator rend() const; iterator insert(const_value_type& value); iterator insert(iterator pos,</pre>	:set <key,compare,fixed_node_pool<> Has all the set functionality. Public Member Functions fixed_set(); fixed_set(const Compare& compare); fixed_set(const this_type& x); fixed_set(InputIterator first,</key,compare,fixed_node_pool<>	float get_max_load_factor() const; void set_max_load_factor(float fMaxLoadFactor); const rehash_policy() type&	: intrusive_hashtable bucketCount, bConstIterators=true, bulniqueKeys=true> Public Types T
const_local_iteraconst const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ insert_return_typ void iterator iterator size_type void iterator const_iterator	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const_iterator,	<pre>set(const allocator_type& allocator = east1::allocator("EASTL classname"); set(const Compare& compare, const allocator type& allocator = east1::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEnd); allocator_type& get_allocator(); set_allocator(</pre>	<pre>:set<key,compare,fixed_node_pool<></key,compare,fixed_node_pool<></pre>	float get_max_load_factor() const; void set_max_load_factor() const; void rehash_policy() const void rehash_policy() const void rehash_policy() const rehash_policy_type&	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types T
const_local_iteraconst const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ insert_return_typ void iterator iterator size_type void iterator const_iterator iterator const_iterator	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const_iterator,	<pre>set(const allocator_type& allocator = east1::allocator("EASTL classname"); set(const Compare& compare, const allocator_type& allocator = east1::allocator("EASTL classname"); set(const this_type& x); set(Iterator itBegin, Iterator itEnd); allocator_type& get_allocator(); set_allocator(const allocator_type&) this_type& operator=(const this_type& x); iterator const iterator</pre>	<pre>:set<key,compare,fixed_node_nool<></key,compare,fixed_node_nool<></pre>	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 value_type&); iterator insert(const_iterator, const value_type&); void insert(InputIterator first, InputIterator last); iterator erase(iterator); iterator erase(iterator, iterator); size_type erase(const key_type&); void clear(); void rehash(size_type nBucketCount); iterator find(const key_type& k) const; iterator find(const key_type& k) const; iterator find_as(const U&, UHash, BinaryPredicate); const_iterator find_as(const U&, BinaryPredicate) const; iterator find_as(const U& u); const_iterator find_as(const U& u); find_as(const U& u) const;	: intrusive_hashtable bucketCount, bConstIterators=true, bulic Types T Value Value public Types Value value type; value type; value type& const_return_type; intrusive_node_iterator bConst=true> local_iterator, const_local_iterator; intrusive_hashtable_iterator value walue mapped_type; const_local_iterator; intrusive_hashtable_iterator const_iterator; const_iterator; mapped_type; use_self value mapped_type; extract_key; Public Member Functions intrusive_hash_set(const Hash& h = Hash(), const Equal& eq = Equal()); void swap(this_type& x); iterator const_iterator iterator const_iterator iterator const_iterator iterator const_iterator iterator const_iterator iterator const_iterator iterator const_iterator iterator const_iterator iterator const_iterator iterator const_iterato
const_local_iteraconst const_local_itera bool size_type size_type size_type const; size_type const; float insert_return_typ insert_return_typ void iterator iterator size_type void iterator const_iterator iterator const_iterator iterator	tor begin(size_type n) tor end(size_type) const empty() const; size() const; bucket_count() const; bucket_size(size_type n) bucket(const key_type& k) load_factor() const; e insert(value_type& value); e insert(const iterator,	<pre>set(const allocator_type& allocator = east1::allocator("EASTL classname"); set(const Compare& compare, const allocator type& allocator = east1::allocator("EASTL classname"); set(const this_type& x); set(Iterator itEnd); allocator_type& get_allocator(); set_allocator(</pre>	<pre>:set<key,compare,fixed_node_nool<></key,compare,fixed_node_nool<></pre>	float get_max_load_factor() const; void set_max_load_factor() const; void rehash_policy() const void rehash_policy() const void rehash_policy() const rehash_policy_type&	: intrusive_hashtable bucketCount, bConstIterators=true, bUniqueKeys=true> Public Types T

hool amatu() const.	hypertax sat& energias (sanst yester	set& x); rbtree iterator <va< th=""><th>iluo tymo</th><th>pair<iterator, iterator=""></iterator,></th><th></th><th>litaratar</th><th>hogin().</th></va<>	iluo tymo	pair <iterator, iterator=""></iterator,>		litaratar	hogin().
<pre>bool</pre>	<pre>vector_set& operator=(const vector void</pre>		nst value type*,		mall(const <i>Key</i> & k);	iterator const iterator	<pre>begin(); begin() const;</pre>
	= 7		onst value_type&>	pair <const_iterator, cons<="" td=""><td>_iterator></td><td>iterator</td><td>end();</td></const_iterator,>	_iterator>	iterator	end();
size_type bucket_count() cons size type bucket_size(size ty		rbtree iterator <va< td=""><td>iterator;</td><td>equal_range_s</td><td>mall(const <i>Key</i>& k) const;</td><td>const_iterator</td><td><pre>end() const;</pre></td></va<>	iterator;	equal_range_s	mall(const <i>Key</i> & k) const;	const_iterator	<pre>end() const;</pre>
size_type bucket(const key_ty			onst value type*,	bool validate() const;		local iterator	<pre>begin(size type n)</pre>
	pair <iterator, bool=""></iterator,>		onst value_type&>	_	const_iterator i) const;	local_iterator	<pre>end(size_type)</pre>
float load_factor() const	<pre>insert(const value_type8</pre>	(value);	const_iterator;	Global Functions & Operato	rs (+ Containers Common)	<pre>const_local_itera const</pre>	tor begin (size_type n)
insert return type insert(value type	value); iterator insert(iterator		const_reverse_iterator	void swap (rbtree<>& a,		const local itera	etor end(size type) const
insert return type insert(const itera	or, const value_type8			rbtree<>& b);			
value_type8 void insert(InputIterat		Compare iterator	key_compare; insert_return_type;	11	6 / .	bool empty () size_type size ()	const;
InputItera		±\´.	olic Member Functions	template <key, node<br="" size_t="">bEnableOverflow = true, Co</key,>			
:+	iterator erase (iterator po	multicat/const all	ocator type& allocator =	Allocator = eastl::alloca		size_type bucket_	
<pre>iterator erase(iterator); iterator erase(iterator, iterator, iterator,</pre>		rst. eastl::al	.locator("EASTL classname");	class fixed_multiset		Size_type bucket_	_size (size_type n) const
size_type erase(const key_ty	e&); iterator la	ist); multiset(const Com	npare& compare, .ocator_type& allocator =	:multiset <key,compare,fix< td=""><td>d_node_pool<>></td><td></td><td>cor() const</td></key,compare,fix<>	d_node_pool<>>		cor() const
<pre>void clear();</pre>	size_type erase (const key_type& k)	' eastl::al	.locator("EASTL classname");	Has all the multi	set functionality.		load_factor() const;
voiu cieai(),	iterator find (const key type	& k); multiset(const thi	s_type& x);	Public Memb	er Functions	VOIG SEL_MAX_I	<pre>load_factor(float fMaxLoadFactor);</pre>
iterator find (const key_typ		& k) const; multiset(Iterator Iterator		<pre>fixed_multiset();</pre>		const rehash_poli	
const_iterator find(const key_typeriterator find as(const U&,	& k) const; iterator find_as (const <i>U</i> &		, ,	<pre>fixed_multiset(const Compo fixed multiset(const this</pre>			<pre>policy() const policy(const rehash policy type&</pre>
UHash,	BinaryPredicate		<pre>get_allocator();</pre>	fixed multiset(InputItera		voiu ichushi	rehashPolicy);
BinaryPred	cate); const_iterator find_as(const U& BinaryPredicate	u,	<pre>set_allocator(</pre>	InputItera	or last);	incort roturn tur	on incort(const value type):
const_iterator find_as (const U&,	Bindiyriedicate			size_type max_size() cons	::	iterator	<pre>pe insert(const value_type&); insert(const iterator,</pre>
BinaryPred	<i>cate</i>) const; size_type count (const key_type	be& k); this_type& void	<pre>operator=(const this_type&); swap(this type& x);</pre>		•		const_value_type&);
iterator find_as(const U& u const_iterator find_as(const U& u			Shap (chis_cypea x);	template <value, hash="ha</td"><td>ish<value>. Predicate =</value></td><td>void</td><td><pre>insert(InputIterator first,</pre></td></value,>	ish <value>. Predicate =</value>	void	<pre>insert(InputIterator first,</pre>
tina_us(const ou t	const_iterator lower_bound(const	rev_tyne%) const: literator	begin();	equal_to <value>, Allocato</value>	= eastl::allocator,		inputitefutor last),
size_type count (const <i>key_t</i>)			<pre>begin() const; end();</pre>	bool bCacheHashCode = fal:	ie>	iterator	erase(iterator);
pair <iterator, iterator=""></iterator,>	const_iterator upper_bound(const k	const_iterator	end() const;	<pre>class hash_multiset : public hashtable<, bMu</pre>	abloItorators-falso	iterator size_type	<pre>erase(iterator, iterator); erase(const key type&);</pre>
equal_range(const key_		reverse_iterator const reverse iter	<pre>rbegin(); rator rbegin() const;</pre>	bUniqueKeys=false>	.abieiterators=raise,		
pair <const_iterator, const_iterator=""></const_iterator,>	equal_range(const equal_range(const pair <const const="" iterator,="" iterator<="" td=""><td>cy_typea),, -</td><td>rend();</td><td>Public</td><td>Types</td><td>void void</td><td><pre>clear(); reset();</pre></td></const>	cy_typea),, -	rend();	Public	Types	void void	<pre>clear(); reset();</pre>
0 ,	equal_range(const		rator rend() const;	Value	value_type;	void	<pre>rehash(size_type nBucketCount);</pre>
<pre>bool validate() const; int validate_iterator(const ite</pre>	Inherited from base class, Random/	AccessContainer		hash_node <value_type, bca<="" td=""><td></td><td>iterator</td><td><pre>find(const key_type& k);</pre></td></value_type,>		iterator	<pre>find(const key_type& k);</pre>
const;	allocator_type& get_allocat		empty() const;	iterator	<pre>node_type; insert_return_type;</pre>	const iterator	find(const key_type& k); find(const key type& k) const;
Global Functions & Operators (+ Contain	rs Common) void set_allocat	or(size_type sllocator type&);	<pre>size() const;</pre>	value_type&	reference;	iterator	<pre>find_as(const U&,</pre>
void swap (intrusive_hashtable<>& a,		insert_return_type	<pre>insert(const value_type&);</pre>	const value_type&	const_reference;		UHash, BinaryPredicate);
intrusive_hashtable<>& b);	iterator begind const iterator begind	iterator const;	<pre>insert(iterator pos,</pre>	node_iterator <bconst=true< td=""><td></td><td>const_iterator</td><td>find_as(const U&,</td></bconst=true<>		const_iterator	find_as(const U&,
	iterator end():		const value_typea	<pre>node_iterator<bconst=true const="" iterator;<="" local="" pre=""></bconst=true></pre>	>		UHash, BinaryPredicate) const;
<pre>template <key, compare="less<Key">, A eastl::allocator, RandomAccessContain</key,></pre>	locator = const_iterator end()	const; void	<pre>insert(InputIterator first,</pre>	hashtable iterator <bconst:< td=""><td>true,></td><td>iterator</td><td>find as(const U& u);</td></bconst:<>	true,>	iterator	find as(const U& u);
vector <key, allocator=""> ></key,>	10,0130_1010101 10081	l(); l() const;	<pre>InputIterator last);</pre>	_	iterator;	const_iterator	<pre>find_as(const U& u) const;</pre>
class vector set	reverse_iterator rend()	; Iterator	<pre>erase(iterator pos);</pre>	hashtable_iterator <bconst< td=""><td>const iterator;</td><td>iterator</td><td><pre>find_by_hash(hash code t);</pre></td></bconst<>	const iterator;	iterator	<pre>find_by_hash(hash code t);</pre>
: public RandomAccessContainer	const_reverse_iterator rend()	const; Iterator	erase(iterator first,	Public Memb	-	const_iterator	find_by_hash(hash_code_t) const;
Public Types	bool empty() const	void	<pre>iterator last); erase(const key type* first,</pre>	hash multiset(er i uneciono	ciao tumo	<pre>count(const Value& k) const;</pre>
Key key_type;	<pre>size_type size() const;</pre>		<pre>const key_type* last);</pre>	const allocator_ty		size_type	count (const valuea k) const,
Key value_type Compare key compar		size_type	erase(const Key& k);	eastl::allocator("A	:ASIL classname")	pair <iterator, it<="" td=""><td></td></iterator,>	
Compare value_comp			<pre>clear();</pre>	_ size_type	nBucketCount,	pair <const iterat<="" td=""><td><pre>equal_range(const Value&); cor, const iterator></pre></td></const>	<pre>equal_range(const Value&); cor, const iterator></pre>
value_type& reference; const value_type& const_refe		void	reset();	const Hash&	<pre>hashFunction = Hash(), predicate = Predicate(),</pre>	_	<pre>equal_range(const Value&) const;</pre>
const varue_typea const_rere	vector_set<>& b);	iterator f	ind(const key_type& key);	const allocator_ty		Hash	hash function() const;
pointer, const_poin		const_iterator f	Find(const key_type& key) const;	eastl::allocator("	ASTL classname")	Predicate	equal_function();
iterator, const_iter reverse_iterator, const_reve	co itorator: Lempiale (Key, Compare = less(Key)	',	<pre>find_as(const U& u,</pre>	hash_multiset(ForwardIterator	first,		N
Public Member Functions	- Allocutor = eastr.	const_iterator f	ind_as(const <i>U</i> & u,	ForwardIterator	last,	bool validate int validate	<pre>iterator(const iterator i) const;</pre>
<pre>vector_set();</pre>	<pre>class multiset : public rbtree<, bMutableIterato</pre>	ors=false.	Compare2 compare2) const;	size_type const Hash&	<pre>nBucketCount = 0, hashFunction = Hash(),</pre>	Global Functions	& Operators (+ Containers Common)
vector_set(const allocator_type& allo	ator); bUniqueKevs=false>	iterator low	wer_bound(const key_type&);	const Predicate&	<pre>predicate = Predicate(),</pre>	void swap (hashtab	ole<>& a,
<pre>vector_set(const key_compare& comp const allocator type& allo</pre>			wer_bound(const key_type&) const;	const allocator_ty	<pre>be& allocator = itor("EASTL classname");</pre>	hashtal	ole<>& b);
eastl::allocator(<i>"EASTI</i>	classname"); Key key_ty	pe; const iterator unn	<pre>per_bound(const key_type&); per bound(const key type&) const;</pre>		,		
<pre>vector_set(const vector_set& x);</pre>	Key value rbtree node <value type=""> node_f</value>	type; – ''	_		allocator();	tomplate (Vev. C	omnara - loss/Vaux Allosator
		ence;	<pre>count(const Key& k) const;</pre>	void set	<pre>allocator(const allocator type&);</pre>		mpare = less <key>, Allocator = RandomAccessContainer =</key>
<pre>vector_set(InputIterator first,</pre>							
<pre>InputIterator last);</pre>		reference; pair <iterator, ite<="" td=""><td></td><td></td><td></td><td>vector<key, alloc<="" td=""><td></td></key,></td></iterator,>				vector <key, alloc<="" td=""><td></td></key,>	
	const value_type& const	equal_	erator> _range(const <i>Key</i> & k); _r, const iterator>		rator=(const this_type&); o(this type& x);	<pre>class vector_mu : public RandomAc</pre>	ltiset

Pub	lic Types	iterator	end();		find(const key_type& k) const;	l. ,		Global Functions & (Operators (+ Containers Common)
Кеу	key_type;	const_iterator reverse iterator	<pre>end() const; rbegin();</pre>	iterator	<pre>find_as(const U&,</pre>	bool size type	<pre>empty() const; size() const;</pre>		
Key	value_type;	const reverse ite			BinaryPredicate);	size_type	capacity() const;	template <t, allocat<="" td=""><td>tor = eastl::allocator></td></t,>	tor = eastl::allocator>
Compare Compare	<pre>key_compare; value_compare;</pre>	reverse_iterator	rend();	const_iterator	<pre>find_as(const U&,</pre>	bool	<pre>full() const;</pre>	class vector	
value type&	reference;	const_reverse_ite	rator rend () const;		UHash, BinaryPredicate) const;	void	reside(size type n).	Crubb C C C C	Public Types
const value_type&	<pre>const_reference;</pre>	bool	<pre>empty() const;</pre>	iterator	find as(const U& u);	void	<pre>resize(size_type n); reserve(size type cap);</pre>	464 4	value_type,
pointer,	const pointer,	size type	size() const;		find_as(const U& u) const;			this_type, pointer,	const pointer,
iterator,	const_pointer,		- 0		., .,	reference	<pre>operator[](size_type n);</pre>	reference,	const_reference,
reverse_iterator,	<pre>const_reverse_iterator;</pre>	void	<pre>clear();</pre>	size_type	<pre>count(const key_type& k) const;</pre>	reference	rence operator[](size_type n) const; front();	iterator,	const_iterator,
Public Mer	mber Functions		& Operators (+ Containers Common)	pair <iterator, iter<="" td=""><td>ator></td><td></td><td>erence front() const;</td><td>reverse_iterator,</td><td>const_reverse_iterator</td></iterator,>	ator>		erence front () const;	reverse_iterator,	const_reverse_iterator
<pre>vector_multiset();</pre>		void swap(vector_	multiset<>& a, multiset<>& b);	equa	<pre>1_range(const key_type&);</pre>	reference	back();		c Member Functions
	llocator_type& allocator);	vector_	muitiset(/α b),	pair <const_iterator< td=""><td><pre>, const_iterator> 1 range(const key type&) const;</pre></td><td>const_refe</td><td>erence back() const;</td><td>vector();</td><td>cor type& allocator);</td></const_iterator<>	<pre>, const_iterator> 1 range(const key type&) const;</pre>	const_refe	erence back () const;	vector();	cor type& allocator);
vector_multiset(const ke	ey_compare& comp, llocator type& allocator=		11 1.16. 1 11 1 1.15	equa	1_lange(const key_typea) const,	void push	<pre>back(const value type& value);</pre>	vector(const allocativector(size type	n.
eastl::all	locator("EASTL classname");	Equal = equal to	_t bucketCount, Hash = hash <t>, T> ></t>	bool validate ()		void push	back();	const alloca	ntor_type& allocator =
<pre>vector_multiset(const ve</pre>		class intrusive h			<pre>erator(const_iterator i) const;</pre>	void pop_b	pack();		cator("EASTL classname");
<pre>vector_multiset(InputIte</pre>	erator first,		able <key=t, equal,<="" hash,="" td="" value="T,"><td></td><td>Operators (+ Containers Common)</td><td>void nush</td><td><pre>front(const value type& value);</pre></td><td><pre>vector(size_type</pre></td><td>n, type& value,</td></key=t,>		Operators (+ Containers Common)	void nush	<pre>front(const value type& value);</pre>	<pre>vector(size_type</pre>	n, type& value,
InputIte		bucketCount, bCon		void swap (intrusive		void push		const alloca	ator type& allocator =
<pre>vector_multiset(InputIte</pre>	erator first,	bUniqueKeys=false		intrusive	_hashtable<>& b);	void pop_ f	ront();		cator("EASTL classname");
InputIte			Public Types	_		itorator i	<pre>insert(iterator pos,</pre>	<pre>vector(const this_ty vector(InputIterator)</pre>	
CONST KE	ey_compare& compare);	T	key_type;	template <typename< td=""><td>T1, typename T2></td><td>iterator i</td><td>const value type& value);</td><td>InputIterator</td><td></td></typename<>	T1, typename T2>	iterator i	const value type& value);	InputIterator	
vector_multiset& operato	or=(const	Value Value	value_type; node_type;	struct pair		void i	nsert(iterator pos,	11	
vector_multiset&);		iterator	insert_return_type;		Public Types		size_type n, const value type& value);	allocator_type& void	<pre>get_allocator(); set allocator(</pre>
void swap (th	his_type& x);	value_type&	reference;	T1 first_			const varue_typeα varue),	VOIU	const allocator type&);
key compare key_comp()) const;	const value_type&	<pre>const_reference; erator<value pre="" type,<=""></value></pre>	T1 second		void i	nsert(iterator pos,	_	= 71 75
value_compare value_comp	p() const;		bConst=true> local iterator,		ic Member Variables		InputIterator first,	vector&	<pre>operator=(const</pre>
pair <iterator, bool=""></iterator,>				T1 first; T2 second			<pre>InputIterator last);</pre>	this_type&); void	<pre>swap(this type& x);</pre>
	value type& value);	const_local_itera			, ic Member Functions	iterator e	erase(iterator pos);	void	<pre>assign(size_type n,</pre>
			le_iterator <value_type, bConst=true> iterator,</value_type, 	pair();	ic Melliber Functions	iterator e	erase(iterator first,		<pre>const value_type&); assign(InputIterator first,</pre>
iterator insert(iterato	or pos, value type& value);		const_iterator;	pair(), pair(const <i>T1</i> &	x);	void c	<pre>iterator last); :lear();</pre>	void	InputIterator last);
COLLECT	value_typed value),	Value	mapped_type;	<pre>pair(const T1&</pre>	х,	VOIU (,
<pre>void insert(InputIt</pre>		use_self< <i>Value</i> >	extract_key;	const T2&	y);	container_		iterator	begin();
InputI	terator last);		blic Member Functions	<pre>pair(const pair<u,< pre=""></u,<></pre>		const cont	ainer_type& get_container() const;	const_iterator iterator	<pre>begin() const; end();</pre>
iterator erase(iterator	r pos);	intrusive_hash_se	t(const Hash& h = Hash(), const Equal& eq = Equal());		Global Functions	bool v a	lidate() const;	const iterator	end() const;
iterator erase (iterator			const Equato eq = Equat());	pair <t1, t2=""> make_p</t1,>	T2 b);	int v a	<pre>lidate_iterator(const_iterator i) const;</pre>	reverse_iterator	rbegin();
iterator		void	<pre>swap(this_type& x);</pre>	pair <t1, t2=""> make_p</t1,>	air_ref(const <i>T1</i> % a,	Global F	functions & Operators (+ Containers Common)	<pre>const_reverse_iterat reverse iterator</pre>	<pre>cor rbegin() const; rend();</pre>
size_type erase (const ke	ey_typeα κ);	iterator	begin();		const <i>T2</i> % b);	void swap	(ring_buffer<>& a,	const reverse iterat	
	st key_type& k);	const iterator	begin(); begin() const;				ring_buffer<>& b);		
const_iterator find (cons	st key_type& k) const;	iterator	end();	template <t, contai<="" td=""><td></td><td></td><td></td><td>bool size type</td><td><pre>empty() const; size() const;</pre></td></t,>				bool size type	<pre>empty() const; size() const;</pre>
iterator find as (const U& u,	const_iterator	<pre>end() const;</pre>	class ring_buffer			T, Container = vector <t> ></t>	size_type	capacity() const;
Binaryl	Predicate predicate);	local iterator	<pre>begin(size type n)</pre>		Public Types	class sta c	:k		
const_iterator find_as(local_iterator	<pre>end(size_type)</pre>	Container	container_type;		Public Types	void	resize(size_type n,
вınaryl	Predicate predicate) const;	<pre>const_local_itera const local itera</pre>				value_type		void	<pre>const value_type&); resize(size type n);</pre>
size_type count(cor	nst key_type& k);	const_total_tteld	cor cim(3126_type) const	this_type, reference,	value_type, const reference,	<pre>container_ reference,</pre>		void	<pre>reserve(size_type n);</pre>
	und(const key type&);	bool	<pre>empty() const;</pre>	iterator,	const_iterator,	re referice,	Public Member Functions	void	<pre>set_capacity(size_type n =</pre>
	und(const key_type&); und(const key type&) const;	size_type	<pre>size() const;</pre>	reverse_iterator,	const_reverse_iterator	stack();	1 ablic Welliber Fulletions		прозу,
	und(const key_type&);	size type	<pre>bucket count() const;</pre>	Publ	ic Member Functions		t Container& x);		<pre>data();</pre>
const_iterator upper_bou	<pre>und(const key_type&) const;</pre>	size_type	<pre>bucket_size(size_type n) const;</pre>	ring_buffer(size_ty		,		const_pointer	<pre>data() const;</pre>
pair <iterator, iterator;<="" td=""><td>></td><td>size_type</td><td><pre>bucket(const key_type& k) const;</pre></td><td><pre>ring_buffer(const C ring_buffer(const t</pre></td><td></td><td></td><td><pre>empty() const; size() const;</pre></td><td>reference</td><td><pre>operator[](size type n);</pre></td></iterator,>	>	size_type	<pre>bucket(const key_type& k) const;</pre>	<pre>ring_buffer(const C ring_buffer(const t</pre>			<pre>empty() const; size() const;</pre>	reference	<pre>operator[](size type n);</pre>
equal_rar	nge(const key_type&);	float	<pre>load factor() const;</pre>		nis_cypea x/;	312C_type	3126() (01131)		<pre>operator[](size_type n) const;</pre>
pair <const_iterator, cor<="" td=""><td>nst_iterator></td><td></td><td>= ''</td><td>this_type&</td><td><pre>operator=(const</pre></td><td>reference</td><td></td><td></td><td>at(size_type n);</td></const_iterator,>	nst_iterator>		= ''	this_type&	<pre>operator=(const</pre>	reference			at(size_type n);
equa1_ra	nge(const key_type&) const;	insert_return_typ	e insert(value_type& value);	this_type&); void	<pre>swap(this type& x);</pre>	const_refe	erence top() const;	reference	<pre>at(size_type n) const; front();</pre>
pair <const_iterator, cor<="" td=""><td>nst_iterator></td><td></td><td>e insert(const_iterator, value type& value);</td><td>void</td><td><pre>assign(InputIterator first,</pre></td><td>void push(</td><td>const value type& value);</td><td>const_reference</td><td><pre>front() const;</pre></td></const_iterator,>	nst_iterator>		e insert (const_iterator, value type& value);	void	<pre>assign(InputIterator first,</pre>	void push(const value type& value);	const_reference	<pre>front() const;</pre>
equal_range_sma	all(const key_type&) const;	void	<pre>insert(InputIterator first,</pre>		`InputIterator last);	void pop()	= 21		back();
pair <iterator, iterator:<="" td=""><td>> all(const key type&)</td><td></td><td><pre>InputIterator last);</pre></td><td>iterator</td><td>begin();</td><td>container</td><td><pre>type& get container();</pre></td><td>const_reference</td><td><pre>back() const;</pre></td></iterator,>	> all (const key type&)		<pre>InputIterator last);</pre>	iterator	begin();	container	<pre>type& get container();</pre>	const_reference	<pre>back() const;</pre>
	ass, RandomAccessContainer	iterator	<pre>erase(iterator);</pre>	const iterator	begin(); begin() const;	<pre>container_ const cont</pre>	ainer type& get_container (); ainer type& get container () const;		<pre>push_back(const value_type&);</pre>
	get allocator();	iterator	<pre>erase(iterator, iterator);</pre>	iterator	end();		= 71 · 6 · = · · · · · · · · · · · · · · · ·	void	push_back();
	set_allocator(size_type	<pre>erase(const key_type&);</pre>	const_iterator reverse iterator	<pre>end() const; rbegin();</pre>			void	pop_back();
	<pre>const allocator_type&);</pre>	void	<pre>clear();</pre>	const reverse itera				iterator	<pre>insert(iterator pos,</pre>
iterator	begin();		,,,	reverse_iterator	rend();				<pre>const value_type&);</pre>
const_iterator	<pre>begin(); begin() const;</pre>	iterator	<pre>find(const key_type& k);</pre>	const_reverse_itera	tor rend() const;				

	• • • • •	1					T e (x)	6. 1. 1/		1		
void	<pre>insert(iterator pos,</pre>	2. Algorithm	ς	Query Algorith			ForwardIterator1	<pre>find_end(ForwardIterator1</pre>	first1,	const T& median(cor	ist 1& a, ist T& b,	
	const value_type&);	_		ForwardIterato.	r adjacent_find(ForwardIterator1	last1,		nst <i>T&</i> c);	
void	<pre>insert(iterator</pre>	typename	Meaning		ForwardIterator ForwardIterator			ForwardIterator2	first2,	const T& median(cor		
	InputIterator first,	1	The value type.	ForwardIterato	r adjacent find(1451),		ForwardIterator2	last2,		nst <i>T&</i> b,	
	<pre>InputIterator last);</pre>	Compare	A function which takes two		ForwardIterator ·	first,	ForwardIterator1	BinaryPredicate find first of();		nst T& c, npare compare);	
iterator	<pre>erase(iterator pos);</pre>		arguments and returns the		ForwardIterator :		TOIWUIUILEIULOII	ForwardIterator1	first1,	Con	pure compare),	
iterator	erase(iterator first,	Dun din uta	lesser of the two.		BinaryPredicate)	;		ForwardIterator1	last1,	pair <inputiterator1< td=""><td>I, InputIterator2> mi</td><td>.smatch(</td></inputiterator1<>	I, InputIterator2> mi	.smatch(
	iterator last);	Predicate	A function which takes one	bool	binary_search(ForwardIterator2	first2,	i i	InputIterator1	first1,
	-1().		argument returns true if the	5001	ForwardIterator	first,	C	ForwardIterator2	last2);		InputIterator1	
void void	<pre>clear(); reset();</pre>		argument meets some criteria.		ForwardIterator		ForwardIterator1	<pre>find_first_of(ForwardIterator1</pre>	first1,	nair/InnutIterator1	InputIterator2 I, InputIterator2> mi :	
1010	10000(/)	BinaryPredicate	A function which takes two	h1		value);		ForwardIterator1	last1,	parrinpacrecraeors	InputIterator1	
bool validate()		DillaryFredicate	arguments and returns true if	bool	<pre>binary_search(ForwardIterator -</pre>	first		ForwardIterator2	first2,		InputIterator1	
_	<pre>iterator(const_iterator i) const;</pre>		some criteria is met (e.g. they		ForwardIterator			ForwardIterator2 BinaryPredicate);	last2,		InputIterator2	
Global Functions	& Operators (+ Containers Common)		are equal).			value,	ForwardIterator1	find_first_not_of	(BinaryPredicat	ε),
void swap (vector		StrickWeakOrdering	A BinaryPredicate that		Compare	compare);	7 017/4141414141414141414141414141414141414	ForwardIterator1	`first1,	ForwardIterator1 se	earch(ForwardIterator	1 first1,
vector	<>& b);		compares two objects,	ForwardIterato	r binary_search_i(ForwardIterator1	last1,		ForwardIterator.	
			returning true if the first	, ornararecrato.	ForwardIterator	first,		ForwardIterator2 ForwardIterator2	first2,		ForwardIterator.	
template <t, size_<="" td=""><td></td><td></td><td>precedes the second. Like</td><td></td><td>ForwardIterator</td><td></td><td>ForwardIterator1</td><td>find first not o</td><td>last2); f(</td><td>ForwardTterator1 se</td><td>.ForwardIterator :earch(ForwardIterator</td><td></td></t,>			precedes the second. Like		ForwardIterator		ForwardIterator1	find first not o	last2); f (ForwardTterator1 se	.ForwardIterator : earch (ForwardIterator	
	nableOverflow = true,		Compare but has additional		const T& value);		7 0174414114141414141	ForwardIterator1	first1,	, ormararecraeorr 50	ForwardIterator.	
	or = eastl::allocator>		requirements. Used for	ForwardIterato.	r binary_search_i (ForwardIterator -	first		ForwardIterator1	last1,		ForwardIterator.	
class fixed_vecto			sorting routines.		ForwardIterator			ForwardIterator2	first2,		ForwardIterator.	
_	vector_allocator<> >	Function	A function which takes one			valué,		ForwardIterator2 BinaryPredicate);	last2,	ForwardTterator se	BinaryPredicate earch n(ForwardIterate	
	I the vector functionality.		argument and applies some		Compare	compare)	BidirectionalIter	ator1 find last of(TOTWGTGTCTGTGT SC	ForwardIterati	
Pul	blic Member Functions		operation to the target.	difference type	e count (InputItera	tor first		BidirectionalIterator.			Size	count,
<pre>fixed_vector();</pre>		Size	A count or size.	difference_typ	InputItera			BidirectionalIterator			const <i>T&</i>	value);
<pre>fixed_vector(size_</pre>		Generator	A function which takes no		const T&	value);		ForwardIterator2 ForwardIterator2	first2, last2);	Sorting		
<pre>fixed_vector(size_ const value</pre>			arguments and returns a	difference_typ	e count_if (InputItera		BidirectionalIter	ator1 find last of(143(2),	RandomAccessIterato		
fixed vector(const			value (which will usually be		InputItera Predicate	tor last, predicate);		BidirectionalIterator.			RandomAccessIterator RandomAccessIterator	
fixed_vector(Input	tIterator first,		assigned to an object).		rieuicute	predicate),		BidirectionalIterator			const T&	pivotVal);
Input	tIterator last);	UnaryOperation	A function which takes one	bool equa	1(ForwardIterator2 ForwardIterator2	first2, last2,	RandomAccessIterato		r,
waid cot	<pre>capacity(size type n);</pre>		argument and returns a value	-		or1 first1,		BinaryPredicate);	10512,		Random Access Iterator	
	<pre>_capacity(size_type n); size() const;</pre>		(which will usually be		InputIterate		BidirectionalIter	ator1 find_last_not_of	(RandomAccessIterator	
	I() const;		assigned to second object).	bool equa		or2 first2);		BidirectionalIterator			const T& Compare	pivotVal, compare);
	& Operators (+ Containers Common)	BinaryOperation	A function which takes two	boot cquu		or1 first1,		BidirectionalIterator. ForwardIterator2			Compare	compare),
void swap (fixed v	•		arguments and returns a		InputIterate	or1 last1,		ForwardIterator2	first2, last2);	void heap_sort (RandomAccessIterato	
	vector<>& b);		value (which will usually be		InputIterate		BidirectionalIter	ator1 find last not of			RandomAccessIterato	
			assigned to a third object). An input iterator (iterator you	nair/ForwardIt	BinaryPredio	cate);		BidirectionalIterator.		void heap_sort (StrictWeakOrdering RandomAccessIterato	
		InputIterator	An input iterator (iterator you	equa	l_range(017		BidirectionalIterator		void neup_sore(RandomAccessIterato	
			read from) which allows		ForwardIter			ForwardIterator2 ForwardIterator2	first2, last2,			, ,
			reading each element only once and only in a forward		ForwardIter			BinaryPredicate);	10512,	void insertion_sort	t(BidirectionalIterat	
			direction.	nair/ForwardIt	const T& erator, ForwardIterato	value);					BidirectionalIterate StrictWeakOrdering	
		ForwardIterator	An input iterator which is like	·	l_range(017		<pre>putIterator1 first1, putIterator1 last1,</pre>		void insertion sort	(BidirectionalIterat	
		i oi waranciatoi	InputIterator except it can be	1	ForwardIter			putIterator2 first2,		_	`BidirectionalIterate	
			reset back to the beginning.		ForwardIter			putIterator2 last2);		unid imparties of	s simple/	
		BidirectionalIterator	An input iterator which is like		const T& <i>Compare</i>	value, compare);		putIterator1 first1,		void insertion_sort	:_simpie (RandomAccessIterato:	r first
			ForwardIterator except it can					<pre>putIterator1 last1, putIterator2 first2,</pre>			RandomAccessIterato	
			be read in a backward	Function for_e	ach(InputIterator fir			putIterator2 last2,			Compare	compár);
			direction as well.		InputIterator las- Function fun	t, ction);		naryPredicate predicat	e)	world stable seat/	RandomAccessIterato	r first
		RandomAccessIterato	An input iterator which can		Tunction Tune	CC1011/)	5	1 1/5 /5/		void stable_sort (RandomAccessIterato:	
			be addressed like an array. It	InputIterator	find(rorwaraiterator 1	ower_bound(ForwardIter ForwardIter			StrictWeakOrdering	compare);
			is a superset of all other input	:	InputIterator	first,		const T&	val);	<pre>void stable_sort(</pre>	RandomAccessIterato:	
			iterators.		InputIterator const T&	last, value);	ForwardIterator 1	<pre>ower_bound(ForwardIter</pre>	ator first,	void stable_sort(RandomAccessIterato: RandomAccessIterato:	
		OutputIterator	An output iterator (iterator	InputIterator		vaiue),		ForwardIter		VOID STADIE_SOIT(RandomAccessIterato:	
			you write to) which allows		InputIterator `	first,		const T& Compare);	val,		Allocator&	allocatr,
			writing each element only		InputIterator	last,	ForwardIterator u	pper bound(ForwardIter	ator first.		StrictWeakOrdering	compare);
			once in only in a forward		Predicate	<pre>predicate);</pre>		ForwardIter	ator last,	woid radio cont/	RandomAccessIterato:	r firc+
			direction.	ForwardIterato	r1 find end(C	const T&	val);	void radix_sort(RandomAccessIterato	
		A function which take	an InputIterator will also work	9	ForwardIterator1	first1,	ForwardIterator u	<pre>pper_bound(ForwardIter ForwardIter)</pre>			RandomAccessIterato:	
		with a Forwarditerato	r, Bidirectionaliterator, or		ForwardIterator1	last1,		const T&	val,	1	- 1-1	
		RandomAccessIterato			ForwardIterator2 ForwardIterator2	first2, last2);		Compare);	. ,	void comb_sort(ForwardIterator ForwardIterator	first,
			e is merely the <i>minimum</i> by the iterator must support.		' OTMUTUTEETH FOT 5	10312/,					StrictWeakOrdering	last, compare);
		supported interiousing	y the iterator must support.							void comb_sort(ForwardIterator	first,
		1		I			I			1	ForwardIterator	last);

							In				le ie			h		· .
<pre>void bubble_sort(</pre>	ForwardIterator	first,	Modifying Sequ				OutputlIterator	r remove_copy		tor first,	ForwardIterator m	in_element(ax element(ForwardIte	rator first	bool is_heap (RandomAccessIterator RandomAccessIterator	
VOIG BUBBIC_SOIC(ForwardIterator	last,	OutputlIterator						InputItera		TOTWATATEETATOT III			bool is heap(RandomAccessIterator	
	StrictWeakOrdering			InputIte InputIte		irst, ast,					ForwardIterator m	in_element(- '`	${\it Random} Access Iterator$	last,
void bubble_sort (ForwardIterator	first,		Output1		esult);	E		Predicate)		ForwardIterator ma	ax_element(ForwardIte			Compare	compare);
	ForwardIterator	last);	BidirectionalIt			,,,	ForwardIterator	r remove_1+(rator first, rator last,		ForwardIte: Compare);	rator last,	void make_heap(RandomAccessIterator RandomAccessIterator	
void selection sort	(ForwardIterator	first,			nalIterator1 f				Predicate)		Lovicographic Ord			void make heap(RandomAccessIterator	
-	`ForwardIterator	last,		Bidirection	nalIterator1 l nalIterator2 r	ast,			,		Lexicographic Ord				${\it RandomAccessIterator}$	
	StrictWeakOrdering			DIUITECTI	muiitelutoiz i	csuit),	void	replace(_			DOOL TEXTCOSTABILITY	InputIterator1	first1,		Compare compare);	
void selection_sort	(ForwardIterator): ForwardIterator	first, last);	void	fill(Forward	lIterator	first,			ardIterator ardIterator			InputIterator1	last1,	void pop_heap(RandomAccessIterator RandomAccessIterator	
	TOIWUIUILEIULOI	iast),				last,		cons		old value,		InputIterator2	first2,	void pop_heap(RandomAccessIterator	
<pre>void shaker_sort(</pre>	BidirectionalItera	tor first,	waid	const i fill(char*		value); first,		cons		new_value);	h1 1	InputIterator2	last2)	TOTA POP_HEAP(RandomAccessIterator	
	BidirectionalItera		void	char*		last,	void	replace_if(bool lexicographic	const char*	first1,		Compare compare);	
void shakar sart/	StrictWeakOrdering BidirectionalItera			const		c);			ardIterator ardIterator			const char*	last1,	void push_heap (RandomAccessIterator RandomAccessIterator	
void shaker_sort (BidirectionalItera		void	fill(char*		first,			ururterutor icate	predicate,		const char*	first2,	void push heap (RandomAccessIterator	
	DIWITCCCIONWITCCIW	101 1050/,		char*		last,		cons		new value);	h1 1	const char*	last2);	VOIG PUSII_IICUP(RandomAccessIterator	
<pre>void bucket_sort(</pre>	ForwardIterator fi		void	const i		c); first,	OutputlIterator				bool lexicographic	char*	first1,		Compare	compare)
	ForwardIterator la		VOIU			last,		Inpu	tIterator tIterator	first,		char*	last1,	void remove_heap (RandomAccessIterator	
	ContainerArray& bu HashFunction ha	sh);			ınsigned char&				utlIterator			char*	first2,		Distance Distance	heapSize, pos);
Set Operations	Tidotti directori ild	311/3	void	fill(unsigne		first,		cons		old value,	h1 1	char*	last2);	void remove heap(
OutputlIterator set	difference(const		last, c);	_	cons		new_value);	bool lexicographic	const unsigned ch	nar* first1			heapSize,
		1 first1,	void	fill(signed		first,	OutputlIterator	r replace_copy	y_if (tIterator	first		const unsigned ch	nar* last1,		Distance	pos,
	InputIterator	1 last1,		signed		last,			tIterator			const unsigned ch		void sort heap (Compare RandomAccessIterator	compare);
	InputIterator InputIterator					c);			utlIterator		h1 1	const unsigned ch	nar* last2);	Tota sort_meap(RandomAccessIterator	
	OutputlIterat		void	fill(signed signed		first, last,				oredicate,	bool lexicographic	unsigned char*	first1,	void sort_heap (${\it Random} Access Iterator$	
OutputlIterator set		01 103010/,		const		c);		cons	t /&	new_value);		unsigned char*	last1,		RandomAccessIterator	
,	InputIterator		void	fill(bool*		first,	void	reverse(unsigned char*	first2,	Name and a Almandala		compare);
	InputIterator			bool* const b		last,	1014		tionalItera	tor first,	haal lawisassanhi	unsigned char*	last2);	Numeric Algorith		6:
	InputIterator InputIterator			CONST	ισστα	b);	0 / 177/		tionalItera	tor last);	bool lexicographic	const signed cha	r* first1,	l ac	cumulate(InputIterator InputIterator	
	OutputlIterat		OutputlIterator	r fill_n (Output	lIterator	first,	OutputlIterator	r reverse_copy	y (tionalItera	tor first		const signed char			T	init);
	Compare	_compare);		Size		n,			tionalItera			const signed char		T ac	<pre>cumulate(InputIterator</pre>	r first,
OutputlIterator set	symmetric_ditteren. InputIterator		char*	const fill n(char*		value); first,			Iterator	resuĺt);	bool lexicographic	const signed char	r* last2);		InputIterator	
	InputIterator		Cliai	Size		n,		/ =0			DOOT TEXTCOGRAPHIC	signed char*	first1,		BinaryOperati	init, ion hin on):
	InputIterator	2 first2,		const	char&	c);	void	swap (<i>T&</i> <i>T&</i>		a, b);		signed char*	last1,			,
	InputIterator		unsigned char*		ed char*	first,	ForwardIterator			5/,		signed char*	first2,	OutputIterator ad	jacent_difference(
OutputlIterator set	OutputlIterat			Size	unsigned char8	n,		Fo	rwardIterat		bool lexicographic	signed char*	last2);		InputIterato: InputIterato:	
outputification sec	InputIterator		signed char*	fill_n(signed		first,			rwardIterat	or1 last1, or2 first2);	DOOT TEXTCOGRAPHIC	InputIterator1	first1,		OutputIterato	
	InputIterator			Size		n,	void		ForwardIter			InputIterator1	last1,	OutputIterator ad	jacent_difference(, .
	InputIterator		L1*		signed char&	c);	1014		ForwardIter			InputIterator2	first2,		InputIterator	
	InputIterator OutputlIterat		bool*	fill_n(bool* Size		first, n,	_					InputIterator2 Compare);	last2,		InputIterato: OutputIterato	
	Compare	compare);		const	bool&	b);	OutputlIterator				Heap Operations				BinaryOperati	
									putIterator tputlIterat			(RandomAccessIterator	first.			=
OutputlIterator set	:_intersection(InputIterator	1 first1	Function	for_each(Inpu		first, last,		Und	aryOperatio	n);			topPos,	T in	ner_product(ra firsta
	InputIterator			Funci		func);	OutputlIterator					Distance	pos,		InputIterato: InputIterato:	
	InputIterator					,,			nputIterato. nputIterato.			const T& (RandomAccessIterator	value);		InputIterato:	
	InputIterator		void		rwardIterator				utputlItera				topPos,		T	init);
OutputlIterator set	OutputlIterat intersection(or resurt),			rwardIterator enerator	gen);		В.	inaryOperat	ion);			pos,	ı ın	ner_product(InputIterator1	first1,
			OutputlIterator				ForwardIterator	r unique/Form	ardIterator	first			value, compare);		InputIterator1	
	InputIterator					n,	o o i war ar terator		ardIterator			RandomAccessIterator			InputIterator2	
	InputIterator InputIterator			Ge	enerator	gen);	ForwardIterator						topPos,		T	init,
	OutputlIterat		void random shu	uffle(RandomAco	essIterator	first,			ardIterator				heapSize,		BinaryOperatior BinaryOperatior	
	Compare	comparé);		RandomAco	essIterator	last,	Min and Man	в1па.	тургеатсасе	predicate);			pos, value);		7.1	_11 //
0		(:+-		RandomNun	berGenerator&	rng);	Min and Max					RandomAccessIterator		OutputIterator pa	rtial_sum(InputIterate	
OutputlIterator set	union(InputIterato <u>.</u> InputIterato		ForwardIterator	r remove(orwardIterator	first	const T& min(const T& a			3 - 1	Distance	topPos,		InputIterato OutputIterat	
	InputIterato	r2 first2,	. SIMULUI CCIUCOI		orwardIterator		203c /a max(const T& b					heapSize,	OutputIterator pa	rtial_sum(InputIterate	
	InputIterato			(onst <i>T&</i>	val);	float min(, -				pos, value,	'	InputIterate	or last,
OutputlIterator set	OutputlItera		OutputlIterator				float max(float a				Compare	compare);		OutputIterat	
outputilierator set	InputIterato				inputIterator OutputlIterator		double min/	float b	/,			Random Access Iterator			BinaryOperat	rion pinob);
	InputIterato	r2 first2,			onst T&		double max(double a					heapSize, pos);			
	InputIterato							double b);			RandomAccessIterator	first,			
	OutputlItera Compare	compar);					const T& min(const T& max(const T& a				Distance	heapSize,			
		pu-/;					const ia max(const T& b					pos,			
								Compare c				Compare	compare);			

Security	2. Francisco Obioceto	template < <i>Predicate</i> >	pointer_to_unary_function(template <result, t=""> mem fun t<></result,>	
template disruptions (Particularly International Content in a sugment type; result, type; result	3. Function Ubjects	class unary_negate	Result (*pFunction)(Arg));		
Structurage function ### Apparent		: unary_function< <i>Predicate</i> ::argument_type, bool>	Result operator() (Arg x) const;		
### Struct Change (Trymerer Lype)	template <argument, result=""></argument,>	Public Member Functions			
argument special type; result					
replate dragment, Anymord, Results treplate dragment, Anymord, Results Aspenent Asp					
remains departed, Apparent, Apparent	Argument argument_type;	const Treateuteargument_typed a) const,	: public binary_tunction(Argi, Argz, Result)	(Result (T::*MemberFunction)() const)	
template designates, Argament, Personal Control Productions (Argament) (Argam	result_type;	townlate (Predicate) unary negate (Predicate)		(Nesure (1.1. Hember unceron) (Argument) conse)	
troplate c/b result result (repeated) result result (repeated) result result (repeated) result (repeated			pointer_to_binary_function(t1-t- (01t T)	
Appendix of Struct Systems (Systems 1, 1995) Applied to Systems (S					
Arginent: second arginent types template erredicates template cfs template erredicates template erredicate	/ -	(const Treateured predicate)			
casplate of Common Multiples of Wides casplate of Common Multiples casplate of Common	7.7	townlate (Predicate)		· -	
template of poerator() (const Th a) const public kinary-function of the struct program o		template (Fleulewitz)	template <arg, result=""></arg,>	, · · · · · · · · · · · · · · · · · · ·	
Compared of Predicate's second argument type, bools				, , , , , , , , , , , , , , , , , , , ,	
Public Member Functions Public Member Fu	template <t></t>				
poperator()(const TA a) const redicate:::rirst argument types a) const redicate::second_argument_types b) const redicat	struct plus, minus, multiplies, divides, modulus	Public Member Functions	(Result (*pFunction)(Arg))		
template <pre>const Predicate: strict argument types a const Predicate: strict argument types b const strict argument types b const predicate: strict argument types b const strict argument types const strict argument types const strict ar</pre>	: public binary_function <t, t="" t,=""></t,>	<pre>binary_negate(const Predicate& a)</pre>			
template c/r const 778 b) const; template c/r const 778	T energter()/const T% a			template <result, argument="" t,=""></result,>	
template c7) template c7 temp		const Predicate::second argument type% a,	pointer_to_binary_functionkargi, argz, kesuits		
template of Predicates binary negate/Predicates public unary function(7, 7) operator()(const 78 a) const; template of Department of Departm	232 12 27 68.156,		(Result (*pFunction)(Arg1, Arg2))		
template desired, To predict many function of, To problet many function of, To many	template <t></t>	temmlate (Predicate) himary penate/Predicate		-	
Topatic unity function (7, 7) Const Pa a) const; Const Predicate predicate Const Predicate Const Predicate Predicate Const Predicate	·		template < <i>Result</i> , <i>T</i> >	Result (T::*MemberFunction)(Argument);	
Toperator()(const TB a) const; template <d <d="" template="" template<="" th="" =""><th></th><th></th><th></th><th>Public Member Functions</th></d>				Public Member Functions	
template CD constraints co	·	(const / reareacea preareace)	: unary function <t*, result=""></t*,>	<pre>mem_fun1_ref_t(MemberFunction);</pre>	
class binder1st class equal_to_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_not_experience_to_not_experience_to_not_not_experience_to_not_not_experience_to_not_expe	T operator()(const T& a) const;	template (Oneration)	Public Types	Result operator()(T& pT, Argument arg) const;	
constraint equal to, not equal to, less, greater, construct equal to, not equal to, less, greater, construct equal to, not equal to, less, greater, constraint to peratorisms constraint to			Result (T::*MemberFunction)();		
Struct equal_to, not_equal_to, validate_not_equal_to, validate_not_equal_to, validate_not_equal_to, validate_greater, validate_greater, validate_less_equal_validate_greater, validate_less_equal_validate_greater, validate_not_equal_to, validate_not_equ			Public Member Functions		
Similar Const (Const (Ta b) Const (Ta b) C	struct equal_to, not_equal_to, less, greater,				
Const Operation Const T& a Const Operation		Public Member Functions	Result operator()(T* pT) const;	7= -	
const \$\(\text{operation}\); first argument type \(\text{ype}\); \\ \text{operation}\); \\ \text{template } \(\text{t}\); \\ \text{validate } \text{equal to}, \text{validate } \text{not } \text{qeal to}, \text{validate } \text{poperation}\); \\ \text{template } \(\text{t}\); \\ \text{validate } \text{equal to}, \text{validate } \text{poperation}\); \\ \text{template } \(\text{t}\); \\ \text{validate } \text{geal to}, \text{validate } \text{geater}, \\ \text{desplate} \times \text{geater}, \\ \text{seplate} \times \text{geater}, \\ \text{template} \times \text{geater}, \\ \text{seplate} \times \text{geater}, \\ \tex	: public binary_function <t, bool="" t,=""></t,>			71	
Const T& b) Const T& b	T operator()(const T& a.			, , , ,	
template <t, compare=""> bool. validate_equal_to, validate_not_equal_to, validate_less, validate_greater, validate_less_equal, validate_greater_equal (const T& a, const T& b) const T& a) remplate <t> const Operation::second_argument_type& x) const; validate_less, validate_greater, validate_less_equal, validate_greater_equal (const T& a, const T& b) const; template <t> const Operation : second_argument_type& x) const; validate_less_equal, validate_greater_equal (const T& a, const T& b) const; template <t> const Operation is public operation > bind1st template <t> const Operation is public operation > const T& a, const T& b) const; template <t> const Operation is public operation > const Operation is operation operation is operation on the public operation of the properation of the properation operation operati</t></t></t></t></t></t,>					
template <7, Compare> bool validate_equal_to, validate_greater, validate_less_equal, validate_greater, validate_less_equal, validate_greater, validate_less_equal, validate_greater, validate_less_equal, validate_greater_equal (const 7\(\tilde{t}\) bind1st (const 7\(\tilde{t}\) a, const 7\(\tilde{t}\) b, Compare compare) template <7> perator()(const 7\(\tilde{t}\) a, const 7\(\tilde{t}\) b) const; template <7> template <7			1 2 2 1		
<pre>validate_equal_to, validate_not_equal_to, validate_less_validate_greater, validate_less_validate_greater_equal (const T& a, const T& b, Compare compare) template <operation, t=""> binder1st<operation> bind1st const T& a, const T& b, Compare compare) template <t> template <i> const T& a, const T& b, Compare compare) template <t> template <i> const T& a, const T& b, Compare compare) template <t> template <operation> template <i binary_function<="" pre="" public=""> public Const T& a, const T& b) const; template <t> public binary_function public dember Functions template <operation> templat</operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></operation></t></i></operation></operation></operation></operation></operation></operation></operation></operation></t></i></t></i></t></operation></operation,></pre>	template <t, compare=""> bool</t,>		/1	nesare speciator()(conse na pri) conse;	
<pre>validate_less, validate_greater, validate_less_equal, validate_greater_equal (const 7% a, const 7% b, Compare compare) template <t> template <td <td="" tem<="" template="" th=""><th>validate_equal_to, validate_not_equal_to,</th><th>operation recond_discusting_typed x/ const,</th><th>· · · · · · · · · · · · · · · · · · ·</th><th>template (Pesult T Argument)</th></td></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></pre>	<th>validate_equal_to, validate_not_equal_to,</th> <th>operation recond_discusting_typed x/ const,</th> <th>· · · · · · · · · · · · · · · · · · ·</th> <th>template (Pesult T Argument)</th>	validate_equal_to, validate_not_equal_to,	operation recond_discusting_typed x/ const,	· · · · · · · · · · · · · · · · · · ·	template (Pesult T Argument)
Validate_less_equal, validate_greater_equal (const T& a, const T& b, Compare compare) (const Operation: (const Operation (const T& a, const T& b, Compare compare) (const T& a, const T& b, Compare compare) (const Operation (const T& a, const T& b, Compare compare) (const Operation (const T& a, const T& b, Compare compare) (const Operation (const T& a, const T& b, Compare compare) (const Operation (const T& a, const T& b, Compare compare) (const Operation (const T& a) (const Operation (const T& a) (con	validate_less, validate_greater,	template (Operation To hinder1st(Operation)			
Const T& a, const T& b, Compare compare	validate less equal validate greater equal				
template <t> template <t, bool="" t,=""> template <t, bool="" t,=""> template <t, u=""> template <t, u=""> template <t> template <t> template <t, u=""> template <t> template <t, u=""> template <t> template <t> template <t> template <t, u=""> template <t, u=""> template <t> template <t, u=""> template <t> template <t, u=""> template <t> template <t, u=""> template <t, <t,="" a,="" and,="" b)="" const="" const;="" logical="" not="" or,="" struct="" t&="" template="" u=""> template <t> template <t, u=""> template <t, u=""> template <t, u=""> template <t, (t::*memberfunction)="" <t,="" end="" logical="" struct="" template="" u=""> template <t, (t::*memberfunction)="" <t,="" end="" logical="" struct="" template="" u=""> template <t, (t::*memberfunc<="" (t::*memberfunction)="" <t,="" argument="" end="" logical="" struct="" template="" th=""><th></th><th></th><th>, ,,, , ,</th><th></th></t,></t,></t,></t,></t,></t,></t></t,></t,></t></t,></t></t,></t></t,></t,></t></t></t></t,></t></t,></t></t></t,></t,></t,></t,></t></t></t></t></t></t></t></t></t></t></t></t>			, ,,, , ,		
template <1> struct str_equal_to struct str_equal_to spublic binary_function<1, 7, bool> remplate <1> struct str_equal_to spublic binary_function<1, 7, bool> remplate <1> spublic binary_function<2, 7, bool> remplate <1> spublic dember functions spublic Member functions binder2nd(template <result, t=""></result,>	/1	
struct str_equal_to : public binary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_function : unary_functio	template <t></t>	template <operation></operation>		Public Member Functions	
: public binary_function <t, bool="" t,=""> T</t,>	struct str equal to			<pre>const_mem_fun1_ref_t(MemberFunction);</pre>	
Operation::result_type> Public Member Functions binder2nd(const Operation::result_type operator()(const T& a) const Operation::result_type operator()(const Op		: unary_function <operation::first_argument_type,< th=""><th>· -</th><th>Result operator()(const T& pT, Argument arg) const;</th></operation::first_argument_type,<>	· -	Result operator()(const T& pT, Argument arg) const;	
Public Member Functions Const T& b Const T& b Const T& b	T operator()(const T% a	=	Result (T::*MemberFunction)() const;		
template <1> template <1 template			Public Member Functions	template <result,t> mem_fun_ref_t<></result,t>	
template <t> struct logical_and, logical_or, logical_not : public binary_function<t, bool="" t,=""> T operator()(const T& a), const T& b) const; template <t, u=""> const Operation::second_argument_type& y) Operation::result_type operator()(const Operation::first_argument_type& x) const; Operation::first_argument_type& x) const; template <result, argument="" t,=""> const mem funl ref mem_fun_ref (Result (T::*MemberFunction)()) (Result (T::*MemberFunction)(Argument)) (Result (T::*MemberFunction)(Argument)) (Result (T::*MemberFunction)(Argument) const) template <t, u=""> remplate <result, argument="" t,=""> const; template <result, argument="" t:=""> const; template <result, (result="" (t::*memberfunction)(argument)="" *memberfunction)()="" *memberfunction)(argument)="" <result,="" <t,="" const)="" remplate="" t:="" u=""></result,></result,></result,></t,></result,></t,></t,></t>	, ,			template <pre></pre>	
<pre>struct logical_and, logical_or, logical_not : public binary_function<t, bool="" t,=""> T</t,></pre>	template <t></t>		Result operator()(const T* pT) const;	template <pre>template <pre>te</pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre>	
: public binary_function <t, bool="" t,=""> T</t,>		Operation::result type operator()(
Operation::first_argument_type& x) const; Const T& b) const; template <t, u=""> Operation::first_argument_type& x) const; Const T& b) const; template <operation, t=""> binder2nd<operation> bind2nd Operation::first_argument_type& x) const; Const T& Argument, Result> Public Types Result (T::*MemberFunction)(Argument) const; Public Member Functions Result (T::*MemberFunction)(Argument) const; Public Member Functions</operation></operation,></t,>		Operation receilt type energter()/		(Result (T::*MemberFunction)())	
template <t, u=""> operator()(Const ra a, const ra b) const; template <operation, t=""> binder2nd<operation> bind2nd ibnary_function<const ra="" rangement,="" result=""> Public Types Result (T::*MemberFunction)(Argument) const) Result (T::*MemberFunction)(Argument) const) Public Member Functions result (T::*MemberFunction)(Argument) const)</const></operation></operation,></t,>	T energter()(const T0 -	Operation::first_argument_type& x) const;			
template <operation, t=""> binder2nd<operation> bind2nd template <t, u=""> Result (T::*MemberFunction)(Argument) const; Public Member Functions</t,></operation></operation,>			· · · · · · · · · · · · · · · · · · ·	(Result (T::*MemberFunction)(Argument) const)	
template <t, u=""> bind2nd Public Member Functions</t,>	,,	template <operation, t=""> binder2nd<operation></operation></operation,>			
Pupil Member Functions	template <t. u=""></t.>				
struct equal to 2 not equal to 2 less 2 (const Operation% op, const 7% x)	struct equal to 2, not equal to 2, less 2	(const <i>Operation</i> & op, const <i>T</i> & x)			
const mem_fun1_t(MemberFunction); : public binary_function <t, bool="" u,=""> const mem_fun1_t(MemberFunction); Result operator()(const T* pT, Argument arg) const;</t,>	: public binary function< <i>T</i> , <i>U</i> , bool>				
template <arg. result=""></arg.>		template <arg, result=""></arg,>	Printing and Court		
operator()(const 7% a, const; U% b) const; class pointer_to_unary_function	operator()(const T& a,				
: public unary_function <arg, result=""></arg,>	const ou b) const,				
Public Member Functions					
pointer_to_unary_function()		pointer_to_unary_function()			

12 | EASTL 1.10.03 Quick Reference v1.0

4. Iterators		Global reverse_iterator Operators	
		template <iterator1, iterator2=""></iterator1,>	
Iterators Categor		bool operator==(
struct input_iter	ator_tag	bool operator!=(bool operator< (
<pre>{}; struct output_iterator_tag</pre>		bool operator> (
{};		bool operator<=(bool operator>=(
struct forward_iterator_tag : input iterator tag		const reverse_iterator	
{};	0	const reverse_iterator< <i>Iterator2</i> >& b);	
struct bidirectional_iterator_tag : forward iterator tag		template <iterator1, iterator2=""></iterator1,>	
{};	5	reverse_iterator <pre>reverse_iterator</pre> <pre>operator-(</pre>	
struct random_acc	<pre>ess_iterator_tag : bidirectional iterator tag</pre>	const reverse_iterator	
{};	Didirectional_feetator_tag	const reverse_iterator <pre>const reverse_iterator</pre> <pre>terator2>&</pre> <pre>b);</pre>	
		template < <i>Iterator</i> >	
template <categor< td=""><td>y, T, Distance = ptrdiff_t,</td><td>reverse_iterator</td></categor<>	y, T, Distance = ptrdiff_t,	reverse_iterator	
Pointer = T*, Reference = T&>		<pre>operator+(reverse iterator</pre> //reverse iterator	
struct iterator	Dule II a Tomas	const reverse_iterator< <i>Iterator</i> >&a);	
itaratar satasarı	Public Types		
pointer,	<pre>, value_type, difference_type, reference;</pre>	template <i><container></container></i>	
, ,	•	class back_insert_iterator	
template <iterato< td=""><td>r></td><td><pre>: public iterator<output_iterator_tag, void,="" void,<br="">void, void></output_iterator_tag,></pre></td></iterato<>	r>	<pre>: public iterator<output_iterator_tag, void,="" void,<br="">void, void></output_iterator_tag,></pre>	
struct iterator_tr	aits	Public Member Functions	
	Public Types	<pre>back_insert_iterator(Container& x);</pre>	
iterator_category		<pre>back_insert_iterator& operator=(const_reference); back insert iterator& operator*();</pre>	
pointer,	reference;	back insert iterator& operator++();	
		back_insert_iterator operator++ (int);	
	ct iterator_traits <t*> ct iterator_traits<const t*=""></const></t*>		
tempiate (77 Sero	Public Types	template <container></container>	
iterator category	**	back insert iterator <container></container>	
<pre>iterator_category pointer,</pre>	<pre>, value_type, difference_type, reference;</pre>	back_inserter	
	, value_type, difference_type,		
<pre>pointer, template <it></it></pre>	, value_type, difference_type, reference;	back_inserter	
template <it> class reverse_ite</it>	<pre>, value_type, difference_type, reference;</pre>	back_inserter (Container& x) template <container> class front_insert_iterator</container>	
template <it> class reverse_ite :iterator<iterator< td=""><td>, value_type, difference_type, reference;</td><td><pre>back_inserter (Container& x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, pre="" void,="" void,<=""></output_iterator_tag,></container></pre></td></iterator<></it>	, value_type, difference_type, reference;	<pre>back_inserter (Container& x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, pre="" void,="" void,<=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite :iterator<iterato iterato="" iterato<="" td=""><td><pre>rator r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::idfference_type,</it></it></it></pre></td><td><pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, pre="" void,="" void<=""></output_iterator_tag,></container></pre></td></iterato></it>	<pre>rator r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::idfference_type,</it></it></it></pre>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, pre="" void,="" void<=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite :iterator<iterator iterator="" iterator<="" td=""><td>rator r_traits<it>::iterator_category, r_traits<it>::difference_type, r_traits<it>::iterator_category, r_traits<it>::difference_type, r_traits<it>::pointer,</it></it></it></it></it></td><td>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></td></iterator></it>	rator r_traits <it>::iterator_category, r_traits<it>::difference_type, r_traits<it>::iterator_category, r_traits<it>::difference_type, r_traits<it>::pointer,</it></it></it></it></it>	back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container>	
template <it> class reverse_ite :iterator<iterator iterator="" iterator<="" td=""><td><pre>rator r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::idfference_type,</it></it></it></pre></td><td><pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre></td></iterator></it>	<pre>rator r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::idfference_type,</it></it></it></pre>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre>	
template <it> class reverse ite :iterator<iterator iterator="" iterator<="" td=""><td><pre>rator r traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::difference_type, r_traits<it>::reference></it></it></it></it></it></pre></td><td><pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre></td></iterator></it>	<pre>rator r traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::difference_type, r_traits<it>::reference></it></it></it></it></it></pre>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre>	
template <it> class reverse ite :iterator:iterato iterato iterato</it>	rator r_traits <it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::ointerence_type, r_traits<it>::pointer, r_traits<it>::reference> Public Types inter, reference,</it></it></it></it></it></it>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite :iterator:iterato iterato pedifference_type;</it>	rator r_traits <it>::iterator_category, r_traits<it>::value_type, r_traits<it>::difference_type, r_traits<it>::difference_type, r_traits<it>::pointer, r_traits<it>::reference Public Types inter, reference, tblic Member Functions</it></it></it></it></it></it>	<pre>back_inserter (Container& x) template <container> class front_insert_iterator : public iterator<athermiterator of="" state="" state<="" td="" term="" the=""></athermiterator></container></pre>	
template <it> class reverse_ite :iterator<iterato iterato="" iterator="" iterator<="" td=""><td>rator r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::reference traits<it>::reference Public Types inter, reference, blic Member Functions);</it></it></it></it></it></it></td><td><pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></pre></td></iterato></it>	rator r_traits <it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::reference traits<it>::reference Public Types inter, reference, blic Member Functions);</it></it></it></it></it></it>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite :iterator:iterato iterato iterato iterato iterato iterato iterato iterato iterato riterato iterator reverse_iterator reverse_iterator reverse_iterator</it>	rator r_traits <it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::reference traits<it>::reference Public Types inter, reference, blic Member Functions);</it></it></it></it></it></it>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite :iterator<iterator <u="" iterator="" remplate="" reverse_iterator=""></iterator></it>	rator r_traits <it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::reference traits<it>::reference Public Types inter, reference, blic Member Functions); iterator_type i); const_reverse_iterator& ri);</it></it></it></it></it></it>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite citerator iterator reverse_iterator template <u> reverse_iterator template <u> reverse_iterator reverse_iterator</u></u></it>	<pre>rator r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::pointer, r_traits<it>::reference> Public Types inter, reference, blic Member Functions); iterator_type i);</it></it></it></it></it></pre>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite :iterator<iterator <u="" iterator="" reverse_iterator="" template=""> reverse_iterator(template <u></u></iterator></it>	rator r_traits <it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::reference traits<it>::reference Public Types inter, reference, blic Member Functions); iterator_type i); const_reverse_iterator& ri); const_reverse_iterator</it></it></it></it></it></it>	<pre>back_inserter (Container& x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite :iterator<iterator <u="" iterator="" reverse_iterator(="" template=""> reverse_iterator(template <u> reverse_iterator(template <u> reverse_iterator(template <u> reverse_iterator(</u></u></u></iterator></it>	rator r_traits <it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::reference traits<it>::reference Public Types inter, reference, blic Member Functions); iterator_type i); const_reverse_iterator& ri);</it></it></it></it></it></it>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite :iterator<iterator <u="" iterator="" reverse_iterator="" template=""> reverse_iterator template <u> reverse_iterator construction reverse_iterator reverse_</u></iterator></it>	rator r_traits <it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::reference public Types inter, reference, blic Member Functions); iterator_type i); const_reverse_iterator const_reverse_iterator (J) & ri); Iterator>& operator=(onst_reverse_iterator (J) & ri);</it></it></it></it></it>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite iterator reverse_iterator template <u> reverse_iterator itemplate <u> reverse_iterator iterator_type</u></u></u></u></u></u></u></it>	rator r_traits <it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::difference_type, r_traits<it>::pointer, r_traits<it>::reference> Public Types inter, reference, blic Member Functions); iterator_type i); const reverse_iterator& ri); const reverse_iterator <pre>const reverse_iterator (onst reverse_iterator</pre> // Iterator>& operator=(onst reverse_iterator // base() const;</it></it></it></it></it></it></it>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></pre>	
template <it> class reverse_ite :iterator<iterator <u="" iterator="" reverse_iterator="" template=""> reverse_iterator template <u> reverse_iterator iterator template <u> reverse_iterator reve</u></u></iterator></it>	rator r_traits <it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::reference public Types inter, reference, blic Member Functions); iterator_type i); const reverse_iterator& ri); Const reverse_iterator ### rips ###</it></it></it></it></it>	<pre>back_inserter (Container& x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre>	
pointer, template <it> class reverse_ite :iterator<iterato <u="" iterato="" iterator="" reverse_iterator="" template=""> reverse_iterator template <u> reverse_iterator iterator_type reference pointer reverse_iterator8</u></u></u></u></u></u></iterato></it>	rator r_traits <it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::pointer, r_traits<it>::pointer, r_traits<it>::pointer, r_traits<it>::reference> Public Types inter, reference, blic Member Functions); iterator_type i); const reverse_iterator& ri); const reverse_iterator<u>& ri); Iterator>& operator=(onst reverse_iterator<u>& ri); base() const; operator*() const; operator+();</u></u></it></it></it></it></it></it></it></it></it>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></pre>	
pointer, template <it> class reverse_ite :iterator<iterator <u="" iterator="" reverse_iterator="" template=""> reverse_iterator template <u> reverse_iterator iterator template <u> reverse_iterator reverse_iterator iterator template <u> reverse_iterator reverse_iterator iterator_type reference pointer reverse_iterator reverse_iterator reverse_iterator</u></u></u></iterator></it>	<pre>rator reference; rator r_traits<it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::openator=category, r_traits<it>::difference_type, r_traits<it>::pointer, r_traits<it>::pointer, r_traits<it>::pointer, r_traits<it>::reference> Public Types inter, reference, blic Member Functions); iterator_type i); const reverse_iterator& ri); const reverse_iterator& ri); Iterator>& operator=(onst reverse_iterator<u>& ri); base() const; operator=() const; operator=(); operator+(); operator+(); operator+(); operator+(); operator+(); operator+();</u></it></it></it></it></it></it></it></it></it></pre>	<pre>back_inserter (Container& x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre>	
pointer, template <it> class reverse_ite :iterator<iterato <u="" iterato="" iterator="" reverse_iterator="" template=""> reverse_iterator template <u> reverse_iterator iterator_type reference pointer reverse_iterator8</u></u></u></u></u></u></iterato></it>	rator r_traits <it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::difference_type, r_traits<it>::pointer, r_traits<it>::reference> Public Types inter, reference, bblic Member Functions); iterator_type i); const reverse_iterator& ri); const reverse_iterator& ri); terator>& operator=(onst reverse_iterator<u>& ri); base() const; operator->() const; operator-+(); operator-+(); operator();</u></it></it></it></it></it></it></it></it>	<pre>back_inserter (Container& x) template <container> class front_insert_iterator : public iterator<output_iterator_tag, void="" void,=""> Public Member Functions front_insert_iterator(Container& x); front_insert_iterator& operator*(); front_insert_iterator& operator*(); front_insert_iterator& operator++(int); template<container> front_insert_iterator<ore> front_insert_iterator front_insert_iterator</ore></container></output_iterator_tag,></container></pre> (ContainerA x) template <container> front_inserter (Container& x) template <container> class insert_iterator : public iterator<output_iterator_tag, functions="" insert_iterator&="" member="" operator="(const_insert_i</td" public="" void="" void,=""></output_iterator_tag,></container></container>	
dass reverse_ite :iterator <iterator difference_type;="" iterator_type,="" pod="" reverse_iterator(="" reverse_iterator(<="" td=""><td><pre>rator r_traits<it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::difference_type, r_traits<it>::pointer, r_traits<it>::reference> Public Types inter, reference, biblic Member Functions); iterator_type i); const reverse_iterator& ri); const reverse_iterator& ri); Iterator>& operator=(onst reverse_iterator<u>& ri); base() const; operator+(); operator+(); operator-(); operator-(); operator-(); operator-(int); operator-(difference_type) const</u></it></it></it></it></it></it></it></pre></td><td><pre>back_inserter (Container& x) template <container> dass front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre></td></iterator>	<pre>rator r_traits<it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::difference_type, r_traits<it>::pointer, r_traits<it>::reference> Public Types inter, reference, biblic Member Functions); iterator_type i); const reverse_iterator& ri); const reverse_iterator& ri); Iterator>& operator=(onst reverse_iterator<u>& ri); base() const; operator+(); operator+(); operator-(); operator-(); operator-(); operator-(int); operator-(difference_type) const</u></it></it></it></it></it></it></it></pre>	<pre>back_inserter (Container& x) template <container> dass front_insert_iterator : public iterator<output_iterator_tag, td="" void,="" void<=""></output_iterator_tag,></container></pre>	
pointer, template <it> class reverse_ite :iterator:iterato iterato iterato iterato iterato iterator iterator iterator iterator iterator reverse_iterator template <u> reverse_iterator template <u> reverse_iterator template <u> reverse_iterator template <u> reverse_iterator reverse_iterator</u></u></u></u></it>	rator r_traits <it>::iterator_category, r_traits<it>::iterator_category, r_traits<it>::value_type, r_traits<it>::value_type, r_traits<it>::difference_type, r_traits<it>::pointer, r_traits<it>::pointer, r_traits<it>::reference> Public Types inter, reference, blic Member Functions); iterator_type i); const reverse_iterator& ri); const reverse_iterator<u>& ri); Iterator>& operator=(onst reverse_iterator<u>& ri); base() const; operator+() const; operator+(); operator-(); operator-(); operator-(); operator-(int);</u></u></it></it></it></it></it></it></it></it>	<pre>back_inserter (Container% x) template <container> class front_insert_iterator: : public iterator<output_iterator_tag, void="" void,=""></output_iterator_tag,></container></pre>	

inserter

distance

advance

template <InputIterator>

```
template <Container, Iterator>
                                                    5. Smart Pointers
insert iterator<Container>
(Container& x, Iterator i)
                                                     template <T>
                                                     class intrusive ptr
                                                                         Public Types
                                                                                                          template <U>
iterator traits<InputIterator>::difference type
                                                            element_type;
                                                                    Public Member Functions
(InputIterator first, InputIterator last)
                                                                                                          template <U>
                                                     intrusive_ptr();
                                                                                                          void
                                                     intrusive_ptr(√*
                                                                                                          void
template <InputIterator, Distance> void
                                                                                         bAddRef =
                                                     intrusive_ptr(const intrusive_ptr& ip);
(InputIterator& i, Distance n)
                                                     template <U>
                                                     intrusive_ptr(const intrusive_ptr<U>&);
                                                                                                          int
                                                     intrusive ptr& operator=(const intrusive ptr&);
                                                                                                          bool
                                                     template <U>
                                                     intrusive ptr& operator=(const intrusive ptr<U>&);
                                                                                                           typedef T*
                                                     intrusive_ptr& operator=(T*);
                                                                                                           oool
                                                                    operator *() const;
                                                                    operator ->() const;
                                                                                                          void
                                                                    get() const;
                                                     void
                                                     void
                                                                    swap(this_type& ip);
                                                                    attach(T* pObject);
                                                     void
                                                                    detach();
                                                     typedef T* (this_type::*bool_)() const;
                                                                    operator bool_() const;
                                                     bool
                                                                    operator!() const;
                                                            Global intrusive_ptr Functions & Operators
                                                     template <T>
                                                         get_pointer(
                                                                    const intrusive ptr<T>&
                                                     intrusivePtr)
                                                     template <T>
                                                     void swap(
                                                                                              iPtr1.
                                                                    intrusive ptr<T>&
                                                                                             iPtr2)
                                                                    intrusive ptr<T>&
                                                     template <T, U>
                                                     bool operator == (
                                                     bool operator!=(
                                                     bool operator (intrusive ptr<T> const& iPtr1,
                                                                     intrusive ptr<U> const& iPtr2);
                                                     template <T>
                                                     bool operator == (
                                                     bool operator!=(intrusive ptr<T>const& iPtr1,
                                                                                               p);
                                                     template <T>
                                                     bool operator == (
                                                     bool operator!=(T*
                                                                     intrusive ptr<T> const& iPtr2);
                                                                                                          int
                                                                                                           oool
                                                     template <T, U>
                                                     intrusive ptr<T> static pointer cast(
                                                                                                           :ypedef T*
                                                     intrusive_ptr<T> dynamic_pointer_cast(
                                                                     const intrusive ptr<U>& iPtr);
                                                                                                           ool
                                                                                                          void
                                                     template <T, Deleter = smart ptr deleter<T> >
                                                     class linked ptr
                                                      public linked ptr base
                                                                                                           template <T>
                                                                          Public Types
                                                            element type;
                                                                    Public Member Functions
                                                     linked_ptr();
                                                     template <U>
                                                     linked ptr(U*
                                                                                  pValue);
```

```
linked ptr(const linked ptr&
                                    linkedPtr):
template <Ù, D>
linked_ptr(const linked ptr<U, D>& linkedPtr);
 linked ptr& operator=(const linked ptr&);
 template <U, D>
 linked ptr& operator=(const linked ptr<U, D>&);
linked ptr& operator=(U* pValue);
             reset(U* pValue);
             reset();
             operator*() const;
             operator->() const;
             get() const;
             use count() const;
             unique() const;
            (this type::*bool )() const;
             operator bool () const;
             operator!() const;
              force delete();
         Global linked ptr Functions & Operators
 template <T, D>
 * get_pointer(const linked ptr<T, D>& linkedPtr);
template <T, TD, U, UD>
bool operator == (
bool operator!=(
bool operator<
             const linked ptr<T, TD>& linkedPtr1,
             const linked ptr<U, UD>& linkedPtr2);
template <T, Deleter = smart array deleter<T> >
class linked array
                     Public Types
        element_type;
               Public Member Functions
 linked_array(⊺*
                                  pArray = NULL);
linked_array(const linked array& linkedArray);
linked array& operator=(const linked array&);
linked array& operator=(T* pArray);
 void reset(T* pArray = NULL);
 "& operator[](ptrdiff t i) const;
             operator*() const;
operator->() const;
             get() const;
             use_count() const;
              unique() const;
           (this type::*bool_)() const;
             operator bool_() const;
             operator!() const;
             force_delete();
        Global linked array Functions & Operators
 * get_pointer(const linked_array<T>& linkedArray)
template <T, TD, U, UD>
bool operator==(
bool operator!=(
 bool operator<
         const linked array<T, TD>& linkedArray1,
         const linked array<U, UD>& linkedArray2);
```

reference

operator[](difference_type) const;

class safe_object	template <t, array="" deleter="smart" deleter<t=""> ></t,>	allocator_type& get_allocator();	template <t, a,="" d=""></t,>
Public Member Functions	class scoped_array	<pre>void set_allocator(</pre>	<pre>void swap(shared_array<t, a,="" d="">& sharedArray1,</t,></pre>
bool has_references() const;	Public Types		Silateu_attay(1, A, D/G SilateuAttay2),
bool mas_reverences() conse,	<pre>T element_type;</pre>	Global shared_ptr Functions & Operators template < T, A, D>	template <t, ta,="" td,="" u,="" ua,="" ud=""></t,>
template< <i>T</i> >	Public Member Functions	T* get_pointer (const shared ptr< <i>T</i> , <i>A</i> , <i>D</i> >&);	<pre>bool operator==(bool operator!=(</pre>
class safe ptr	scoped_array(T* pArray = NULL)		bool operator (
		template <t, a,="" d=""> void swap(shared ptr<t, a,="" d="">& sharedPtr1,</t,></t,>	<pre>const shared_array<t, ta,="" td="">& sharedArray1,</t,></pre>
Public Member Functions	<pre>void reset(T* pArray = NULL) void swap(this type& scopedArray)</pre>	shared ptr <t, a,="" d="">& sharedPtr2);</t,>	const shared_array <u, ua,="" ud="">& sharedArray2);</u,>
<pre>safe_ptr(); safe ptr(T* p0bject);</pre>	word Swap(this_typed scopedniray)		struct smart inter-deleter
<pre>safe_ptr(const this type& safePtr);</pre>	<pre>78</pre>	template <t, ta,="" td,="" u,="" ua,="" ud=""></t,>	<pre>struct smart_ptr_deleter template <t> struct smart_ptr_deleter</t></pre>
	T* get() const;	<pre>bool operator==(bool operator!=(</pre>	void operator()(const T* p) const;
this_type& operator= (const this_type& safePtr); this type& operator= (T* const pObject);	typedef T* (this_type::*bool_)() const;	bool operator (
this_typed operator=(1 const pobject),	<pre>operator bool_() const;</pre>	const shared_ptr <t, ta,="" td="">& sharedPtr1,</t,>	<pre>template <> struct smart_ptr_deleter<void> void operator()(const void* p) const;</void></pre>
<pre>bool operator==(const this_type& safePtr) const;</pre>	bool operator!() const;	const shared_ptr <u, ua,="" ud="">& sharedPtr2);</u,>	void operator()(const void p) const,
bool empty() const;	Global scoped_array Functions & Operators	template <t, ta,="" td,="" u,="" ua,="" ud=""></t,>	<pre>template <> struct smart_ptr_deleter<const void=""></const></pre>
void reset(safe object* pObject);	<pre>template <t, d=""> T* get_pointer(const scoped array<t,d>&);</t,d></t,></pre>	shared_ptr <t, ta,="" td=""> static shared pointer cast (</t,>	void operator()(const void* p) const;
<pre>void reset();</pre>	get_pointer(const scoped_array(1,074),	dynamic_shared_pointer_cast(struct smart array deleter
<pre>operator T*() const;</pre>	template <t, d=""></t,>	const shared_ptr <u, ua,="" ud="">&);</u,>	template <t> struct smart array deleter</t>
T& operator*() const;	<pre>void swap(scoped_array <t, d="">& scopedArray1,</t,></pre>	tomplate of th	<pre>void operator()(T* p) const;</pre>
<pre>T* operator->() const;</pre>	scopeu_array (1, b)a scopeumray2);	<pre>template <t, u=""> shared ptr<t, allocator,="" deleter<t="" ptr="" smart=""> ></t,></t,></pre>	1
T* get() const;	template <t, d=""></t,>	static_pointer_cast (<pre>template <> struct smart_array_deleter<void> void operator()(void* p) const;</void></pre>
bool unique() const;	<pre>bool operator<(</pre>	dynamic_pointer_cast(
• " .	const scoped array<7, D>& scopedArray2);	<pre>const_pointer_cast (</pre>	template <t, allocator="eastl::allocator"></t,>
<pre>typedef T* (this_type::*bool_)() const; operator bool_() const;</pre>			class weak_ptr
bool operator!() const;	template <t, allocator="eastl::allocator," deleter<="" td=""><td>template <t, ta,="" td,="" u,="" ua,="" ud=""> shared ptr<t, ta,="" td=""></t,></t,></td><td></td></t,>	template <t, ta,="" td,="" u,="" ua,="" ud=""> shared ptr<t, ta,="" td=""></t,></t,>	
Global safe_ptr Functions & Operators	= smart ptr deleter <t> ></t>	const shared pointer cast(Public Types
template <t></t>	class shared_ptr	const shared_ptr <u, ua,="" ud="">&);</u,>	7 element_type;
bool operator==(Public Types		Public Member Functions weak ptr(const allocator type& allocator =
bool operator!= (const safe_ptr< <i>T</i> >& safePtr, const <i>T</i> * pObject);	<pre>7 element_type;</pre>	template <t, allocator="eastl::allocator," deleter<="" td=""><td>eastl::allocator("EASTL classname"));</td></t,>	eastl::allocator("EASTL classname"));
template <t></t>	Public Member Functions	= smart array deleter <t> ></t>	<pre>weak_ptr(const weak_ptr& weakPtr);</pre>
bool operator (const safe_ptr< <i>T</i> >& safePtrA,	<pre>shared_ptr(const allocator_type& allocator =</pre>	class shared_array	template <u></u>
	eastl::allocator("EASTL classname");	Public Types	<pre>weak_ptr(const weak_ptr<u, allocator="">& weakPtr);</u,></pre>
<pre>bool operator<(const safe_ptr<t>& safePtrA,</t></pre>	<pre>east1::allocator("EASTL classname"); template <u></u></pre>	_ <i>,</i>	
<pre>bool operator<(const safe_ptr<t>& safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr);</u>	Public Types 7 element_type; Public Member Functions	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators&="" d="" template="" weakptr);=""> weak_ptr(const shared_ptr<u, a,="" d="">& sharedPtr);</u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>% safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""></u,></u>	Public Types 7 element_type; Public Member Functions shared_array(7* pArray = NULL,	<pre>weak_ptr(const weak_ptr<u, allocator="">& weakPtr); template <u, a,="" d=""></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>& safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr);</u,></u,></u>	Public Types 7	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators%="" d="" template="" weakptr);=""> 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,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>% safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr);</u,></u,></u,></u,></u>	Public Types 7 element_type; Public Member Functions shared_array(7* pArray = NULL,	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators&="" d="" template="" weakptr);=""> 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="">&); template <u, a,="" d=""></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr<7>% safePtrA, const safe_ptr<7>% safePtrB); template <7, Deleter = smart ptr deleter<7> > class scoped_ptr Public Types 7	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const weak_ptr<u, a="">& weakPtr); template <u, a,="" d=""></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators%="" d="" template="" weakptr);=""> 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,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>% safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template (U)* shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template (U, A, D)* 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,</u,></u,></u,>	Public Types 7	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators&="" d="" template="" weakptr);=""> 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="">&); template <u, a,="" d=""></u,></u,></u,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>% safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr, static_cast_tag); template <u, a,="" d=""></u,></u,></u,></u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const;</t,></u,></u,></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr<7>% safePtrA, const safe_ptr<7>% safePtrB); template <7, Deleter = smart ptr deleter<7> > class scoped_ptr Public Types 7	eastl::allocator("EASTL classname"); template (U)* shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template (U, A, D)* 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="">& shared_ptr(U, A, D>& shared_ptr(U, A, D)* shared_ptr(const shared_ptr<u, a,="" d="">& shared_ptr(U, A, D>& shared_ptr(U, A, D)* shared_ptr(const shared_ptr<u, a,="" d="">& shared_ptr(U, A, D>& shared_ptr(U, A, D)*</u,></u,></u,></u,></u,></u,></u,>	Public Types 7	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators%="" d="" template="" weakptr);=""> 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="">%); template <u, a,="" d=""> weak_ptr% operator=(const shared_ptr<u, a,="" d="">%);</u,></u,></u,></u,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>% safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr, static_cast_tag); template <u, a,="" d=""></u,></u,></u,></u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>& safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template (U)* shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template (U, A, D)* shared_ptr(const shared_ptr <u, a,="" d="">& sharedPtr); template <u, a,="" d=""> shared_ptr(const weak_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,>	Public Types 7	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators%="" d="" template="" weakptr);=""> 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="">&); template <u, a,="" d=""> weak_ptr% operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int use_count() const;</t,></u,></u,></u,></u,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>& safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr&U, A, D>& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr <t>& safePtrA, const safe_ptr<t>& safePtrB); template <t, deleter="smart" deleter<t="" ptr="">> class scoped_ptr Public Types T element_type; Public Member Functions scoped_ptr(T* pValue = NULL); void reset(T* pValue = NULL); void swap(this_type& scopedPtr); T& operator*() const; T* operator->() const; T* get() const;</t,></t></t>	eastl::allocator("EASTL classname"); template (U)* shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template (U, A, D)* shared_ptr(const shared_ptr <u, a,="" d="">& sharedPtr); template <u, a,="" d=""> shared_ptr(const weak_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,>	Public Types 7	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators%="" d="" template="" weakptr);=""> 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>&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>& safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> 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,</u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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="" a,="" operator="(const" weak_ptr&="" weak_ptr<u,="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr <t>& safePtrA, const safe_ptr<t>& safePtrB); template <t, deleter="smart" deleter<t="" ptr="">> class scoped_ptr Public Types T element_type; Public Member Functions scoped_ptr(T* pValue = NULL); void reset(T* pValue = NULL); void swap(this_type& scopedPtr); T& operator*() const; T* operator->() const; T* get() const;</t,></t></t>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const weak_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators%="" d="" template="" weakptr);=""> 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>\Uext{\\te\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\</u,></u,></u,></pre>
<pre>bool operator<(const safe ptr<t>% safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> 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,</u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr <t>& safePtrA, const safe_ptr<t>& safePtrB); template <t, deleter="smart" deleter<t="" ptr="">> class scoped_ptr Public Types Telement_type; Public Member Functions scoped_ptr(T* pValue = NULL); void reset(T* pValue = NULL); void swap(this_type& scopedPtr); T& operator*() const; T* operator->() const; T* operator->() const; T* operator->() const; Typedef T* (this_type::*bool_)() const; operator bool_() const; bool operator!() const; Global scoped_ptr Functions & Operators template <t, d=""></t,></t,></t></t>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr, template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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="" a,="" operator="(const" weak_ptr&="" weak_ptr<u,="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>& safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr, template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators%="" d="" template="" weakptr);=""> 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>\Uext{\\te\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\</u,></u,></u,></pre>
bool operator<(const safe_ptr <t>& safePtrA, const safe_ptr<t>& safePtrB); template <t, deleter="smart" deleter<t="" ptr="">> class scoped_ptr Public Types Telement_type; Public Member Functions scoped_ptr(T* pValue = NULL); void reset(T* pValue = NULL); void swap(this_type& scopedPtr); T& operator*() const; T* operator->() const; T* operator->() const; T* operator->() const; Typedef T* (this_type::*bool_)() const; operator bool_() const; bool operator!() const; Global scoped_ptr Functions & Operators template <t, d=""></t,></t,></t></t>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr, template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>weak_ptr(const weak_ptr<u, <u,="" a,="" allocators%="" d="" d,="" shared_ptr<u,="" template="" weak_ptr(const="" weakptr);="">& sharedPtr); weak_ptr& operator=(const weak_ptr& weakPtr); template <u, a="" a,="" operator="(const" weak_ptr&="" weak_ptr<u,="">&); template <u, a,="" d="" d,="" operator="(const" shared_ptr<u,="" weak_ptr&="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>& safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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="" a,="" operator="(const" weak_ptr&="" weak_ptr<u,="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></pre>
<pre>bool operator<(const safe_ptr<t>& safePtrA,</t></pre>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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,="" d=""> weak_ptr& operator=(const weak_ptr<u, a="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr<7>& safePtrA, const safe_ptr<7>& safePtrB); template <7, Deleter = smart ptr deleter<7>> class scoped_ptr Public Types 7	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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="" a,="" operator="(const" weak_ptr&="" weak_ptr<u,="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr <t>& safePtrA, const safe_ptr<t>& safePtrB); template <t, deleter="smart" deleter<t="" ptr=""> > class scoped_ptr Public Types Telement_type; Public Member Functions scoped_ptr(T* pValue = NULL); void reset(T* pValue = NULL); void swap(this_type& scopedPtr); T& operator*() const; T* operator->() const; T* get() const; typedef T* (this_type::*bool_)() const; operator bool_() const; Global scoped_ptr Functions & Operators template <t, d=""> void swap(scoped_ptr<t, d="">& scopedPtr); template <t, d=""> bool operator</t,></t,></t,></t,></t></t>	<pre>eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u,></u></pre>	Public Types 7	<pre>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,="" d=""> weak_ptr& operator=(const weak_ptr<u, a="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr<7>& safePtrA, const safe_ptr<7>& safePtrB); template <7, Deleter = smart ptr deleter<7>> class scoped_ptr Public Types 7	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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,="" d=""> weak_ptr& operator=(const weak_ptr<u, a="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr <t>& safePtrA, const safe_ptr<t>& safePtrB); template <t, deleter="smart" deleter<t="" ptr=""> > class scoped_ptr Public Types Telement_type; Public Member Functions scoped_ptr(T* pValue = NULL); void reset(T* pValue = NULL); void swap(this_type& scopedPtr); T& operator*() const; T* operator->() const; T* get() const; typedef T* (this_type::*bool_)() const; operator bool_() const; Global scoped_ptr Functions & Operators template <t, d=""> void swap(scoped_ptr<t, d="">& scopedPtr); template <t, d=""> bool operator</t,></t,></t,></t,></t></t>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a=""> shared_ptr(const shared_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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,="" d=""> weak_ptr& operator=(const weak_ptr<u, a="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></u,></pre>
bool operator<(const safe_ptr <t>& safePtrA, const safe_ptr<t>& safePtrB); template <t, deleter="smart" deleter<t="" ptr=""> > class scoped_ptr Public Types Telement_type; Public Member Functions scoped_ptr(T* pValue = NULL); void reset(T* pValue = NULL); void swap(this_type& scopedPtr); T& operator*() const; T* operator->() const; T* get() const; typedef T* (this_type::*bool_)() const; operator bool_() const; Global scoped_ptr Functions & Operators template <t, d=""> void swap(scoped_ptr<t, d="">& scopedPtr); template <t, d=""> bool operator</t,></t,></t,></t,></t></t>	eastl::allocator("EASTL classname"); template <u> shared_ptr(U* pValue); shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr& sharedPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr); template <u, a,="" d=""> shared_ptr(const weak_ptr<u, a="">& weakPtr); template <u, a,="" d=""> shared_ptr(const shared_ptr<u, a,="" d="">& sharedPtr,</u,></u,></u,></u,></u,></u,></u,></u>	Public Types 7	<pre>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,="" d=""> weak_ptr& operator=(const weak_ptr<u, a="">&); template <u, a,="" d=""> weak_ptr& operator=(const shared_ptr<u, a,="" d="">&); shared_ptr<t, allocator=""> lock() const; int</t,></u,></u,></u,></u,></u,></u,></u,></pre>

EASTL 1.10.03 Quick Reference v1.0