Container	Operation		Short Description
	Category	Function-like Macros Prototype	
CCXLL	Create	*CCXLL* ccxll (TYPE) list;	Create a ccxll container list of type TYPE. This is implemented by a C struct to construct a list container.
		CCXLL ccxll_pckd (TYPE) list;	Create a packed ccxll container list of type TYPE. This is implemented by an aligned C struct to construct a list container.
		CCXLL ccxll_extd (TYPE, unsigned *num*, **align**) list;	Create a ccxll container list of type TYPE with *num* iterators. The container is packed when **align** is PACKED. Otherwise, set NORMAL for default.
	Initialize	void ccxll_init (CCXLL);	Initialize the ccxll container. CAUTION: Every container must be initialized right after its creation.
		void ccxll_iter_init (ITER, CCXLL);	Initialize the iterator for the ccxll container. CAUTION: Every iterator is implicitly initialized when the container it belongs to is initialized.
	Destroy	*stat* ccxll_free (CCXLL);	Deallocate all elements in the container manually. CAUTION: Every container should be destroyed before the program terminates.
	Access	TYPE& ccxll_front (CCXLL);	Return a reference to the first element. It's an undefined behavior if the container is empty.
		TYPE& ccxll_back (CCXLL);	Return a reference to the last element. It's an undefined behavior if the container is empty.
	Capacity	int ccxll_size (CCXLL);	Return the number of the elements in the container. Return 0 if the container is empty.
		int ccxll_empty (CCXLL);	Check whether the container is empty. Return 1 if the container is empty, and return 0 if it is not.
	Modifiers	*stat* ccxll_push_front (CCXLL, TYPE **value**);	Insert an element at the beginning. This makes a copy of **value** into the container.
		stat ccxll_push_back (CCXLL, TYPE **value**);	Insert an element at the end. This makes a copy of **value** into the container.
		stat ccxll_pop_front (CCXLL);	Remove the first element. There is nothing modified if the container is empty.
		stat ccxll_pop_back (CCXLL);	Remove the last element. There is nothing modified if the container is empty.
		stat ccxll_insert (ITER, TYPE **value**);	Insert an element at the position where the iterator points. This makes a copy of **value** into the container.
		stat ccxll_erase (ITER);	Erase an element at the position where the iterator points. There is nothing modified if the container is empty.
		stat ccxll_swap (CCXLL_A, CCXLL_B);	Swap two containers of the same type. It may cause unexpected errors if two containers are of different types.
		stat ccxll_resize (CCXLL, int *num*, TYPE **value**);	Resize the container to contain *num* elements. If the current size is smaller than *num* elements, then fills with **value**. Otherwise, it truncates.
		stat ccxll_clear (CCXLL);	Remove all elements in the container. This does not deallocate all elements in the container.
	Operations	*stat* ccxll_move_range (ITER_P, ITER_L, ITER_R);	Move the elements in the range [ITER_L, ITER_R) to position where ITER_P points. These three iterators should be affiliated to the same ccxll container.
		stat ccxll_merge[_extd] (CCXLL_A, CCXLL_B[, (*LEQ)()]);	Merge two sorted lists from CCXLL_B into CCXLL_A. Merge with the default comparator CCXLL_LEQ_COMPAR if _extd postfix is not specified.
		stat ccxll_sort[_extd] (CCXLL[, (*LEQ)()]);	Sort all elements in CCXLL. Sort with the default comparator CCXLL_LEQ_COMPAR if _extd postfix is not specified.
		stat ccxll_reverse_range (ITER_L, ITER_R);	Reverse the elements in the range [ITER_L, ITER_R]. This performs in constant time no matter how large the range is.
	Comparators	LEQ CCXLL_LEQ_COMPAR (ITER_L, ITER_R); (abbrev. XLEQ)	Compare values by passing and dereferencing two iterators for sorting algorithms. Return 1 iff the value pointed by ITER_L is not greater than the value pointed by ITER_R.
	Iterators	ITER ITER[_NTH] (CCXLL[, *num*]);	Return the *num*-th iterator of CCXLL. Return the zeroth iterator if _NTH postfix is not specified.
		TYPE& DREF (ITER);	Return a reference to the element. It's an undefined behavior if the iterator is not invalid.
		TYPE& DREF_[PREV|NEXT] (ITER);	Return a reference to the previous/next element. It's an undefined behavior if the iterator is not invalid.
		void ccxll_iter_copy (ITER_DST, ITER_SRC);	Copy the iterator from ITER_SRC to ITER_DST. It's not acceptable to assign the iterator by assignment operator.
		void ccxll_iter_[head|tail] (ITER);	Set the iterator to the head/tail of the container. The head/tail of the container is the sentinel node pointing to the first/last element.
		void ccxll_iter_[begin|end] (ITER);	Set the iterator to the first/last element usually. Set the iterator to the tail/head if the container is empty.
		int ccxll_iter_at_[head|tail] (ITER);	Check whether the iterator points to the head/tail of the container. Return 1 if it is true. Otherwise, return 0.
		int ccxll_iter_at_[begin|end] (ITER);	Check whether the iterator points to the first/last element. Return 1 if it is true. Otherwise, return 0.
		void* ccxll_iter_[incr|decr] (ITER);	Move the iterator forward/backward by one element. Return NULL iff the iterator doesn't point to any element before and after moving.
		stat ccxll_iter_advance (ITER, int **diff**);	Move the iterator by **diff** element(s). (regard forward as positive) The iterator will stop at the sentinel node if there is no element left to iterate over.
		stat ccxll_iter_distance (ITER_A, ITER_B, int ***dist**);	Return the distance between ITER_A and ITER_B through the pointer **dist**. Return 0 if the distance between them cannot be determined.
	Traversal	*loop* CCXLL_[INCR|DECR] (ITER) *stat*;	Traverse all elements forward/backward. This is implemented by a single for statement.
		loop CCXLL_[INCR|DECR]_DREF (TYPE ***pval**, CCXLL) *stat*;	Traverse all elements forward/backward. This macro will not be activated if CCC_STRICT is defined.