**Design List DS**

**Goals:**

* Provide an API similar to **C++ std::list**
* Support **generic elements** using elementSize plus **typed wrappers**:
  + List\_int,
  + List\_char,
  + List\_float,
  + List\_objSPtr

Efficient implementation based on circular array: O(1) per push/pop.

* **Iterator support** with a unified API across DS (next/prev/is\_end/equals/get/set + advance/distance).
* **Performance**: O(1) push/pop /front/back/size/empty

O(n) clear and advance/distance.

insert/erase with iterator.

* Consistent style: macros for generic + typed API declarations, implementations, and binding.

**API List**

* push\_back(T value) - Append a new element to the end of the list. O(1).
* push\_front(T value) - Insert a new element at the front of the list. O(1).
* pop\_back(T\* out) - Remove the last element; optionally return it via out. O(1).
* pop\_front(T\* out) - Remove the first element; optionally return it via out. O(1).
* insert(iterator pos, T value) -Insert a new element before the given iterator position. O(1) if iterator is valid.
* erase(iterator pos, T\* out, iterator\* out\_next) - Remove the element at pos; optionally copy it to out and return the next iterator. O(1).
* head(T\* out) - Access the first element without removing it. Throws if list is empty. O(1).
* tail(T\* out)  
  Access the last element without removing it. Throws if list is empty. O(1).
* clear() - Remove all elements from the list. O(n).
* size() - Return the number of elements currently stored. O(1).
* empty() - Return true if the list has no elements. O(1).

**API – Iterator List**

* equals(object\*other,bool\*out\_equal) - Compares two iterators: true iff both point to the same
* next() - Move forward one element:

node = node ? node->next : NULL.

If already end(), stays NULL.

* prev() - Move backward one element.

If currently end() (node == NULL), jump to the list’s tail;

otherwise node = node->prev. If list is empty, remains NULL.

* get\_ref(void\*\* out\_ptr) - Returns a non-owning pointer to the current element bytes (i.e., node->payload). Throws if end().
* get\_cref(const void\*\* out\_ptr) - Same as get\_ref but const-qualified view.
* distance(object\* other, ptrdiff\_t\* out\_dist) - Linear distance between two iterators over the same list (O(n)). Positive if other is forward, negative if backward. Throws if different owners.
* advance(ptrdiff\_t n) - Move n steps (forward if n>0, backward if n<0). Stops early at end() / begin() boundaries. O(|n|).

**Error Handling:**

* Uses COOP’s THROW\_MSG / THROW\_MSG\_UNLESS macros for runtime errors.

**Expected errors:**

* **"List is empty"** → when calling pop\_front(), pop\_back(), front(), or back() on an empty list.
* **"Invalid Data Size"** → when the provided size ≠ elementSize (e.g., in push\_\*\_generic, pop\_\*\_generic, insert\_\*\_generic, erase\_node).
* **"Output pointer is NULL"** → when a required output pointer (out\_ptr, out\_node, out\_size) is NULL. (Note: in *pop\_\** a NULL *dst* is allowed to skip copying.)
* **"Iterator from foreign list"** → when using an iterator/node that does not belong to the current list (e.g., in insert\_before, insert\_after, erase).
* **"Iterator at end"** → when calling get\_ref/get\_cref on an iterator pointing to end().
* **"Invalid position"** → when attempting erase or insert with a NULL/invalid iterator.
* **"Out of memory"** → when allocation of a new node fails during push/insert.

**MACROS List (Int):**

#define DECLARE\_SPECIFIC\_LIST\_TYPE\_INT() \

DEF\_DERIVED\_CLASS(List\_int, GenericList); \

END\_DEF\_DERIVED(List\_int); \

DERIVED\_FUNCTIONS(List\_int, GenericList); \

MEM\_FUN\_DECL(List\_int, push\_back, int val); \

MEM\_FUN\_DECL(List\_int, push\_front, int val); \

MEM\_FUN\_DECL(List\_int, pop\_back, int\* out); \

MEM\_FUN\_DECL(List\_int, pop\_front, int\* out); \

MEM\_FUN\_DECL(List\_int, front, int\* out); \

MEM\_FUN\_DECL(List\_int, back, int\* out); \

MEM\_FUN\_DECL(List\_int, insert\_before, struct ListIterator\* pos, int val, struct ListIterator\*\* out\_it); \

MEM\_FUN\_DECL(List\_int, insert\_after, struct ListIterator\* pos, int val, struct ListIterator\*\* out\_it); \

MEM\_FUN\_DECL(List\_int, erase, struct ListIterator\* pos, int\* out, struct ListIterator\*\* out\_next); \

MEM\_FUN\_DECL(List\_int, size, MEM\_SIZE\_T\* out\_size); \

MEM\_FUN\_DECL(List\_int, clear); \

END\_DERIVED\_FUNCTIONS(List\_int)

**MACRO Iterator List:**

#define LIST\_ITER\_EQUALS(IT\_A, IT\_B, OUT\_BOOL) \

MFUN((IT\_A), equals), (object\*)(IT\_B), (OUT\_BOOL) CALL

#define LIST\_ITER\_NEXT(IT) \

MFUN((IT), next) CALL

#define LIST\_ITER\_PREV(IT) \

MFUN((IT), prev) CALL

#define LIST\_ITER\_GET\_REF(IT, OUT\_VOIDPTR) \

MFUN((IT), get\_ref), (void\*\*)(OUT\_VOIDPTR) CALL

#define LIST\_ITER\_GET\_CREF(IT, OUT\_CVOIDPTR) \

MFUN((IT), get\_cref), (const void\*\*)(OUT\_CVOIDPTR) CALL

#define LIST\_ITER\_DISTANCE(IT\_A, IT\_B, OUT\_DIST) \

MFUN((IT\_A), distance), (object\*)(IT\_B), (OUT\_DIST) CALL

#define LIST\_ITER\_ADVANCE(IT, N) \

MFUN((IT), advance), (N) CALL

**Data Structure Fields:**

* MEM\_SIZE\_T size;  
  Number of elements currently in the list
* MEM\_SIZE\_T elementSize;  
  Size in bytes of a single element stored in the list.
* \_\_ListNode\* head;  
  Pointer to the first node in the list (NULL if empty).
* \_\_ListNode\* tail;  
  Pointer to the last node in the list (NULL if empty).

**Unit tests:**

**Core behavior (int typed wrapper)**

* PushBack\_PopBack\_Int\_Basic — push\_back 0..N−1, then pop\_back into out; verify LIFO order, size transitions, and empty at end.
* PushFront\_PopFront\_Int\_Basic — push\_front 0..N−1, then pop\_front; verify FIFO order, size transitions, and empty at end.
* FrontBack\_ReadOnly\_Int — push a few values, call front/back (no removal) and check values; size unchanged.
* Size\_Empty\_Clear — interleave pushes/pops; check size() and empty() at each step; clear() leaves empty.

**Edge cases**

* SingleElement\_HeadTailOps — insert one element; exercise front/back/push\_\*/pop\_\* ensuring head/tail become NULL on last removal.
* EmptyList\_OpsThrow — on empty list: pop\_front/pop\_back/front/back must throw "List is empty".
* NullDst\_Allowed\_InPop — pop\_\* with dst == NULL skips copy but removes element; size decrements, no throw.

**Insert/erase via iterator**

* InsertBefore\_AtHead — insert\_before at begin(); new value becomes first; iterator returned points to new node.
* InsertAfter\_AtTail — insert\_after on last element; new value becomes last.
* Insert\_InMiddle — insert\_before/after in middle; links updated; values correct.
* Erase\_Head\_Tail\_Middle — erase at head, tail, and middle; out receives correct value when non-NULL; out\_next is valid; size decrements.
* Erase\_OnEnd\_Throws — erase(end, …) → "Invalid position".