Chapter 1

The auxiliary interfaces

1.1 Mask

A mask is a sequence that contains boolean data used for selection of items in a sequential container. It is not specified if a mask is a bit string (i.e. a strictly boolean array) or an array of chars or other integers used to hold the binary data. In all cases a value of the mask at a given position means *select* if it is different than zero, or *do not select* if it is zero.

The interface offered by the mask object is very small. Masks can't be resized but they have an allocator to be able to reclaim the memory they use when created. This allocator will be initialized to the current allocator when the mask is created.

```
typedef struct _Mask Mask;
typedef struct tagMaskInterface {
    int (*And)(Mask *src1,Mask *src2);
    int (*Clear)(Mask *m);
    Mask *(*Copy)(const Mask *src);
    Mask *(*Create)(size_t length);
    Mask *(*CreateFromMask)(size_t length,char *data);
    int (*Finalize)(Mask *m);
    int (*Or)(Mask *src1,Mask *src2);
    size_t (*PopulationCount)(const Mask *m);
    int (*Set)(Mask *m,size_t idx,int val);
    size_t (*Size)(Mask *);
    size_t (*Sizeof)(Mask *);
} iMask;
```

Operation	**Description**
And	Stores into src1 the result of a logical AND operation between each element of src1 with the corresponding element of src2.
Clear	Sets all elements of the mask to zero.

Copy	Allocates a new mask and copies the contents of the given one into it.
CreateFromMask	Creates a new mask with the specified length and copies the given data into the mask. Each character in the input data is transformed into the mask internal representation. The storage is obtained using the CurrentAllocator pointer.
Create	Creates a new mask with the specified length. The storage is obtained using the CurrentAllocator pointer. The data is initialized to zero.
Finalize	The memory used by the mask is reclaimed.
Not	Stores into src the result of a logical NOT operation: each bit is inverted.
Or	Stores into src1 the result of a logical OR operation between each element of src1 with the corresponding element of src2.
PopulationCount	Counts the number of entries different from zero in the given mask, returning the sum.
Set	Sets the given position to the given value if the value fits in the internal representation of the mask. If not, an implementation defined conversion occurs.
Size	The number of elements in the mask is returned.
Sizeof	The number of bytes used by the given mask. If the argument is NULL the number of bytes of the header structure is returned.

1.2 Memory management

Several interfaces implement different memory allocation strategies. This should give flexibility to the implementations, allowing it to use several memory allocation strategies within the same container.

The library starts with the `default` memory manager, that contains pointers to the default C memory management functions: malloc, free, realloc and calloc. Another memory manager is the `debug` memory manager that should implement more checking and maybe offer hooks to the debugger. The sample implementation shows how to implement several simple checks, but other implementations can extend this simple interface providing much more sophisticated controls.

```
typedef struct tagAllocator {
    void *(*malloc)(size_t);
    void  (*free)(void *);
    void *(*realloc)(void *,size_t);
    void *(*calloc)(size_t,size_t);
} ContainerAllocator;
extern ContainerAllocator * CurrentAllocator;
```

Each of the interface functions corresponds exctly to the specifications of the C language fnctions of the same name. The default memory management interface is initialized with the corresponding C Library functions.

1.3 The Heap interface: iHeap

Some containers can benefit from a cacheing memory manager that manages a stock of objects of the same size. This is not required and not all implementations may provide it. If they do, the interface is:

```
int (*UseHeap)(Container *c);
ContainerHeap *(*GetHeap)(Container *c);
```

In the sample implementation, many complex data structures are implemented using a heap. This allows automatically to have an iterator, since for looping all elements of the container it suffices to iterate the underlying heap. The standard interface for the heap is:

```
typedef struct tagContainerHeapInterface {
   void (*Clear)(ContainerHeap *heap);
   ContainerHeap *(*Create)(size_t ElementSize,
                  const ContainerAllocator *m);
   void (*Finalize)(ContainerHeap *heap);
   int (*FreeObject)(ContainerHeap *heap,void *element);
   ContainerHeap *(*InitHeap)(void *heap,size_t nbElements,
                  const ContainerAllocator *allocator);
   Iterator *(*NewIterator)(ContainerHeap *);
   void *(*NewObject)(ContainerHeap *heap);
   size_t (*Sizeof)(ContainerHeap *heap);
   int (*deleteIterator)(Iterator *it);
} ContainerHeapInterface;
```

Operation	**Description**
Clear	Releases all memory used by the free list and resets the heap object to its state as it was when created.
Create	Creates a new heap object that will use the given memory manager to allocate memory. All elements will have the given size. If the memory manager object pointer is `NULL` , the object pointed by CurrentAllocator will be used.

InitHeap	Initializes the given buffer to a heap header object designed to hold objects of `ElementSize` bytes. The heap will use the given memory manager. If the memory manager parameter is NULL the default memory manager is used. This function supposes that the `heap` parameter points to a contiguous memory space at least enough to hold a `ContainerHeap` object. The size of this object can be obtainer by using the `iHeap.Size` API with a NULL parameter. **Returns:**A pointer to the new ContainerHeap object or NULL if there is an error. Note that the pointer returned can be different from the passed in pointer due to alignment requirements.
newObject	The heap returns a pointer to a new object or NULL if no more memory is left.
FreeObject	Adds the given object to the list of free objects, allowing for recycling of memory without new allocations. The element pointer can be NULL .
Finalize	Destroys all memory used by the indicated heap and frees the heap object itself.
Sizeof	Returns the number of bytes used by the given heap, including the size of the free list. If the argument `"heap"` is NULL , the result is the size of the heap header structure (i.e. `sizeof(ContainerHeap)`.

1.4 Pooled memory interface: iPool

Many containers could benefit from a memory pool. A memory pool groups all allocations done in a specific context and can be released in a single call. This allows the programmer to avoid having to manage each single piece of memory like the basic interface.

```
typedef struct _tagPoolAllocatorInterface {
    Pool  *(*Create)(ContainerAllocator *m);
    void  *(*Alloc)(Pool *pool,size_t size);
    void  *(*Calloc)(Pool *pool,size_t size);
    void   (*Clear)(Pool *);
    void   (*Finalize)(Pool *);
} PoolAllocatorInterface;
```

Operation	**Description**
Create	Creates a new pool object that will use the given memory manager. If the argument is NULL , the object pointed by the CurrentAllocator will be used.

Alloc	Allocates size bytes from the pool pool. If there isn't enough memory to resize the pool the result is NULL .
Calloc	Allocates n objects of size "size" in a single block. All memory is initialized to zero. If there is no memory left it returns NULL .
Clear	Reclaims all memory used by the pool and leaves the object as it was when created.
Finalize	Reclaims all memory used by the pool and destroys the pool object itself.

1.5 Error handling Interface: iError

The "iError" interface provides a default strategy for handling errors. The "RaiseError" function will be used as the default error function within the creation function for all containers that support a per container instance error function.

```
typedef (*ErrorFunction)(const char *,int,...);
typedef struct {
  void        (*EmptyErrorFunction)(const char *fname,int code,...);
  int         (*NullPtrError)(const char *);
  void        (*RaiseError)(const char *fname,int code,...);
  ErrorFunction (*SetErrorFunction)(ErrorFunction);
  const char *(*StrError)(int errorCode);
} ErrorInterface;
```

Operation	**Description**
EmptyErrorFunction	This function can be used to ignore all errors within the library. It does nothing.
NullPtrError	Calls `RaiseError`, then returns `CONTAINER_ERROR_BADARG`
RaiseError	The parameter "fname" should be the name of the function where the error occurs. The "errcode" parameter is a negative error code. See 1.5. Other parameters can be passed depending on the error.

Error codes

The error codes defined by this specification are:

- CONTAINER_ERROR_BADARG One of the parameters passed to a function is invalid.
- CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

- CONTAINER_ERROR_INDEX The index is out of bounds. The library passes extra parameters when this error is invoked: the `container` pointer, and a `size_t` containing the the out of bounds index.
- CONTAINER_ERROR_READONLY The object is read-only and the operation would modify it
- CONTAINER_ERROR_INTERNAL Unspecified error provoked by a problem in the implementation.
- CONTAINER_ERROR_OBJECT_CHANGED A change in the underlying object has invalidated an iterator.
- CONTAINER_ERROR_FILE_READ Input error in a stream.
- CONTAINER_ERROR_FILE_WRITE Output error in a stream.
- CONTAINER_ERROR_CONTAINER_FULL Implementations can limit the maximum number of elements a container can hold. This error indicates that the limit is reached.
- CONTAINER_ERROR_BADPOINTER The debug implementation of `free()` has discovered an incorrect pointer attempting to be freed
- CONTAINER_ERROR_BUFFEROVERFLOW The debug implementation of `free()` discovered a buffer overflow.
- CONTAINER_ERROR_WRONGFILE You are trying to read a container from a stream that has no such container saved
- CONTAINER_ERROR_DIVISION_BY_ZERO The library has detected an attempt to divide by zero.
- CONTAINER_ERROR_OVERFLOW An overflow was detected in an arithmetic operation. Implementations are encouraged to detect overflow in all operations that can generate one and report it through this error.
- CONTAINER_ERROR_BADMASK The mask given to a `Select` or `SelectCopy` operation is of a different length than the length of the associated container. The library passes two pointers to the error function: The first to the container and the second to the mask.
- CONTAINER_ERROR_NOENT The library wants to open a file that doesn't exist or is not readable. A pointer to the name of the file is passed to the error function

1.6 The iterator interface

The iterator object exposes at least the functions "GetFirst", for initializing the loop, and "GetNext", for getting the next element in the sequence. The functions "NewIterator" and "deleteIterator" are specific to each container interface even if they all have the same syntax.

```
typedef struct _Iterator {
    void *(*GetNext)(Iterator *);
    void *(*GetPrevious)(Iterator *);
    void *(*GetFirst)(Iterator *);
    void *(*GetCurrent)(Iterator *);
    void *(*GetLast)(Iterator *);
    void *(*Seek)(Iterator *it,size_t pos);
    int (*Replace)(Iterator *it, void *data, int direction);
} Iterator;
```

Operation	**Description**
GetCurrent	Returns the element at the cursor position.
GetFirst	This function initializes the given iterator to the first element in the container. For sequential operators this is the element with index zero. In associative operators which element is the first is implementation defined and can change if elements are added or removed from the container.
GetNext	Positions de cursor at the next element and returns a pointer to its contents. If the iterator is at the end of the container the result is `NULL` and the iterator remains at the last position, a subsequent call to GetCurrent returns the last element.
GetPrevious	Positions de cursor at the previous element and returns a pointer to its contents. If the pointer is at the beginning of the container the result is `NULL` and the iterator remains at the beginning, a subsequent call to GetCurrent will return the first element of the container. This function is meaningful only in sequential containers. Its existence in associative containers is implementation defined. Even in sequential containers, it can be very expensive to find a previous element, for instance in single linked lists.
GetLast	Positions the cursor at the last element and returns a pointer to it. Returns `NULL` if the container is empty. If the container is read-only, a pointer to a copy of the element is returned.
Seek	Positions the given iterator at the indicated position and then returns a pointer to the element's data at that position. If the position is bigger than the last element of the container, the last element position will be used.

Replace	Replaces the current object pointed by the given iterator with the new data. If the `data` argument is NULL the element is erased from the container. If the *`direction`* parameter is different from zero, in sequential containers the iterator will point to the next element, otherwise it will point to the previous element. In associative containers this parameter is ignored and the iterator is always set to the next element, if any.

1.7 The observer interface

In its general form, the observer design pattern can be defined as a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

When a container changes its state, specifically when elements are added or removed, it is sometimes necessary to update relationships that can be very complex. The observer interface is designed to simplify this operation by allowing the container to emit *notifications* to other objects that have previously manifested interest in receiving them by *subscribing* to them. In general notifications are sent only when one of the defined operations for a container occur, mostly operations that change the number of elements.

This interface then, establishes a relationship between two software entities:

1. The container, that is responsible for sending the notifications when appropriate
2. The receiver, that is an unspecified object represented by its callback function that is called when a change occurs that matches the notifications specified in the subscription.

Since this relationship needs both objects, it will be finished when either object goes out of scope or breaks the relationship for whatever reason. Both objects can unsubscribe (terminate) their relationship.

Caveats

- It is in general a bad idea to modify the object being observed during a notification since this could trigger other notification messages. Implementations are not required to avoid this situation that is the responsibility of the programmer. Contrary to the iterator interface no error is issued when a possible infinite loop is started. Implementations may catch the error by limiting the number of recursive invocations of this interface but they are not required to do so.
- Since all messages sent by the containers have different type of information in the same two arguments that each message is associated with, there is no possible compile time control of the usage of the received pointers or numbers. The observer function must correctly discriminate between the different messages it can receive.

```
typedef void (*ObserverFunction)(const void *ObservedObject,
                                 unsigned Operation,
                                 void *ExtraInfo[]);

typedef struct tagObserverInterface {
    int (*Subscribe)(void *ObservedObject,
                     ObserverFunction callback, unsigned Operations);
    int (*Notify)(const void *ObservedObject,unsigned operation,
                  void *ExtraInfo1,void *ExtraInfo2);
    size_t (*Unsubscribe)(void *ObservedObject,
                          ObserverFunction callback);
} ObserverInterface;
extern ObserverInterface iObserver;
```

Operation	**Description**
Subscribe	Establishes the relationship between the observed object (argument 1) and the observer, represented by its callback (argument 2). The third argument establishes which operations are to be observed. This operation performs an allocation to register the relationship in the observer interface tables, therefore it can fail with an out of memory condition.
Notify	Used by the container to send a message to the receiver callback. The arguments correspond roughly to the arguments the callback function will receive. "Notify" will call all the objects that are observing `ObservedObject` and that have subscribed to one of the operations specified in the `Operation` argument. This implies a search through the observer interface table, and possibly several calls, making this function quite expensive. The time needed is roughly proportional to the number of registered callbacks and the complexity of the callbacks themselves.

<table>
<tr>
<td>Unsubscribe</td>
<td>Breaks the relationship between the observed object and the observer. There are several combinations of both arguments:

The `ObservedObject` argument is NULL . This means that the `callback` object wants to break its relationship to all objects it is observing. The observer interface will remove all relationships that contain this callback from its tables.
The `callback` argument is NULL . This means that the given `ObservedObject` is going out of scope and wants to break all relationships to all its observers. The interface removes from its tables all relationships that have this object as the observed object. This happens normally immediately after the notification `FINALIZE` is sent.
If both `callback` and `ObservedObject` are non NULL , only the matching relationship will be removed from the tables.

</td>
</tr>
<tr>
<td>ObserverFunction</td>
<td>This function will be called by the interface when a notification is received for an observed object. The call happens after all arguments have been processed, the actual work of the function is finished (when adding an object) or not yet done (when destroying an object). The container is in a consistent state. For the callbacks that are called when an object is deleted from a container the call happens before any call to `free()` and before any call to a destructor (if any) is done. For the calls that add an object the callback is called after the container has been modified.
Arguments:

`ObservedObject`: Specifies the object that sends the notification, i.e. the container that has the subscription. It is assumed that this container conforms to the `iGeneric` interface.
`Operation`: The operation that provoked the notification. Since it is possible to subscribe to several operations with only one callback function, this argument allows the callback to discriminate between the operation notifications.
`ExtraInfo`: This argument is specific to each operation and conveys further information for each operation.

None of the arguments will be ever NULL or zero.</td>
</tr>
</table>

Notification messages

Operation	Argument 1	Argument 2
Add	Pointer to the new object	NULL or slice specs if any
AddRange	A `size_t` with the number of objects added	Pointer to a table of n elements that were added
Append	A pointer to the object being appended. It is of the same type as the object emitting the notification	NULL
Clear	Pointer to the container being cleared	NULL
Copy	Pointer to the copy of the container	NULL
Erase	Pointer to the object being deleted. The object is still valid	NULL
EraseAt	Pointer to object being deleted	Position (as size_t)
Finalize	NULL	NULL
Insert	Pointer to the new object being inserted	A `size_t` with the position of the object being inserted if applicable
InsertIn	Pointer to the object being inserted, that has the same type as the object sending the notification	NULL
Pop	Pointer to the object being popped	NULL
Push	Pointer to the object being pushed	NULL
ReplaceAt	Pointer to the old value	Pointer to the new value

Chapter 2

The containers

2.1 The List interfaces: iList, iDlist

The list container appears in two flavors:

- single linked lists: the iList type
- double linked lists the iDlist type

The space overhead of single linked lists is smaller at the expense of more difficult access to the elements. It is up to the application programmer to decide which container fits best in his/her application [1].

It is often more efficient to get the next element from a list starting with the previous element instead of searching the whole list starting from the beginning. For this, the list and the Dlist containers provide:

- `FirstElement` Start of the list
- `LastElement` End of the list
- `NextElement` Returns a pointer to the next element
- `PreviousElement` Only in double linked lists. Returns a pointer to the previous element.
- `ElementData` Extracts a pointer to the element data
- `SetElementData` Modifies one element of the list.
- `Advance` Returns the data of an element and advances the given pointer in one operation.
- `MoveBack` Returns the data of an element and moves back the pointer one element. This operation is available only in double linked lists.

These operations can't be done in a read-only list.

The exact layout of the `ListElement` structure is undefined and private to each implementation. This is the reason for providing the `ElementData` function: it hides the

[1] The single linked list container corresponds to the C++ STL `forward_list`.

exact position and layout of the data from user code, that remains independent from implementation details.

The interfaces of both containers are very similar. Double linked lists support all functions in single linked ones, and add a few more. To avoid unnecessary repetition we document here all the single linked list interface, then only the functions that the Dlist interface adds to it.

```
typedef struct tagListInterface {
   int (*Add)(List *L,const void *newval);
   int (*AddRange)(List *L, size_t n,const void *data);
   void *(*Advance)(ListElement **pListElement);
   int (*Append)(List *l1,List *l2);
   int (*Apply)(List *L,int(Applyfn)(void *,void *),void *arg);
   void *(*Back)(const List *l);
   int (*Clear)(List *L);
   int (*Contains)(const List *L,const void *element);
   List *(*Copy)(const List *L);
   int (*CopyElement)(const List *list,size_t idx,void *OutBuffer);
   List *(*Create)(size_t element_size);
   List *(*CreateWithAllocator)(size_t elementsize,
         const ContainerAllocator *mm);
   void *(*ElementData)(ListElement *le);
   int (*Equal)(const List *l1,const List *l2);
   int (*Erase)(List *L,const void *);
   int (*EraseAll)(List *l,const void *);
   int (*EraseAt)(List *L,size_t idx);
   int (*EraseRange)(List *L,size_t start,size_t end);
   int (*Finalize)(List *L);
   ListElement *(*FirstElement)(List *l);
   void *(*Front)(const List *l);
   const ContainerAllocator *(*GetAllocator)(const List *list);
   void *(*GetElement)(const List *L,size_t idx);
   size_t (*GetElementSize)(const List *l);
   unsigned (*GetFlags)(const List *L);
   ContainerHeap *(*GetHeap)(const List *l);
   List *(*GetRange)(const List *l,size_t start,size_t end);
   int (*IndexOf)(const List *L,const void *SearchedElement,
        void *ExtraArgs,size_t *result);
   List *(*Init)(List *aList,size_t element_size);
   int (*InitIterator)(List *L,void *buf);
   List *(*InitWithAllocator)(List *aList,size_t element_size,
         const ContainerAllocator *mm);
   List *(*InitializeWith)(size_t elementSize,size_t n,
         const void *data);
```

```
   int (*InsertAt)(List *L,size_t idx,const void *newval);
   int (*InsertIn)(List *l, size_t idx,List *newData);
   ListElement *(*LastElement)(List *l);
   List *(*Load)(FILE *stream, ReadFunction loadFn,void *arg);
   Iterator *(*NewIterator)(List *L);
   ListElement *(*NextElement)(ListElement *le);
   int (*PopFront)(List *L,void *result);
   int (*PushFront)(List *L,const void *str);
   int (*RemoveRange)(List *l,size_t start, size_t end);
   int (*ReplaceAt)(List *L,size_t idx,const void *newval);
   int (*Reverse)(List *l);
   int (*RotateLeft)(List *l, size_t n);
   int (*RotateRight)(List *l,size_t n);
   int (*Save)(const List *L,FILE *stream, SaveFunction saveFn,
        void *arg);
   int (*Select)(List *src,const Mask *m);
   List *(*SelectCopy)(const List *src,const Mask *m);
   CompareFunction (*SetCompareFunction)(List *l,CompareFunction fn);
   DestructorFunction (*SetDestructor)(List *v,DestructorFunction fn);
   int (*SetElementData)(List *l, ListElement *le,void *data);
   ErrorFunction (*SetErrorFunction)(List *L,ErrorFunction);
   unsigned (*SetFlags)(List *L,unsigned flags);
   size_t (*Size)(const List *L);
   size_t (*Sizeof)(const List *l);
   size_t (*SizeofIterator)(const List *);
   ListElement *(*Skip)(ListElement *l,size_t n);
   int (*Sort)(List *l);
   List *(*SplitAfter)(List *l, ListElement *pt);
   int (*UseHeap)(List *L, const ContainerAllocator *m);
   int (*deleteIterator)(Iterator *);
} ListInterface;
```

General remarks

Lists are containers that store each element in a sequence, unidirectionally (single linked lists) or bidirectionally (double linked lists). The advantage of linked lists is their flexibility. You can easily and with a very low cost remove or add elements by manipulating the links between the elements. Single linked lists have less overhead than their double linked counterparts (one pointer less in each node), but they tend to use a lot of computer power when inserting elements near the end of the list: you have to follow all links from the beginning until you find the right one.

The list nodes themselves do not move around, only their links are changed. This can be important if you maintain pointers to those elements. Obviously, if you delete a node, its contents (that do not move) could be recycled to contain something else than what you expect.

The iList interface consists (as all other interfaces) of a table of function pointers. The interface describes the behavior of the List container.

The stack operations push and pop are provided with PushFront and PopFront because they have a very low cost, insertion at the start of a single linked list is very fast. PushBack is the equivalent of the `Add` operation, but PopBack would have a very high cost since it would need going through all the list.

The list container features in some implementations a per list error function. This is the function that will be called for any errors, except in cases where no list object exists: the creation function, or the error of getting a NULL pointer instead of a list pointer. In those cases the general iError interface is used, and iError.RaiseError is called. The default value of the list error function is the function iError.RaiseError at the moment the list is created.

Other implementations of this interface may specialize list for a certain category of uses: lists of a few elements would try to reduce overhead by eliminating a per list error function and replace it with the standard error function in iError, for instance, eliminating their fields in the header. If the read-only flag support is dropped, the whole "Flags" field can be eliminated. In such an implementation, the SetFlags primitive would always return an error code.

The sample implementation of the list container supports the following state flags:

```
#define CONTAINER_READONLY          1
```

If this flag is set, no modifications to the container are allowed, and the Clear and Finalize functions will not work. Only copies of the data are handed out, no direct pointers to the data are available.

```
#define CONTAINER_SORTED_FRONT      2
#define CONTAINER_SORTED_BACK       4
```

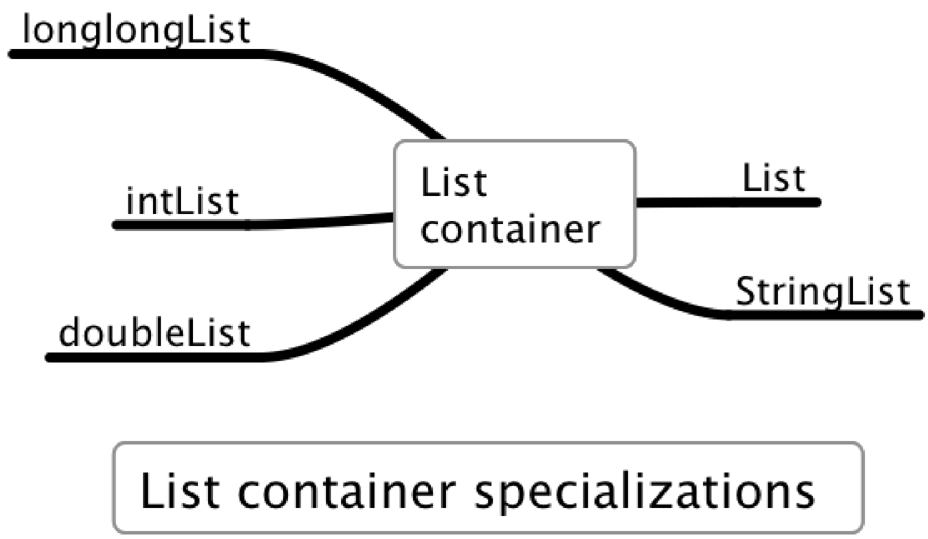
If this flag is set, the container is maintained always in sorted order, with the biggest element at the index zero for `CONTAINER_SORTED_FRONT` or with the biggest element at the end if `CONTAINER_SORTED_BACK` is set. It is an error if both flags are set, and the results in that case are implementation defined.

Specializations

All "specialized" containers share the same interface with the following exceptions:

- The functions where a `void *` to the element data is passed or where a `void *` is the result of the operation are replaced with the actual data type of the specialization. For instance the `GetElement` API instead of returning a void pointer returns a pointer to the specific data type: an integer for `intList`, a double for `doubleList` etc.

- The creation and initialization functions that construct a new container receive one argument less than its generic counterparts since the size of each element is fixed.
  To make things clear and to save work from the library user some specializations are delivered with the sample implementation to show how a *file templated* container looks like.



In the right side of the drawing we see the generic list container using generic pointers (`void *`) and the stringlist container. Strings are special because in C their length is the result of a function call instead of being fixed like other data types.

In the left side, we see three specialized containers for some numeric data types. Those containers are generated using two types of source files:

- Parameter files: They define the data type and some other parameters like the comparison expression.
- Templated files: They implement the specialized container. The pre-processor does the editing work on the templated file to yield several different type definitions. Using this interface has the advantage of ensuring compile time checking of the arguments to the API, what is not possible using generic pointers.

Operation	Description

Add	Adds the given element to the container. In its generic form it is assumed that "data" points to a contiguous memory area of at least ElementSize bytes. In its specialized form the data is passed by value. Returns a value greater than zero if the addition of the element to the list completed successfully, a negative error code otherwise.
Advance	Given the address of a pointer to an element, it returns a pointer to the data stored into that element and writes the address of the next element into its argument `ppElement`. If ppElement isNULL it returnsNULL . If `*ppElement` isNULL it also returnsNULL , and obviously there is no advancing done.
AddRange	Adds the n given elements to the end of the container. It is the same operations as the PushBack operation. It is assumed that "data" points to a contiguous memory area of at least n*ElementSize bytes. If n is zero no error is issued even if the array pointer or the data pointer are NULL .
Append	Appends the contents of list2 to list1 and destroys list2.
Apply	Will call the given function for each element of the list. The first argument of the callback function receives an element of the list. The second argument of the callback is the arg argument that the Apply function receives and passes to the callback. This way some context can be passed to the callback, and from one element to the next. Note that the result of the callback is not used. This allows all kinds of result types to be accepted after a suitable cast. If the list is read-only, a copy of the element will be passed to the callback function.
Back	Returns the last element of the given list or NULL if the list is empty.
Clear	Erases all stored data and releases the memory associated with it. The list header will not be destroyed, and its contents will be the same as when the list was initially created. It is an error to use this function when there are still active iterators for the container.
Contains	Returns one if the given data is stored in the list, zero otherwise. The "data" argument is supposed to point to an element at least ElementSize bytes. The list's comparison function is used for determining if two elements are equal. This comparison function defaults to memcmp.

Copy	A shallow copy of the given list is performed. Only ElementSize bytes will be copied for each element. If the element contains pointers, only the pointers are copied, not the objects they point to. The new memory will be allocated using the given list's allocator.
CopyElement	Copies the element data at the given position into the given buffer, assuming that at least ElementSize bytes of storage are available at the position pointed by the output buffer. The main usage of this function is to access data in a read only container for later modification.
Create	The creation function returns an empty List container, initialized with all the default values. The current memory manager is used to allocate the space needed for the List header. The list is supposed to contain elements of the same size. If the elements you want to store are of different size, use a pointer to them, and create the list with sizeof(void *) as the size parameter.
deleteIterator	Reclaims the memory used by the given iterator object
Equal	Compares the given lists using the list comparison function of either list1 or list2 that must compare equal. If the list differ in their length, flags, or any other characteristic they compare unequal. If any of their elements differ, they compare unequal. If both list1 and list2 are NULL they compare equal. If both list1 and list2 are empty they compare equal.
EraseRange	Removes from the list the given range, starting with the `start` index, until the element before the `end` index. If `end` is greater than the length of the list, it will be 'rounded' to the length of the list.
EraseAt	Removes from the list the element at the given position.
EraseRage	Removes from the list the given range, starting with the `start` index, until the element before the `end` index. If `end` is greater than the length of the list, it will be 'rounded' to the length of the list.
Finalize	Reclaims all memory used by the list, including the list header object itself.
FirstElement	Finds the first element of the list and returns a pointer to it. This is a pointer to the element, **not** to the data stored at that element. It is an error to attempt to use this function with a read-only list.
Front	Returns a pointer to the first element of the given list or NULL if the list is empty.
GetAllocator	Returns the list's allocator object. If the list pointer is NULL it returns NULL .

GetElementSize	Retrieves the size of the elements stored in the given list. Note that this value can be different than the value given to the creation function because of alignment requirements.
GetElement	Returns a read only pointer to the element at the given index, or NULL if the operation failed. This function will return NULL if the list is read only. Use the CopyElement function to get a read/write copy of an element of the list.