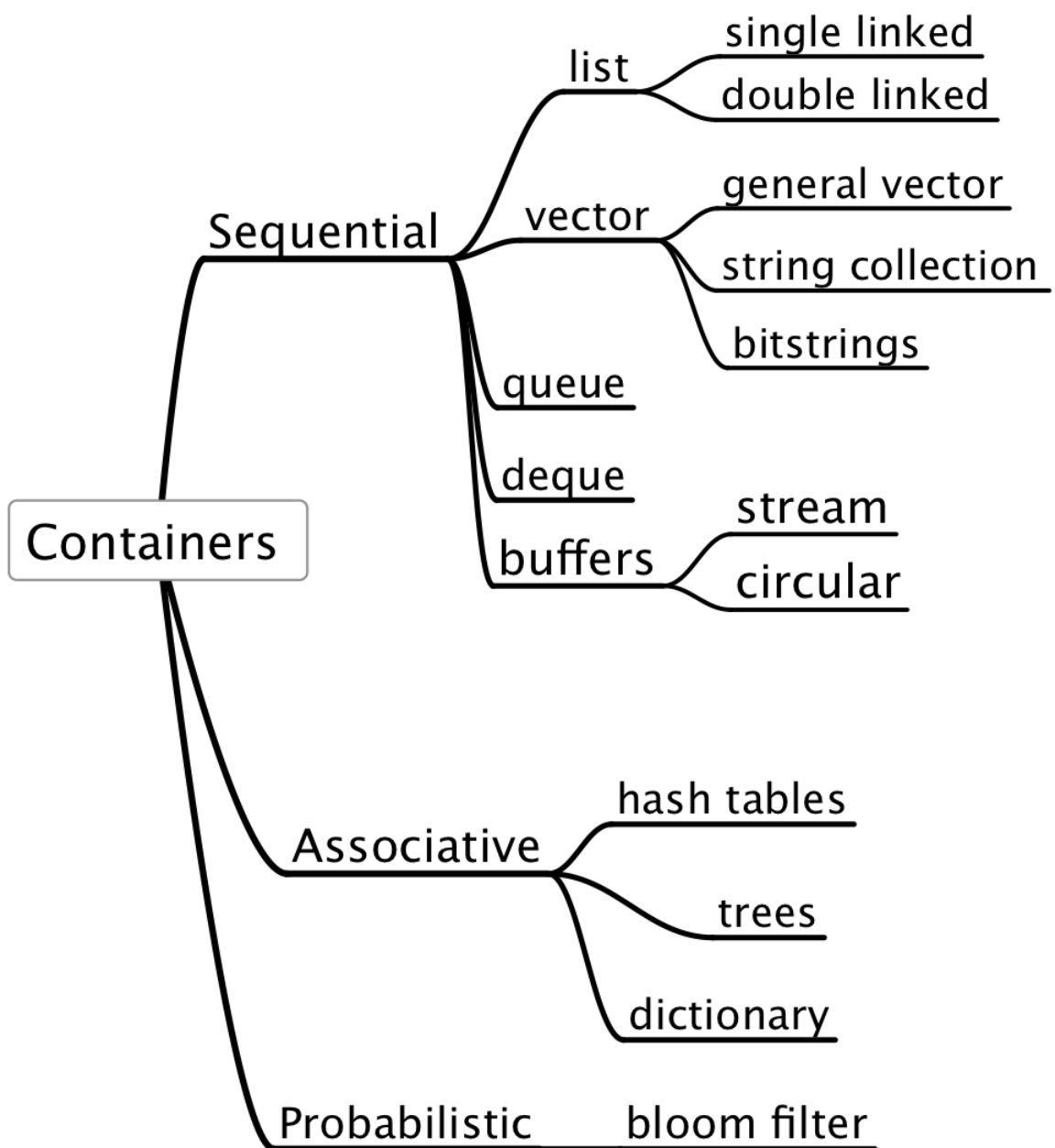


A CONTAINER LIBRARY FOR C

Jacob Navia



Contents

1	Introduction	13
1.1	Containers	14
1.2	The interface concept	15
1.3	Error handling	15
1.4	The different containers	16
1.4.1	Single and double linked lists	17
1.4.2	Flexible arrays (vector)	17
1.4.3	String collection	18
1.4.4	Bit-string	18
1.4.5	Dictionary	18
1.4.6	Hash Table	18
1.4.7	AVL trees	18
1.4.8	“Scapegoat” trees	18
1.4.9	Bloom Filter	19
1.4.10	Queue	19
1.4.11	Deque	19
1.4.12	Buffers	19
1.5	Types used by the library	19
1.5.1	CompareInfo	19
1.5.2	CompareFunction	20
1.5.3	SaveFunction	20
1.5.4	ReadFunction	20
1.5.5	ErrorFunction	20
1.5.6	DestructorFunction	20
1.6	Design goals	21
1.6.1	Error analysis	21
1.6.2	Full feature set	22
1.6.3	Abstraction	22
1.6.4	Performance	23
1.7	How the functions are specified in this document.	23
2	The common vocabulary: iGenericContainer	25
2.1	Creation of a container: Create	26
2.2	Destruction of a container: Clear and Finalize	26

2.2.1	Other creation functions	26
2.3	Adding an element to a container: Add and AddRange	27
2.4	Removing an element from a container: Erase	28
2.5	Retrieving an element from a container: GetElement	28
2.6	Sorting a sequential container: Sort	28
2.7	Copying a container: Copy	28
2.8	Saving and loading a container to or from disk: Save and Load	29
2.9	Inserting a container into another: InsertIn	29
2.9.1	Sequential containers	29
2.9.2	Associative containers	29
2.10	Replace an element with another	29
2.10.1	Sequential containers: ReplaceAt	29
2.10.2	Associative containers: Replace	29
2.11	Looping through all elements of a container	30
2.11.1	Using a simple loop to iterate a container	30
2.11.2	Using the “Apply” function.	31
2.11.3	Using iterators	31
2.12	Setting and retrieving the state: GetFlags and SetFlags	32
2.13	Retrieving the number of elements stored: Size	32
2.14	Space used: Sizeof	32
2.15	Memory management	33
2.15.1	Memory manager objects	33
2.15.2	Pooled memory management	34
2.15.3	Heap of same size objects	34
3	The auxiliary interfaces	35
3.1	Memory management	35
3.1.1	The traditional memory manager	35
3.1.2	The Heap interface: iHeap	36
	Create	37
	InitHeap	37
	newObject	37
	AddToFreeList	37
	DestroyFreeList	38
	Finalize	38
	Sizeof	38
3.2	Pooled memory interface: iPool	39
	Create	39
	Alloc	39
	Calloc	40
	Clear	40
	Finalize	40
3.3	Error handling Interface: iError	40
	RaiseError	41

	EmptyErrorFunction	41
	StrError	41
	SetErrorFunction	41
3.4	The iterator interface	41
3.4.1	The interface	42
	GetCurrent	42
	GetFirst	42
	GetNext	43
	GetPrevious	43
	GetCurrent	44
	GetLast	44
4	The containers	45
4.1	The List interfaces: iList, iDlist	45
4.1.1	General remarks	47
	Add	48
	AddRange	49
	Append	50
	Apply	51
	Clear	52
	Contains	52
	Copy	52
	CopyElement	53
	Create	53
	CreateWithAllocator	54
	deleteIterator	54
	Equal	55
	Erase	55
	EraseAt	55
	EraseRange	56
	Finalize	57
	GetAllocator	57
	GetElementSize	57
	GetElement	57
	GetFlags / SetFlags	58
	GetRange	58
	IndexOf	59
	Init	59
	InitWithAllocator	59
	InsertAt	60
	InsertIn	60
	Load	62
	newIterator	62
	PopFront	62

	PushFront	63
	ReplaceAt	63
	Reverse	64
	Seek	64
	Save	65
	SetCompareFunction	65
	SetAllocator	65
	SetDestructor	66
	SetErrorFunction	66
	Size	66
	Sizeof	66
	Sort	67
	UseHeap	67
4.2	Double linked lists: iDlist	68
	PopBack	70
	Splice	71
4.3	The Vector interface: iVector	72
4.3.1	The interface	73
4.3.2	The API	74
	Add	74
	AddRange	74
	Append	75
	Apply	75
	Clear	76
	Contains	76
	Copy	77
	Create	77
	CreateWithAllocator	77
	Contains	78
	CopyTo	78
	deleteIterator	78
	Equal	78
	Erase	79
	EraseAt	79
	Finalize	80
	GetCapacity	80
	GetElementSize	80
	GetElement	80
	GetFlags / SetFlags	81
	GetRange	81
	IndexIn	82
	IndexOf	82
	InsertAt	82
	InsertIn	83

	Load	84
	newIterator	84
	Mismatch	85
	PopBack	85
	ReplaceAt	86
	Reverse	86
	Save	86
	SetCapacity	87
	SetCompareFunction	87
	SetDestructor	87
	SetErrorFunction	88
	Size	88
	Sizeof	88
	Sort	88
4.4	The bit-string container: iBitString	90
4.4.1	The interface	91
4.4.2	API	92
	Add	92
	And	93
	AndAssign	94
	BitBlockCount	95
	CopyBits	96
	GetBits	96
	GetRange	96
	LeftShift	96
	Not	97
	NotAssign	97
	ObjectToBitString	97
	Or	97
	OrAssign	98
	PopulationCount	98
	Print	98
	Reverse	98
	RemoveAt	99
	Set	100
	StringToBitString	100
	Xor	100
	XorAssign	101
4.5	The string collection container: iStringCollection	102
4.5.1	The interface	102
4.5.2	API	103
	AddRange	103
	CastToArray	104
	CreateFromFile	104

	FindFirstText	104
	FindNextText	104
	FindTextPositions	105
	Init	105
	InitWithAllocator	105
	InsertIn	105
	Mismatch	107
	PopBack	108
	WriteToFile	108
4.6	The dictionary container: iDictionary	109
4.6.1	The dictionary interface	110
4.6.2	The API	111
	Add	111
	Apply	111
	Clear	112
	Contains	113
	Copy	113
	Create	113
	deleteIterator	113
	Equal	114
	Erase	114
	Finalize	114
	GetElementSize	115
	GetElement	115
	Init	116
	InitWithAllocator	116
	Insert	116
	Load	116
	newIterator	117
	SetDestructor	117
	Size	117
	Save	118
	Sizeof	118
	SetErrorFunction	118
	Size	119
4.7	The TreeMap interface: iTreeMap	120
	The comparison function must be consistent	120
4.7.1	The interface	120
4.8	Hash Table: iHashTable	122
4.8.1	The interface	122
4.8.2	The API	123
	Add	123
	Apply	124
	Clear	125

	Copy	125
	Create	125
	deleteIterator	125
	Erase	125
	GetElement	126
	GetFlags	126
	Load	126
	Merge	126
	newIterator	127
	Overlay	128
	Resize	128
	Replace	128
	Save	128
	SetErrorFunction	129
	Size	129
	Sizeof	129
4.9	Queues: iQueue	130
4.9.1	Interface	130
4.9.2	The API	130
	Front	131
	Back	131
	GetList	131
4.10	Deque: iDeque	131
4.10.1	Interface	132
	Apply	133
	Back	133
	Clear	133
	Contains	133
	Copy	133
	Create	134
	Equal	134
	Front	134
	Erase	134
	Finalize	135
	GetFlags	135
	Load	135
	PopBack	136
	PopFront	136
	PushBack	136
	PushFront	136
	Save	137
4.11	Bloom filters	138
4.11.1	The interface: iBloomFilter	138
4.11.2	The API	139

	CalculateSpace	139
	Create	139
	Add	139
	Find	140
	Clear	140
	Finalize	140
4.12	Buffers	141
4.12.1	Stream buffers	141
	The interface	142
	The API	142
	Clear	142
	Create	142
	CreateWithAllocator	142
	Finalize	143
	GetData	143
	GetPosition	143
	Read	143
	SetPosition	144
	Size	144
	Write	144
4.12.2	Circular buffers	146
	The interface: iCircularBuffer	146
	The API	146
	Add	146
	Clear	147
	CreateWithAllocator	147
	Create	147
	Finalize	147
	PeekFront	148
	PopFront	148
	Size	149
	Sizeof	149
4.13	The generic interfaces	150
4.13.1	Generic containers	150
4.13.2	Sequential containers	151
4.13.3	Associative containers	152
5	Enhancing the library	153
6	Applications	155
6.1	Mapcar	155
7	The sample implementation	159
7.1	Data structures	159

7.1.1	The generic part	159
7.1.2	Lists	160
7.1.3	Double linked lists	161
7.1.4	Vector	162
7.1.5	Dictionary	163
7.1.6	String collection	163
7.1.7	The iterator data structure	164
7.2	The code	165
7.2.1	List	165
7.2.2	Queues	195
7.2.3	The dictionary	197
	Hashing	198
	Creation	199
	Adding elements	200
	Implementing iterators	202
7.2.4	The bloom filter	203
	Debugging malloc	204
8	Building generic components	207
	Index	213

1 Introduction

The objective of this proposal is to standardize the usage of common data structures within the context of the C language. The existence of a common standard interface for lists, hash tables, flexible arrays, and other containers has several advantages:

- User code remains portable across machines and operating systems
- The portable specifications provide a common framework for library writers and compiler/system designers to build compatible yet strongly specialized implementations.
- The language becomes more expressive: it is not necessary to build the *nth* hash table function from scratch. You can use a standard one.

The big innovation of C in the eighties was its standard library, that made input/output portable across machines and implementations. The container library would replicate again that idea, at a higher level.

The specifications presented here are completely scoped by the C99 specifications, and can be implemented even in compilers that do not implement C99 and stayed within the C94 context. No language extensions are needed nor any are proposed.

The interfaces proposed try to present complete packages, i.e. interfaces with all the necessary functions to allow the widest usage: Serialization, searching, and many other functionalities are included in the proposed standard to allow for maximum code portability. It can be argued that this makes for "fat" containers, but if you read carefully you will notice that many things can be left out in systems that run in low memory or with feeble computing power.

This documentation is composed of several parts:

1. An introductory part where the general lines of the library are explained.
2. A specifications part where each function of the library is fully specified. This is the proposal for the next C standard.
3. An "examples" part that shows the uses of the library and allows you to have a better idea of how the usage of the library looks like.
4. An implementation part where the code of the sample implementation is discussed. This is designed as a guide for implementors to give them a basis to start with.

1.1 Containers

In the context of this library, a container is a data structure used to organize data within a single logical object that allows for adding, searching and removing data. The data is not further specified. It can be anything, images, numbers, text, whatever. The only thing that the container knows is the size of the data, if we store a series of objects of the same size, or its address, if we store objects of different sizes. In the later case we store just a pointer in the container. Each container has a way of iterating through all its elements by using an “iterator” auxiliary object, that returns each stored object in sequence.

All objects stored by the library are copied into the library, and the library is responsible for the management of the associated storage. If you do not want this, just store a pointer to the data and manage the data yourself.

We have basically two different kinds of containers

- 1. Sequential containers
- 2. Associative containers

A sequential container is organized in a linear order. We have a sequence starting at index zero up to the number of elements stored. Data items can be retrieved by index, and it makes sense to speak of a “next” and a “previous” element.

Sequential containers can be contiguous (arrays) or disjoint (lists). In the first case access is very fast since it implies multiplying the index by the size of each element to get to any position in the data. In the second case access the *n*th element can be a lengthy operation since the chain of “next” or “previous” pointers must be followed for each access to a given position.

An associative container stores an object divided in two parts: a key, that is used as a token for the data, and the data itself. It associates key/value pairs. Speed of access is fast, but not linear, and can degrade as new items are stored in it.

In all cases, we have some basic properties of an abstract container that are common to all of them.

- A function to report errors. This function (like all other function pointers) can be changed, and defaults to a simple error function that prints the error in the standard error stream.
- Each change in a container is recorded. This permits to validate pointers to a container: if the container has changed after the creation of the pointer, the pointer could be invalid.
- All containers use a standard object to allocate and manage memory. The library provides a default allocator that contains the standard C functions `malloc`, `free`, `realloc` and `calloc`. Each container class can contain an allocator pointer, or each container can contain an allocator. The provided sample implementation has a per container allocator, but in many applications a per class allocator could be enough, or even a single global allocator that would be used by the whole library.

Managing a sequence involves trade offs what performance is concerned. If the usage will involve frequent insertion and deletion of objects you will prefer a container that handles those operations in constant time: the time to add or delete an object doesn't increase with the number of elements in the container. Such a container will be unlikely to provide also access to a given element in constant time. Access is likely to be much slower, and what you gain in flexibility you loose in another dimension. It is the user of the library, the programmer, that decides what container fits best the intended usage.

Since usage patterns change, however, the library tries to ensure that you can change the container you are using with minimal effort. If at the beginning of an application a list looked like a good solution but later an array, that provides constant time access is better suited, you can change the type of container without changing every line that uses it. The common vocabulary of the library makes this possible.

1.2 The interface concept

Each container is defined by its interface, i.e. the table of functions it supports. For each interface, its name is composed of a lower case "i" followed by the container name: `iList`, `iVector`, `iStringCollection`, etc.

Each function of the interface receives always the container as its first argument. Obviously, the big exception is the creation function, that receives various arguments depending of which container or from what input, the container is to be created.

For each container interface a global object exists that allows direct access to the function table without the need of creating a container to access it.

This interface allows for simple access to each container using a very similar vocabulary:

```
iList.Add(list,object);  
iStringCollection.Add(strcol,object);
```

The objects stored in a container have always the same size. When storing objects of different sizes just store a pointer to the objects, since pointers have always the same size.

1.3 Error handling

This specification describes the basic error handling that each function of the library must do. Other errors can appear in different implementations. At each error, the library should call the container instance specific error handling when there is one, or call the general error handling function in the `iError` interface. When it is not possible to call the instance specific error function, for instance when the instance parameter is `NULL`, the library calls the general error handling function in the `iError` interface¹.

¹There is no automatic cleanup of objects left by active functions in the stack. This can be a problem or not, depending if your use a garbage collection or not. If you use a garbage collector, this

The user of the library can either replace the default `iError` interface with a function that handles the error with a jump to a previously set recovery point, or treat the error locally using the return code. All errors are negative constants, it suffices to test if the result is less than zero.

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.
- `CONTAINER_ERROR_READONLY` The object is readonly and the operation is not allowed.
- `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_NOT_EMPTY` Operation can be performed in an object with no elements only.
- `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.

Other errors can be defined by each implementation.

The treatment of each error is done in the object defined by the `iError` interface.

1.4 The different containers

All data structures in this section are known and used for several decades. Lists are a common feature of any data processing task since the sixties for instance. The library provides for abstract containers, and some examples of concrete ones for the elementary types. We have:

problem doesn't even appear: the unused objects will be automatically collected. If you don't, you should test for the return code of each function.

- **Vectors.** The general abstract vector container is implemented in the “Vector” container. This is a flexible array that allows for insertion/deletions, with no cost for insertion at the end in most cases. Concrete implementations for the elementary types are provided for bits (bit-strings), strings (null terminated), int/double/long double numeric data in the form of templates.
- **Lists.** Single linked lists (List) and double linked lists (Dlist) are provided.
- **Queue, Deque**
- **Trees** (red/black trees, AVL trees)
- **Dictionary.** This is a simple implementation of a hash table with character keys.
- **Hash Table.** More complex implementation of a hash table with arbitrary (binary) keys, and automatic hash table resizing.
- **Buffers.** Stream buffers (linear buffers that resize to accomodate more data) and circular buffers are provided.

1.4.1 Single and double linked lists

This containers consist of a header and a list of elements containing each a pointer to the next element in the chain, and a pointer to the data item stored. The end of the list is marked by a node that contains a NULL “next” pointer. Double linked lists contain an additional pointer to the previous element.

This is a very flexible container, allowing you to add and delete elements easily just by rewriting some pointers. You can even split them in two sublists just by zeroing somewhere the “next” pointer.

The price you pay for this flexibility is that sequential access is expensive, the cost of accessing the n th element increases linearly with n .

Storage overhead is one or two pointers per element stored in the list for single/double linked lists..

The data is stored directly after the pointer, there is no pointer to the data. This is a variable length structure with a fixed and a variable part. To avoid using a standard C99 feature that could be absent in older compilers, we use a semi-generic pointer indexed either by one (for older compilers) or by nothing (standard C) .

1.4.2 Flexible arrays (vector)

This container is an array with added operations that allow the user to insert and delete elements easily. It will resize itself if needed.

The access time is essentially the same as with a normal array. Insertion and deletion are possible but they are in general more expensive than with lists since the container must copy the elements to make place for a new element or to delete an element. An

exception to this rule is the deletion of the last element that will be done in constant time since it implies only decrementing the number of elements in the container.

The storage overhead for each element is zero since this container doesn't require any pointers per object stored.

This container uses a reserve storage to avoid allocating new memory for each addition operation. This allows the "Add" operation to be done in constant time in most occasions.

1.4.3 String collection

This container is designed to handle a collection of C strings. It is essentially an application of the flexible array container with some extra functionality to handle strings.

1.4.4 Bit-string

This container is designed to handle arbitrary sequences of bits. Some algorithms that are easy to program with strings are much more complicated for bit-strings, like to one that mimics "strstr" ("bit-strstr").

The bits are packed with 8 bits per character unit. The overhead per bit is the size of the bit-string header only. No pointers are associated with each bit.

1.4.5 Dictionary

This is an associative container based on a hash table. It associates a text key with some arbitrary data. This container is not ordered. Access time to each element depends on how much elements are stored in it and on the efficacy of the hash function to maintain elements in different slots. Storage overhead per element is one pointer each, plus the size of the slot table. This is for a hash table with linked lists in each slot for managing collisions. Other implementations exist of course.

1.4.6 Hash Table

This is a more sophisticated version of the dictionary hash table. It allows for keys of binary data and it has automatic resizing in case the table gets too crowded.

1.4.7 AVL trees

This data structure allows for fast searching for data. You can store millions of records and find a given record with a few comparisons.

1.4.8 "Scapegoat" trees

This is another form of trees. They can be more efficient than AVL trees, but from a container perspective they share the same characteristics.

1.4.9 Bloom Filter

This is a probabilistic data structure used to quickly check if an element is not in a larger set of elements. It returns false positives with a given probability set when the container is built. Elements can be added to it but they can't be removed from the container. It stores no data, just a key.

1.4.10 Queue

Queues are designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other. This container can be implemented as an adaptor using a single linked list as its base container. The sample implementation uses this strategy to show how adapters can look like. Other implementations can implement this container directly presenting the same interface.

1.4.11 Deque

This is a linear container that allows for cheap insertions/deletions at both ends.

1.4.12 Buffers

Buffers are containers used to hold data temporarily, either to be transmitted or stored into some medium, or to be filtered and used later by other parts of the application. The library provides two types of buffers:

- Stream buffers. They are a linear sequence of bytes, like a file. They resize automatically if they need to, and they have a *cursor* that points to the position where the next item will be stored.
- Circular buffers. They store the last *n* items of a stream. They can contain any item as in the vector container, or they can contain character strings, as in the string collection.

1.5 Types used by the library

1.5.1 CompareInfo

```
typedef struct tagCompareInfo {  
    void *ExtraArgs;  
    void *Container;  
} CompareInfo;
```

This structure will be passed to the comparison functions. The “ExtraArgs” pointer will receive the pointer that was passed to the calling function. The “Object” pointer will receive the address of the container where the elements are stored. If the two elements being compared are in different containers, this pointer will be NULL .

1.5.2 CompareFunction

```
typedef int (*CompareFunction)(const void *elem1,  
                               const void *elem2,  
                               CompareInfo *ExtraArgs);
```

This type defines the function used to compare two elements. The result should be less than zero if elem1 is less than elem2, zero if they are equal, and bigger than zero if elem1 is bigger than element 2.

1.5.3 SaveFunction

```
typedef int (*SaveFunction)(const void *element,  
                            void *ExtraArg,  
                            FILE *OutputStream);
```

This function should save the given element into the given stream. The “ExtraArg” argument receives the address of the container and any argument passed to the Save function. The result should be bigger than zero if the operation completed successfully, zero or less than zero otherwise.

1.5.4 ReadFunction

```
typedef int (*ReadFunction)(void *element,  
                            void *arg,  
                            FILE *InputStream);
```

This function should read into the given element from the given stream. The “ExtraArg” argument is passed to the container Save function and allows to pass an argument to the user defined save function. The result should be bigger than zero if the operation completed successfully, zero or less than zero otherwise.

1.5.5 ErrorFunction

```
typedef void (*ErrorFunction)(const char *functionName,int code);
```

This function type is used to handle errors in each container. The first argument is the function name where the error occurred, the second is a negative error code.

1.5.6 DestructorFunction

```
typedef int (*DestructorFunction)(void *object);
```

This function type is called when an object is being destroyed from the container. An object is destroyed when:

- An `Erase` call is done.
- A `Replace` call is done.
- The `Clear` call is done.

This function should free any memory used by pointers within the object **without** freeing the object memory itself. In most cases the memory used by the library is **not** allocated with `malloc`. Its result type is less than zero when an error occurred or greater than zero when it finished successfully.

1.6 Design goals

1.6.1 Error analysis

It has been a tradition in C to place raw performance as the most important quality of specifications. To follow this sacred cow C specifications ignored any error analysis arguing that any specification of failure modes would damage "performance". No matter that raw machine performance increased by several orders of magnitude, the cost of a check for NULL was always "too expensive" to afford.

This kind of mental framework was described by one of the people in the discussion group "comp.lang.c++" as follows:²

In C++, the program is responsible for ensuring that **all** parameters to the standard library functions are valid, not only the third parameter of `std::mismatch()`. For example, also the first range for `std::mismatch()` must be valid, one may not pass a start iterator from one container and end iterator from another, for example. However, STL does not guarantee any protection against such errors, this is just UB.

This specifications try to break away from that frame of thought. Each function specifies a minimal subset of failure modes as a consequence of its error analysis. This allows user code to:

- Detect and handle errors better.
- Ensure that errors will always have the **same** consequences. One of the worst consequences of undefined behavior is that the same error can have completely different consequences depending on apparently random factors like previous contents of memory or previous allocation pattern.

²We were discussing the specifications of the `mismatch` function of the C++ STL and why any error analysis is absent. The C++ STL prescribes a bounded region for the first container, but just a starting point for the second one. If the second is shorter than the specified range of the first *undefined behavior* ensues and anything can happen. In many cases this "anything" is different each time the same error occurs. In our specific case `mismatch` would read from memory that doesn't belong to the container it started with. Depending on the contents of that memory a crash could happen, or worst, a wrong result returned to the calling software, etc.

At the same time, the mandatory error checking consists mainly of checks that can be implemented with a few integer comparisons. For instance a check for zero is a single instruction in most processors. If implemented correctly the conditional jump after the comparison with zero is not taken in the normal case and correctly predicted by the processor. This means that the pipeline is not disturbed and the cost for the whole operation is much less than a cycle.

Why is error analysis an essential part of any program specifications?

Because **mistakes are a fact of life**. Good programmers are good most of the time only. Even very good programmers *do make mistakes*³. Software must be prepared to cope with this fact in an orderly fashion because if failure modes are not specified they have catastrophic consequences and lead to brittle software that crashes randomly.

Note that error *analysis* is not error *handling*. Error handling is taking an action after an error, a task only the application can do. What the library can do is to establish a framework where a user defined procedure receives enough information about the specific problem at hand.

Error analysis means that for each function and each API:

- An analysis is performed of what are the consequences of any error in its inputs. Error codes are used to pass detailed error information to the error procedure.
- During its execution, an analysis is done of each step that can fail.
- The outputs of the function are left in a consistent state, errors provoking the undo of the previous steps in most cases, leaving the inputs as they were before the function was called. This feature allows library functions to be restartable after an error. For instance an out of memory condition can be corrected by freeing memory and retrying.

The library provides hooks for the users that can control each step and provide functions that can do the error handling, for instance logging the error and jumping to a pre-established recovery point.

1.6.2 Full feature set

Another design goal is to offer to the user a full feature set, complete with serializing, iterators, search, read-only containers and all the features needed in most situations. Other features are planned for later (observers, multi-threading support).

1.6.3 Abstraction

The library is designed with the possibility of implementing abstraction like serial and associative containers that allow software to treat several containers in a way that ab-

³Donald Knuth, the author of the TeX typesetting program can be without doubt be qualified as a good programmer (and an excellent computer scientist). But he, like anybody else, is not without flaws. See: www.tug.org/texmf-dist/doc/generic/knuth/errata/errorlog.pdf. There are hundreds of entries in that log.

stract most of their features, improving code reuse by allowing to implement algorithms for a class of objects. This is specially true in the iterators feature.

It can be argued that the C language lacks many of the abstractions constructs of other languages like templates, inheritance, and many others. All that is true, but the objective of this proposal is to show that those constructs are just an aid to developing abstractions, an aid that is paid in added complexity for the resulting language, and in a limitation of what is feasible within a given framework. Since C has no framework, no preferred inheritance model, it is possible to create abstractions that are quite unconstrained: there is no framework precisely.

1.6.4 Performance

Even with all the tests, the performance of the library has been maintained at a high level compared to similar libraries in other languages. The performance should improve if standardized because compiler writers could specialize their optimizations targeting this code.

1.7 How the functions are specified in this document.

The specifications part of the proposal uses the same building blocks for each of the functions proposed.

Name

The name of the function. Note that when using this name, the container interface should be always before: `iList.Add`, `iDictionary.Add`, etc.

The name is followed by the prototype defined as a function pointer. For the function "Add" of the container "List" we have

```
int (*Add)(List *list,void *data);
```

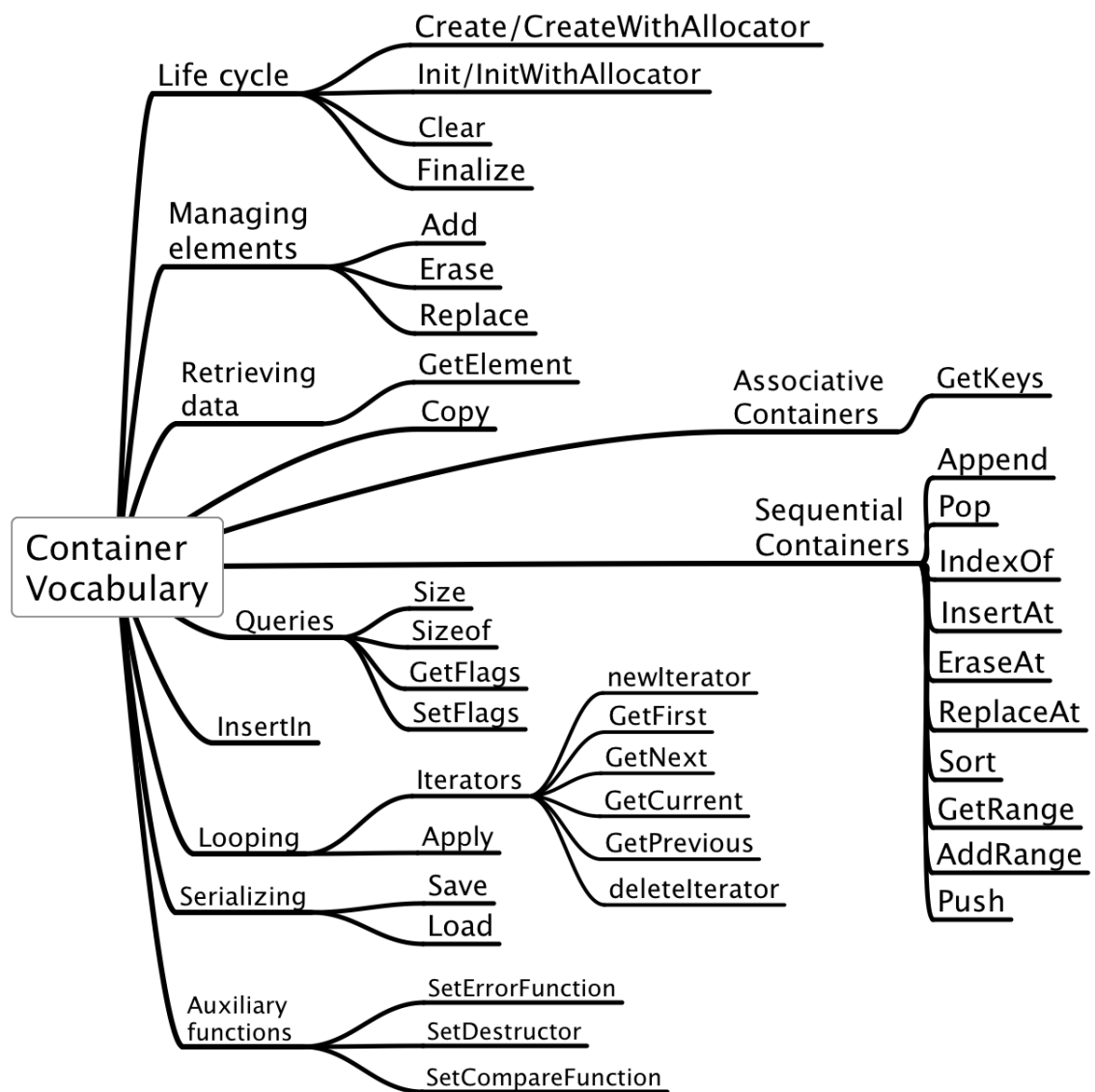
This means that "Add" is a function pointer in the interface `iList`. It would be used as: `iList.Add(list,data)`.

Errors:

The minimal set of errors that can appear during the execution of the function is listed. Each implementation is free to add implementation specific errors to this list. Note that how the library behaves after an error is defined by the current error function in the container (if any), then by the behavior of the error function in the `iError` interface. This can be changed by the user by using the `iError` interface.

Returns: The return value of the operation. Normally, negative values are error codes, positive values means success, and zero means non fatal errors, more in the sense of a warning.

2 The common vocabulary: iGenericContainer



The library uses always the same words to represent similar actions in all containers.

2.1 Creation of a container: Create

Containers are created with a call to their “Create” function. The first argument is the size of the objects that will be stored in the container. The second is optional and is a hint to the number of elements that will be stored in the container. Note that if you want to store objects of different sizes you just store a pointer to those objects instead of the objects themselves. The creation functions can have several arguments, the first being always the size of the elements that the container will hold. The prototype can be:

```
Container * iContainer.Create(size_t elementsize,...);
```

The creation function needs to allocate memory to hold the container. This memory will be allocated using the current memory manager that is always an implicit argument to all creation functions. The rationale behind this design decision is that you don’t change your memory allocation strategy at each call to a container creation function. This simplifies the interface at the expense of making the change of allocation strategy more expensive.

2.2 Destruction of a container: Clear and Finalize

All containers support two cleanup functions:

1. Clear: remove all elements. The header structure remains untouched. This can be used to free the memory when the container was created with the `Init` function.
2. Finalize: Remove all elements and the memory used by the container object using the allocator for this container. The container should NOT have been created using the `Init` function.

The syntax is:

1. `int iContainer.Clear(Container *);`
2. `int iContainer.Finalize(Container *);`

The result of those functions is less than zero when something goes wrong, greater than zero otherwise.

2.2.1 Other creation functions

1. An implicit argument to all the creation functions is the current allocator, that is used to retrieve space for the container being built. To avoid changing the current allocator, what in multi-threaded environment would need acquiring a lock to that global variable, some containers support a creation function that receives an extra argument: a custom allocator.

```
Container * iContainer.CreateWithAllocator(size_t elementsize,  
                                          ContainerMemoryManager *allocator, ...);
```

2. Sometimes it can be useful for some containers (specially lists) to create the header structure using an already existing space, for instance in the space for local variables. For this an 'Init' function can exist, that initializes a container within an existing space. Since normally the detailed structure (and the size of course) of each container header is implementation dependent, you use the Sizeof function with an argument of NULL to get the size of the header. This can be used within a C99 compiler environment to allocate the space for that variable.¹ The declaration of the container header in C99 would be:

```
int function(void)  
{  
    char listSpace[iList.Sizeof(NULL)];  
    iList.Init(listSpace);  
}
```

If C99 is not available, the best way is to just print the size of the container you are interested in, and then use that value that should stay fixed for a given version. This can be automated and you can find in the Appendix 1, a small program that generates a series of **#defines** with the values of the sizes of the containers described in this documentation.

2.3 Adding an element to a container: Add and AddRange

This operation adds the given element to a container. In sequential containers it is added at the end, in associative containers it is added at an unspecified position.

```
int iContainer.Add(Container *, void *element);
```

The result of this operation is a positive integer if success, or an error code less than zero if the operation fails.

Sequential containers support also the **AddRange** API:

```
int iContainer.AddRange(Container *,size_t n, void element[]);
```

This API allows you to pass a table of elements into a sequential container and add it with a single call.

¹This incredibly useful feature has been made now optional by the C99 committee, even if it was mandatory when the C99 standard was published.

2.4 Removing an element from a container: Erase

Removes the given element from the container. The result is an integer greater or equal to zero with the number of elements in the container after the remove operation, or an error code less than zero if the element couldn't be added.

```
int iContainer.Erase(Container *,void *element);
```

This function needs to search for the given element before erasing it. For sequential containers you can use the “RemoveAt” function, that will remove a container at a given position.

```
int iContainer.EraseAt(Container *,size_t idx);
```

For associative containers you use RemoveKey:

```
int iContainer.RemoveKey(Container *,void *Key);
```

2.5 Retrieving an element from a container: GetElement

The GetElement function retrieves an element from a container. It comes in two different flavors, one for sequential containers, and another for associative ones.

```
void *iContainer.GetElement(Container *,size_t index);  
void *iContainer.GetElement(Container *,void *Key);
```

These functions return a pointer to the requested element or NULL if the element can't be retrieved. The resulting pointer points directly to the data stored in the container. This could be used to bypass all the flags that control the access to the container. For read-only containers, use the CopyElement function that returns a copy of the requested data into a buffer.

2.6 Sorting a sequential container: Sort

The “Sort” function will sort a container in place. To keep the old, unsorted contents, make a copy of the container first.

```
int iContainer.Sort(Container *);
```

2.7 Copying a container: Copy

The “Copy” function will make a fresh copy of a container. Some fields of the header are copied: the error and compare functions, the flags, and others. Memory will be allocated with the source container allocator.

```
newContainer * iContainer.Copy(Container *);
```

2.8 Saving and loading a container to or from disk: Save and Load

The functions “Save” and “Load” will save / load the contents, state, and characteristics of a container into / from disk. They need an open file stream, open in binary mode, and in the correct direction: saving needs a stream open in the write direction, loading needs a stream open in the read direction.

2.9 Inserting a container into another: InsertIn

2.9.1 Sequential containers

```
int (*InsertIn)(Container *destination,
                size_t position,
                Container *source);
```

This function will insert into the “destination” container the contents of the “source” container at the given position. The source is not modified in any way, and a copy of its data will be used. Both containers must be of the same type and store elements of the same type. The library only tests the element size of each one.

2.9.2 Associative containers

```
int (*InsertIn)(Container *destination, Container *source);
```

This function will insert into the destination container the source container using the source container keys. Otherwise the same conditions apply as to the sequential containers: the containers must be of the same type and store elements of the same type.

2.10 Replace an element with another

2.10.1 Sequential containers: ReplaceAt

```
int (*ReplaceAt)(Container *dst, size_t position, void *newData);
```

Replaces the element at the given position with the new data.

2.10.2 Associative containers: Replace

```
int (*Replace)(Dictionary *Dict, const unsigned char *Key, void *Value);
```

Replaces the element with the given key. If the element is absent nothing is done.

2.11 Looping through all elements of a container

The user has three methods for looping through all elements:

1. Using a simple loop construct
2. Using the “Apply” function
3. Using an iterator

One the most familiar design patterns is the ITERATOR pattern, which ‘provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation’.

Traditionally, this is achieved by identifying an ITERATOR interface that presents operations to initialize an iteration, to access the current element, to advance to the next element, and to test for completion; collection objects are expected to implement this interface, usually indirectly via an auxiliary object.

This is exactly the case in the iterator proposal here. Essential to the pattern is the idea that elements are accessed sequentially, but independently of their ‘position’ in the collection; for example, labeling each element of a tree with its index in left-to-right order fits the pattern, but labeling each element with its depth does not. This traditional version of the pattern is sometimes called an EXTERNAL ITERATOR.

An alternative INTERNAL ITERATOR approach assigns responsibility for managing the traversal to the collection instead of the client: the client needs only to provide an operation, which the collection applies to each of its elements. The latter approach is simpler to use, but less flexible; for example, it is not possible for the iteration to affect the order in which elements are accessed, nor to terminate the iteration early. This is the algorithm followed by the “Apply” function.

2.11.1 Using a simple loop to iterate a container

You can iterate any sequential container with a simple loop. You use the “Size” function to limit the loop. At each loop step you get the corresponding element with the “GetElement” function, present in this form in all containers.

```
// "Container" is a pointer to some container
for (size_t i=0; i<iContainer.Size(Container); i++) {
    someType *element = iContainer.GetElement(Container,i);
    // Use "element" here.
}
```

For associative containers you retrieve first a StringCollection containing all keys using the `GetKeys` function, present in all associative containers. Then, you retrieve each element by looping through the string collection that you have obtained in a similar manner to the sequential containers.

2.11.2 Using the “Apply” function.

The “Apply” function will iterate through all elements calling a given function for each one.¹ Its prototype is:

```
void iContainer.Apply(Container, //pointer to some container
                     int(*Applyfn)(void *elem,void *arg),
                     void *arg);
```

This function receives three arguments:

1. A pointer to the container.
2. A function pointer that should point to a function that receives two arguments: the element of the container, and an extra argument where it can receive (and write to) global information about the search. This extra argument is
3. The third one passed to the “Apply” function. Apply will pass this argument to the given function together with a pointer to the element retrieved from the container.

2.11.3 Using iterators

Iterators are objects returned by each container that allow you to iterate (obviously) through all elements of a container. You use iterators like this:

```
Iterator *it = iContainer.newIterator(Container *);
mytype *myobject;
for (myobject = it->GetFirst(it);
     myobject!= NULL;
     myobject = it->GetNext(it)) {
    // Work with "myobject" here
}
iContainer.DeleteIterator(it); // dispose the iterator object
```

Iterators provide a container-independent way of iterating that will work with any container, both sequential or associative. In associative containers the specific sequence is implementation defined, and in sequential containers is the natural sequence. Iterators always support always at least two methods:

```
void *iterator->GetFirst(iterator);
void *iterator->GetNext(iterator);
void *iterator->GetCurrent(iterator);
```

All containers support the “newIterator” and “deleteIterator” methods:

```
iterator *iContainer.newIterator(Container);
int iContainer.deleteIterator(iterator);
```

Iterators must be destroyed since they are allocated using the containers default allocator.

Sequential containers can support additional functions:

```
void *iterator->GetLast(iterator);  
void *iterator->GetPrevious(iterator);
```

This interface allows users to write fully general algorithms that will work with any container, independently of its internal structure. Obviously the performance can differ from container to container depending on usage.

All iterators will become invalid if the underlying container changes in any way, except through the iterator itself.²

2.12 Setting and retrieving the state: GetFlags and SetFlags

Each container has a set of flags that can be read and written to change the container's behavior. The only flag that is defined by all containers is the read-only flag. Implementations can extend this to offer different services like copy-on-write, or other applications.

2.13 Retrieving the number of elements stored: Size

All containers support querying the number of elements stored. The prototype is:

```
size_t iContainer.Size(Container *);
```

There is no error return. If an error occurs the result is zero.

2.14 Space used: Sizeof

This computes the total size used by the container in bytes, including the header structure, the data stored, and any related storage, for instance any free lists, spare space used to grow an array, etc.

²This is completely different to the C++ language. In C++ you may have an invalid iterator if you change the underlying container or not, depending on the operation and the specific container. This is a bad interface for the following reasons:

1. There are many rules to remember without underlying principles. You have to know the specifics of each container to know if the iterators are invalidated or not.
2. Any error leads directly to catastrophic consequences instead of being caught and signalled in an orderly fashion. Worst, errors do not produce always the same consequences, depending on what were the contents of the invalid memory you are using, on the memory allocation pattern, etc. In short, any error leads to very difficult maintenance problems.
3. Any modifications of the container type lead to a review of all code that uses that container since the rules change from container to container. Iterators that worked could be invalid now. This another source of errors.


```
size_t iContainer.Sizeof(Container *);
```

If its argument is `NULL`, `Sizeof` returns the size of the container header. This can be used to allocate space for a container as a local variable for instance.

2.15 Memory management

All containers have a pointer to their allocator object. An allocator object is a simple interface that provides 4 functions:

1. `malloc`: A function that receives a `size_t` and returns a `void *` pointing to a memory block of the requested size, or `NULL` if no more memory is available
2. `realloc`: A function that will resize a previously allocated block
3. `free`: A function that will release the memory allocated previously with `malloc/realloc`.
4. `calloc`: a function that will allocate `n` objects of `m` size and clear the memory block to zero before returning it.

At the start of the library runtime a default allocator object exists that uses the four functions of the standard C library. Other allocator objects can be used, and the user can change the global allocator at any time. Each container retrieves the default allocator object when created, and stores it in the container descriptor. Any further change to the default allocator will not affect existing containers that have already an allocator. When changing the allocator you should do that before creating the container.

Some containers are created without any heap management by default. You can introduce heap management by calling the “`UseHeap`” function, that will install a new heap in the container. Other containers are always created with a heap, and you should pass them an allocator object for object creation.

2.15.1 Memory manager objects

The library provides two memory manager objects:

1. The default memory manager, that receives the standard C library functions; `malloc`, `free`, `realloc` and `calloc`.
2. The debug memory manager that implements the same functions with added functionality designed to:
 - Catch the “double free” problem.
 - Catch the overflow of a memory block
 - Catch freeing a block that wasn’t allocated

2.15.2 Pooled memory management

The problem with the traditional C memory management is that it requires that the programmer cares about each piece of RAM that is allocated by the program and follows the lifetime of each piece to ensure that it gets returned to the system for reuse. In today's software world, this is just impractical.

A better strategy is to use a pool of memory where related memory allocations can be done from a common pool. When the module finishes, all the allocated pool is freed just by destroying the whole pool. This is much easier to manage, and in many cases more efficient. The proposed interface has the following functionalities:

1. Creation. The creation function receives a memory allocator to use for this pool.
2. Alloc. This function receives a pool and a size and returns a memory block, or NULL if there is no more memory.
3. Clear. This erases all objects allocated in the pool without returning the memory to the system.
4. Destroy. This releases all memory and destroys all objects.

Note that there is no realloc, and that the "Clear" function is optional. Not all pools support it. The rationale for these decisions being that realloc would need to store the size of each block, what in a pool maintained by a single stack like pointer would be very expensive.

2.15.3 Heap of same size objects

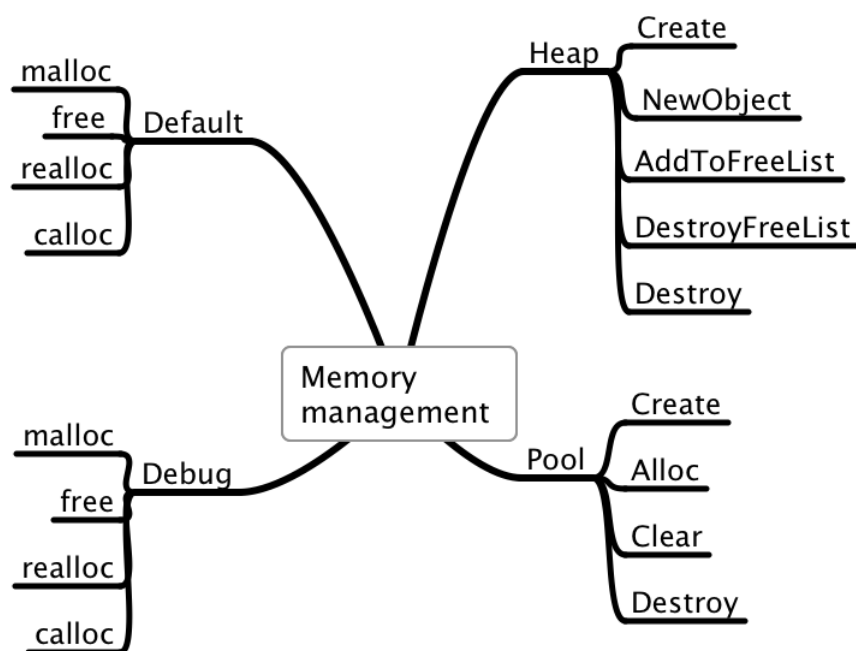
Many containers are used to store sets of objects of the same size. The library provides a specialized heap management software for this application. It stores vectors of objects of the same size. The interface provided is as follows:

- Create. This function receives a memory manager object that will be used to allocate memory.
- NewObject: returns an object to the application
- AddToFreeList: Adds an object to the list of available objects
- Size: Returns the size of the heap in bytes
- DestroyFreeList: reclaims memory used by the free list
- Destroy: Reclaims all memory used by the heap and the heap object

3 The auxiliary interfaces

3.1 Memory management

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



3.1.1 The traditional memory manager

The heap memory manager for a collection of objects of the same size the pool memory manager that manages a pool of different sized objects The basic memory managers The C language provides several functions to manage memory. The MemoryManager object presents an uniform interface for all memory managers that accept this interface.

```
typedef struct tagMemoryManager {  
    void *(*malloc)(size_t);
```

```

    void (*free)(void *);
    void *(*realloc)(void *,size_t);
    void *(*calloc)(size_t,size_t);
} ContainerMemoryManager;
extern ContainerMemoryManager * CurrentMemoryManager;

```

At startup, the CurrentMemoryManager points to an object constructed with the functions of the C standard library. This is a required interface. The user can change the object that “CurrentMemoryManager” points to to another object that should have the same interface.

The library can also include a debug version on top of the standard C functions, offering the same interface. Changing the CurrentMemoryManager to point to that object allows to switch to the debug version. The debug version offers:

- Detection of free() of a memory block not allocated by malloc().
- Detection of writing past the end of the block in some cases.
- Detection of freeing a memory block twice.

```
extern ContainerMemoryManager iDebugMalloc;
```

This interface is optional. The sample implementation documents a possible implementation, see 7.2.4 on page 204 .

3.1.2 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);
```

The standard interface for the heap is:

```

typedef struct tagHeapObject ContainerHeap;
typedef struct _HeapAllocatorInterface {
    ContainerHeap *(*Create)(size_t ElementSize,
                             ContainerMemoryManager *m);
    void *(*newObject)(ContainerHeap *heap);
    void (*AddToFreeList)(ContainerHeap *heap,void *element);
    void (*DestroyFreeList)(ContainerHeap *heap);
    void (*Destroy)(ContainerHeap *heap);
    ContainerHeap * (*InitHeap)(size_t ElementSize,void *heap,
                                ContainerMemoryManager *m);
    size_t (*Sizeof)(ContainerHeap *heap);
} HeapInterface;
extern HeapInterface iHeap;

```

Create

```
ContainerHeap *iHeap.Create(size_t elementSize, MemoryManager *m);
```

Description: 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 CurrentMemoryManager will be used.

Returns: a pointer to the new heap object or NULL, if an error occurred.

Errors:

CONTAINER_ERROR_BADARG The element size is bigger than what the heap implementation can support..

CONTAINER_ERROR_NOMEMORY Not enough memory is available to complete the operation.

InitHeap

```
ContainerHeap * (*InitHeap)(void *heap, size_t ElementSize,  
                             ContainerMemoryManager *m);
```

Description: 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 obtained 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

```
void *iHeap.newObject(ContainerHeap *heap);
```

Description: The heap returns a pointer to a new object or NULL if no more memory is left.

Errors:

CONTAINER_ERROR_NOMEMORY Not enough memory is available to complete the operation.

Returns: A pointer to an object or NULL if there is not enough memory to complete the operation.

AddToFreeList

```
size_t iHeap.AddToFreeList(ContainerHeap *heap, void *element);
```

Description: Adds the given object to the list of free objects, allowing for recycling of memory without new allocations. The element pointer can be NULL .

Errors:

CONTAINER_ERROR_BADARG The heap pointer is NULL .

Returns: The number of objects in the free list.

DestroyFreeList

```
void iHeap.DestroyFreeList(ContainerHeap *heap);
```

Description: Releases all memory used by the free list and resets the heap object to its state as it was when created.

Errors:

CONTAINER_ERROR_BADARG The heap pointer is NULL .

Finalize

```
void iHeap.Finalize(ContainerHeap *heap);
```

Description: Destroys all memory used by the indicated heap and frees the heap object itself.

Errors:

CONTAINER_ERROR_BADARG The heap pointer is NULL .

Sizeof

```
size_t iHeap.Sizeof(ContainerHeap *heap);
```

Description: 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)).

Errors:

None.

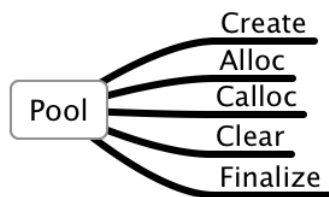
Example:

```
void SomeFunction(void)
{
    char buffer[iHeap.Sizerof(NULL)];
    ContainerHeap *ch;

    ch = iHeap.InitHeap(buffer,200,NULL);
    // ...
    iHeap.DestroyFreeList(ch);
}
```

This example uses the variable length arrays that have been introduced in the C language by the latest standard (C99). The `Sizeof` function returns the size of the header object that is used to specify the size of the buffer. The buffer is passed to the `InitHeap` function using a number of objects of 200 and the default memory allocator.

3.2 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)(ContainerMemoryManager *m);
    void >(*Alloc)(Pool *pool,size_t size);
    void >(*Calloc)(Pool *pool,size_t size);
    void  (*Clear)(Pool *);
    void  (*Finalize)(Pool *);
} PoolAllocatorInterface;
```

Note that there is no `realloc` function. Pooled memory is often implemented without storing the size of the block to cut overhead. Since a `realloc` function could be expensive, implementations are not required to provide it.

Create

```
Pool *iPool.Create(ContainerMemoryManager *m);
```

Description: Creates a new pool object that will use the given memory manager. If `m` is null, the object pointed by the `CurrentMemoryManager` will be used.

Errors:

`CONTAINER.ERROR.NOMEMORY` Not enough memory to complete the operation.

Returns: A pointer to the new object or `NULL` if the operation couldn't be completed.

Alloc

```
void *iPool.Alloc(Pool *pool,size_t size);
```

Description: Allocates size bytes from the pool pool. If there isn't enough memory to resize the pool the result is NULL .

Errors:

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A pointer to the allocated memory or NULL if error.

Calloc

```
void *iPool.Calloc(Pool *pool,size_t n,size_t size);
```

Description: 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 ;

Errors:

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A pointer to the allocated memory or NULL if error.

Clear

```
void iPool.Clear(Pool *);
```

Description: Reclaims all memory used by the pool and leaves the object as it was when created.

Errors:

CONTAINER_ERROR_BADARG The pool pointer is NULL .

Finalize

```
void iPool.Finalize(Pool *);
```

Description: Reclaims all memory used by the pool and destroys the pool object itself.

Errors:

CONTAINER_ERROR_BADARG The pool pointer is NULL .

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


RaiseError

```
void      iError.RaiseError(const char *fname,int errcode,...);
```

Description: The parameter “fname” should be the name of the function where the error occurs. The “errcode” parameter is a negative error code. The actual value of the code is implementation defined. Other parameters can be passed depending on the error. The behavior of the default error function is implementation specific. In the sample code this function will just print the error message in the standard error stream. Other implementations could end the program, or do nothing.

Returns: No return value

EmptyErrorFunction

```
void      iError.EmptyErrorFunction(const char *fname,int errcode,...);
```

Description: This function can be used to ignore all errors within the library. It does nothing.

StrError

```
const char *iError.StrError(int errorCode);
```

Description: Converts the given error code in a character string. If the error code doesn’t correspond to any error defined by the implementation a character string with an implementation defined value is returned.

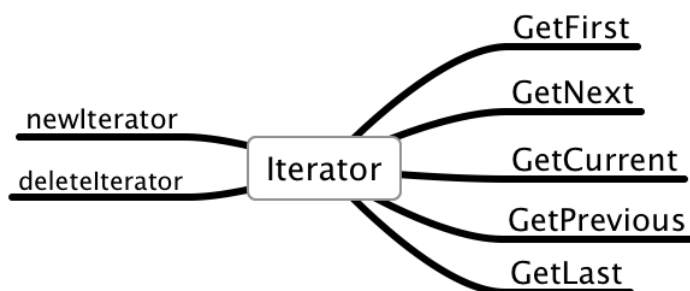
SetErrorFunction

```
ErrorFunction iError.SetErrorFunction(ErrorFunction);
```

Description: Changes the value of the default error function. If its argument is NULL , nothing is done, and the call is interpreted as a query of the current value.

Returns: The old value of the default error function.

3.4 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.

3.4.1 The interface

```
typedef struct _Iterator {
    void (*GetNext)(Iterator *);
    void (*GetPrevious)(Iterator *);
    void (*GetFirst)(Iterator *);
    void (*GetCurrent)(Iterator *);
    void (*GetLast)(Iterator *);
} Iterator;
```

GetCurrent

```
void (*GetCurrent)(Iterator *);
```

Description: Returns the element at the cursor position.

Errors:

CONTAINER_ERROR_BADARG The iterator pointer is NULL .

Returns: A pointer to the first element or NULL , if the container is empty or an error occurs. If the container is read-only, a pointer to a copy of the element is returned. This pointer is valid only until the next iterator function is called.

GetFirst

```
void (*GetFirst)(Iterator *);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG The iterator pointer is NULL .

Returns: A pointer to the first element or NULL , if the container is empty or an error occurs. If the container is read-only, a pointer to a copy of the element is returned. This pointer is valid only until the next iterator function is called.

Example:

```
Iterator *myIterator;
List *myList;
myType *obj; // "myList" stores objects of type "myType"
myIterator = iList.newIterator(myList); // Request iterator
```

```
for (obj = myIterator->GetFirst(myIterator);
    obj != NULL;
    obj = myIterator->GetNext(myIterator)) {
    //Use obj here
}
iList.deleteIterator(myIterator); // Reclaim memory
```

GetNext

```
void *(*GetNext)(Iterator *);
```

Description: Positions de cursor at the next element and returns a pointer to its contents. If the container is read-only, a pointer to a copy of the object is returned. This pointer is valid only until the next iterator function is called.

Errors:

CONTAINER_ERROR_BADARG The iterator pointer is NULL .

CONTAINER_ERROR_OBJECT_CHANGED The container has been modified and the iterator is invalid. Further calls always return NULL .

Returns: A pointer to the next element or NULL , if the cursor reaches the last element. If the container is read-only, a pointer to a copy of the element is returned, valid until the next element is retrieved

GetPrevious

```
void *(*GetPrevious)(Iterator *);
```

Description: Positions de cursor at the next element and returns a pointer to its contents. 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. In those cases it can always return NULL .

Errors:

CONTAINER_ERROR_BADARG The iterator pointer is NULL .

CONTAINER_ERROR_OBJECT_CHANGED The container has been modified and the iterator is invalid. Further calls always return NULL .

Returns: A pointer to the next element or NULL , if the cursor reached the element already. If the container is read-only, a pointer to a copy of the element is returned.

Example:

```
Iterator *myIterator;
List *myList;
myType *obj; // "myList" stores objects of type "myType"
myIterator = iList.newIterator(myList); // Request iterator
for (obj = myIterator->GetLast(myIterator);
    obj != NULL;
```

3. THE AUXILIARY INTERFACES

```
    obj = myIterator->GetPrevious(myIterator)) {  
        //Use obj here  
    }  
iList.deleteIterator(myIterator); // Reclaim memory
```

GetCurrent

```
void *GetCurrent(Iterator *);
```

Description: Returns a pointer to the current element's data without moving the cursor.

GetLast

```
void *(*GetLast)(Iterator *);
```

Description: 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.

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 the last element, for instance in single linked lists. In those cases it can always return NULL .

Errors:

CONTAINER_ERROR_BADARG The iterator pointer is NULL .

CONTAINER_ERROR_OBJECT_CHANGED The container has been modified and the iterator is invalid. Further calls always return NULL .

4 The containers

4.1 The List interfaces: iList, iDlist

The list container appears in two flavors:

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

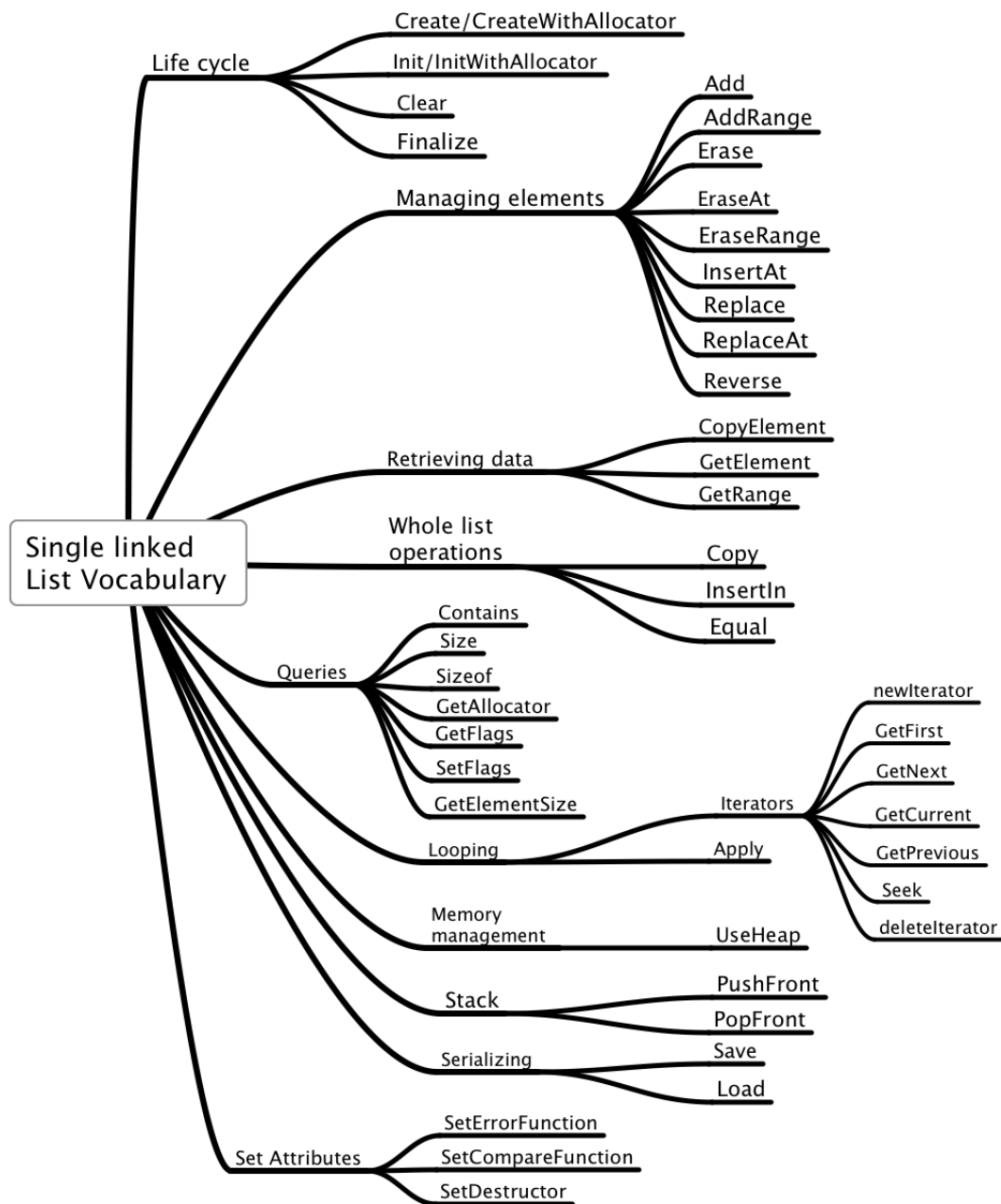
The space overhead of single linked lists is smaller at the expense of more difficult access to the elements. It is up to the application programmer to decide which container fits best in his/her application.

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 _List List;
typedef struct {
    int (*Add)(List *L,void *newval);
    int (*AddRange)(List * AL,size_t n, void *data);
    List *(*Append)(List *l1,List *l2);
    void (*Apply)(List *L,int(Applyfn)(void *elem,void *arg),void *arg);
    int (*Clear)(List *L);
    int (*Contains)(List *L,void *element);
    List *(*Create)(size_t element_size);
    List *(*CreateWithAllocator)(size_t elementsize,
                                ContainerMemoryManager *allocator);
    List *(*Copy)(List *L);
    int (*deleteIterator)(Iterator *);
    int (*Erase)(List *L,void *);
    int (*EraseAt)(List *L,size_t idx);
    int (*EraseRange)(List *L,size_t start,size_t end);
    int (*Equal)(List *l1,List *l2);
    ContainerMemoryManager *(*GetAllocator)(List *list);
    void *(*GetElement)(List *L,int idx);
    size_t (*GetElementSize)(List *l);
}
```

```
unsigned (*GetFlags)(List *L);
List (*GetRange)(List *l,size_t start,size_t end);
int (*Finalize)(List *L);
int (*IndexOf)(List *L,void *SearchedElement,size_t *result);
List (*Init)(List *aList,size_t element_size);
List (*InitWithAllocator)(List *aList,size_t element_size,
                          ContainerMemoryManager *allocator);
int (*Insert)(List *L,void *);
int (*InsertAt)(List *L,size_t idx,void *newVal);
int (*InsertIn)(List *Destination, size_t position, List *source);
List (*Load)(FILE *stream, ReadFunction loadFn,void *arg);
Iterator (*newIterator)(List *L);
int (*PushFront)(List *L,void *str);
int (*PopFront)(List *L,void *result);
int (*ReplaceAt)(List *L,size_t idx,void *newVal);
List (*Reverse)(List *l);
int (*Save)(List *L,FILE *stream, SaveFunction saveFn,void *arg);
void (*Seek)(Iterator *it,size_t pos);
CompareFunction (*SetCompareFunction)(List *l,CompareFunction fn);
DestructorFunction SetDestructor(List *l,DestructorFunction fn);
ErrorFunction (*SetErrorFunction)(List *L,ErrorFunction);
int (*Size)(List *L);
size_t (*Sizeof)(List *l);
int (*Sort)(List *l);
unsigned (*SetFlags)(List *L,unsigned flags);
int (*UseHeap)(List *L, ContainerMemoryManager *m);
} ListInterface;

extern ListInterface iList;
```



4.1.1 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 com-

puter 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 List container supports the following state flags:

```
#define CONTAINER_LIST_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.

Add

```
int (*Add)(List *l,void *data);
```

Description: Adds the given element to the container. It is assumed that “data” points to a contiguous memory area of at least ElementSize bytes. Returns a value greater than zero if the addition of the element to the list completed successfully, a negative error code otherwise. The error codes returned can be:

CONTAINER_ERROR_BADARG The list or the data pointers are NULL .

CONTAINER_ERROR_READONLY The list is read-only. No modifications allowed.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: Returns the number of elements stored if it is less than INT_MAX, or INT_MAX if there are more elements stored than the value of INT_MAX. If the return value is negative, an error occurred.

Example:

```
/* This example shows how to:
(1) Create a linked list of "double" data
(2) Fill it using the "Add" function
(3) Print it using the GetElement function */
#include <containers.h>
static void PrintList(List *AL)
{
    size_t i;
    for (i=0; i<iList.Size(AL);i++) {
        printf("%g ",*(double *)iList.GetElement(AL,i));
    }
    printf("\n");
}
static void FillList(List * AL,size_t siz)
{
    size_t i;

    for (i=0; i<siz;i++) {
        double d = i;
        iList.Add(AL,&d);
    }
}

int main(void)
{
    List *AL = iList.Create(sizeof(double));
    FillList(AL,10);
    PrintList(AL);
    return 0;
}
OUTPUT:
0 1 2 3 4 5 6 7 8 9
```

AddRange

```
int (*AddRange)(List * AL,size_t n, void *data);
```

Description: 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 .

Errors:

CONTAINER_ERROR_BADARG The list or the data pointers are NULL , and n is not zero.

CONTAINER_ERROR_READONLY The list is read-only. No modifications allowed.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive number if the operation completed, negative error code otherwise.

Append

```
int (*Append)(List *list1,List *list2);
```

Description: Appends the contents of list2 to list1 and destroys list2.

Errors:

CONTAINER_ERROR_BADARG Either list1 or list2 are NULL .

CONTAINER_ERROR_READONLY One or both lists are read only.

Returns: A positive value if the operation succeeded, or a negative error code otherwise.

Example:

```
#include <containers.h>
static void PrintList(List *AL)
{
    size_t i;
    for (i=0; i<iList.Size(AL);i++) {
        printf("%g ",*(double *)iList.GetElement(AL,i));
    }
    printf("\n");
}
static void FillList(List * AL,size_t siz)
{
    size_t i;

    for (i=0; i<siz;i++) { double d = i; iList.Add(AL,&d);}
}

int main(void)
{
    List *L1 = iList.Create(sizeof(double));
    List *L2 = iList.Create(sizeof(double));
    FillList(L1,10);
    FillList(L2,10);
    iList.Append(L1,L2);
    PrintList(L1);
    return 0;
}
```

OUTPUT:

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

Apply

```
int (*Apply)(List l,int (Applyfn)(void *,void *),void *arg);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG Either list or Applyfn are NULL .

CONTAINER_ERROR_NOMEMORY : The list is read-only and there is no more memory to allocate the buffer to copy each element.

Notes:

The list container of C++ has no direct equivalent, but in the algorithm part of the STL there is a “for_each” construct, that does essentially the same. Java and C# offer a similar “ForEach” functionality.

Example:

```
#include <containers.h>
static int Callback(void *pElement,void *pResult)
{
    double *p = pElement;
    double *result = pResult;
    *result += *p;
    return 1;
}

void main(void)
{
    double sum = 0;
    List *list = iList.Create(sizeof(double));
    double d = 2;
    iList.Add(list,&d);
    d = 3;
    iList.Add(list,&d);
    iList.Apply(list,Callback,&sum);
    // Here sum should be 5.
    printf("%g\n",sum);
}
```

The above example shows a function callback as used by “Apply”. It receives two pointers, one to the current element and another to an extra argument that in this case

contains a pointer to the sum. For each call to the callback, the function adds the contents of the element to the sum.

The main function creates a list, adds two elements with the values 2 and 3, and then calls "Apply" to get their sum using the callback.

Clear

```
int (*Clear)(List *l);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG The list pointer is NULL .

CONTAINER_ERROR_READONLY The list is read only.

Returns: The result is greater than zero if successful, or an error code if an error occurs.

Notes:

Java, C++ and C# have a similar "Clear" functionality.

Example:

```
List *l;  
int m = iList.Clear(l);
```

Contains

```
int (*Contains)(List *list,void *data);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG Either list or data are NULL .

Notes:

C++ has std::find that does essentially the same . Java and C# have a "Contains" method.

Example:

```
List *list;  
int r = iList.Contains(list,&data);
```

Copy

```
List *(*Copy)(List *L);
```

Description: 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.

Errors:

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

CONTAINER_ERROR_BADARG The given list pointer is NULL .

Notes:

C++ has no direct equivalent but the assignment operator should work, Java and C# support a copy method.

Example:

```
List *newList,*OldList;
newList = iList.Copy(OldList);
if (newList == NULL) { /* Error handling */ }
```

CopyElement

```
int (*CopyElement)(List *list,size_t idx,void *outBuffer);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG The given list pointer or the output buffer are NULL .

CONTAINER_ERROR_INDEX The given position is out of bounds.

Returns: A positive value if the operation succeeded, or a negative error code if it failed.

Notes:

Neither C# nor Java provide this functionality because the treatment of pointers in those languages makes the need for such a construct unnecessary.

Example:

```
List *list; double d;
if (iList.CopyElement(list,3,&d) > 0)
    printf("The value at position 3 is %g\n",d);
```

Create

```
List *(*Create)(size_t element_size);
```

Description: 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.

Returns: A pointer to a newly created List or NULL if an error occurs.

Errors:

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

CONTAINER_ERROR_BADARG The given element size is zero or greater than what the implementation allows for maximum object size.

Errors provoke the call of the current default error function of the library since this is the creation function and there isn't a container specific error function yet.

Example:

```
List *ListOfDoubles = iList.Create(sizeof(double));
```

CreateWithAllocator

```
List *(*CreateWithAllocator)(size_t elem_size,  
                             ContainerMemoryManager *allocator);
```

Description: The creation function returns an empty List container, initialized with all the default values. The given 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.

Returns: A pointer to a newly created List or NULL if an error occurs.

Errors:

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

CONTAINER_ERROR_BADARG The given element size is zero or greater than what the implementation allows for maximum object size, or the given allocator pointer is NULL .

Errors provoke the call of the current default error function of the library since this is the creation function and there isn't a container specific error function yet.

Example:

```
ContainerMemoryManager *myAllocator;  
List *ListOfDoubles =  
    iList.CreateWithAllocator(sizeof(double),myAllocator);
```

deleteIterator

```
int deleteIterator(Iterator *it);
```

Description: Reclaims the memory used by the given iterator object

Errors:

CONTAINER_ERROR_BADARG The iterator pointer is NULL .

Returns: A positive value if successful or a negative error code.

Equal

```
int (*Equal)(List *list1, List *list2);
```

Description: Compares the given lists using the list comparison function of either list1 or list2 that must compare equal. If the lists 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.

Errors:

None

Returns: The result is one if the lists are equal, zero otherwise.

Erase

```
int (*Erase)(List *list, void *data);
```

Description: Removes from the list the element that matches the given data, that is assumed to be a pointer to an element.

Returns: A negative error code if an error occurred, or a positive value that indicates that a match was found and the element was removed. If the element is not in the list the result is CONTAINER_ERROR_NOTFOUND .

Errors:

CONTAINER_ERROR_BADARG One or both arguments are NULL .

Example:

```
double d = 2.3;
List *list;
int r = iList.Erase(list, &d);
if (r > 0)
    printf("2.3 erased\n");
else if (r == CONTAINER_ERROR_NOTFOUND)
    printf("No element with value 2.3 present\n");
else
    printf("2.3 not erased. Error is %s\n", iError.StrError(r));
```

EraseAt

```
int (*EraseAt)(List *list, size_t idx);
```

Description: Removes from the list the element at the given position.

Returns: A negative error code if an error occurred or a positive value that indicates that the element was removed.

Errors:

CONTAINER_ERROR_BADARG The given list pointer is NULL .

CONTAINER_ERROR_INDEX The given position is out of bounds.

Example:

```
List *list;
int r = iList.EraseAt(list,2);
if (r > 0)
    printf("Element at position 2 erased\n");
else
    printf("Error code %d\n",r);
```

EraseRange

```
int (*EraseRange)(List *L,size_t start,size_t end);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG The given list pointer is NULL .

Returns: A positive number indicates success, zero means nothing was erased, and a negative number an error.

Example:

```
#include <containers.h>
static void print_list(List *li)
{
    int i;
    for (i=0; i<iList.Size(li); i++)
        printf(" %d",*(int *)iList.GetElement(li,i));
    printf("\n");
}
int main(void)
{
    List *li = iList.Create(sizeof(int));
    int i;
    for (i=0; i<10;i++) {
        iList.Add(li,&i);
    }
    print_list(li);
    iList.EraseRange(li,3,8);
```



```
    print_list(li);
}
```

OUTPUT:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 8 9
```

Finalize

```
int (*Finalize)(List *list);
```

Description: Reclaims all memory used by the list, including the list header object itself.

Errors:

CONTAINER_ERROR_BADARG The given list pointer is NULL .

CONTAINER_ERROR_READONLY The list is read-only. No modifications allowed.

Returns: A positive value means the operation completed. A negative error code indicates failure.

Example:

```
List *list;
int r = iList.Finalize(list);
if (r < 0) { /* error handling */ }
```

GetAllocator

```
ContainerMemoryManager *(*GetAllocator)(List *l);
```

Description: Returns the list's allocator object. If the list pointer is NULL it returns NULL .

GetElementSize

```
size_t (*GetElementSize)(List *l);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG The given list pointer is NULL .

Returns: The element size.

Example:

```
List *l;
size_t siz = iList.GetElementSize(l);
```

GetElement

```
void *(*GetElement)(List *list,size_t idx);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG The given list pointer is NULL .

CONTAINER_ERROR_INDEX The given position is out of bounds.

CONTAINER_ERROR_READONLY The list is read only.

Example:

```
List *list;
double d = *(double *)iList.GetElement(list,3);
```

GetFlags / SetFlags

```
unsigned (*GetFlags)(List *l);
unsigned (*SetFlags)(List *l,unsigned newFlags);
```

Description: GetFlags returns the state of the container flags, SetFlags sets the flags to a new value and returns the old value.

Errors:

CONTAINER_ERROR_BADARG The given list pointer is NULL .

Returns: The flags or zero if there was an error.

GetRange

```
List *(*GetRange)(List *list,size_t start,size_t end);
```

Description: Selects a series of consecutive elements starting at position start and ending at position end. Both the elements at start and end are included in the result. If start is greater than end start and end are interchanged. If end is bigger than the number of elements in list, only elements up to the number of elements will be used. If both start and end are out of range an error is issued and NULL is returned. The selected elements are copied into a new list. The original list remains unchanged.

Errors:

CONTAINER_ERROR_BADARG The given list pointer is NULL

CONTAINER_ERROR_INDEX Both start and end are out of range.

Returns: A pointer to a new list containing the selected elements or NULL if an error occurs.

Example:

```
List *list;
List *range = iList.GetRange(list,2,5);
if (range == NULL) { /* Error handling */ }
```

IndexOf

```
int (*IndexOf)(List *l,void *ElementToFind,void *args,size_t *result);
```

Description: Searches for an element in the list. If found its zero based index is returned in the passed pointer "result".

Otherwise the result of the search is `CONTAINER_ERROR_NOTFOUND` and the passed pointer will remain unmodified. The "args" argument will be passed to the comparison function that is called by `IndexOf`.

Errors:

`CONTAINER_ERROR_BADARG` The given list pointer or element are NULL .

Returns: A positive value if element is found or a negative value if not found or an error occurs.

Example:

```
List *list;
double data;
size_t idx;
int r = iList.IndexOf(list,&data,&idx);
if (r == CONTAINER_ERROR_NOTFOUND)
    printf("Not found\n");
else if (r < 0)
    printf("Error\n");
else printf("Found at position %ld\n",idx);
```

Init

```
List *(*Init)(List *aList,size_t element_size);
```

Description: Initializes the memory pointed by the `aList` argument. The new list will use the allocator pointed by the current memory allocator. It is assumed that the memory pointed by `aList` contains at least the size of the header object. This size can be obtained by calling the `Sizeof` function with a NULL argument.

Errors:

`CONTAINER_ERROR_BADARG` The given list pointer is NULL .

Example:

```
// This example uses C99
void Example(void)
{
    char aList[iList.Sizeof(NULL)];
    List *list = iList.Init((List *)aList);
}
```

InitWithAllocator

```
List *(*InitWithAllocator)(List *aList,  
                           size_t element_size,  
                           ContainerMemoryManager *allocator);
```

Description: Initializes the memory pointed by the `aList` argument. The new list will use the given allocator. It is assumed that the memory pointed by `aList` contains at least the size of the header object. This size can be obtained by calling the `Sizeof` function with a `NULL` argument.

Errors:

`CONTAINER_ERROR_BADARG` The given list pointer is `NULL` .

InsertAt

```
int (*InsertAt)(List *L,size_t idx,void *newData);
```

Description: Inserts the new element. The new element will have the given index, that can go from zero to the list count inclusive, i.e. one more than the number of elements in the list. In single linked lists the cost for this operation is proportional to `idx`.

Errors:

`CONTAINER_ERROR_BADARG` The given list pointer or the element given are `NULL` .

`CONTAINER_ERROR_READONLY` The list is read only.

`CONTAINER_ERROR_INDEX` The given position is out of bounds.

`CONTAINER_ERROR_NOMEMORY` There is not enough memory to complete the operation.

Returns: A positive value if the operation succeeded, or a negative error code if the operation failed.

Example:

```
double d;  
List *list;  
int r = iList.InsertAt(list,2,&d);  
if (r < 0) { /* Error handling */ }
```

InsertIn

```
int (*InsertIn)(List *Destination, size_t position, List *source);
```

Description: Inserts the list given in its third argument at the given position in the list pointed to by its first argument. The data is copied, and the source argument is not modified in any way. Both lists must have elements of the same type. The library only tests the size of each one.

Errors:

`CONTAINER_ERROR_BADARG` The source or the destination lists are `NULL` .

`CONTAINER_ERROR_READONLY` The destination list is read only.

`CONTAINER_ERROR_INDEX` The given position is out of bounds.

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

CONTAINER_ERROR_INCOMPATIBLE The lists store elements of different size.

Returns: A positive value if the operation succeeded, or a negative error code if the operation failed.

Example:

```
#include <containers.h>
/* Prints the contents of a list */
static void PrintList(List *AL)
{
    size_t i;
    printf("Count %ld\n", (long)iList.Size(AL));
    for (i=0; i<iList.Size(AL);i++) {
        printf("%g ", *(double *)iList.GetElement(AL,i));
    }
    printf("\n");
}

/* Fills a list with 10 numbers. The 10 is hardwired... */
static void FillList(List * AL,int start)
{
    size_t i;

    for (i=0; i<10;i++) {
        double d = i+start;
        iList.Add(AL,&d);
    }
}

/* Creates two lists: one with the numbers from 0 to 9, another
   with numbers 100 to 109, then inserts the second into the
   first at position 5 */
int main(void)
{
    List *AL = iList.Create(sizeof(double));
    List *AL1 =iList.Create(sizeof(double));
    FillList(AL,0);
    FillList(AL1,100);
    iList.InsertIn(AL,5,AL1);
    PrintList(AL);
    return 0;
}
```

OUTPUT:

Count 20

0 1 2 3 4 100 101 102 103 104 105 106 107 108 109 5 6 7 8 9

Load

```
List *(*Load)(FILE *stream, ReadFunction readFn, void *arg);
```

Description: Reads a list previously saved with the Save function from the stream pointed to by stream. If readFn is not NULL, it will be used to read each element. The “arg” argument will be passed to the read function. If the read function is NULL, this argument is ignored and a default read function is used.

Errors:

CONTAINER_ERROR_BADARG The given stream pointer is NULL.

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

Returns: A new list or NULL if the operation could not be completed. Note that the function pointers in the list are NOT saved, nor any special allocator that was in the original list. Those values will be the values by default. To rebuild the original state the user should replace the pointers again with the new list.

newIterator

```
Iterator *(*newIterator)(List *list);
```

Description: Allocates and initializes a new iterator object to iterate this list.

Errors:

CONTAINER_ERROR_NOMEMORY No more memory is available.

Returns: A pointer to a new iterator or NULL if there is no more memory left.

Example:

```
List *list;
Iterator *it = iList.newIterator(list);
double *d;
for (d=it->GetFirst(it); d != NULL; d = it->GetNext(it)) {
    double val = *d;
    // Work with the value here
}
iList.deleteIterator(it);
```

PopFront

```
int (*PopFront)(List *L, void *result);
```

Description: Pops the element at position zero copying it to the result pointer. If the “result” pointer is NULL, the first element is removed without any copying. The library supposes that result points to at least ElementSize bytes of contiguous storage.

Errors:

CONTAINER_ERROR_BADARG The list or the result pointer are NULL.

CONTAINER_ERROR_READONLY The list is read only.

Returns: A positive value if an element was popped, zero if the list was empty, or a negative error code if an error occurred.

Example:

```
double d;
int r = iList.PopFront(L,&d);
if (r==0)
    printf("List empty\n");
else if (r < 0) {
    printf("Error %d\n",r);
else    printf("OK, popped value %g\n",d);
```

PushFront

```
int (*PushFront)(List *L,void *element);
```

Description: Inserts the given element at position zero.

Errors:

CONTAINER_ERROR_BADARG The list or the element pointer are NULL .

CONTAINER_ERROR_READONLY The list is read only.

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

Returns: A positive value if the operation completed, or a negative error code otherwise.

Example:

```
double d = 2.3;
if (iList.PushFront(list,&d) < 0)
    printf("Error\n");
```

ReplaceAt

```
int (*ReplaceAt)(List *list,size_t idx,void *newData);
```

Description: Replaces the list element at position idx with the new data starting at the position pointed to by “newData” and extending ElementSize bytes.

Errors:

CONTAINER_ERROR_BADARG The list or the new element pointer are NULL .

CONTAINER_ERROR_READONLY The list is read only.

CONTAINER_ERROR_INDEX The given position is out of bounds.

Returns: A negative error code if an error occurs, or a positive value if the operation succeeded.

Example:

```
List *list;
double d = 6.7;
int r = iList.ReplaceAt(list,2,&d);
if (r < 0) { /* Error handling */ }
```

Reverse

```
int (*Reverse)(List *list);
```

Description: Reverses the order of the given list: the head becomes the tail and the tail becomes the head. The original order is lost.

Errors:

CONTAINER_ERROR_BADARG The list pointer is NULL .

CONTAINER_ERROR_READONLY The list is read only.

Returns: A negative error code if an error occurs, or a positive value if the operation succeeded.

Example:

```
#include <containers.h>
static void print_list(List *li)
{
    int i;
    for (i=0; i<iList.Size(li); i++)
        printf(" %d",*(int *)iList.GetElement(li,i));
    printf("\n");
}
int main(void)
{
    List *li = iList.Create(sizeof(int));
    int i;
    for (i=0; i<10;i++) {
        iList.Add(li,&i);
    }
    print_list(li);
    iList.Reverse(li);
    print_list(li);
}
```

OUTPUT

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

Seek

```
void *(*Seek)(Iterator *it,size_t pos);
```


Description: Positions the given iterator at the indicated position and then returns a pointer to the element at that position. If the position is bigger than the last element of the list, the last element position will be used.

Errors:

CONTAINER_ERROR_BADARG The list or the new element pointer are NULL .

Returns: A pointer to the data of the given element or NULL if error.

Save ---

```
int (*Save)(List *l, FILE *stream, SaveFunction SaveFn, void *arg);
```

Description: The contents of the given list are saved into the given stream. If the save function pointer is not NULL , it will be used to save the contents of each element and will receive the arg argument passed to Save. Otherwise a default save function will be used and arg will be ignored.

Errors:

CONTAINER_ERROR_BADARG The list pointer or the stream pointer are NULL . EOF A disk input/output error occurred.

Returns: A positive value if the operation completed, a negative value or EOF otherwise.

SetCompareFunction ---

```
CompareFunction (*SetCompareFunction)(List l, CompareFunction f);
```

Description: if the f argument is non NULL , it sets the list comparison function to f.

Errors:

CONTAINER_ERROR_BADARG The list pointer is NULL .

CONTAINER_ERROR_READONLY The list is read only and the function argument is not NULL .

Returns: The old value of the comparison function.

Example:

```
ErrorFunction fn, newfn;  
List *list;  
fn = iList.SetCompareFunction(list, newfn);
```

SetAllocator ---

```
List *SetAllocator(List *l, ContainerMemoryManager *allocator);
```

Description: Replaces the current allocator for the given list with the new one function if different from NULL . The list must be empty, and the new allocator must be able to allocate at least the size of the list header.

Errors:

CONTAINER_ERROR_BADARG The list pointer is NULL .

`CONTAINER_ERROR_READONLY` The list is read only and the function argument is not `NULL` .

Returns: The old value of the allocator, or `NULL` if there is an error.

SetDestructor

```
DestructorFunction SetDestructor(List *l, DestructorFunction fn);
```

Description: Sets the destructor function to its given argument. If the function argument is `NULL` nothing is changed and the call is interpreted as a query since the return value is the current value of the destructor function. If the list argument is `NULL` , the result is `NULL` .

Returns: The old value of the destructor.

SetErrorFunction

```
ErrorFunction (*SetErrorFunction)(List *L, ErrorFunction);
```

Description: Replaces the current error function for the given list with the new error function if different from `NULL` .

Errors:

`CONTAINER_ERROR_BADARG` The list pointer is `NULL` .

`CONTAINER_ERROR_READONLY` The list is read only and the function argument is not `NULL` .

Returns: The old value of the error function, or `NULL` if there is an error.

Size

```
size_t (*Size)(List *l);
```

Description: Returns the number of elements stored in the list.

Errors:

If the given list pointer is `NULL` , it returns `SIZE_MAX`.

Example:

```
List *li;  
size_t bytes = iList.Size(li);
```

Sizeof

```
size_t (*Sizeof)(List *list);
```

Description: Returns the total size in bytes of the list, including the header, and all data stored in the list. If `list` is `NULL` , the result is the size of the `List` structure.

Returns: The number of bytes used by the list or the size of the empty `List` container if the argument is `NULL` .

Example:

```
List *list;  
size_t siz = iList.Sizeof(list);
```

Sort

```
int Sort(List *list);
```

Description: Sorts the given list using the list comparison function. The order of the original list is destroyed. You should copy it if you want to preserve it.

Returns: A positive number if sorting succeeded, a negative error code if not.

Example:

```
List *list;  
if (iList.Sort(list) < 0) { /* Error handling */ }
```

UseHeap

```
int (*UseHeap)(List *list, ContainerMemoryManager *m);
```

Description: Adds a heap manager to the given list, that should be empty. The heap manager will manage the free list and the allocation of new objects. Use this function when the list will hold a great number of elements. This function is optional and may not be present in all implementations. If m is NULL, the current memory manager object will be used for allocating and reclaiming memory. Otherwise m should be a memory manager object.

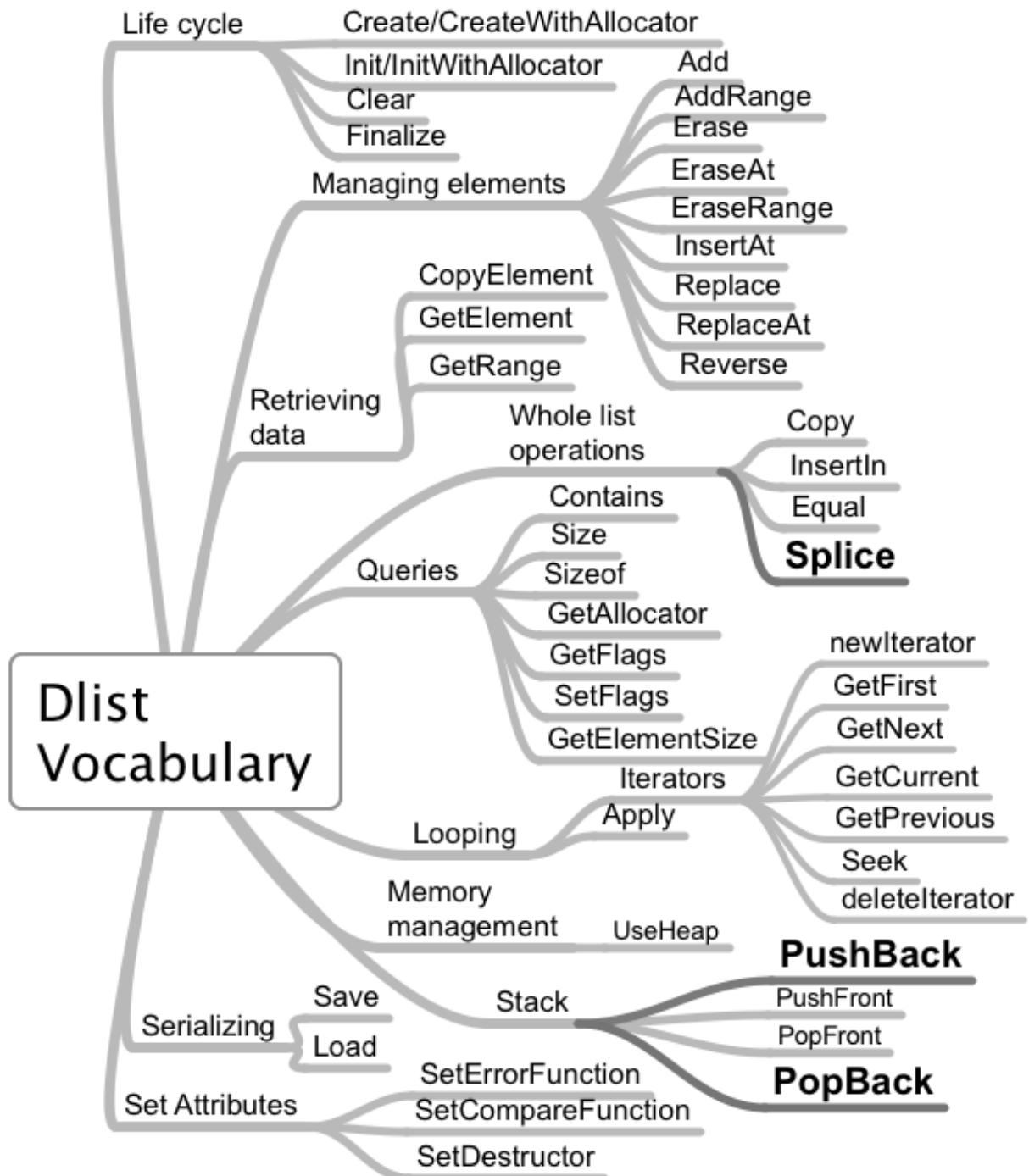
Errors:

CONTAINER_ERROR_BADARG The list pointer is NULL .

CONTAINER_ERROR_NOT_EMPTY The list is not empty or has already a heap.

Portability: This function is optional and may not be present in all implementations.

4.2 Double linked lists: iDlist



Differences with the list vocabulary are marked in bold.

Double linked lists have a pair of pointers pointing to the next and to the previous element in the list. It is easy then, to move in either direction through the list. The price to pay is a higher overhead for each element. This container shares most of its interface with the single linked list container. Here we document the functions that aren't already described for the list container.

```
typedef struct Dlist Dlist;
typedef struct {
    int (*Add)(Dlist *dlist,void *newVal);
    Dlist *(*Append)(Dlist *l1,Dlist *l2);
    int (*Apply)(Dlist *L,int(ApplyFn)(void *,void *),void *arg);
    int (*Clear)(Dlist *dlist);
    int (*Contains)(Dlist *dlist,void *element);
    Dlist *(*Copy)(Dlist *dlist);
    Dlist *(*Create)(size_t elementSize);
    Dlist *(*CreateWithAllocator)(size_t elementSize,
                                ContainerMemoryManager *,allocator);
    int (*deleteIterator)(Iterator *);
    int (*Equal)(Dlist *l1,Dlist *l2);
    int (*Erase)(Dlist *dlist,void *);
    int (*EraseAt)(Dlist *dlist,size_t idx);
    int (*Finalize)(Dlist *dlist);
    size_t (*GetElementSize)(Dlist *);
    void *(*GetElement)(Dlist *dlist,int idx);
    unsigned (*GetFlags)(Dlist *dlist);
    Dlist *(*GetRange)(Dlist *l,size_t start,size_t end);
    int (*IndexOf)(Dlist *dlist,void *SearchedElement,size_t *result);
    Dlist *(*Init)(Dlist *dlist,size_t elementsize);
    int (*Insert)(Dlist *dlist,void *);
    int (*InsertAt)(Dlist *dlist,size_t idx,void *newVal);
    int (*InsertIn)(Dlist *l, size_t idx,Dlist *newData);
    Dlist *(*Load)(FILE *stream, ReadFunction loadFn,void *arg);
    Iterator *(*newIterator)(Dlist *);
    int (*PopBack)(Dlist *AL,void *result);
    int (*PopFront)(Dlist *AL,void *result);
    int (*PushBack)(Dlist *AL,void *str);
    int (*PushFront)(Dlist *AL,void *str);
    int (*ReplaceAt)(Dlist *dlist,size_t idx,void *newVal);
    Dlist *(*Reverse)(Dlist *l);
    int (*Save)(Dlist *L,FILE *stream, SaveFunction saveFn,void *arg);
    CompareFunction (*SetCompareFunction)(Dlist *l,CompareFunction Fn);
    DestructorFunction (*SetDestructor)(Dlist *l,
                                       DestructorFunction fn);
    ErrorFunction (*SetErrorFunction)(Dlist *L,ErrorFunction);
}
```

```
    unsigned (*SetFlags)(Dlist *dlist,unsigned flags);
    int (*Size)(Dlist *dlist);
    int (*Sort)(Dlist *l);
    Dlist *(*Splice)(Dlist *list,
                     void *Pos,Dlist *toInsert,int direction);
    int (*UseHeap)(Dlist *L, ContainerMemoryManager *m);
} DlistInterface;

extern DlistInterface iDlist;
```

PopBack

```
int (*PopBack)(List *L,void *result);
```

Description: Pops the element at position zero copying it to the result pointer. If the “result” pointer is NULL , the last element is removed without any copying. Otherwise, the library supposes that result points to at least ElementSize bytes of contiguous storage.

Errors:

CONTAINER_ERROR_BADARG The list or the result pointer are NULL .

CONTAINER_ERROR_READONLY The list is read only.

Returns: A positive value if an element was popped, zero if the list was empty, or a negative error code if an error occurred.

Example:

```
double d;
int r = iList.PopBack(L,&d);
if (r==0)
    printf("List empty\n");
else if (r < 0) {
    printf("Error %d\n",r);
} else
    printf("OK, popped value %g\n",d);
```

PushBack

Synopsis:

```
int (*PushBack)(List *L,void *element);
```

Description: Inserts the given element at position zero.

Errors:

CONTAINER_ERROR_BADARG The list or the element pointer are NULL .

CONTAINER_ERROR_READONLY The list is read only.

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

Returns: A positive value if the operation completed, or a negative error code otherwise.

Example:

```
double d = 2.3;
if (iList.PushFront(list,&d) < 0)
    printf("Error\n");
```

Splice

Synopsis:

```
Dlist *(*Splice)(Dlist *list, void *Pos, Dlist *toInsert,int direction);
```

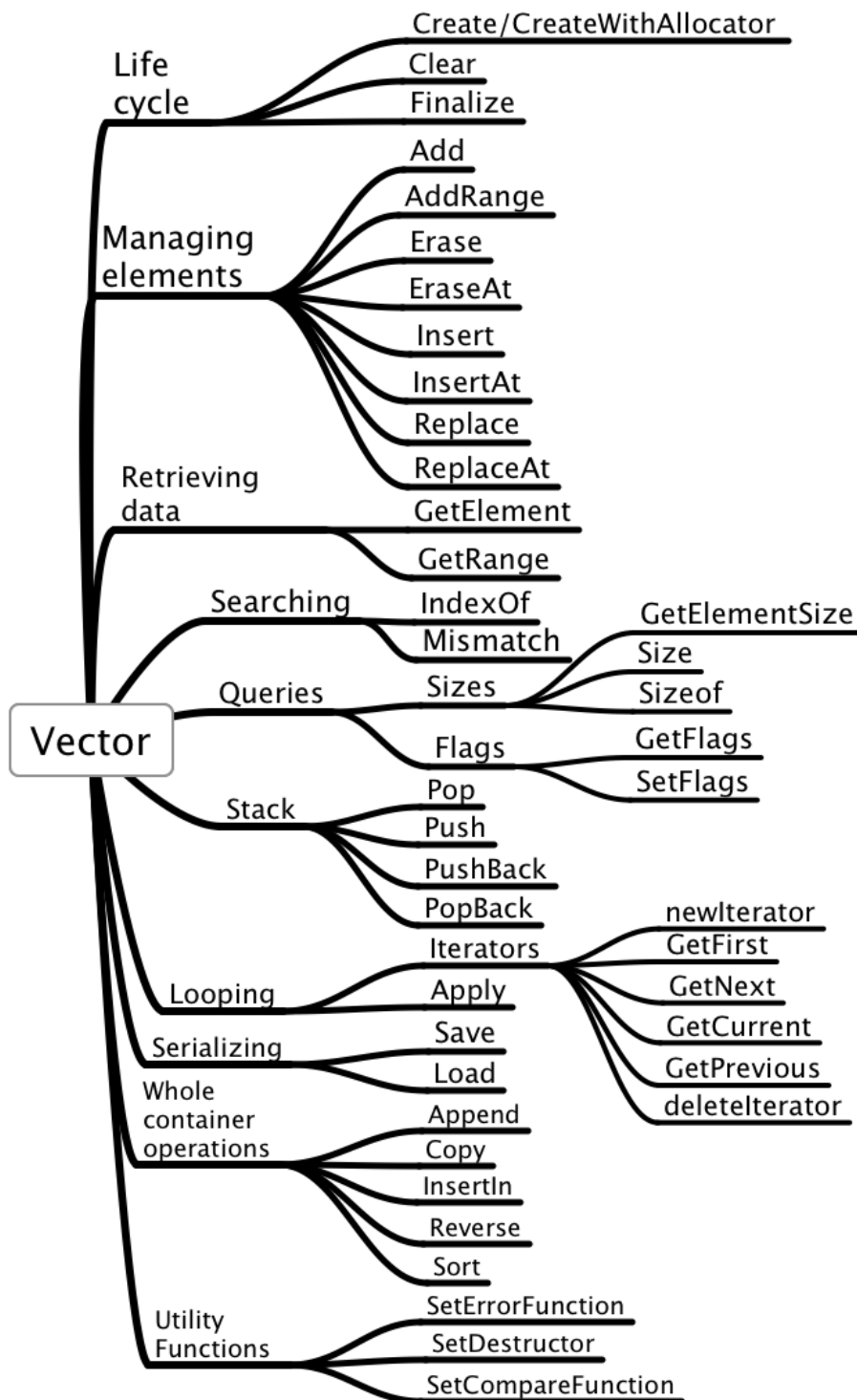
Description: Inserts a list (parameter “toInsert”) into another one (parameter “list”) at the given position that should be an element of “list”. The direction argument means to insert before the position if zero, after the position if not zero.

Errors:

CONTAINER_ERROR_BADARG The list, the list to be inserted or the element pointer are NULL .

CONTAINER_ERROR_READONLY The list is read only.

4.3 The Vector interface: iVector



Vector vocabulary.

The "vector" container is an array that resizes to accommodate new elements. Access is always checked against the array bounds. Insertion and deletion of items are more expensive than in lists, and the cost increases linearly with the array size. Access is very cheap, since a multiplication suffices to get to any array position.

Most functions of the interface are shared with the list, Dlist and the other sequential containers.

4.3.1 The interface

```
typedef struct {
    int (*Add)(Vector *AL,void *newval);
    int (*AddRange)(Vector *AL,size_t n, void *newvalues);
    int (*Append)(Vector *AL1,Vector *AL2);
    int (*Apply)(Vector *AL,
                 int (*Applyfn)(void *elm,void *arg),
                 void *arg);
    int (*Clear)(Vector *AL);
    Vector *(*Create)(size_t elementsize,size_t startsize);
    Vector *(*CreateWithAllocator)(size_t elementsize,
                                   size_t startsize,ContainerMemoryManager *allocator);
    int (*Contains)(Vector *AL,void *str,void *ExtraArgs);
    Vector *(*Copy)(Vector *AL);
    void **(*CopyTo)(Vector *AL);
    int (*deleteIterator)(Iterator *);
    int (*Erase)(Vector *AL,void *);
    int (*EraseAt)(Vector *AL,size_t idx);
    int (*Finalize)(Vector *AL);
    size_t (*Finalize)(Vector)(const Vector *AL);
    void *(*GetElement)(Vector *AL,size_t idx);
    size_t (*GetElementSize)(const Vector *AL);
    unsigned (*GetFlags)(const Vector *AL);
    Vector *(*GetRange)(Vector *AL,size_t start,size_t end);
    int (*IndexOf)(Vector *AL, void *elemToSearch,
                  void *ExtraArgs, size_t *result);
    int (*Insert)(Vector *AL,void *);
    int (*InsertAt)(Vector *AL,size_t idx,void *newval);
    int (*InsertIn)(Vector *l, size_t idx,Vector *newData)
    Vector *(*Load)(FILE *stream, ReadFunction readFn,void *arg);
    int (*Mismatch)(const Vector *a1,const Vector *a2,
                   size_t *result);

    Iterator *(*newIterator)(Vector *AL);
    int (*PushBack)(Vector *AL,void *element);
    int (*PopBack)(Vector *AL,void *result);
    int (*ReplaceAt)(Vector *AL,size_t idx,void *newval);
```

```
int (*Reverse)(Vector *AL);
int (*SetCapacity)(Vector *AL,size_t newCapacity);
CompareFunction (*SetCompareFunction)(Vector *AL,
                                      CompareFunction f);
DestructorFunction (*SetDestructor)(Vector *v,
                                   DestructorFunction fn);
ErrorFunction (*SetErrorFunction)(Vector *AL,ErrorFunction);
unsigned (*SetFlags)(Vector *AL,unsigned flags);
size_t (*Size)(const Vector *AL);
size_t (*Sizeof)(Vector *AL);
int (*Sort)(Vector *AL);
int (*Save)(Vector *AL,FILE *stream, SaveFunction Fn,void *arg);
} VectorInterface;
```

4.3.2 The API

Add

```
int (*Add)(Vector *AL,void *data);
```

Description: Adds the given element 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 ElementSize bytes. Returns a value greater than zero if the addition completed successfully, a negative error code otherwise.

Errors:

CONTAINER_ERROR_BADARG The vector or the data pointers are NULL .

CONTAINER_ERROR_READONLY The vector is read-only. No modifications allowed.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive number if the operation completed, negative error code otherwise.

Example:

```
Vector *AL;
double data = 4.5;
int result = iVector.Add(AL,&data);
if (result < 0) { /* Error handling */ }
```

AddRange

```
int (*Add)(Vector *AL,size_t n, void *data);
```

Description: 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. Returns a value greater than zero if the addition completed successfully, a negative error code otherwise. If n is zero no error is issued even if the array pointer or the data pointer are NULL .

Errors:

CONTAINER_ERROR_BADARG The vector or the data pointers are NULL , and n is not zero.

CONTAINER_ERROR_READONLY The vector is read-only. No modifications allowed.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive number if the operation completed, negative error code otherwise.

Example:

```
Vector *AL;
double data[] = {4.5, 4.6, 4.7 };
int result = iVector.Add(AL,3, data);
if (result < 0) { /* Error handling */ }
```

Append

```
int (*Append)(Vector *AL1, Vector *AL2);
```

Description: Adds all elements of AL2 at the end of the first container AL1.

Errors:

CONTAINER_ERROR_BADARG One of the Vector pointer is NULL .

CONTAINER_ERROR_READONLY The first argument is read-only. No modifications allowed.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive number if the operation completed, negative error code otherwise.

Apply

```
int (*Apply)(Vector l,int (Applyfn)(void *,void *),void *arg);
```

Description: Will call the given function for each element of the array. The first argument of the callback function receives an element of the array. 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 array is read-only, a copy of the element will be passed to the callback function.

Errors:

CONTAINER_ERROR_BADARG Either list or Applyfn are NULL .

CONTAINER_ERROR_NOMEMORY The list is read-only and there is no more memory to allocate the buffer to copy each element.

Returns: A positive value if no errors or a negative error code.

Example:

```
static int Callback(void *pelement,void *pResult)
{
    double *p = pelement;
```

```
        double *result = pResult;
        *pResult += *p;
        return 1;
    }
    double AddVector(Vector *l) {
        double sum = 0;
        Vector *alist = iVector.Create(sizeof(double));
        double d = 2;
        iVector.Add(list,&d);
        d = 3;
        iVector.Add(alist,&d);
        iList.Apply(alist,Callback,&sum);
        // Here sum should be 5.
        return sum;
    }
```

Clear

```
int (*Clear)(Vector *l);
```

Description: Erases all stored data and releases the memory associated with it. The vector header will not be destroyed, and its contents will be the same as when the array was initially created. It is an error to use this function when there are still active iterators for the container.

Returns: The result is greater than zero if successful, or an error code if an error occurs.

Errors:

CONTAINER_ERROR_BADARG The vector pointer is NULL .

CONTAINER_ERROR_READONLY The vector is read only.

Example:

```
Vector *A1;
int m = iVector.Clear(A1);
```

Contains

```
int (*Contains)(Vector *a,void *data);
```

Description: Searches the given data in the array. 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.

Errors:

CONTAINER_ERROR_BADARG Either array or data are NULL .

Returns: One if the given data is stored in the list, zero otherwise. If either daata pointer or the array pointer are NULL returns a negative error code.

Example:

```
Vector *a;  
int r = iVector.Contains(a,&data);
```

Copy

```
Vector *(*Copy)(Vector *A);
```

Description: A shallow copy of the given array 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 array's allocator.

Errors:

`CONTAINER_ERROR_NOMEMORY` There is not enough memory to complete the operation.

`CONTAINER_ERROR_BADARG` The given vector pointer is `NULL`.

Example:

```
Vector *newVector,*OldVector;  
newVector = iVector.Copy(OldVector);
```

Create

```
Vector *(*Create)(size_t element_size,size_t startsize);
```

Description: The creation function returns an empty array, initialized with all the default values. The current memory manager is used to allocate the space needed for the header. The array 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.

Returns: A pointer to a newly created List or `NULL` if an error occurs.

Errors:

`CONTAINER_ERROR_NOMEMORY` There is not enough memory to complete the operation.

`CONTAINER_ERROR_BADARG` The given element size is zero.

Any errors provoke the call the current default error function of the library since this is the creation function.

Example:

```
Vector *DArray = iVector.Create(sizeof(double),100);
```

CreateWithAllocator

```
Vector *(*CreateWithAllocator)(size_t elementsize,  
                               size_t startsize,ContainerMemoryManager *allocator);
```

Description: This function is identical to `Create` with the difference that it accepts a pointer to an allocator object. Actually, `Create` can be written as:

```
return CreateWithAllocator(elementsize,startsize,CurrentMemoryManager);
```

Contains

```
int (*Contains)(Vector *AL,void *data);
```

Description: Searches for the given data in the array. The “data” argument is supposed to point to an element at least `ElementSize` bytes. The array’s comparison function is used for determining if two elements are equal. This comparison function defaults to `memcmp`.

Returns: One if the given data is stored in the list, zero otherwise. If an error occurs, it returns a negative error code.

Errors:

`CONTAINER_ERROR_BADARG` Either list or data are NULL .

Example:

```
Vector *AL;  
int r = iVector.Contains(AL,&data);
```

CopyTo

```
void **(*CopyTo)(Vector *AL);
```

Description: Copies the whole contents of the given array list into a table of pointers to newly allocated elements, finished by a NULL pointer.

Errors:

`CONTAINER_ERROR_BADARG` The iterator pointer is NULL .

`CONTAINER_ERROR_NOMEMORY` There is not enough memory to complete the operation.

Returns: A pointer to a table of pointers or NULL if an error occurs.

deleteIterator

```
int deleteIterator(Iterator *it);
```

Description: Reclaims the memory used by the given iterator object

Returns: Integer smaller than zero with error code or a positive number when the operation completes.

Errors:

`CONTAINER_ERROR_BADARG` The iterator pointer is NULL .

Equal

```
int (*Equal)(Vector *first,Vector *second);
```

Description: Compares the given arrays. If they differ in their length, flags, or element size they compare unequal. If any of their elements differ, they compare unequal. If both first and second are NULL they compare equal.

Errors:

None

Returns: The result is one if the lists are equal, zero otherwise.

Erase

```
int (*Erase)(Vector *AL,void *data);
```

Description: Removes from the list the element that matches the given data, that is assumed to be a pointer to an element.

Returns: A negative error code if an error occurred, or a positive value that indicates that a match was found and the element was removed. If the element is not in the list the result value is `CONTAINER_ERROR_NOTFOUND`.

Errors:

`CONTAINER_ERROR_BADARG` One or both arguments are NULL.

Example:

```
double d = 2.3;
Vector *AL;
int r = iVector.Erase(AL,&d);
if (r > 0)
    printf("2.3 erased\n");
else if (r == 0)
    printf("No element with value 2.3 present\n");
else
    printf("error code %d\n",r);
```

EraseAt

```
int (*EraseAt)(Vector *AL,size_t idx);
```

Description: Removes from the array the element at the given position.

Returns: A negative error code if an error occurred or a positive value that indicates that the element was removed.

Errors:

`CONTAINER_ERROR_BADARG` The given vector pointer is NULL.

`CONTAINER_ERROR_INDEX` The given position is out of bounds.

Example:

```
Vector *AL;
int r = iVector.EraseAt(AL,2);
if (r > 0)
    printf("Element at position 2 erased\n");
```

```
else
    printf("Error code %d\n",r);
```

Finalize

```
int (*Finalize)(Vector *AL);
```

Description: Reclaims all memory used by the container, including the array header object itself.

Errors:

CONTAINER_ERROR_BADARG The given list pointer is NULL .

CONTAINER_ERROR_READONLY The container is read-only. No modifications allowed.

Returns: A positive value means the operation completed. A negative error code indicates failure.

Example:

```
Vector *AL;
int r = iVector.Finalize(AL);
if (r < 0) { /* error handling */ }
```

GetCapacity

```
size_t (*GetCapacity)(Vector *AL);
```

Description: Returns the number of elements the array can hold before it needs to reallocate its data.

Errors:

CONTAINER_ERROR_BADARG The given array is NULL .

Returns: The array capacity or zero if there was an error.

GetElementSize

```
size_t (*GetElementSize)(Vector *AL);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG The given list pointer is NULL .

Returns: The element size.

Example:

```
Vector *AL;
size_t siz = iVector.GetElementSize(AL);
```

GetElement

```
void *(*GetElement)(Vector *AL,size_t idx);
```

Description: 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.

Errors:

CONTAINER_ERROR_BADARG The given array pointer is NULL .

CONTAINER_ERROR_INDEX The given position is out of bounds.

CONTAINER_ERROR_READONLY The array is read only.

Example:

```
Vector *AL;
double *d = iList.GetElement(AL,3);
if (d == NULL) { /* Error handling */ }
```

GetFlags / SetFlags

```
unsigned (*GetFlags)(Vector *AL);
unsigned (*SetFlags)(Vector *AL,unsigned newFlags);
```

Description: GetFlags returns the state of the container flags, SetFlags sets the flags to a new value and returns the old value.

The Vector container supports the following flags:

CONTAINER_LIST_READONLY If this flag is set, no modifications to the container are allowed, and the Clear and Finalize functions will not work. The GetElement function will always return NULL . You should use the CopyElement function to access the data

GetRange

```
Vector *(*GetRange)(Vector *AL,size_t start,size_t end);
```

Description: Selects a series of consecutive elements starting at position start and ending at position end. Both the elements at start and end are included in the result. If start > end or start > list->count, an empty list is returned. If end is bigger than the number of elements in list, only elements up to the number of elements will be used. The selected elements are copied into a new list. The original list remains unchanged.

Errors:

CONTAINER_ERROR_BADARG The given list pointer or the element given are NULL

Returns: : A pointer to a new list containing the selected elements or NULL if an error occurs.

Example:

```
Vector *AL;
Vector *range = iVector.GetRange(AL,2,5);
if (range == NULL) { /* Error handling */ }
```

IndexIn

```
Vector *(*IndexIn)(Vector *SC, Vector *AL);
```

Description: Returns an array built from indexing the first argument ("SC") with the array of indexes "AL" that should be an array of `size_t` elements. The number of elements of the resulting array is equal to the number of elements of the indexes array.

Errors:

CONTAINER_ERROR_BADARG The given array pointer or the indexes array are NULL .

CONTAINER_ERROR_INDEX Any given position is out of bounds.

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

Returns: A new arraylist or NULL if an error occurs. No partial results are returned. If any index is out of bounds the whole operation fails.

IndexOf

```
int (*IndexOf)(Vector *l, void *data, void *ExtraArgs, size_t *result);
```

Description: Searches for an element in the array. If found its zero based index is returned in the pointer "result". Otherwise the result of the search is CONTAINER_ERROR_NOTFOUND . The "extraArgs" argument will be passed to the comparison function, that is used to compare elements.

Errors:

CONTAINER_ERROR_BADARG The given array pointer or the element given are NULL .

Returns: A positive number if the element is found, or a negative number containing an error code or the negative constant CONTAINER_ERROR_NOTFOUND .

Example:

```
Vector *AL;
double data = 6.8;
size_t pos;
int r = iVector.IndexOf(AL, &data, NULL, &pos);
if (r == CONTAINER_ERROR_NOTFOUND)
    printf("Not found\n");
```

InsertAt

```
int (*InsertAt)(Vector *AL, size_t idx, void *newData);
```

Description: Inserts the new element. The new element will have the given index, that can go from zero to the list count inclusive, i.e. one more than the number of elements in the list. In single linked lists the cost for this operation is proportional to `idx`.

Errors:

CONTAINER_ERROR_BADARG The given list pointer or the element given are NULL .

CONTAINER_ERROR_READONLY The list is read only.

CONTAINER_ERROR_INDEX The given position is out of bounds.

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

Returns: A positive value if the operation succeeded, or a negative error code if the operation failed.

Example:

```
double d;
Vector *AL;
int r = iVector.InsertAt(AL,2,&d);
if (r < 0) { /* Error handling */ }
else { /* Normal processing */ }
```

InsertIn

```
int (*InsertIn)(Vector *Destination, size_t pos, Vector *src);
```

Description: Inserts the array given in its third argument at the given position in the array pointed to by its first argument. The data is copied, and the source argument is not modified in any way. Both arrays must have elements of the same type. The library only tests the size of each one.

Errors:

CONTAINER_ERROR_BADARG The source or the destination lists are NULL .

CONTAINER_ERROR_READONLY The destination list is read only.

CONTAINER_ERROR_INDEX The given position is out of bounds.

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

CONTAINER_ERROR_INCOMPATIBLE The lists store elements of different size.

Returns: A positive value if the operation succeeded, or a negative error code if the operation failed.

Example:

```
#include <containers.h>
static void PrintVector(Vector *AL)
{
    size_t i;
    printf("Count %ld, Capacity %ld\n", (long)iVector.Size(AL),
          (long)iVector.GetCapacity(AL));
    for (i=0; i<iVector.Size(AL);i++) {
        printf("%g ", *(double *)iVector.GetElement(AL,i));
    }
    printf("\n");
}
```

```
static void FillVector(Vector * AL,int start)
```

```
{
    size_t i;

    for (i=0; i<10;i++) {
        double d = i+start;
        iVector.Add(AL,&d);
    }
}

int main(void)
{
    Vector *AL = iVector.Create(sizeof(double),10);
    Vector *AL1 =iVector.Create(sizeof(double),10);
    FillVector(AL,0);
    FillVector(AL1,100);
    iVector.InsertIn(AL,5,AL1);
    PrintVector(AL);
    return 0;
}
```

OUTPUT:

Count 20, Capacity 20

0 1 2 3 4 100 101 102 103 104 105 106 107 108 109 5 6 7 8 9

Load

```
Vector *(*Load)(FILE *stream,ReadFunction readFn,void *arg);
```

Description: Reads an array previously saved with the Save function from the stream pointed to by stream. If readFn is not NULL , it will be used to read each element. The “arg” argument will be passed to the read function. If the read function is NULL , this argument is ignored and a default read function is used.

Errors:

CONTAINER_ERROR_BADARG The given stream pointer is NULL .

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

Returns: A new array or NULL if the operation could not be completed. Note that the function pointers in the array are NOT saved, nor any special allocator that was in the original list. Those values will be the values by default. To rebuild the original state the user should replace the pointers again in the new array.

newIterator

```
Iterator *(*newIterator)(Vector *AL);
```

Description: Allocates and initializes a new iterator object to iterate this array.

Errors:

If no more memory is available it returns NULL .

Returns: A pointer to a new iterator or NULL if there is no more memory left.

Example:

```
Vector *AL;
Iterator *it = iVector.newIterator(AL);
double *d;
for (d=it->GetFirst(it); d != NULL; d = it->GetNext(it)) {
    double val = *d;
    // Work with the value here
}
iVector.deleteIterator(it);
```

Mismatch

```
int (*Mismatch)(const Vector *a1,const Vector *a2,
                size_t *mismatch);
```

Description: Returns the index of the first element that is different when comparing both arrays in the passed pointer *mismatch*. If one array is shorter than the other the comparison stops when the last element from the shorter array is compared. The comparison stops when the first difference is spotted.

Errors:

CONTAINER_ERROR_BADARG Any of the arguments is NULL .

Returns: If a mismatch is found the result is greater than zero and the *mismatch* argument will contain the index of the first element that compared unequal. This will be always the case for arrays of different length.

If both arrays are the same length and no differences are found the result is zero and the value pointed to by the *mismatch* argument is one more than the length of the arrays.

If an error occurs, a negative error code is returned. The *mismatch* argument contains zero.

PopBack

```
int (*PopBack)(Vector *AL,void *result);
```

Description: Copies the last element into the given result buffer and deletes the element from the container. If the result buffer is NULL , no copy is performed.

Errors:

CONTAINER_ERROR_BADARG The array is NULL .

CONTAINER_ERROR_READONLY The array is read only.

Returns: A negative value if an error occurs, zero if the array is empty or greater than zero if the operation succeeded.

ReplaceAt

```
int (*ReplaceAt)(Vector *AL, size_t idx, void *newData);
```

Description: Replaces the array element at position `idx` with the new data starting at the position pointed to by “newData” and extending `ElementSize` bytes.

Errors:

CONTAINER_ERROR_BADARG The array or the new element pointer are NULL .

CONTAINER_ERROR_READONLY The array is read only.

CONTAINER_ERROR_INDEX The given position is out of bounds.

Returns: A negative error code if an error occurs, or a positive value if the operation succeeded.

Example:

```
Vector *AL;
double d = 6.7;
int r = iVector.ReplaceAt(AL, 2, &d);
if (r < 0) { /* Error handling */ }
```

Reverse

```
int (*Reverse)(Vector *AL);
```

Description: Reverses the order of the elements of the given Vector.

Errors:

CONTAINER_ERROR_BADARG The array pointer is NULL .

CONTAINER_ERROR_READONLY The array is read only.

CONTAINER_ERROR_NOMEMORY Not enough memory for intermediate storage available

Returns: A negative error code if an error occurs, or a positive value if the operation succeeded.

Save

```
int (*Save)(Vector *AL, FILE *out, SaveFunction Fn, void *arg);
```

Description: The contents of the given list are saved into the given stream. If the save function pointer is not NULL , it will be used to save the contents of each element and will receive the `arg` argument passed to `Save`, together with the output stream. Otherwise a default save function will be used and `arg` will be ignored. The output stream must be opened for writing and must be in binary mode.

Errors:

CONTAINER_ERROR_BADARG The array pointer or the stream pointer are NULL . EOF
A disk input/output error occurred.

Returns: A positive value if the operation completed, a negative value or EOF otherwise.

Example:

```
Vector *AL;
FILE *outFile;
if (iVector.Save(AL,outFile,NULL,NULL) < 0) {
    /* Handle error here */
}
```

SetCapacity

```
int (*SetCapacity)(Vector *AL,size_t newCapacity);
```

Description: Resizes the given Vector to a new value. The new capacity means there will be that number of elements allocated, avoiding costly resizing operations when new elements are added to the Vector. If the number given is less than the number of elements present in the array, elements are discarded from the end of the array.

Errors:

CONTAINER_ERROR_BADARG The Vector pointer is NULL .

CONTAINER_ERROR_READONLY The array is read only.

Returns: A positive value if resizing was completed, a negative error code otherwise.

SetCompareFunction

```
CompareFunction (*SetCompareFunction)(Vector *AL,
                                       CompareFunction f);
```

Description: if the f argument is non NULL , it sets the array comparison function to f.

Errors:

CONTAINER_ERROR_BADARG The array pointer is NULL .

CONTAINER_ERROR_READONLY The array is read only and the function argument is not NULL .

Returns: The old value of the comparison function.

Example:

```
ErrorFunction fn,newfn;
Vector *AL;
fn = iVector.SetErrorFunction(AL,newfn);
```

SetDestructor

```
DestructorFunction (*SetDestructor)(Vector *v,DestructorFunction fn);
```

Description: Sets the destructor function to its given argument. If the function argument is NULL nothing is changed and the call is interpreted as a query since the return value is the current value of the destructor function. If the vector argument is NULL , the result is NULL .

Returns: The old value of the destructor.

SetErrorFunction

```
ErrorFunction (*SetErrorFunction)(Vector *V,ErrorFunction);
```

Description: Replaces the current error function for the given list with the new error function if different from NULL .

Errors:

CONTAINER_ERROR_BADARG The list pointer is NULL .

CONTAINER_ERROR_READONLY The list is read only and the function argument is not NULL .

Returns: The old value of the error function or NULL if there is an error.

Size

```
size_t (*Size)(Vector *AL);
```

Description: Returns the number of elements stored in the array.

Example:

```
Vector *AL;  
size_t elem = iVector.Size(AL);
```

Sizeof

```
size_t (*Sizeof)(Vector *AL);
```

Description: Returns the total size in bytes of the list, including the header, and all data stored in it. If the argument is NULL , the size of the header only is returned.

Returns: The number of bytes used by the list or the size of the Vector header if the argument is NULL .

Example:

```
Vector *AL;  
size_t size = iVector.Sizeof(AL);
```

Sort

```
int Sort(Vector *AL);
```

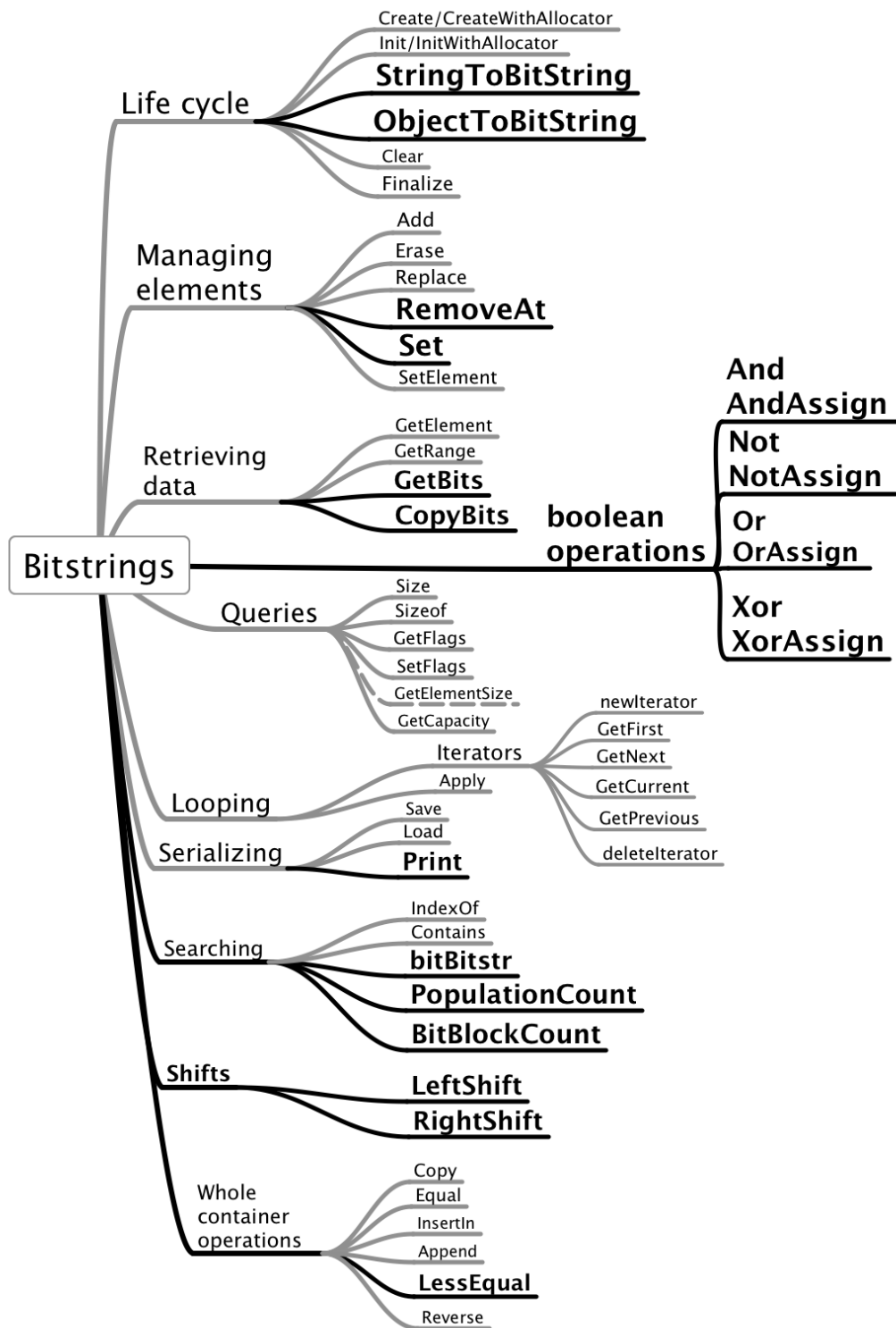

Description: Sorts the given array using the its comparison function. The order of the original array is destroyed. You should copy it if you want to preserve it.

Returns: A positive number if sorting succeeded, a negative error code if not.

Example:

```
Vector *AL;  
if (iVector.Sort(AL) < 0) { /* Error handling */ }
```

4.4 The bit-string container: iBitString



BitString vocabulary. Specific items are in bold.

A bit string is a derivation from the Vector container, specialized to hold a sequence of bits. It is a sequential container designed to save space in the storage of boolean values ¹.

4.4.1 The interface

```
typedef struct {
    int (*Add)(BitString *BitStr,int);
    BitString *(*And)(BitString *bsl,BitString *bsr);
    int (*AndAssign)(BitString *bsl,BitString *bsr);
    int (*Append)(BitString *left,BitString *right);
    int (*Apply)(BitString *B,int (*Applyfn)(int ,void *),void *);
    uintmax_t (*BitsBlockCount)(BitString *b);
    int (*Clear)(BitString *BitStr);
    int (*Contains)(BitString *B,BitString *str,void *ExtraArgs);
    BitString *(*Copy)(BitString *);
    int (*CopyBits)(BitString *b, unsigned char *buffer);
    BitString *(*Create)(size_t bitlen);
    int (*deleteIterator)(Iterator *);
    int (*Equal)(BitString *bsl,BitString *bsr);
    int (*Erase)(BitString *BitStr,bool bit);
    int (*EraseAt)(BitString *BitStr,size_t idx);
    int (*Finalize)(BitString *BitStr);
    unsigned char *(*GetBits)(BitString *b);
    size_t (*GetCapacity)(BitString *BitStr);
    int (*GetElement)(BitString *BitStr,size_t idx);
    size_t GetElementSize(BitString *b);
    unsigned (*GetFlags)(BitString *BitStr);
    BitString *(*GetRange)(BitString *b,size_t start,size_t end);
    int (*IndexOf)(BitString *B,bool SearchedBit);
    BitString *(*Init)(BitString *BitStr,size_t bitlen);
    size_t (*Insert)(BitString *BitStr,bool bit);
    size_t (*InsertAt)(BitString *BitStr,size_t idx,bool bit);
    int (*LeftShift)(BitString *bs,size_t shift);
    BitString *(*Load)(FILE *stream, ReadFunction Fn,void *arg);
    int (*LessEqual)(BitString *bsl,BitString *bsr);
    Iterator *(*newIterator)(BitString *);
    BitString *(*Not)(BitString *bsl);
    int (*NotAssign)(BitString *bsl);
    BitString *(*ObjectToBitString)(unsigned char *p,size_t size);
    BitString *(*Or)(BitString *left,BitString *right);
```

¹ The equivalent in C# is the `BitArray` class in `System.Collections`. In Java the equivalent is the `BitSet` class.

```
int (*OrAssign)(BitString *bsl, BitString *bsr);
int (*Pop)(BitString *BitStr);
uintmax_t (*PopulationCount)(BitString *b);
size_t (*Print)(BitString *b, size_t bufsiz, unsigned char *out);
int (*Push)(BitString *BitStr, int val);
int (*ReplaceAt)(BitString *BitStr, size_t idx, bool newval);
BitString (*Reverse)(BitString *b);
int (*RemoveAt)(BitString *bitStr, size_t idx);
int (*RightShift)(BitString *bs, size_t shift);
int (*Save)(BitString *B, FILE *out, SaveFunction Fn, void *arg);
int (*Set)(BitString *B, size_t start, size_t stop, bool newval);
int (*SetCapacity)(BitString *BitStr, size_t newCapacity);
int (*SetElement)(BitString *bs, size_t position, bool b);
ErrorFunction (*SetErrorFunction)(BitString *, ErrorFunction);
unsigned (*SetFlags)(BitString *BitStr, unsigned flags);
size_t (*Size)(BitString *BitStr);
size_t (*Sizeof)(BitString *b);
void (*ExtraArgs, size_t *result);
BitString (*StringToBitString)(unsigned char *);
BitString (*Xor)(BitString *bsl, BitString *bsr);
int (*XorAssign)(BitString *bsl, BitString *bsr);
} BitStringInterface;
```

4.4.2 API

Contrary to the other containers presented above like `iList` or `iVector`, `bitstring` receives and returns not pointers but values of bits. This is an important difference and makes for significant changes in the interface of many functions.

Other functions like `Apply` do not make much sense for bits and are provided just to be coherent in the overall design of the library. Obviously a function that needs a function call per bit is not very fast. The function `GetElementSize` is provided for compatibility purposes only and returns always 1. Actually it should return 0.125 assuming 8 bits bytes.

Add

```
int (*Add)(BitString *BitStr, int);
```

Description: Adds a bit at the end of the given `bitstring`.

Errors:

`CONTAINER_ERROR_BADARG` The given pointer is `NULL`.

`CONTAINER_ERROR_NOMEMORY` There is no memory to carry out the operation.

Returns: A positive number if the bit is added or a negative error code otherwise.

Example:

```
#include "containers.h"
int main(int argc, char *argv[])
{
    size_t i;
    BitString *b;
    unsigned char buf[512];

    b = iBitString.Create(32);
    for (i=0; i<32;i++)
        iBitString.Add(b,i&1);
    iBitString.Print(b,sizeof(buf),buf);
    printf("%s\n",buf);
    return 0;
}
OUTPUT:
1010 1010  1010 1010  1010 1010  1010 1010
```

And

```
BitString *(*And)(BitString *left, BitString *right);
```

Description: Makes a logical AND between the left and right arguments. The result is returned in a new bit string, both arguments are not modified. The length of the resulting bit string is the smallest length of both strings.

Returns: A pointer to the newly allocated result or NULL in case of error.

Errors:

CONTAINER_ERROR_BADARG One of both bitstring pointers are NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory is available to complete the operation.

Example:

```
#include "containers.h"
int main(int argc, char *argv[])
{
    size_t i;
    BitString *b,*c,*d;
    unsigned char buf[512];

    b = iBitString.Create(32);
    c = iBitString.Create(32);
    for (i=0; i<32;i++) {
        iBitString.Add(b,i&1);
        iBitString.Add(c,i<16);
    }
}
```

```

    iBitString.Print(b,sizeof(buf),buf);
    printf("%s\n",buf);
    printf(" AND\n");
    iBitString.Print(c,sizeof(buf),buf);
    printf("%s\n",buf);
    printf("=\n");
    d = iBitString.And(b,c);
    iBitString.Print(d,sizeof(buf),buf);
    printf("%s\n",buf);
    return 0;
}

```

OUTPUT:

```

1010 1010  1010 1010  1010 1010  1010 1010
AND
0000 0000  0000 0000  1111 1111  1111 1111
=
0000 0000  0000 0000  1010 1010  1010 1010

```

AndAssign

```
int (*AndAssign)(BitString *left, BitString *right);
```

Description: Makes a logical AND of its two arguments and assigns the result into the left bit string. If the bit strings have a different length, the operation uses the bits of the right argument until either the end of the right argument or the end of the destination string is reached.

Returns: A positive number or a negative error code in case of error.

Errors:

CONTAINER_ERROR_BADARG One or both arguments are NULL .

Example:

```

#include "containers.h"
int main(int argc, char *argv[])
{
    size_t i;
    BitString *b,*c;
    unsigned char buf[512];

    b = iBitString.Create(32);
    c = iBitString.Create(32);
    for (i=0; i<32;i++) {
        iBitString.Add(b,i&1);
        iBitString.Add(c,i<16);
    }
}

```

```
iBitString.Print(b,sizeof(buf),buf);
printf("%s\n",buf);
printf(" AND\n");
iBitString.Print(c,sizeof(buf),buf);
printf("%s\n",buf);
printf("=\n");
iBitString.AndAssign(b,c);
iBitString.Print(b,sizeof(buf),buf);
printf("%s\n",buf);
return 0;
}
```

OUTPUT:

```
1010 1010  1010 1010  1010 1010  1010 1010
AND
0000 0000  0000 0000  1111 1111  1111 1111
=
0000 0000  0000 0000  1010 1010  1010 1010
```

BitBlockCount

```
uintmax_t (*BitBlockCount)(BitString *b);
```

Description: Computes the number of blocks where 1 or more bits are set.

Returns: The number of blocks of set bits.

Errors:

CONTAINER_ERROR_BADARG The given argument is NULL .

Example:

```
#include "containers.h"
int main(int argc,char *argv[])
{
    size_t i;
    BitString *b,*c,*d;
    unsigned char buf[512];

    b = iBitString.Create(32);
    c = iBitString.Create(32);
    for (i=0; i<32;i++) {
        iBitString.Add(b,i&1);
        iBitString.Add(c,i<16);
    }
    iBitString.Print(b,sizeof(buf),buf);
    printf("%s BitBlockCount=%ld\n",buf,iBitString.BitBlockCount(b));
    iBitString.Print(c,sizeof(buf),buf);
}
```

```
    printf("%s BitBlockCount=%ld\n",buf,iBitString.BitBlockCount(c));  
    return 0;  
}
```

OUTPUT:

```
1010 1010  1010 1010  1010 1010  1010 1010 BitBlockCount=16  
0000 0000  0000 0000  1111 1111  1111 1111 BitBlockCount=1
```

CopyBits

```
int (*CopyBits)(BitString *b, unsigned char *buffer);
```

Description: Copies the bits into the given buffer. The size of the buffer is at least:

```
1+iBitstring.GetSize(bitstr)/8
```

Errors:

CONTAINER_ERROR_BADARG Either the bitstring or the buffer pointer are NULL .

Returns: A positive number if the bits are copied, a negative error code otherwise.

GetBits

```
unsigned char *(*GetBits)(BitString *b);
```

Description: Returns a pointer to the bits stored in the bitstring. If the string is read-only the result is NULL . The size of the needed buffer can be calculated according to:

```
BitString *bitstr;  
size_t bytesize;
```

```
bytesize = 1+iBitString.GetSize(bitstr)/CHAR_BIT;
```

Errors:

CONTAINER_ERROR_BADARG The bit string pointer is NULL .

CONTAINER_ERROR_READONLY The bitstring is read-only.

GetRange

```
BitString *(*GetRange)(BitString *b,size_t start,size_t end);
```

Description: Returns all the bits between the start (inclusive) and the end (inclusive) indices. If *end* is smaller than *start*, *start* and *end* are exchanged. If *end* is greater than the size of the bit string, all elements up to the last one are returned. If both *start* and *end* are out of range, an error is issued and the result is NULL .

Returns: A new bit string with the specified contents.

Errors:

CONTAINER_ERROR_BADARG The given argument is NULL .

LeftShift

```
int (*LeftShift)(BitString *bs,size_t shift);
```

Description: Shifts left the given bit string by the specified number of bits. New bits introduced by the right are zeroed.

Errors:

CONTAINER_ERROR_BADARG The bit string pointer is NULL .

Returns: An integer bigger than zero if successful, a negative error code otherwise.

Not

```
BitString *(*Not)(BitString *src);
```

Description: Makes a logical NOT of its argument. The result is returned in a new bit string. The length of the resulting bit string is the same as the length of the argument.

Returns: A pointer to the newly allocated bit string or NULL in case of error.

Errors:

CONTAINER_ERROR_BADARG The argument is NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory is available to complete the operation.

NotAssign

```
int (*NotAssign)(BitString *src);
```

Description: Makes a logical NOT of its argument and assigns the result into it.

Returns: A pointer to its argument or NULL in case of error.

Errors:

CONTAINER_ERROR_BADARG The argument is NULL .

Returns: A positive number or a negative error code in case of error.

ObjectToBitString

```
BitString *(*ObjectToBitString)(unsigned char *p,size_t size);
```

Description: The bits starting by the given pointer are copied into a new bit string using the size (in bytes) indicated by the second parameter **size**.

Errors:

CONTAINER_ERROR_BADARG The pointer is NULL

CONTAINER_ERROR_NOMEMORY There is not enough resources to finish the operation.

Returns: A new bit string or NULL if there is an error.

Or

```
BitString *(*Or)(BitString *left,BitString *right);
```

Description: Makes a logical OR between the left and right arguments. The result is returned in a new bit string, both arguments are not modified. The length of the resulting bit string is the smallest length of both strings.

Errors:

CONTAINER_ERROR_BADARG One of both bitstring pointers are NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory is available to complete the operation.

OrAssign

```
int (*OrAssign)(BitString *left, BitString *right);
```

Description: Makes a logical OR of its two arguments and assigns the result into the left bit string. If the bit strings have a different length, the operation uses the bits of the right argument until either the end of the right argument or the end of the destination string is reached.

Errors:

CONTAINER_ERROR_BADARG One or both arguments are NULL .

Returns: A positive number or a negative error code in case of error.

PopulationCount

```
uintmax_t (*PopulationCount)(BitString *b);
```

Description: Computes the number of 1 bits in the bit string.

Returns: The number of set bits in the string.

Errors:

CONTAINER_ERROR_BADARG The given argument is NULL .

Print

```
size_t (*Print)(BitString *b, size_t bufsiz, unsigned char *out);
```

Description: Prints into the given buffer the contents of the bitstring `b` without exceeding the length of the given buffer `bufsiz`. The bits will be grouped into 4 bits separated by a space. Each group of 8 bits will be separated from the rest by two spaces.

Errors:

CONTAINER_ERROR_BADARG . The bit string pointer is NULL .

Returns: The number of characters written to the output string, including the terminating zero. If the output string pointer is NULL , it returns the number of characters that would be needed to print the contents of the bitstring.

Reverse

```
BitString (*Reverse)(BitString *b);
```

Description: The bit sequence of the argument is reversed

Returns: A new bit string containing the reversed argument.

Errors:

CONTAINER_ERROR_BADARG The given argument is NULL .

Example:

```
#include "containers.h"
int main(int argc, char *argv[])
{
    size_t pos;
    BitString *b, *c;
    unsigned char buf[512];

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <bitstring>\n", argv[0]);
        return 1;
    }
    b = iBitString.StringToBitString(argv[1]);
    iBitString.Print(b, sizeof(buf), buf);
    printf("Reversing bits of %s\n", buf);
    c = iBitString.Reverse(b);
    iBitString.Print(c, sizeof(buf), buf);
    printf("%s\n", buf);
    return 0;
}
```

OUTPUT:

```
Reversing bits of 1111 1100 0000 1111 1111 1111
1111 1111 1111 0000 0011 1111
```

RemoveAt

```
int (*RemoveAt)(BitString *bitStr, size_t idx);
```

Description: Removes the bit at the specified position. If the position is greater than the length of the string the last position will be used.

Errors:

CONTAINER_ERROR_BADARG The given bit string pointer is NULL

Returns: A positive number when the bit was removed, a negative error code otherwise. If the bit string is empty the result is zero.

Example:

```
#include "containers.h"
int main(int argc, char *argv[])
{
    size_t pos;
```

```
BitString *b;
unsigned char buf[512];

if (argc < 3) {
    fprintf(stderr,"Usage: %s bitstring pos\n",argv[0]);
    return 1;
}
b = iBitString.StringToBitString(argv[1]);
pos = atoi(argv[2]);
iBitString.Print(b,sizeof(buf),buf);
printf("Erasing bit %d of %s\n",pos,buf);
iBitString.EraseAt(b,pos);
iBitString.Print(b,sizeof(buf),buf);
printf("%s\n",buf);
return 0;
}
```

OUTPUT:

```
Erasing bit 2 of 11 1000 1110 0011 1000
1 1100 0111 0001 1100
```

Set

```
int (*Set)(BitString *B,size_t start,size_t stop,bool newvalue);
```

Description: Sets the range of bits delimited by its *start* and *end* arguments to the value given by its *newvalue* argument. If the new value is different than zero a '1' bit is written, otherwise the bit is set to zero. If the *stop* argument is bigger than the length of the bitstring, the end of the string will be used.

CONTAINER_ERROR_BADARG The bit string pointer is NULL.

CONTAINER_ERROR_INDEX The *start* argument is bigger or equal to the length of the bitstring.

StringToBitString

```
BitString *(*StringToBitString)(unsigned char *);
```

Reads a bitstring from a character string. The character string should contain only the characters '1', '0', space and tab.

Errors:

CONTAINER_ERROR_BADARG The character string pointer is NULL .

Returns: A pointer to the new bitstring or NULL if there was an error or the given character string did not contain any '1' or '0'.

Xor

```
BitString *(*Xor)(BitString *left,BitString *right);
```

Description: Makes a logical XOR between the left and right arguments. The result is returned in a new bit string, both arguments are not modified. The length of the resulting bit string is the smallest length of both strings.

Returns: A pointer to its result or NULL in case of error.

Errors:

CONTAINER_ERROR_BADARG One of both bitstring pointers are NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory is available to complete the operation.

XorAssign

```
int (*XorAssign)(BitString *left, BitString *right);
```

Description: Makes a logical XOR of its two arguments and assigns the result into the left bit string. If the bit strings have a different length, the operation uses the bits of the right argument until either the end of the right argument or the end of the destination string is reached.

Returns: A positive number or a negative error code in case of error.

Errors:

CONTAINER_ERROR_BADARG Its argument is NULL .

4.5 The string collection container: iStringCollection

A string collection is a derivation from the Vector container, specialized to hold character strings.

4.5.1 The interface

```
typedef struct {
    int (*Add)(StringCollection *SC,char *newval);
    int (*AddRange)(StringCollection *SC,char **newvalues);
    int (*Apply)(StringCollection *SC,
                 int (*Applyfn)(char *,void * ExtraArg),
                 void *ExtraArg);
    Vector *(*CastToArray)(StringCollection *SC);
    int (*Clear)(StringCollection *SC);
    int (*Contains)(StringCollection *SC,char *str);
    StringCollection *(*Copy)(StringCollection *SC);
    char **(*CopyTo)(StringCollection *SC);
    StringCollection *(*Create)(size_t startsize);
    StringCollection *(*CreateWithAllocator)(size_t startsize,
                                             ContainerMemoryManager *allocator);
    StringCollection *(*CreateFromFile)(unsigned char *fileName);
    int (*deleteIterator)(Iterator *);
    int (*Equal)(StringCollection *SC1,StringCollection *SC2);
    int (*Erase)(StringCollection *SC,char *);
    int (*EraseAt)(StringCollection *SC,size_t idx);
    int (*Finalize)(StringCollection *SC);
    size_t (*FindFirstText)(StringCollection *SC,char *text);
    size_t (*FindNextText)(StringCollection *SC,char *txt,
                           size_t start);
    Vector *(*FindTextPositions)(StringCollection *SC,char *text);
    Vector *(*FindTextPositions)(StringCollection *SC,char *text);
    ContainerMemoryManager *(*GetAllocator)(StringCollection *AL);
    int (*GetCapacity)(StringCollection *SC);
    char *(*GetElement)(StringCollection *SC,size_t idx);
    unsigned (*GetFlags)(StringCollection *SC);
    int (*IndexOf)(StringCollection *SC,
                  char *SearchedString,size_t *result);
    StringCollection *(*InitWithAllocator)(StringCollection *result,
                                             size_t startsize,
                                             ContainerMemoryManager *allocator);
    StringCollection *(*Init)(StringCollection *result,
                              size_t startsize);
    int (*Insert)(StringCollection *SC,char *);
```

```
int (*InsertAt)(StringCollection *SC,size_t idx,char *newval);
int (*InsertIn)(StringCollection *source, size_t idx,
                StringCollection *newData);
StringCollection *(*Load)(FILE *stream,
                          ReadFunction readFn,void *arg);
int (*Mismatch)(const Vector *a1,const Vector *a2,
                size_t *result);

Iterator *(*newIterator)(StringCollection *SC);
size_t (*PopBack)(StringCollection *SC,char *buffer,size_t buflen);
int (*PushBack)(StringCollection *SC,char *str);
int (*ReplaceAt)(StringCollection *S,size_t idx,char *newV);
int (*SetCapacity)(StringCollection *SC,size_t newCapacity);
StringCompareFn (*SetCompareFunction)(StringCollection *SC,
                                       StringCompareFn StrCmp);
ErrorFunction (*SetErrorFunction)(StringCollection *S,
                                  ErrorFunction Fn);
unsigned (*SetFlags)(StringCollection *SC,unsigned flags);
size_t (*Size)(StringCollection *SC);
size_t (*Sizeof)(StringCollection *SC);
int (*Save)(StringCollection *SC,
            FILE *stream, SaveFunction saveFn, void *arg);
int (*Sort)(StringCollection *SC);
int (*WriteToFile)(StringCollection *SC,unsigned char *fileName);
} StringCollectionInterface;

extern StringCollectionInterface iStringCollection;
```

4.5.2 API

Most of the functions present in the interface are exactly like the functions in Vector. Only those that differ will be documented here.

AddRange

```
int (*AddRange)(StringCollection *SC,size_t n,char *data[]);
```

Description: Adds each string of the array of string pointers at the end of the container. It is assumed that “data” points to a contiguous array of string pointers whose size is given by the “n” parameter. Returns a value greater than zero if the addition completed successfully, a negative error code otherwise. If n is zero nothing is done and no errors are issued, even if the array pointer or the data pointer are NULL .

Errors:

CONTAINER_ERROR_BADARG The StringCollection pointer or the data pointers are NULL .

CONTAINER_ERROR_READONLY The list is read-only. No modifications allowed.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive number if the operation completed, negative error code otherwise.

Example:

```
StringCollection *SC;
char *data[] = { "One","two","three"};
int result = iStringCollection.AddRange(SC,3,data);
if (result < 0) { /* Error handling */ }
```

CastToArray

```
Vector *(*CastToArray)(StringCollection *SC);
```

Description: Converts a string collection into an vector.

Errors:

CONTAINER_ERROR_BADARG The StringCollection pointer is NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive number if the operation completed, negative error code otherwise.

CreateFromFile

```
StringCollection *(*CreateFromFile)(unsigned char *fileName);
```

Description: Reads the given text file and stores each line in a string of the collection. The end of line characters are discarded.

Errors:

CONTAINER_ERROR_BADARG The fileName pointer is NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

CONTAINER_ERROR_NOENT The file doesn't exist or can't be opened for reading.

Returns: A pointer to a new string collection with the contents of the file. If an error occurs the result is NULL and the current error function (in the iError interface) is called.

FindFirstText

```
size_t (*FindFirstText)(StringCollection *SC,char *text);
```

Description: Finds the first occurrence of the given text in the string collection.

Errors:

CONTAINER_ERROR_BADARG One or both arguments are NULL .

Returns: The zero based index of the line that contains the given text or the constant **CONTAINER_ERROR_NOTFOUND** if the text is not found.

FindNextText

```
size_t (*FindNextText)(StringCollection *SC,char *txt,size_t start);
```


Description: Starts searching for the given text at the specified line.

Errors:

CONTAINER_ERROR_BADARG The StringCollection or the text pointer are NULL .

Returns: The one based index of the line that contains the text or zero if the text is not found or an error occurred.

FindTextPositions

```
Vector *(*FindTextPositions)(StringCollection *SC,char *text);
```

Description: Searches all occurrences of the given text in the given string collection.

Errors:

CONTAINER_ERROR_BADARG The StringCollection or the text pointer are NULL .

CONTAINER_ERROR_NOMEMORY Not enough storage for holding the result array list.

Returns: An array list containing a pair of integers for each occurrence containing the zero based position of the line where the text was found and a second number indicating the character index within the line where the searched text occurs. The result is NULL if there wasn't any occurrences of the searched text in the string collection or an error was detected.

Init

```
StringCollection *(*Init)(StringCollection *result, size_t startsize);
```

Description: Initializes the given string collection to contain at least the number of strings given. Uses the current memory manager.

Errors:

CONTAINER_ERROR_NOMEMORY There is no more memory left to complete the operation.

CONTAINER_ERROR_BADARG The string collection pointer is NULL

Returns: A pointer to the initialized string collection or NULL if an error occurs.

InitWithAllocator

```
StringCollection *(*InitWithAllocator)(StringCollection *result,  
                                       size_t startsize,  
                                       ContainerMemoryManager *allocator);
```

Description: Initializes the given string collection to contain at least the number of strings given. Uses the given memory manager.

Errors:

CONTAINER_ERROR_NOMEMORY There is no more memory left to complete the operation.

CONTAINER_ERROR_BADARG The string collection pointer is NULL

Returns: A pointer to the initialized string collection or NULL if an error occurs.

InsertIn

```
int (*InsertIn)(StringCollection *dst, size_t pos,
                StringCollection *newData);
```

Description: Inserts the given StringCollection into the destination StringCollection at the given position. If the position is greater than the actual length of the string collection the new data will be inserted at the end.

Errors:

CONTAINER_ERROR_BADARG The source or destination pointers are NULL .

CONTAINER_ERROR_READONLY The destination is read only.

Example:

```
#include <containers.h>
static void PrintStringCollection(StringCollection *AL)
{
    size_t i;
    printf("Count %ld, Capacity %ld\n",
           (long)iStringCollection.Size(AL),
           (long)iStringCollection.GetCapacity(AL));
    for (i=0; i<iStringCollection.Size(AL);i++) {
        printf("%s ",iStringCollection.GetElement(AL,i));
    }
    printf("\n");
}

static void FillStringCollection(StringCollection * AL,int start)
{
    size_t i;
    char buf[256];

    for (i=0; i<10;i++) {
        double d = i+start;
        sprintf(buf,"%g",d);
        iStringCollection.Add(AL,buf);
    }
}

int main(void)
{
    StringCollection *AL = iStringCollection.Create(10);
    StringCollection *AL1 =iStringCollection.Create(10);
    FillStringCollection(AL,0);
    FillStringCollection(AL1,100);
    iStringCollection.InsertIn(AL,5,AL1);
    PrintStringCollection(AL);
    return 0;
}
```

The example creates two string collections, fills them with the string representation of the numbers from 0 to 9 and from 100 to 109, then inserts the second collection into the first one at position 5.

OUTPUT:

Count 20, Capacity 20

0 1 2 3 4 100 101 102 103 104 105 106 107 108 109 5 6 7 8 9

Mismatch

```
int (*Mismatch)(const StringCollection *a1,
                const StringCollection *a2,
                size_t *mismatch);
```

Description: Returns the index of the first element that is different when comparing both collections in the passed pointer *mismatch*. If one is shorter than the other the comparison stops when the last element from the shorter array is compared. The comparison also stops when the first difference is spotted.

Errors:

CONTAINER_ERROR_BADARG Any of the arguments is NULL .

Returns: If a mismatch is found the result is greater than zero and the *mismatch* argument will contain the index of the first element that compared unequal. This will be always the case for arrays of different length.

If both arrays are the same length and no differences are found the result is zero and the value pointed to by the *mismatch* argument is one more than the length of the arrays.

If an error occurs, a negative error code is returned. The *mismatch* argument contains zero.

Example:

```
#include "containers.h"
char *table[] = {"String 1", "String 2","String 3","String 4",};

int main(void)
{
    size_t idx;
    StringCollection *sc = iStringCollection.Create(4);
    StringCollection *sc2;
    iStringCollection.AddRange(sc,sizeof(table)/sizeof(table[0]),table);
    sc2 = iStringCollection.Copy(sc);
    iStringCollection.ReplaceAt(sc,2,"String456");
    iStringCollection.Mismatch(sc,sc2,&idx);
    printf("String collections differ at position %d\n",idx);
}
```

OUTPUT:

String collections differ at position 2

PopBack

```
size_t (*PopBack)(StringCollection *SC,char *buffer,size_t buflen);
```

Description: If the string collection is not empty, it will copy at most buflen characters into the given buffer. If the buffer pointer is NULL or the length of the buffer is zero it will return the length of the element that would be popped.

Errors:

CONTAINER_ERROR_BADARG The StringCollection pointer is NULL .

Returns: Zero if there was an error or the string collection is empty. Otherwise returns the length of the string stored at the position to pop, including the terminating zero.

WriteToFile

```
int (*WriteToFile)(StringCollection *SC,unsigned char *fileName);
```

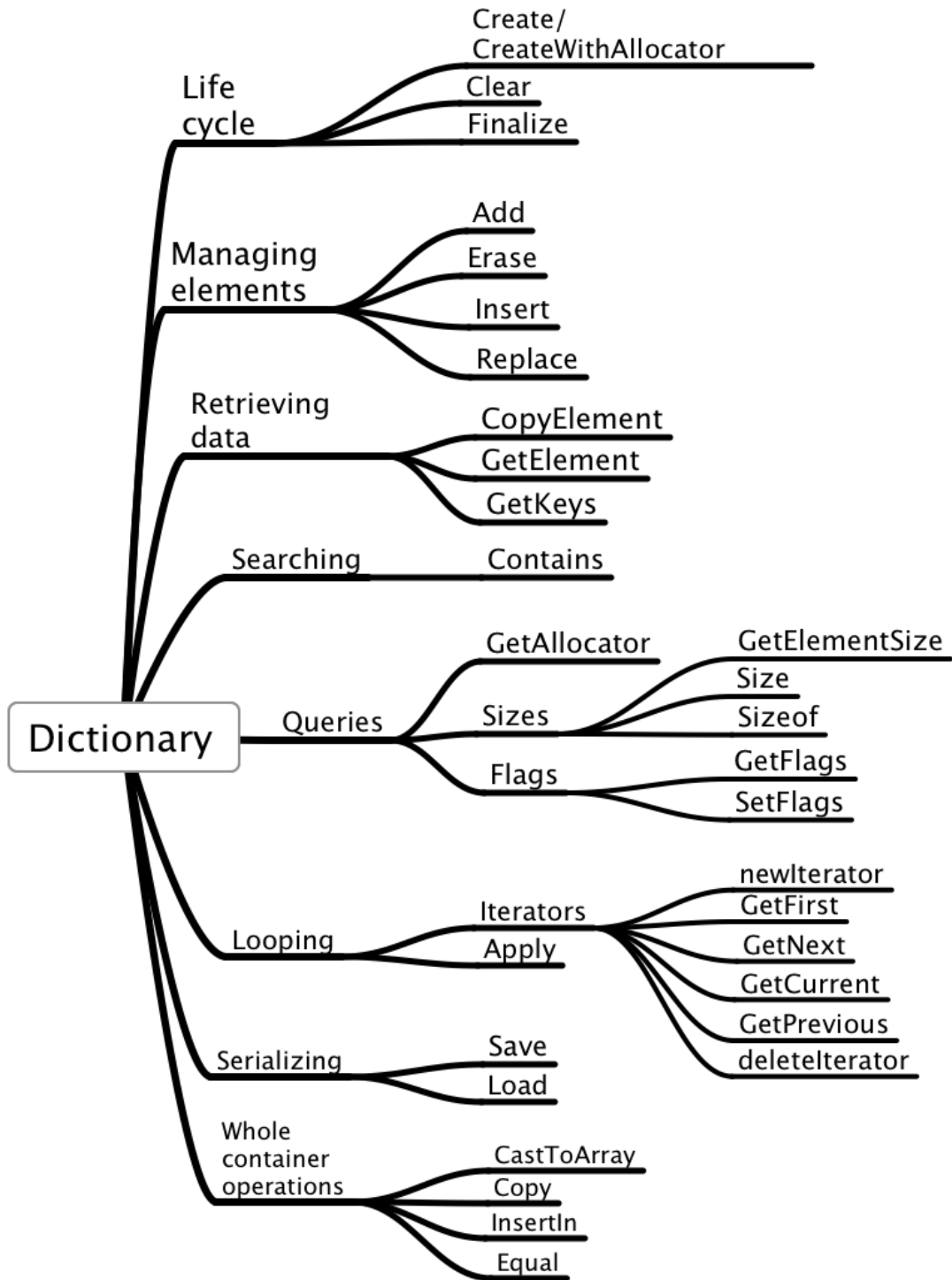
Description: Writes the contents of the given string collection into a file with the given name. If the collection is empty an empty file is created. The resulting file contains a line for each string in the collection.

Errors:

CONTAINER_ERROR_BADARG The StringCollection pointer or the fileName are NULL .

Returns: A positive number if the operation completes, or a negative error code otherwise. If the collection is empty the result is zero.

4.6 The dictionary container: iDictionary



The dictionary vocabulary.

A dictionary is an associative container that associates a text key with a piece of data. It can be implemented by means of a hash table that uses a hash function to map the key into a restricted integer range, used in a table.

4.6.1 The dictionary interface

```
typedef struct _Dictionary Dictionary;

typedef struct {
    int (*Add)(Dictionary *Dict,
               const unsigned char *key,void *Data);
    int (*Apply)(Dictionary *Dict,
                 int (*Applyfn)(const unsigned char *Key,
                                const void *data,void *arg),
                 void *arg);
    Vector *(*CastToArray)(Dictionary *);
    int (*Clear)(Dictionary *Dict);
    Dictionary *(*Copy)(Dictionary *dict);
    Dictionary *(*Create)(size_t ElementSize,size_t hint);
    Dictionary *(*CreateWithAllocator)(size_t elementsize,size_t hint,
                                       ContainerMemoryManager *allocator);
    int (*deleteIterator)(Iterator *);
    int (*Equal)(Dictionary *dict1,Dictionary *dict2);
    int (*Erase)(Dictionary *Dict,const unsigned char *);
    int (*Finalize)(Dictionary *Dict);
    size_t (*GetElementSize)(Dictionary *d);
    ContainerMemoryManager *(*GetAllocator)(Dictionary *Dict);
    const void *(*GetElement)(const Dictionary *Dict,
                              const unsigned char *Key);
    unsigned (*GetFlags)(Dictionary *Dict);
    StringCollection *(*GetKeys)(Dictionary *Dict);
    Dictionary *(*Init)(Dictionary *Dict,size_t elemsize,size_t hint);
    Dictionary *(*InitWithAllocator)(Dictionary *Dict,
                                     size_t elementsize, size_t hint,
                                     ContainerMemoryManager *allocator);
    int (*Insert)(Dictionary *Dict,const unsigned char *Key,void *Value);
    int (*InsertIn)(Dictionary *dst,Dictionary *src);
    Dictionary * (*Load)(FILE *stream, ReadFunction readFn, void *arg);
    Iterator *(*newIterator)(Dictionary *dict);
    int (*Save)(Dictionary *Dict, FILE *stream,
                SaveFunction Fn, void *arg);
    DestructorFunction (*SetDestructor)(Dictionary *dict,
                                       DestructorFunction fn);
    ErrorFunction (*SetErrorFunction)(Dictionary *Dict,ErrorFunction f);
```

```
    unsigned (*SetFlags)(Dictionary *Dict,unsigned flags);
    size_t (*Size)(Dictionary *Dict);
    size_t (*Sizeof)(Dictionary *dict);
} DictionaryInterface;
```

4.6.2 The API

Add ---

```
int (*Add)(Dictionary *Dict,char *key,void *data);
```

Description: Adds the given element to the container using the given “key” string. It is assumed that “data” points to a contiguous memory area of at least ElementSize bytes. Both the key and the data are copied into the container.

If an element exists with the given key, its contents are replaced with the new data. For a different behavior use **Insert** or **Replace**.

Errors:

CONTAINER_ERROR_BADARG The dictionary, the key or the data pointers are NULL .

CONTAINER_ERROR_READONLY The dictionary is read-only. No modifications allowed.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive number if the operation added a new element, zero if the data was written into an existing element, or a negative error code if an error occurred.

Example:

```
Dictionary *dict;
double data = 4.5;
int result = iDictionary.Add(dict,"Interest rate",&data);
if (result < 0) { /* Error handling */ }
```

Apply ---

```
int (*Apply)(Dictionary *Dict,
             int (Applyfn)(const unsigned char *key,
                           void *data,
                           void *extraArg),
             void *extraArg);
```

Description: Will call the given function for each element of the array. The first argument of the callback function receives the key, the second is a pointer to the element of the Dictionary. The third argument of the callback is the “extraArg” 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 function type cast. If the dictionary is read-only, a copy of the element will be passed to the callback function.

Errors:

CONTAINER_ERROR_BADARG Either the dictionary pointer or Applyfn are NULL .

CONTAINER_ERROR_NOMEMORY The dictionary is read-only and there is no more memory to allocate the buffer to copy each element.

Example:

```
static int print(const char *key,
                void *pElement,
                void *pResult)
{
    double *p = pElement;
    printf("%s: %g\n",key,*p);
    return 1;
}
int main(void) {
    Dictionary *dict = iDictionary.Create(sizeof(double),5);
    double d = 2;
    iDictionary.Add(dict,"First item",&d);
    d = 3;
    iDictionary.Add(dict,"Second item",&d);
    iDictionary.Apply(dict,print,NULL);
    return 0;
}
```

Output should be:

First item: 2
Second item: 3

Clear

```
int (*Clear)(Dictionary *dict);
```

Description: Erases all stored data and releases the memory associated with it. The dictionary header is not destroyed, and its contents will be the same as when it was initially created. It is an error to use this function when there are still active iterators for the container.

Returns: The result is greater than zero if successful, or an error code if an error occurs.

Errors:

CONTAINER_ERROR_BADARG The vector pointer is NULL .

CONTAINER_ERROR_READONLY The vector is read only.

Example:

```
Dictionary *Dict;
int m = iDictionary.Clear(Dict);
```


Contains

```
int (*Contains)(Dictionary *Dict,const unsigned char *Key);
```

Description: Returns one if the given key is stored in the dictionary, zero otherwise. If an error occurs it returns a negative error code.

Errors:

CONTAINER_ERROR_BADARG Either Dict or Key are NULL .

Example:

```
Dictionary *dict;  
int r = iDictionary.Contains(dict,"Item 1");
```

Copy

```
Dictionary *(*Copy)(Dictionary *Dict);
```

Description: A shallow copy of the given dictionary 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.

Errors:

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

CONTAINER_ERROR_BADARG The given vector pointer is NULL .

Example:

```
Dictionary *newDict,*Old;  
newDict = iDictionary.Copy(Old);
```

Create

```
Dictionary *(*Create)(size_t ElementSize,size_t hint);  
Dictionary *(*CreateWithAllocator)(size_t elementsize,size_t hint,  
                                   ContainerMemoryManager *allocator);
```

Description: Creates a new dictionary with the given element size and with a table big enough to store hint entries. The Create function uses the current memory manager as the allocator for the new dictionary. CreateWithAllocator uses the given allocator object.

Errors:

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A pointer to the new dictionary or NULL if there is not enough memory to create it.

deleteIterator

```
int deleteIterator(Iterator *it);
```

Description: Reclaims the memory used by the given iterator object

Returns: Integer smaller than zero with error code or a positive number when the operation completes.

Errors:

CONTAINER_ERROR_BADARG The iterator pointer is NULL .

Equal

```
int (*Equal)(Dictionary *d1,Dictionary *d2);
```

Description: Compares the given dictionaries using their comparison function. If the dictionaries differ in their size, flags, or hash functions they compare unequal. If any of their elements differ, they compare unequal. If both d1 and d2 are NULL they compare equal. If Both d1 and d2 are empty they compare equal.

Errors:

None

Returns: The result is one if the dictionaries are equal, zero otherwise.

Erase

```
int (*Erase)(Dictionary *Dict,const char *key);
```

Description: Removes from the dictionary the element that matches the given key.

Returns: A positive value that indicates that a match was found and the element was removed. If no element matched the result is CONTAINER_ERROR_NOTFOUND . If an error occurs, a negative error code is returned.

Errors:

CONTAINER_ERROR_BADARG One or both arguments are NULL .

Example:

```
double d = 2.3;
Vector *AL;
int r = iVector.Erase(AL,&d);
if (r > 0)
    printf("2.3 erased\n");
else if (r == 0)
    printf("No element with value 2.3 present\n");
else
    printf("error code %d\n",r);
```

Finalize

```
int (*Finalize)(Dictionary *dict);
```

Description: Reclaims all memory used by the dictionary, including the array header object itself.

Errors:

CONTAINER_ERROR_BADARG The given pointer is NULL .

CONTAINER_ERROR_READONLY The dictionary is read-only. No modifications allowed.

Returns: A positive value means the operation completed. A negative error code indicates failure.

Example:

```
Dictionary *AL;
int r = iDictionary.Finalize(AL);
if (r < 0) { /* error handling */ }
```

GetElementSize

```
size_t (*GetElementSize)(Dictionary *Dict);
```

Description: Retrieves the size of the elements stored in the given dictionary. Note that this value can be different than the value given to the creation function because of alignment requirements.

Errors:

CONTAINER_ERROR_BADARG The given pointer is NULL .

Returns: The element size or zero if an error.

Example:

```
Dictionary *Dict;
size_t siz = iDictionary.GetElementSize(Dict);
```

GetElement

```
void *(*GetElement)(Dictionary *Dict, const unsigned char *key);
```

Description:

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 dictionary is read only.

Use the CopyElement function to get a read/write copy of an element of the dictionary.

Errors:

CONTAINER_ERROR_BADARG The given array pointer or the key are NULL .

CONTAINER_ERROR_READONLY The array is read only.

Example:

```
Dictionary *Dict;
double *d = iDictionary.GetElement(Dict, "Index");
if (d == NULL) { /* Error handling */ }
```

Init

```
Dictionary *(*Init)(Dictionary *Dict, size_t elementszize, size_t hint);
```

Description: Initializes the indicated storage for use as a dictionary object. This procedure is completely equivalent to **Create** with the difference that there is no allocation done for the dictionary header. Uses the current memory manager for the allocations of the slot table.

Returns: A pointer to its first argument if successful or NULL if there is no memory to complete the operation.

InitWithAllocator

```
Dictionary *(*InitWithAllocator)(Dictionary *Dict,  
                                size_t elementszize, size_t hint,  
                                ContainerMemoryManager *allocator);
```

Description: Initializes the indicated storage for use as a dictionary object. This procedure is completely equivalent to **CreateWithAllocator** with the difference that there is no allocation done for the dictionary header. Uses the given memory manager for the allocations of the slot table.

Returns: A pointer to its first argument if successful or NULL if there is no memory to complete the operation.

Insert

```
int (*Insert)(Dictionary *Dict, const unsigned char *key, void *Data);
```

Description: Inserts the new key and its corresponding data into the given dictionary. If the key is already present, nothing is changed. This contrasts with the behavior of **Add** that will replace an existing key.

Errors:

CONTAINER_ERROR_BADARG Any of the given pointers is NULL .

CONTAINER_ERROR_READONLY The array is read only.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive value if the key was inserted, zero if the key was already present, or a negative error code.

Load

```
Dictionary *(*Load)(FILE *stream, ReadFunction readFn, void *arg);
```

Description: Reads a dictionary previously saved with the **Save** function from the stream pointed to by stream. If readFn is not NULL , it will be used to read each element. The “arg” argument will be passed to the read function. If the read function is NULL , this argument is ignored and a default read function is used.

Errors:

CONTAINER_ERROR_BADARG The given stream pointer is NULL .

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

Returns: A new dictionary or NULL if the operation could not be completed. Note that the function pointers in the array are NOT saved, nor any special allocator that was in the original dictionary. Those values will be the values by default. To rebuild the original state the user should replace the pointers again in the new array.

newIterator

```
Iterator *(*newIterator)(Dictionary *Dict);
```

Description: Allocates and initializes a new iterator object to iterate this dictionary. The exact sequence is implementation defined but it will be the same for the same dictionary with the same number of elements.

Errors:

If no more memory is available it returns NULL .

Returns: A pointer to a new iterator or NULL if there is no more memory left.

Example:

```
Dictionary *Dict;
Iterator *it = iDictionary.newIterator(Dict);
double *d;
for (d=it->GetFirst(it); d != NULL; d = it->GetNext(it)) {
    double val = *d;
    // Work with the value here
}
iDictionary.deleteIterator(it);
```

SetDestructor

```
DestructorFunction SetDestructor(Dictionary *d,DestructorFunction fn);
```

Description: Sets the destructor function to its given argument. If the function argument is NULL nothing is changed and the call is interpreted as a query since the return value is the current value of the destructor function. If the dictionary argument is NULL , the result is NULL .

Returns: The old value of the destructor.

Size

```
size_t (*Size)(Dictionary *Dict);
```

Description: Returns the number of elements stored in the dictionary or SIZE_MAX if the dictionary pointer is NULL .

Errors:

CONTAINER_ERROR_BADARG The given array pointer or the key are NULL .

Example:

```
Dictionary *Dict;
size_t elem = iDictionary.Size(Dict);
```

Save

```
int (*Save)(Dictionary *D, FILE *out, SaveFunction Fn, void *arg);
```

Description: The contents of the given dictionary are saved into the given stream. If the save function pointer is not NULL , it will be used to save the contents of each element and will receive the arg argument passed to Save, together with the output stream. Otherwise a default save function will be used and arg will be ignored. The output stream must be opened for writing and must be in binary mode.

Errors:

CONTAINER_ERROR_BADARG The dictionary pointer or the stream pointer are NULL .
EOF A disk input/output error occurred.

Returns: A positive value if the operation completed, a negative value or EOF otherwise.

Example:

```
Dictionary *Dict;
FILE *outFile;
if (iDictionary.Save(Dict,outFile,NULL,NULL) < 0) {
    /* Handle error here */
}
```

Sizeof

```
size_t (*Sizeof)(Dictionary *Dict);
```

Description: Returns the total size in bytes of the dictionary, including the header, and all data stored in the dictionary, including the size of the dictionary header. If Dict is NULL , the result is the size of the Dictionary structure.

Returns: The number of bytes used by the dictionary or the size of the Dictionary structure if the argument is NULL .

Example:

```
Dictionary *Dict;
size_t size = iDictionary.Sizeof(Dict);
```

SetErrorFunction

```
ErrorFunction (*SetErrorFunction)(Dictionary *dict,ErrorFunction efn);
```

Description: Replaces the current error function for the given dictionary with the new error function if different from NULL .

Errors:

CONTAINER_ERROR_BADARG The dictionary pointer is NULL .

CONTAINER_ERROR_READONLY The dictionary is read only and the function argument is not NULL .

Returns: The old value of the error function or NULL if there is an error.

Size

```
size_t (*Size)(Dictionary *d);
```

Description: Returns the number of elements stored in the dictionary. If the argument is NULL the result is zero.

Example:

```
Dictionary *d;  
size_t elem = iDictionary.Size(d);
```

4.7 The TreeMap interface: iTreeMap

The tree map container uses a tree to associate keys to values. Trees are extremely efficient data structures that allow access to millions of items with a few comparisons. Disadvantages include a greater overhead than other containers, and a complex machinery to maintain them.

This associative container is special in that it contains no separate key, the elements themselves are the key. Obviously they need imperatively a comparison function, and that comparison function could use some parts of the stored object as a key, but that is transparent to the interface.

An essential point in this container is the comparison function. Since all insertions searches and deletions from/to the tree are done using that function, it is essential that is defined correctly. Like all other comparison functions it can receive an extra argument that conveys some kind of context to it. This implies that functions like 'Add' have an extra argument to be able to pass this context to the comparison function.

The comparison function must be consistent

It is important to stress that for this container it is **essential** that the comparison function returns always the **same** result for two given elements. The context passed through this auxiliary arguments must not be used to change the result of the element comparison according to some external factor. Any inconsistency in the comparison function will destroy completely the whole container and the user will be unable to retrieve the data stored or (worst) retrieve the wrong data.

4.7.1 The interface

```
typedef struct tagTreeMapInterface {
    int (*Add)(TreeMap *ST, void *Data,void *ExtraArgs);
    int (*Apply)(TreeMap *ST,
                 int (*Applyfn)(const void *data,void *arg),
                 void *arg);
    TreeMap *(*Copy)(TreeMap *src);
    TreeMap *(*CreateWithAllocator)(size_t ElementSize,
                                    ContainerMemoryManager *m);
    TreeMap *(*Create)(size_t ElementSize);
    unsigned (*GetFlags)(TreeMap *ST);
    int (*Clear)(TreeMap *ST);
    int (*Contains)(TreeMap *ST,void *element,void *ExtraArgs);
    int (*deleteIterator)(Iterator *);
    int (*Erase)(TreeMap *tree, void *element,void *ExtraArgs);
    int (*Equal)(TreeMap *t1, TreeMap *t2);
    int (*Finalize)(TreeMap *ST);
    void *(*Find)(TreeMap *tree,void *element,void *ExtraArgs);
```



```
size_t (*GetElementSize)(TreeMap *d);
int (*Insert)(TreeMap *RB, const void *Data, void *ExtraArgs);
Iterator *(*newIterator)(TreeMap *);
TreeMap *(*Load)(FILE *stream, ReadFunction loadFn,void *arg);
int (*Save)(TreeMap *src,FILE *stream,
            SaveFunction saveFn,void *arg);
CompareFunction (*SetCompareFunction)(TreeMap *ST,
                                      CompareFunction fn);
DestructorFunction (*SetDestructor)(TreeMap *Tree,
                                    DestructorFunction fn);
ErrorFunction (*SetErrorFunction)(TreeMap *ST, ErrorFunction fn);
unsigned (*SetFlags)(TreeMap *ST, unsigned flags);
size_t (*Sizeof)(TreeMap *ST);
size_t (*Size)(TreeMap *ST);
} TreeMapInterface;
```

All the above functions were described for the sequential containers and their syntax is here the same.

4.8 Hash Table: iHashTable

Hash table is a similar container as dictionary, but allows for more features at the expense of a slightly more complicated interface. Keys aren't restricted to zero terminated strings but can be any kind of data. The table resizes itself as it grows. Merging two hash tables

4.8.1 The interface

```
typedef struct {
    int (*Add)(HashTable *HT,const void *key,
               size_t keyLength,const void *Data);
    int (*Apply)(HashTable *HT,
                 int (*ApplyFn)(void *Key,
                                size_t keyLength,
                                void *data,
                                void *ExtraArg),
                 void *ExtraArg);
    int (*Clear)(HashTable *HT);
    HashTable *(*Copy)(const HashTable *Orig,Pool *pool);
    HashTable *(*Create)(size_t ElementSize);
    int (*deleteIterator)(Iterator *);
    int (*Erase)(HashTable *HT,void *key,size_t klen);
    int (*Finalize)(HashTable *HT);
    void *(*GetElement)(const HashTable *HT,
                        const void *Key ,size_t keyLength);
    unsigned (*GetFlags)(const HashTable *HT);
    HashTable *(*Load)(FILE *stream, ReadFunction readFn, void *arg);
    HashTable *(*Merge)(Pool *p,
                        const HashTable *overlay,
                        const HashTable *base,
                        void * (*merger)(Pool *p,
                                         const void *key,
                                         size_t keyLength,
                                         const void *h1_val,
                                         const void *h2_val,
                                         const void *data),
                        const void *data);
    Iterator *(*newIterator)(HashTable *);
    HashTable *(*Overlay)(Pool *p,
                          const HashTable *overlay,
                          const HashTable *base);
    int (*Resize)(HashTable *HT,size_t newSize);
    int (*Replace)(HashTable *HT,
                  const void *key,
```

```
        size_t keyLength,const void *data);
int (*Save)(HashTable *HT,
        FILE *stream, SaveFunction saveFn,void *arg);
int (*Search)(HashTable *ht,
        int (*Comparefn)(void *rec,
                        const void *key,
                        size_t keyLength,
                        const void *value),
        void *rec);
ErrorFunction (*SetErrorFunction)(HashTable *HT,ErrorFunction fn);
unsigned (*SetFlags)(HashTable *HT,unsigned flags);
HashFunction (*SetHashFunction)(HashTable *ht, HashFunction hf);
size_t (*Size)(const HashTable *HT);
size_t (*Sizeof)(const HashTable *HT);
} HashTableInterface;
extern HashTableInterface iHashTable;
```

4.8.2 The API

Add

```
int (*Add)(HashTable *ht,
        void *key,
        size_t keyLength,
        const void *data);
```

Description: Adds the given element to the container using the given “key” string. It is assumed that “data” points to a contiguous memory area of at least `ht->ElementSize` bytes. Both the key and the data are copied into the container.

If an element exists with the given key, its contents are replaced with the new data.

Errors:

`CONTAINER_ERROR_BADARG` The hash table, the key or the data pointers are NULL .
`CONTAINER_ERROR_READONLY` : The hash table is read-only. No modifications allowed.
`CONTAINER_ERROR_NOMEMORY` Not enough memory to complete the operation.

Returns: A positive number if the operation added a new element, zero if the data was written into an existing element, or a negative error code if an error occurred.

Example:

```
HashTable *ht;
double data = 4.5;
int result = iHashTable.Add(ht,"Interest rate",
                        strlen(\Interest rate",&data);
if (result < 0) { /* Error handling */ }
```

Apply

```
int (*Apply)(HashTable *ht,
              int (Applyfn)(const unsigned char *key,
                             size_t keyLength,
                             void *data,
                             void *extraArg),
              void *extraArg);
```

Description: Apply will call the given function for each element of the array. The first argument of the callback function receives the key, the second is the length of the key. The third is a pointer to one element of the table. The fourth argument of the callback is the “extraArg” 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 function type cast.

If the dictionary is read-only, a copy of the element will be passed to the callback function.

Errors:

CONTAINER_ERROR_BADARG Either the hash table pointer or Applyfn are NULL .

CONTAINER_ERROR_NOMEMORY The hash table is read-only and there is no more memory to allocate the buffer to copy each element.

Example:

```
static int print(const char *key,
                 void *pElement,
                 void *pResult)
{
    double *p = pElement;
    printf("%s: %g\n",key,*p);
    return 1;
}

int main(void) {
    Dictionary *dict = iDictionary.Create(sizeof(double),5);
    double d = 2;
    iDictionary.Add(dict,"First item",&d);
    d = 3;
    iDictionary.Add(dict,"Second item",&d);
    iDictionary.Apply(dict,print,NULL);
    return 0;
}
```

Output should be:

First item: 2

Second item: 3

Clear

```
int (*Clear)(HashTable *ht);
```

Description: Erases all stored data and releases the memory associated with it. The hash table header is not destroyed, and its contents will be the same as it was when initially created. It is an error to use this function when there are still active iterators for the container.

Returns: The result is greater than zero if successful, or an error code if an error occurs.

Errors:

CONTAINER_ERROR_BADARG The hash table pointer is NULL .

CONTAINER_ERROR_READONLY The hash table is read only.

Example:

```
HashTable *ht;  
int m = iHashTable.Clear(ht);
```

Copy

```
HashTable *(*Copy)(const HashTable *Orig, Pool *pool);
```

Description: Copies the given hash table using the given pool. If “pool” is NULL ,the pool of the given hash table will be used.

Errors:

CONTAINER_ERROR_BADARG The hash table pointer is NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Create

```
HashTable *(*Create)(size_t ElementSize);
```

Description: Creates a new hash table and initializes all fields. The table will use the current memory manager for its pool.

Errors:

CONTAINER_ERROR_BADARG The parameter is zero or bigger than the maximum size the implementation supports.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

deleteIterator

```
int (*deleteIterator)(Iterator *);
```

Description: Releases the memory used by the given iterator.

Errors:

CONTAINER_ERROR_BADARG The parameter is NULL .

Returns: A positive value if successful or a negative error code.

Erase

```
int (*Erase)(HashTable *HT,void *key,size_t keyLength);
```

Description: Removes from the hash table the element with the given key.

Errors:

CONTAINER_ERROR_BADARG The hash table parameter or the key pointer are NULL , or the keyLength is zero.

Returns: A positive number if the operation completed, a negative error code otherwise.

Finalize Synopsis: int (*Finalize)(HashTable *HT); Description: Releases all memory used by the hash table and destroys the hash table header itself.

Errors:

CONTAINER_ERROR_BADARG The parameter is NULL .

GetElement

```
void *(*GetElement)(const HashTable *H,const void *Key,size_t keyLen);
```

Description: Returns a pointer to the given hash table element.

Errors:

CONTAINER_ERROR_BADARG The hash table parameter or the key pointer are NULL , or the keyLen parameter is zero.

Returns: A pointer to the element or NULL if no element with the specified key exists.

GetFlags

```
unsigned (*GetFlags)(const HashTable *HT);
```

Description: Returns an unsigned integer with the state of the table.

Load

```
HashTable *(*Load)(FILE *stream,ReadFunction readFn,void *arg);
```

Description: Reads a table previously saved with the Save function from the stream pointed to by stream. If readFn is not NULL , it will be used to read each element. The “arg” argument will be passed to the read function. If the read function is NULL , this argument is ignored and a default read function is used.

Errors:

CONTAINER_ERROR_BADARG The given stream pointer is NULL .

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

Returns: A new table or NULL if the operation could not be completed. Note that the function pointers in the array are NOT saved in most implementations, nor any special allocator that was in the original table. In most implementations those values will be the values by default. To rebuild the original state the user should replace the pointers again in the new table.

Merge

```
HashTable *(*Merge)(Pool *p,
                    const HashTable *overlay,
                    const HashTable *base,
                    void * (*merger)(Pool *p,
                                    const void *key,
                                    size_t keyLength,
                                    const void *h1_val,
                                    const void *h2_val,
                                    const void *data),
                    const void *data);
```

Description: Merge two hash tables into one new hash table. If the same key is present in both tables, call the supplied merge function to produce a merged value for the key in the new table. Both hash tables must use the same hash function. The arguments should be:

1. The pool to use when allocating memory. If NULL , the pool of the “base” hash table will be used.
2. The first table to be used in the merge.
3. The second table
4. An argument to pass to the merger function.

newIterator

```
Iterator *(*newIterator)(HashTable *HT);
```

Description: Allocates and initializes a new iterator object to iterate this table. The exact sequence of each object returned is implementation defined but it will be the same for the same dictionary with the same number of elements.

Errors:

CONTAINER_ERROR_BADARG The parameter is NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A pointer to a new iterator or NULL if the operation couldn't be completed.

Example:

```
HashTable *HT;
Iterator *it = iHashTable.newIterator(HT);
double *d;
for (d=it->GetFirst(it); d != NULL; d = it->GetNext(it)) {
    double val = *d;
    // Work with the value here
}
iHashTable.deleteIterator(it);
```

Overlay

```
HashTable *(*Overlay)(Pool *p,  
                      const HashTable *overlay,  
                      const HashTable *base);
```

Description: Copies overlay into base. If conflicts arise, the data in base will be copied in the result.

Errors:

CONTAINER_ERROR_BADARG One of the arguments is NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Resize

```
int (*Resize)(HashTable *HT, size_t newSize);
```

Description: Will resize the given hash table to a new size. If the given new size is zero, the new size is implementation defined, and equal to the amount when automatic resizing occurs.

Errors:

CONTAINER_ERROR_BADARG The parameter is NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive value if the operation completed, a negative error code otherwise.

Replace

```
int (*Replace)(HashTable *HT, const void *key,  
              size_t keyLength, const void *data);
```

Description: Will replace the contents of the given element if found.

Errors:

CONTAINER_ERROR_BADARG The hash table pointer, the key or the replacement data are NULL , or the keyLength is zero.

Returns: A positive number if the element was replaced or zero if the element wasn't found. If the operation didn't complete a negative error code is returned.

Save

```
int (*Save)(HashTable *HT, FILE *out, SaveFunction Fn, void *arg);
```

Description: The contents of the given table are saved into the given stream. If the save function pointer is not NULL , it will be used to save the contents of each element and will receive the arg argument passed to Save, together with the output stream. Otherwise a default save function will be used and arg will be ignored. The output stream must be opened for writing and must be in binary mode.

Errors:

CONTAINER_ERROR_BADARG The array pointer or the stream pointer are NULL .

EOF A disk input/output error occurred.

Returns: A positive value if the operation completed, a negative value or EOF otherwise.

Example:

```
HashTable *HT;
FILE *outFile;
if (iHashTable.Save(HT,outFile,NULL,NULL) < 0) {
    /* Handle error here */
}
```

SetErrorFunction

```
ErrorFunction (*SetErrorFunction)(HashTable *HT,ErrorFunction fn);
```

Description: Replaces the current error function for the given table with the new error function if the parameter is different from NULL . Otherwise no replacement is done.

Errors:

CONTAINER.ERROR_BADARG The table pointer is NULL .

CONTAINER.ERROR_READONLY The table is read only and the function argument is not NULL .

Returns: The old value of the error function or NULL if there is an error.

Size

```
size_t (*Size)(const HashTable *HT);
```

Description: Returns the number of elements stored in the given table.

Errors:

CONTAINER.ERROR_BADARG The table pointer is NULL .

Returns: The number of elements stored in the table

Sizeof

```
size_t (*Sizeof)(const HashTable *HT);
```

Description: Returns the number of bytes of storage used in the given table including the size of the elements stored in it. If HT is NULL the result is the size of the HashTable header.

Returns: The number of elements stored in the table or the size of the HashTable header if the HT pointer is NULL .

4.9 Queues: iQueue

Queues are a type of container adaptors, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

The sample implementation shows how to implement this container as an “adaptor” container, i.e. based on another container. The implementation uses a linked list to implement a queue ².

4.9.1 Interface

```
typedef struct _Queue Queue;

typedef struct _QueueInterface {
    Queue    *(*Create)(size_t elementSize);
    Queue    *(*CreateWithAllocator)(size_t elementSize,
                                     ContainerMemoryManager *allocator);
    int       (*Size)(Queue *Q);
    size_t    (*Sizeof)(Queue *q);
    int       (*Enqueue)(Queue *Q, void *Element);
    void      *(*Dequeue)(Queue *Q);
    int       (*Clear)(Queue *Q);
    int       (*Finalize)(Queue *Q);
    void *    (*Front)(Queue *Q);
    void *    (*Back)(Queue *Q);
    List *    (*GetList)(Queue *q);
} QueueInterface;

extern QueueInterface iQueue;
```

4.9.2 The API

All methods are exactly like the ones in other containers except for Enqueue, that is equivalent to “Add” since adds one element at the end of the container, and Dequeue, that is the same as PopFront, i.e. pops the first element of the container.

²The Java language provides an interface in `java.util`. C# offers a `Queue` class in `System.Collections`, implemented as a circular array that is increased automatically if needed. There is also a generic `Queue` class.

In C++ the definition is: `template < class T, class Container = deque<T> > class queue;` Where

- **T**: Type of the elements.
- **Container**: Type of the underlying container object used to store and access the elements.

Front

```
int    (*Front)(Queue *Q,void *result);
```

Description: Returns the contents of the first element in the given memory area that should be at least the size of the element size of the queue. Note that nothing is changed, and the first element is not erased from the container.

Returns: A positive number for success, zero if the queue is empty or a negative error code.

Errors:

CONTAINER_ERROR_BADARG The Queue pointer is NULL .

Back

```
int    (*Back)(Queue *Q,void *result);
```

Description: Returns the contents of the last element in the given memory area that should be at least the size of the element size of the queue. Note that nothing is changed, and the last element is not erased from the container.

Returns: A positive number for success, zero if the queue is empty or a negative error code.

Errors:

CONTAINER_ERROR_BADARG The Queue pointer is NULL .

GetList

```
List *(*GetList)(Queue *q);
```

Description: Queues are based on the list container. It is not necessary to duplicate all the list functions in the queue interface: this function allows you to access the underlying list and use all the list specific APIs with it.

Returns: A pointer to the list container or NULL if the queue pointer passed is NULL .

4.10 Deque: iDeque

Deque (usually pronounced like "deck") is an irregular acronym of double-ended queue. Double-ended queues are a kind of sequence containers. As such, their elements are ordered following a strict linear sequence. Deques may be implemented by specific libraries in different ways, but in all cases they allow for adding and retrieving elements at both ends, with storage always handled automatically (expanding and contracting as needed).

Operations to insert and retrieve elements in the middle are not provided because if users need a plain sequential container they can use one. Individual implementation can offer those if they think it is useful. This differs from the C++ implementation.

Here is a little table with a Rosetta stone for deque:

C	Ada	C++	Java	Perl	PHP	Python
PushBack	Append	push_back	offerLast	push	array_push	append
PushFront	Prepend	push_front	offerFirst	unshift	array_unshift	appendleft
PopBack	Delete_Last	pop_back	pollLast	pop	array_pop	pop
PopFront	Delete_First	pop_front	pollFirst	shift	array_shift	popleft
Back	Last_Element	back	peekLast	\$array[-1]	end	<obj>[-1]

Some functions that the C++ interface provides like `is_empty()` can be obtained in this implementation simply by invoking:

```
iDeque.Size(deque) == 0
```

4.10.1 Interface

The interface `iDeque` is as follows:

```
typedef struct deque_t Deque;
typedef struct _DequeInterface {
    void (*Apply)(Deque *d,int (*fn)(void *e,void * arg),void *arg);
    int (*Back)(Deque *d,void *outbuf);
    int (*Clear)(Deque *Q);
    size_t (*Contains)(Deque * d, void* item);
    Deque *(*Copy)(Deque *d);
    Deque *(*Create)(size_t elementSize);
    int (*deleteIterator)(Iterator *);
    int (*Equal)(Deque *d1,Deque *d2);
    int (*Erase)(Deque * d, void* item);
    int (*Finalize)(Deque *Q);
    unsigned (*GetFlags)(Deque *Q);
    Deque *(*Load)(FILE *stream, ReadFunction readFn,void *arg);
    Iterator *(*newIterator)(Deque *Deq);
    int (*Save)(Deque *d,FILE *stream, SaveFunction saveFn,void *arg);
    unsigned (*SetFlags)(Deque *Q,unsigned newFlags);
    size_t (*Size)(Deque *Q);
    ErrorFunction (*SetErrorFunction)(Deque *d,ErrorFunction);
    size_t (*Sizeof)(Deque *d);
    int (*PushBack)(Deque *Q, void *Element);
    int (*PushFront)(Deque *Q, void *Element);
    int (*PopBack)(Deque *d,void *outbuf);
    int (*Front)(Deque *d,void *outbuf);
    int (*PopFront)(Deque *d,void *outbuf);
} DequeInterface;
```

```
extern DequeInterface iDeque;
```

The deque container can be implemented as an adaptor container, for instance based on a double linked list or in an vector. In any case the underlying container interface is not visible.

Apply

```
void (*Apply)(Deque *d,int (Applyfn)(void *,void *),void *arg);
```

Description: Will call the given function for each element. The first argument of the callback function receives an element of the array. 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 array is read-only, a copy of the element will be passed to the callback function.

Errors:

CONTAINER_ERROR_BADARG Either the deque or Applyfn are NULL .

CONTAINER_ERROR_NOMEMORY The list is read-only and there is no more memory to allocate the buffer to copy each element.

Back

```
int (*Back)(Deque *d,void *outbuf);
```

Description: Copies into the given buffer the last element stored in the Deque d.

Errors:

CONTAINER_ERROR_BADARG Either d or outbuf are NULL .

Returns: A positive value of the operation completed, zero if the container is empty, or a negative error code otherwise.

Clear

```
int (*Clear)(Deque *Q);
```

Description: Erases all elements stored in the queue and reclaims the memory used. The Deque object itself is not destroyed. Errors

CONTAINER_ERROR_BADARG The deque pointer is NULL .

CONTAINER_ERROR_READONLY The deque is read-only. No modifications allowed.

Contains

```
size_t (*Contains)(Deque * d, void* item);
```

Description: Searches the deque for the given data, returning its (index one based) position or zero if not found. Errors

CONTAINER_ERROR_BADARG The deque pointer is NULL .

Returns: The index of element or zero if not found.

Copy

```
Deque *(*Copy)(Deque *d);
```

Description: Makes a copy of the given deque.

Errors:

CONTAINER_ERROR_BADARG The deque pointer is NULL .

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A pointer to the new container or NULL if the operation did not complete.

Create

```
Deque *(*Create)(size_t elementSize);
```

Description: Creates a new Deque container using “elementSize” as the size that each element will have.

Errors:

CONTAINER_ERROR_BADARG The elementSize parameter is zero or bigger than what the implementation supports.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A pointer to the new container or NULL if the operation did not complete.

Example:

```
Deque *d = iDeque.Create(sizeof(myType));
if (d == NULL) { /* Error handling */ }
```

Equal

```
int (*Equal)(Deque *d1,Deque *d2);
```

Description: Compares the given deques using their comparison function. If they differ in their size, flags, or compare functions they compare unequal. If any of their elements differ, they compare unequal. If both d1 and d2 are NULL they compare equal. If both are empty, they compare equal.

Errors:

None

Returns: The result is one if the deques are equal, zero otherwise.

Front

```
int (*PeekFront)(Deque *d,void *outbuf);
```

Description: Copies into the given buffer the first element stored in the Deque d.

Errors:

CONTAINER_ERROR_BADARG Either d or outbuf are NULL .

Returns: A positive value of the operation completed, zero if the container is empty, or a negative error code otherwise.

Erase

```
int (*Erase)(Deque * d, void* item);
```

Description: Erases the first occurrence of the given element from the container if found, starting from the front.

Errors:

CONTAINER_ERROR_BADARG The deque pointer or the item pointer are NULL .

CONTAINER_ERROR_READONLY The deque is read-only. No modifications allowed.

Returns: A positive number if the item was found and erased, zero if the item wasn't found, or a negative error code if the operation did not complete.

Finalize

```
int (*Finalize)(Deque *d);
```

Description: Reclaims all memory used by the container erasing all elements, if any. Then it destroys the container object itself.

Errors:

CONTAINER_ERROR_BADARG The deque or the element pointers are NULL .

CONTAINER_ERROR_READONLY The deque is read-only. No modifications allowed.

Returns: A positive number if the operation completed, a negative error code otherwise.

GetFlags

```
unsigned (*GetFlags)(Deque *d);
```

Description: Retrieves the state of the flags. If the implementation doesn't support this field this function always returns zero.

Errors:

CONTAINER_ERROR_BADARG The deque pointer is NULL .

Returns: The state of the flags field.

Load

```
Deque *(*Load)(FILE *stream, ReadFunction readFn, void *arg);
```

Description: Reads a deque previously saved with the Save function from the stream pointed to by stream. If readFn is not NULL , it will be used to read each element. The "arg" argument will be passed to the read function. If the read function is NULL , this argument is ignored and a default read function is used.

Errors:

CONTAINER_ERROR_BADARG The given stream pointer is NULL .

CONTAINER_ERROR_NOMEMORY There is not enough memory to complete the operation.

Returns: A new deque or NULL if the operation could not be completed. Note that the function pointers in the deque are NOT saved in most implementations, nor any special allocator that was in the original table. In most implementations those values will be

the values by default. To rebuild the original state the user should replace the pointers again in the new table.

PopBack

```
int (*PopBack)(Deque *d,void *outbuf);
```

Description: Copies into the given buffer the last element stored in the Deque d, then erases the element from the deque.

Errors:

CONTAINER_ERROR_BADARG Either d or outbuf are NULL .

Returns: A positive value of the operation completed, zero if the container is empty, or a negative error code otherwise.

PopFront

```
int (*PopFront)(Deque *d,void *outbuf);
```

Description: Copies into the given buffer the first element stored in the Deque d, then erases the element from the deque.

Errors:

CONTAINER_ERROR_BADARG Either d or outbuf are NULL .

Returns: A positive value of the operation completed, zero if the container is empty, or a negative error code otherwise.

PushBack

```
int (*PushBack)(Deque *d,void *element);
```

Description: Adds the given element to the end of the deque. It is assumed that “element” points to a contiguous memory area of at least ElementSize bytes.

Errors:

CONTAINER_ERROR_BADARG The deque or the element pointers are NULL .

CONTAINER_ERROR_READONLY The deque is read-only. No modifications allowed.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive number if the operation added a new element, or a negative error code if an error occurred.

Example:

```
Deque *d;
double data = 4.5;
int result = iDeque.PushBack(d,&data);
if (result < 0) { /* Error handling */ }
```

PushFront

```
int (*PushFront)(Deque *d,void *element);
```


Description: Adds the given element to the start of the deque. It is assumed that “element” points to a contiguous memory area of at least ElementSize bytes.

Errors:

CONTAINER_ERROR_BADARG The deque or the element pointers are NULL .

CONTAINER_ERROR_READONLY The deque is read-only. No modifications allowed.

CONTAINER_ERROR_NOMEMORY Not enough memory to complete the operation.

Returns: A positive number if the operation added a new element, or a negative error code if an error occurred.

Example:

```
Deque *d;
double data = 4.5;
int result = iDeque.PushFront(d,&data);
if (result < 0) { /* Error handling */ }
```

Save

```
int (*Save)(Deque *d, FILE *out, SaveFunction Fn, void *arg);
```

Description: The contents of the given deque are saved into the given stream. If the save function pointer is not NULL , it will be used to save the contents of each element and will receive the arg argument passed to Save, together with the output stream. Otherwise a default save function will be used and arg will be ignored. The output stream must be opened for writing and must be in binary mode.

Errors:

CONTAINER_ERROR_BADARG The deque pointer or the stream pointer are NULL . EOF A disk input/output error occurred.

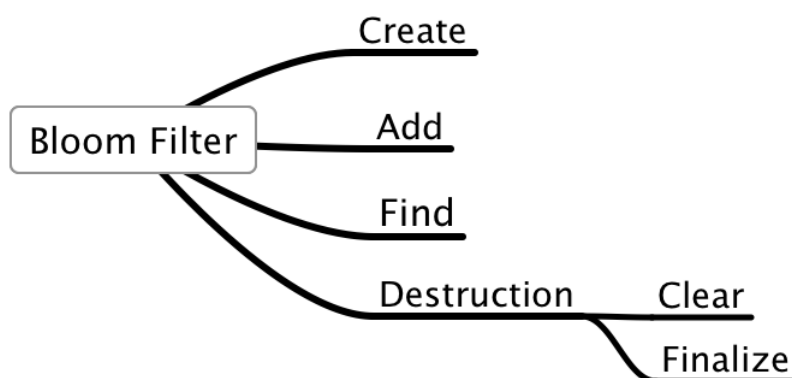
Returns: A positive value if the operation completed, a negative value or EOF otherwise.

Example:

```
Deque *d;
FILE *outFile;
if (iDeque.Save(d,outFile,NULL,NULL) < 0) {
    /* Handle error here */
}
```

4.11 Bloom filters

Bloom filters allow you to determine cheaply and quickly if an element is member of a set without actually looking into the large set. This container doesn't store any data, just a series of bits indicating whether the element is there. It can return false answers, specifically a false positive meaning it can answer "yes, the element is there" when in fact it is not. When it tells you however that the element is **not** there you can be sure it is not in the set. The probability that a false answer occurs can be calculated in function of the size reserved for the bit table: the bigger the table, the smaller the probability of a false answer for a fixed number of elements.³



4.11.1 The interface: iBloomFilter

```
typedef struct tagBloomFilterInterface {
    size_t (*CalculateSpace)(size_t maxElements, double probability);
    BloomFilter *(*Create)(size_t maxElements, double probability);
    size_t (*Add)(BloomFilter *b, const void *key, size_t keylen);
    int (*Find)(BloomFilter *b, const void *key, size_t keylen);
    int (*Clear)(BloomFilter *b);
    int (*Finalize)(BloomFilter *b);
};
```

³More about bloom filters in: <http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html>, and at the NIST: <http://xw2k.nist.gov/dads/html/bloomFilter.html>

The original paper about them was published by Burton Bloom: **Space/time trade-offs in hash coding with allowable errors. Communications of ACM, pages 13(7):422-426, July 1970.**

The idea behind this data structure is to allocate a vector of m bits, initially all set to 0, and then choose k independent hash functions, h_1, h_2, \dots, h_k , each with range $\{1, \dots, m\}$. For each element $a \in A$, the whole set, the bits at positions $h_1(a)$, $h_2(a)$, ..., $h_k(a)$ in v are set to 1. (A particular bit might be set to 1 multiple times).

Given a query for some key b we check the bits at positions $h_1(b)$, $h_2(b)$, ..., $h_k(b)$. If any of them is 0, then certainly b is not in the set A . Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a "false positive". The parameters k (the maximum number of elements) and m (the probability) should be chosen such that the probability m of a false positive (and hence a false hit) is acceptable.

```
} BloomFilterInterface;
```

4.11.2 The API

CalculateSpace

```
size_t (*CalculateSpace)(size_t maxElements,double probability);
```

Description: Returns the space in bytes that would occupy a bloom filter to hold the given number of elements with the given probability. The probability parameter should be greater than zero and smaller than 1.0. For values very close to the values zero and one, a huge number of bits can be necessary and the filter creation function will return NULL because of lack memory problems.

Errors:

CONTAINER_ERROR_BADARG The probability is smaller or equal than zero, or bigger or equal than one.

Returns: The number of bytes needed or zero in case of error.

Create

```
BloomFilter *(*Create)(size_t maxElements,double probability);
```

Description: Creates and initializes a filter with space enough to hold *MaxElements* with the given probability for a false answer. The probability parameter should be greater than zero and smaller than 1.0. For values very close to the values zero and one, a huge number of bits can be necessary and the filter creation function will return NULL because of lack memory problems.

Errors:

CONTAINER_ERROR_BADARG The probability is smaller or equal than zero, or bigger or equal than one.

CONTAINER_ERROR_NOMEM There is no memory for the allocation of the necessary data structures.

Returns: A pointer to a newly allocated bloom filter or NULL in case of error.

Add

```
size_t (*Add)(BloomFilter *b,const void *key,size_t keylen);
```

Description: Adds the given key to the filter. The *keylen* argument should be the length of the key, that should never be zero.

Errors:

CONTAINER_ERROR_BADARG The filter pointer or the key pointer are NULL , or the *keylen* is zero.

CONTAINER_ERROR_CONTAINER_FULL . The maximum number of elements has been reached.

Returns: The number of elements in the filter or zero if there is an error.

Find

```
int (*Find)(BloomFilter *b,const void *key,size_t keylen);
```

Description: Searches the given key in the filter.

Errors:

CONTAINER_ERROR_BADARG The filter pointer or the key pointer are NULL , or the keylen is zero.

Returns: One if the element is found, zero if it is not, or a negative error code if an error occurs.

Clear

```
int (*Clear)(BloomFilter *b);
```

Description: Removes all elements from the filter. No memory is released.

Errors:

CONTAINER_ERROR_BADARG The given pointer is NULL .

Returns: One if all elements were cleared, a negative error code otherwise.

Finalize

```
int (*Finalize)(BloomFilter *b);
```

Description: Releases all memory held by the filter.

Errors:

CONTAINER_ERROR_BADARG The given pointer is NULL .

Returns: One if all elements were cleared, a negative error code otherwise.

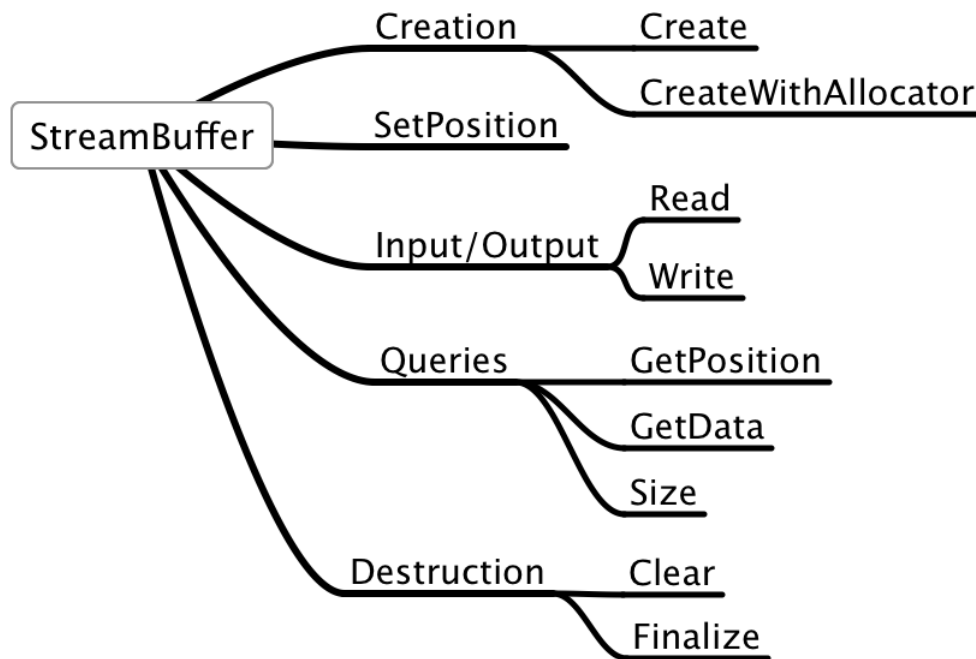
4.12 Buffers

The buffers interface is greatly simplified compared to the interface of a general container. The usage of a buffer as an intermediate storage means there is no sense in including all the functionality of a container. The library provides two types:

1. Stream buffers provide linear storage
2. Circular buffers store the last n items of a stream

Other languages provide similar features⁴.

4.12.1 Stream buffers



⁴ The Java language provides a typed buffer functionality. These buffers are not resizable, have a cursor and other more sophisticated operations than the buffers proposed here like slicing and compacting. Their place in the Java class hierarchy is: `Object` → `Native I/O` → `ByteBuffer`. There are methods for viewing the buffer as float, doubles, etc.

The C# language provides also a typed buffer class of the primitive types (char, float, int, etc). It is called `System.Buffer` and provides a few methods for determining its length and read/write a single byte. The language itself doesn't provide any circular buffers class but several implementations are available in the net. The same situation applies for Java.

The design objective in this library is to keep buffers small and, while providing functionality, reduce the interface to a minimum. Compacting is not feasible in C due to the wide use of pointers. If there is a pointer to the data in the buffer, moving it would invalidate the pointer making for hard to debug crashes.

This objects are designed to store sequentially arbitrary data, resizing themselves as necessary. There is a cursor, a pointer that indicates where the next data item will be written. You can move the cursor, overwriting old data, or leaving holes in the buffer structure.

The interface

```
typedef struct tagStreamBufferInterface {
    StreamBuffer *(*Create)(size_t startsize);
    StreamBuffer *(*CreateWithAllocator)(size_t startsize,
                                         ContainerMemoryManager *allocator);
    size_t (*Read)(StreamBuffer *b, void *data, size_t siz);
    size_t (*Write)(StreamBuffer *b, void *data, size_t siz);
    int (*SetPosition)(StreamBuffer *b, size_t pos);
    size_t (*GetPosition)(StreamBuffer *b);
    char *(*GetData)(StreamBuffer *b);
    size_t (*Size)(StreamBuffer *b);
    int (*Clear)(StreamBuffer *b);
    int (*Finalize)(StreamBuffer *b);
} StreamBufferInterface;
```

The API

Clear

```
int (*Clear)(StreamBuffer *b);
```

Description: Sets the cursor at position zero and zeroes the whole buffer.

Errors:

CONTAINER_ERROR_BADARG The given buffer pointer is NULL

Returns: A positive value if successful, a negative error code otherwise.

Create

```
StreamBuffer *(*Create)(size_t startsize);
```

Description: Creates a new buffer with the given start size. If the size is zero it will use a default start value. The allocator used is the current memory manager.

Errors:

CONTAINER_ERROR_NOMEMORY . There is no more memory to create the buffer.

Returns: A pointer to a newly created buffer or NULL if there is no more memory left.

CreateWithAllocator

```
StreamBuffer *(*CreateWithAllocator)(size_t startsize,
                                     ContainerMemoryManager *allocator);
```

Description: Creates a new buffer using the given allocator and start size. If the start size is zero a default value is used.

Errors:

CONTAINER_ERROR_NOMEMORY There is no more memory to complete the operation.

Returns: A pointer to the new buffer or NULL if there is no memory left.

Finalize

```
int (*Finalize)(StreamBuffer *b);
```

Description: Releases all memory used by the buffer.

Errors:

CONTAINER_ERROR_BADARG The given buffer pointer is NULL .

Returns: A positive value if successful or a negative error code.

GetData

```
char *(*GetData)(StreamBuffer *b);
```

Description: Returns a pointer to the data stored in the buffer.

Errors:

CONTAINER_ERROR_BADARG The given buffer pointer is NULL

Returns: A pointer to the buffer's data or NULL, if an error occurs.

GetPosition

```
size_t (*GetPosition)(StreamBuffer *b);
```

Description: Returns the current cursor position.

Errors:

CONTAINER_ERROR_BADARG The stream buffer pointer is NULL

Returns: The cursor position or zero if there is an error. Note that zero is also a valid cursor position.⁵

Read

```
size_t (*Read)(StreamBuffer *b, void *data, size_t siz);
```

Description: Reads *siz* bytes from the given buffer, starting from the position of the cursor. If the buffer finishes before *siz* characters are read, reading stops, and less characters than requested are returned. It is assumed that the *data* buffer contains at least *siz* characters.

Errors:

CONTAINER_ERROR_BADARG Either the stream buffer, the data buffer are NULL .

⁵Here, as in other APIs from the `buffer` interface it was preferred to have a friendly interface than to cater for errors. In case of a zero return, you should test for a NULL pointer, but it is even better to test for it before calling this function.

Returns: The number of characters copied or zero if there is an error. Note that if the number of requested characters is zero, this function will also return zero.

SetPosition

```
int (*SetPosition)(StreamBuffer *b, size_t pos);
```

Description: Sets the cursor at the given position. If the position is bigger than the size of the buffer the cursor is moved to the end of the buffer.

Errors:

CONTAINER_ERROR_BADARG The given buffer pointer is NULL

Returns: A positive value if successful, a negative error code otherwise.

Size

```
size_t (*Size)(StreamBuffer *b);
```

Description: Returns the allocated size of the buffer. If the buffer pointer is NULL returns the size of the buffer header.

Errors:

None

Returns: The size of the buffer.

Write

```
size_t (*Write)(StreamBuffer *b, void *data, size_t siz);
```

Description: Writes into the buffer *siz* characters from the passed pointer *data*. The characters are written starting at the cursor position. If the buffer is too small to hold the data, it will be enlarged using its allocator.

Errors:

CONTAINER_ERROR_NOMEMORY . There is no more memory to enlarge the buffer.

CONTAINER_ERROR_BADARG The stream buffer pointer or the data pointer is NULL .

Returns: The number of characters written.

Example:

```
#include <containers.h>
int main(void)
{
    StreamBuffer *sb = iStreamBuffer.Create(10);
    int i;
    char buf[20], *p;

    for (i=0; i<10; i++) {
        sprintf(buf, "item %d", i+1);
        iStreamBuffer.Write(sb, buf, 1+strlen(buf));
    }
}
```



```
    buf[0]=0;
    iStreamBuffer.Write(sb,&buf,1);
    printf("Buffer size is: %d, position is %d\n",
           (int)iStreamBuffer.Size(sb),
           (int) iStreamBuffer.GetPosition(sb));
    iStreamBuffer.SetPosition(sb,0);
    p = iStreamBuffer.GetData(sb);
    while (*p) {
        printf("%s\n",p);
        p += 1 + strlen(p);
    }
    iStreamBuffer.Finalize(sb);
    return 1;
}
```

OUTPUT:

Buffer size is: 82, position is 72

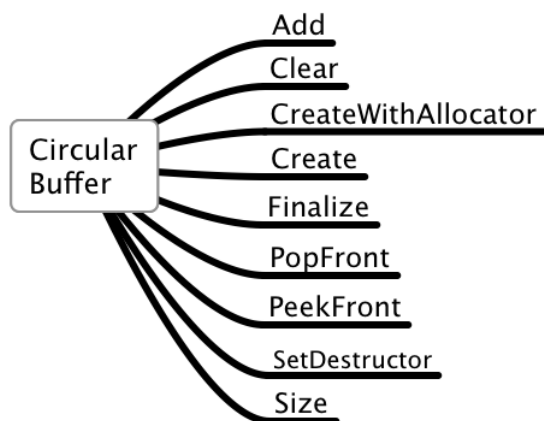
item 1
item 2
item 3
item 4
item 5
item 6
item 7
item 8
item 9
item 10

This example does the following:

- Creates a stream buffer. It assumes success and does not test the return value of the creation function. The buffer is dimensioned too small for the data it will contain so it has to resize several times.
- Prepares a string buffer with `sprintf` and writes the resulting string including its terminating zero in the stream buffer. Note that zeroes have no special significance in buffers. It loops ten times doing this operation.
- It ends the buffer with a terminating double zero.
- It prints the buffer size and the number of characters it has written. Note that they are not the same. The buffer has been resized several times, and at each time the new capacity is determined by an internal algorithm. Since we did not move the cursor the position of the cursor give us the number of characters written.
- It obtains a pointer to the data in the buffer

- It prints all the strings in the buffer to standard output. Each character string from 1 to 9 is 7 bytes long, including its terminating zero. The tenth string is 9 bytes, also including the terminating zero. We have then: $(7*9)+9 = 72$.
- It destroys the buffer.

4.12.2 Circular buffers



This objects are designed to store the last n items of a stream. When they are full, the new items are stored in the same place as the oldest item.

The interface: iCircularBuffer

```

typedef struct tagCircularBufferInterface {
    int (*Add)(CircularBuffer * b, void *data_element);
    int (*Clear)(CircularBuffer *cb);
    CircularBuffer *(*CreateWithAllocator)(size_t sizElement,
                                           size_t sizeBuffer,
                                           ContainerMemoryManager *allocator);
    CircularBuffer *(*Create)(size_t sizElement, size_t sizeBuffer);
    int (*Finalize)(CircularBuffer *cb);
    int (*PopFront)(CircularBuffer *b, void *result);
    int (*PeekFront)(CircularBuffer *b, void *result);
    size_t (*Size)(CircularBuffer *cb);
    DestructorFunction SetDestructor(CircularBuffer *cb,
                                     DestructorFunction NewFn);
} CircularBufferInterface;
  
```

The API

Add

```
int (*Add)( CircularBuffer * b, void *data_element);
```

Description: Adds the given data element to the circular buffer. If the buffer is full, the oldest element's place will be overwritten with the new data and the container remains full with the same number of elements.

Errors:

CONTAINER_ERROR_BADARG One or both arguments are NULL .

Returns: A negative error code if an error occurs. If the container is full zero is returned. If a new element was added a positive number is returned.

Clear

```
int (*Clear)(CircularBuffer *cb);
```

Description: Resets the number of elements inside the container to empty without freeing the memory used by the buffer.

Errors:

CONTAINER_ERROR_BADARG The buffer pointer *b* is NULL .

Returns: A negative error code if an error occurs, or a positive number when the container is reset.

CreateWithAllocator

```
CircularBuffer *(*CreateWithAllocator)(size_t ElementSize,  
                                       size_t sizeBuffer, ContainerMemoryManager *allocator);
```

Description: Creates an empty circular buffer that can hold at most *sizeBuffer* elements, each element being of size *ElementSize*. Uses the given allocator to allocate memory.

Errors:

CONTAINER_ERROR_BADARG One or both sizes are zero, or the allocator pointer is NULL .

CONTAINER_ERROR_NOMEM There is no memory left.

Returns: A pointer to a new circular buffer or NULL if an error occurs.

Create

```
CircularBuffer *(*Create)(size_t ElementSize, size_t sizeBuffer);
```

Description: Creates an empty circular buffer that can hold at most *sizeBuffer* elements, each element being of size *ElementSize*. Uses the CurrentMemoryManager to allocate memory.

Errors:

CONTAINER_ERROR_BADARG One or both arguments are zero.

CONTAINER_ERROR_NOMEM There is no memory left.

Returns: A pointer to a new circular buffer or NULL if an error occurs.

Finalize

```
int (*Finalize)(CircularBuffer *cb);
```

Description: Reclaims all memory used by the given buffer.

Errors:

CONTAINER_ERROR_BADARG The buffer pointer is NULL .

Returns: A positive value if the container is destroyed, a negative error code otherwise.

PeekFront

```
int (*PeekFront)(CircularBuffer *b,void *result);
```

Description: Copies one item from the front of the circular buffer into the given buffer without removing the item from the container.

Errors:

CONTAINER_ERROR_BADARG The buffer pointer or the result buffer are NULL .

Returns: A negative error code if an error occurs, zero if the buffer was empty, or a positive number if an item was copied.

PopFront

```
int (*PopFront)(CircularBuffer *b,void *result);
```

Description: Copies one item from the front of the circular buffer into the given buffer and removes the item from the container. If the *result* pointer is NULL the item is removed but nothing is copied.

Errors:

CONTAINER_ERROR_BADARG The buffer pointer is NULL .

Returns: A negative error code if an error occurs, zero if the buffer was empty, or a positive number if an item was removed.

Example:

```
#include <containers.h>
int main(void)
{
    CircularBuffer *cb = iCircularBuffer.Create(sizeof(int),10);
    int i, integer;

    for (i=0; i<20;i++) {
        iCircularBuffer.Add(cb,&i);
    }
    printf("There are %d elements\n",iCircularBuffer.Size(cb));
    printf("The container uses %d bytes\n",iCircularBuffer.Sizeof(cb));
    printf("The size of the header is %d\n",
           iCircularBuffer.Sizeof(NULL));
    /* Print all elements */
    while (iCircularBuffer.PopFront(cb,&integer) > 0) {
```

```
        printf("%d ",integer);  
    }  
    return 0;  
}
```

OUTPUT:

There are 10 elements

The container uses 88 bytes

The size of the header is 48

10 11 12 13 14 15 16 17 18 19

Size

```
size_t (*Size)(CircularBuffer *cb);
```

Description: Computes the number of items in the given circular buffer.

Errors:

CONTAINER_ERROR_BADARG The buffer pointer is NULL .

Returns: The number of items in the buffer.

Sizeof

```
size_t (*Sizeof)(CircularBuffer *cb);
```

Description: Computes the number of bytes used by given circular buffer. If the pointer is NULL returns the size of the circular buffer header structure.

Returns: The number of bytes used by the buffer.

4.13 The generic interfaces

This interface allows the user to use containers in a generic way, ignoring its specific type. Note that there is no "GenericContainer" object; you can't create any generic container. Once a specific container is created, it can be used as a generic container at any time since all containers comply with the generic interface. This interface just dispatches internally to the actual container and therefore incurs in a slight performance cost.⁶

Conceptually, the generic interfaces represent a base class (GenericContainer) and two derived classes: Sequential and Associative containers. It would be possible to derive more classes, for instance a numeric container class that could be implemented in the future. This is left open for future releases of this specification.⁷

4.13.1 Generic containers

The general generic interface that encloses associative and sequential containers is as follows:

```
typedef struct GenericContainer GenericContainer;
typedef struct tagGenericContainerInterface {
    size_t (*Size)(GenericContainer *Gen);
    unsigned (*GetFlags)(GenericContainer *Gen);
    unsigned (*SetFlags)(GenericContainer *Gen, unsigned flags);
    int (*Clear)(GenericContainer *Gen);
    int (*Contains)(GenericContainer *Gen, void *Value);
    int (*Erase)(GenericContainer *Gen, void *objectToDelete);
    int (*Finalize)(GenericContainer *Gen);
    void (*Apply)(GenericContainer *Gen,
                  int (*Applyfn)(void *, void * arg),
                  void *arg);
    int (*Equal)(GenericContainer *Gen1, GenericContainer *Gen2);
    GenericContainer *(*Copy)(GenericContainer *Gen);
    ErrorFunction (*SetErrorFunction)(GenericContainer *Gen,
                                      ErrorFunction fn);
    size_t (*Sizeof)(GenericContainer *Gen);
```

⁶ The Objective-C language has a similar constructs with its "Protocols". Several classes can share a common interface without any inheritance between them. Obviously in C there is no compiler support for this kind of programming, what forces your implementation to be careful about the order of the function pointers within all objects. A change in the order of those function pointers makes the object incompatible with the protocol specifications, and this can't be checked by the compiler. All of this can be avoided, of course, if you just use the protocols designed and implemented by someone else.

⁷ Two special cases of specialized arrays have been presented: an array of strings and an array of bits. Many other arrays are possible and surely necessary for numeric work, and they could be used as basis for vector extensions with hardware support. Another subject not mentioned in this specification is multi-dimensional arrays.

```

    Iterator *(*newIterator)(GenericContainer *Gen);
    int (*deleteIterator)(Iterator *);
    int (*Save)(GenericContainer *Gen, FILE *stream,
                SaveFunction saveFn, void *arg);
    GenericContainer *(*Load)(FILE *stream,
                              ReadFunction readFn, void *arg);
    size_t GetElementSize(GenericContainer *Gen);
} GenericContainerInterface;
extern GenericContainerInterface iGenericContainer;

```

These functions return the obvious results already described in the documentation of their container-specific counterparts and not repeated here. We only note the absence of a creation function, or any means to add an object.⁸

Based on the generic interface, we have generic sequential and associative interfaces. They contain generic functions for adding and removing objects.

4.13.2 Sequential containers

These containers include all the functions of the GenericContainer interface, adding functions to use any sequential container as a stack, and functions for managing object replacement or addition.

```

typedef struct SequentialContainer SequentialContainer;
typedef struct tagSequentialContainerInterface {
    GenericContainerInterface Generic;
    int (*Add)(SequentialContainer *SC, void *Element);
    void *(*GetElement)(SequentialContainer *SC, size_t idx);
    int (*Push)(SequentialContainer *Gen, void *Element);
    int (*Pop)(SequentialContainer *Gen, void *result);
    int (*InsertAt)(SequentialContainer *SC, size_t idx, void *newval);
    int (*EraseAt)(SequentialContainer *SC, size_t idx);
    int (*ReplaceAt)(SequentialContainer *SC,
                    size_t idx, void *element);
    int (*IndexOf)(SequentialContainer *SC,
                  void *ElementToFind, size_t *result);
    int (*Append)(SequentialContainer *SC1, SequentialContainer *SC2);
} SequentialContainerInterface;
extern SequentialContainerInterface iSequentialContainer;

```

⁸The "Erase" function has been added to the generic interface because it has the same interface both in associative and sequential containers. The "Add" function was left out because in associative containers you need a key argument to add data. This could have been fixed by defining a structure with two fields that would be passed as a single argument, but that would make things more complex than they need to be

4.13.3 Associative containers

These containers include all the functions of the GenericContainer interfaces and add functions for inserting and removing objects.

```
typedef struct AssociativeContainer AssociativeContainer;
typedef struct tagAssociativeContainerInterface {
    GenericContainerInterface Generic;
    int (*Add)(SequentialContainer *SC,void *key,void *Element);
    void (*GetElement)(AssociativeContainer *SC,void *Key);
    int (*Replace)(AssociativeContainer *SC, void *Key, void *element);
} AssociativeContainerInterface;
extern AssociativeContainerInterface iAssociativeContainer;
```


5 Enhancing the library

No design can ever cover all special cases that can arise during development. The advantage of the interface design is that you can enhance the library by subclassing functions that add functionality you need when absent. Subclassing means in this context that you replace a function of the library with a new function written by you that either replaces completely the functionality of the library or that either before or after the library function adds some code that implements an enhancement.

There are several ways to enhance the library in this way:

1. Replace the function in the container interface object. This affects all containers of this type, including those that are already created. This involves simply assigning to the function you want to replace a new function pointer that points to a compatible function. You can save the old value and add some functionality, call the old function pointer to do what the library does, then you can add code that runs after the old library function has finished.
2. Replace the function in a copy of the functions table of a single object. This way is less intrusive than the former, since only one container is affected: the one where you modify the function table. The downside is that instead of using the simple syntax:

```
iList.Add
```

you have to use the container's table:

```
Container->VTable->Add(...)
```

This represents quite a different syntax, but this can be less of a problem if you hide it under some convenient macros ¹.

On the up side, another advantage of this syntax is that you do not need to change your source code if you change the type of the container. If you write:

```
myContainer->Vtable->Add(myContainer,object);
```

this will stay the same for lists, arrays, string collections or whatever sequential container you are using. You can then change completely the type of the container just by changing the declaration.

¹For instance you can use `iList.Add` for `iList.Add`, or similar conventions. The specifications of the library do not define those macros to avoid invasion of the user's name space

6 Applications

6.1 Mapcar

The lisp function "mapcar" produces a map by applying a given function to each element of a list.

```
(mapcar #'abs '(3 -4 2 -5 -6)) => (3 4 2 5 6)
```

We can reproduce this function by using "Apply". In the extra argument we pass a structure of two members:

- A function to call (in the above example it would be a function to calculate the absolute value)
- A list container where the result would be stored

Our function receives then (as all functions called by Apply) two arguments, the element and a pointer to our structure. Here is a sketch of how could it be done:

```
#include <containers.h>
struct MapcarArgs {
    void *(*fn)(void *);
    List *Result;
};
```

We keep some generality by using a general prototype and definition for the function we are using. We could have defined the callback as:

```
int (*fn)(int *);
```

That prototype would have been unusable for lists that use doubles, for instance. With the current definition we can use this "MapcarArgs" structure with any other list.

The actual function we are calling encapsulates all knowledge about the data stored in the list and the operation we perform with that data. The other parts of the software do not need to know anything about it. It returns a static pointer to the result of the operation it performs using the given element as data that will be overwritten at each call. The intended usage is to save that result before making the next call. It can be defined as follows:

```
void *DoAbsValue(void *element)
{
    static int result = *(int *)element;
    if (result < 0)
        result = -result;
    return &result;
}
/* This function will be directly called by "Apply". */
static int Applyfn(void *element, struct MapcarArgs *args)
{
    void *result = args->fn(element);
    int r = iList.Add(args->Result,result);
    return r;
}
List *mapcar(List *li,void *(*fn)(void *))
{
    struct MapcarArgs args;

    args.fn = fn;
    args.Result = iList.Create(iList.GetElementSize(li));
    if (args.Result == NULL)
        return NULL;
    iList.Apply(li,Applyfn,(void *)&args);
    return args.Result;
}

int main(void)
{
    List *li = iList.Create(sizeof(int));
    List *newList;
    int i;
    int tab[] = {3,-4,2,-5,6};
    for (i=0; i<5;i++) {
        iList.Add(li,&tab[i]);
    }
    newList = mapcar(li,DoAbsValue);
}
```

Still, our version of mapcar is still specific to lists. A more general version would use a sequential container to make a mapcar function that would be able to work with any type of sequential container.

The basic idea is to provide an empty container of the desired result type as an extra argument to mapcar. We use an iterator instead of "Apply", obtaining a single compact function that will take any sequential container as input and add the result of the function

to any type of sequential container.

```
int mapcar(SequentialContainer *src,          /* The source container */
           void *(*fn)(void *), /* Function to call with each element */
           SequentialContainer *result) /* The resulting container */
{
    Iterator *it = iSequentialContainer.newIterator(src);
    int r=1;
    void *obj;
    if (it == NULL)
        return CONTAINER_ERROR_NOMEMORY;
    for (obj = it->GetFirst(it);
         obj != NULL;
         obj = it->GetNext(it)) {
        void *tmp = fn(obj);
        int r = iSequentialContainer.Add(result,tmp);
        if (r < 0) {
            /* In case of any error return a partial result
               and the error code */
            break;
        }
    }
    deleteIterator(it);
    return r;
}
```

Other similar functions can be built from this model. For instance "mapcon", a function that needs two containers to build a resulting container. The result is made out of the results of a binary function that will receive one element from each container.

Its implementation is trivially deduced from the above function:

```
int mapcon(SequentialContainer *src1,*src2, /* The input containers */
           void *(*fn)(void *,void *), /* Function with 2 arguments */
           SequentialContainer *result) /* The resulting container */
{
    Iterator *it1 = iSequentialContainer.newIterator(src1);
    Iterator *it2 = iSequentialContainer.newIterator(src2);
    int r=1;
    void *obj1,*obj2;
    if (it == NULL)
        return CONTAINER_ERROR_NOMEMORY;
    if (iSequentialContainer.GetElementSize(src1) !=
        iSequentialContainer.GetElementSize(src2)) {
        return CONTAINER_ERROR_INCOMPATIBLE;
    }
}
```

```
    for (obj1 = it1->GetFirst(it1), obj2 = it2->GetFirst(it2);
        obj1 != NULL && obj2 != NULL;
        obj2 = it2->GetNext(it2),
        obj1 = it1->GetNext(it1)) {
        void *tmp = fn(obj1, obj2);
        int r = iSequentialContainer.Add(result, tmp);
        if (r < 0) {
            /* In case of any error return a partial result
            and the error code */
            break;
        }
    }
    deleteIterator(it1);
    deleteIterator(it2);
    return r;
}
```

We can use it with a function that adds its two arguments to add two containers:

```
void *DoAdd(void *element1, void *element2)
{
    static int result = *(int *)element1 + *(int *)element2;
    return &result;
}
```

Note that not all errors are detected, and we stop at the smallest container, producing a result compatible with the smallest of both arguments. Note too that we make a very superficial compatibility test to see if the arguments contain the same type of object, using their size as an indication. This test would ignore elements of the same size but incompatible, for instance floats and 32 bit integers, or 64 bit integers and double precision elements, etc.

The standard answer to the above problems is to point out that C has a tradition of keeping things simple and expecting programmers that take care of low level details. If you want more error support, you will find out with minimal research a lot of languages ready to make all kinds of hand holding for you.

7 The sample implementation

The objective of the sample implementation is to serve as a guide for the implementers of this proposal. It is not the fastest implementation and it is not the most efficient or compact one. As any other software, it contains bugs, that I hope to iron out with time.

7.1 Data structures

All container data structures are composed of two parts:

1. A header part, containing a pointer to the functions table and some other fields. This 'generic' part is at the start of all container header structures.
2. A container specific part, containing auxiliary structures and data needed for the specific container at hand.

7.1.1 The generic part

The first part of all container data structures is the same for each container. This allows to implement conceptually an abstract class of objects: the 'generic' container.

```
struct GenericContainer {
    GenericContainerInterface *vTable;
    size_t Size;
    unsigned Flags;
    size_t ElementSize;
};
```

1. **Vtable.** All containers in the sample implementation contain a pointer to the table of functions of their interface.
2. **Size.** The number of elements this container stores.
3. **Flags.** Stores the state of the container. The only flag the sample implementation uses is the `READ_ONLY_FLAG` but many others are possible, for instance a 'locked' flag for multi-threading access, or a 'copy on write' flag for lazy copy, etc.

4. **ElementSize.** All containers in the sample implementation can store objects of the same size. This is not really a limitation since you can store objects of ANY size by storing a pointer in the container. An alternative design would store objects of any size but it would need to store the size of each object in addition to the data used by the object. The specialized containers like bitstrings, string collections or integer/double arrays do not need this field obviously, and its presence is optional.

7.1.2 Lists

Single linked lists use a single pointer to the next element. The data for the element comes right behind that pointer to avoid the overhead that yet another pointer would represent.

```
typedef struct _list_element {
    struct _list_element *Next;
    char Data[MINIMUM_ARRAY_INDEX];    // See below
} list_element;
```

The list header uses this structure to store the elements¹. As you can see, there is no space wasted in a pointer to the element stored. The element stored is placed just behind the `Next` pointer. The downside of this decision is that we can't recycle this object to store other different objects of different size.

```
struct _List {
    ListInterface *VTable;
    size_t count;
    unsigned Flags;
    unsigned timestamp;
    size_t ElementSize;
    list_element *Last;
    list_element *First;
    CompareFunction Compare;
    ErrorFunction RaiseError;
    ContainerHeap *Heap;
    ContainerMemoryManager *Allocator;
};
```

In the public `containers.h` header file we refer always to an abstract structure `_List`. We define it here. This schema allows other implementation to use the same header with maybe radically different implementations of their data structure.

¹The constant `MINIMUM_ARRAY_INDEX` is defined as 1 if we are compiling in C90 mode or as nothing if we are compiling in C99 mode. In C99 mode we have a flexible structure, that consists of a fixed and a variable part. The fixed part is the pointer to the next element. The variable part is the object we are storing in the list.

1. **Vtable, count, Flags, ElementSize.** These fields were described in the generic container section.
2. **timestamp.** This field is incremented at each modification of the list, and allows the iterators to detect if the container changes during an iteration: they store the value of this field at the start of the iteration, and before each iteration they compare it with its current value. If there are any changes, they return `NULL`.
3. **Last.** Stores a pointer to the last element of the list. This allows the addition of an element at the end of the list to be fast, avoiding a complete rescan of the list. This field is an optimization, all algorithms of a single linked list would work without this field.
4. **First.** The start of the linked list.
5. **Compare.** A comparison function for the type of elements stored in the list.
6. **RaiseError.** A function that will be called when an error occurs. This field is necessary only if you want to keep the flexibility of having a different error function for each list that the client software builds. An alternative implementation would store a pointer to an error function in the interface.
7. **Allocator.** A set of functions that allocates memory for this list. In an implementation that needs less flexibility and is more interested in saving space it could be replaced by the default allocator.

The sample implementation has certainly a quite voluminous header because of a design decision to keep things very flexible. Other implementations could trim most of the fields, and an absolute minimal implementation would trim **Last**, **Compare**, **RaiseError**, **Heap**, and **Allocator**. If the implementation assumes that only one iterator per container is allowed, the **timestamp** field could be replaced by a single bit ('changed') in the **Flags** field.²

7.1.3 Double linked lists

This container has a very similar structure to the single linked ones

```
typedef struct _dlist_element {
    struct _dlist_element *Next;
    struct _dlist_element *Previous;
    char Data[MINIMUM_ARRAY_INDEX];
} dlist_element;
```

²The function `newContainer` would clear the 'changed' bit, and the iterator functions would test if it is still clear. All modifications function would set it to one. This simple schema becomes problematic when you consider what happens when an invalid iterator is used again. In the simple one bit schema if the flag has been cleared, the iterator goes on, in the more expensive schema of the sample implementation, the stalled iterators are never restartable until the counter wraps around to the same value.

We have now two pointers followed by the stored data. All other fields are exactly identical to the ones in the single linked list. The single difference is the existence of a free list. This could have been done in the single linked list implementation too.

```
struct Dlist {
    DlistInterface *VTable;
    size_t count;
    unsigned Flags;
    unsigned timestamp;
    size_t ElementSize;
    dlist_element *Last;
    dlist_element *First;
    dlist_element *FreeList;
    CompareFunction Compare;
    ErrorFunction RaiseError;
    ContainerHeap *Heap;
    ContainerMemoryManager *Allocator;
};
```

7.1.4 Vector

Arrays are the containers that use the smallest overhead per element: zero. The only overhead is the header structure, whose cost is amortized since it is fixed for all elements that the array can hold.

This is a 'flexible' array however, what means that there is some spare space allocated for allowing further growth, and that different allocation strategies can be followed when allocating a new chunk of array space when the existing array is full.

```
struct _Vector {
    VectorInterface *VTable;
    size_t count;
    unsigned int Flags;
    size_t ElementSize;
    void *contents;
    size_t capacity;
    unsigned timestamp;
    CompareFunction CompareFn;
    ErrorFunction RaiseError;
    ContainerMemoryManager *Allocator;
} ;
```

1. Vtable, count, Flags, ElementSize. This fields were described in the generic container section.
2. CompareFn, RaiseError, timestamp and Allocator were described in the List container.

3. `capacity`. Stores the number of elements this container can hold without resizing.
4. `contents`. Points to an array of `capacity` elements, each of size `ElementSize`.

7.1.5 Dictionary

This container consists of an array of single linked lists. It could have been done with an `Vector` of `List` containers but a dedicated implementation is justified because of a greater efficiency. The advantages of the `Vector` container (secured access, flexible expansion) are not needed since the array has a fixed length that never changes.

```
struct _Dictionary {
    DictionaryInterface *VTable;
    size_t count;
    unsigned Flags;
    size_t size;
    ErrorFunction RaiseError;
    unsigned timestamp;
    size_t ElementSize;
    ContainerMemoryManager *Allocator;
    unsigned (*hash)(const unsigned char *Key);
    struct DataList {
        struct DataList *Next;
        unsigned char *Key;
        char *Value;
    } **buckets;
};
```

1. `Vtable`, `count`, `Flags`, `ElementSize`. These fields were described in the generic container section.
2. `RaiseError`, `timestamp` and `Allocator` were described in the `List` container.
3. `capacity`. Stores the number of elements this container can hold without resizing.
4. `size`. The number of different lists that the hash table can contain. This is normally a prime number.
5. `hash`. A hash function for character strings.
6. `buckets`. A table of pointers to lists of `DataList` structures.

7.1.6 String collection

String collections are just flexible arrays of pointers to C character strings. They share all the fields of the `Vector` container, the only specific field is a context that is passed to

the string comparison function. This context can contain flags or other information to use with special text encodings (wide characters for instance) or other data like regular expressions, etc.

```
struct StringCollection {
    StringCollectionInterface *VTable;
    size_t count;
    unsigned int Flags;
    unsigned char **contents;
    size_t capacity;
    size_t timestamp;
    ErrorFunction RaiseError;
    StringCompareFn strcmpare;
    CompareInfo *StringCompareContext;
    ContainerMemoryManager *Allocator;
};
```

7.1.7 The iterator data structure

This data structure has two main parts:

- A public part declared in `containers.h`:

```
typedef struct _Iterator {
    void *(*GetNext)(struct _Iterator *);
    void *(*GetPrevious)(struct _Iterator *);
    void *(*GetFirst)(struct _Iterator *);
    void *(*GetCurrent)(struct _Iterator *);
    void *(*GetLast)(struct _Iterator *);
    void *(*CopyCurrent)(struct _Iterator *);
} Iterator;
```

This part contains only the functions that the interface offers.

- A private, container specific part that comes right behind the public part and stores additional information that is needed for each container. For instance the list container will add following fields:

```
struct ListIterator {
    Iterator it;
    List *L;
    size_t index;
    list_element *Current;
    size_t timestamp;
    char ElementBuffer[1];
};
```

User code should only see and use the public part, as if the iterator was only the public part. Internally all iterator functions are completely different functions, specific for the container they should iterate. It looks like from user code, as you were always calling the same function because the syntax and name is the same. This allows for a certain abstraction in the source code that uses these functions, allowing to express a whole range of algorithms in terms of general concepts.

Each of the functions that implement `GetNext` `GetFirst`, etc starts with a cast of the input argument that is declared as an `Iterator` structure to a concrete container iterator like our `ListIterator` above.

In all those structures there is a common ground. They have:

1. A pointer to the container the iterator is using.
2. Some fields for storing the current position within the container, i.e. a cursor.
3. A `timestamp` field to detect if the container has changed during the iteration.
4. A buffer that allows the iterator to store an element of the container before returning a pointer to this area that contains a copy of the current element instead of a pointer directly to the element data. This allows to maintain the read only semantics.

There is currently no way to know when you delete a container if there are iterators that are using it. This could be detected by simply having a counter of the number of iterators a container has, but that would mean more overhead for the already fat header objects...

7.2 The code

Only one container will be shown here in full: the `List` container. For the others, only some functions will be explained to save space. You are invited to read the distributed code of course that is part of this work.

7.2.1 List

Add

```
static int Add(List *l,void *elem)
{
    list_element *newl;

    /* Error checking ellided */
    newl = new_link(l,elem,"iList.Add");
    if (newl == NULL) return CONTAINER_ERROR_NOMEMORY;
    if (l->count == 0) {                /* 1 */
```

```
        l->First = newl;
    }
    else {
        l->Last->Next = newl;
    }
    l->Last = newl;
    l->timestamp++;
    ++l->count;                /* 2 */
    return 1;
}
```

This function adds one element at the end. If the list is empty it just establishes the start of the list, if not, it adds it after the last element and makes the new list element the last. Errors leave the list unchanged. Exclusive access to the list is needed between the point marked (1) and the point marked (2) in the code. This operation is a modification of the list, and it needs to update the `timestamp` value to notify possible iterators that they are invalid.

AddRange

```
static int AddRange(List * AL, size_t n, void *data)
{
    unsigned char *p;
    list_element *oldLast;

    /* Error checking snipped */
    p = data;
    oldLast = AL->Last;
    while (n > 0) {
        int r = Add(AL, p);
        if (r < 0) {
            AL->Last = oldLast;
            if (AL->Last)
                AL->Last->Next = NULL;
            return r;
        }
        p += AL->ElementSize;
        n--;
    }
    return 1;
}
```

This function calls repeatedly `Add` for each element of the given array. Note that at compile time we do not know the size of each element and we can't index into this array. We just setup a generic pointer to the start of the data area, and increment it by the

size of each element at each iteration. This implementation supposes that the size of the elements as assumed by the list is the same as the size of then element as assumed by the calling program.

If an error occurs when adding elements the new elements are discarded.

Append ---

```
static int Append(List *l1,List *l2)
{
    /* Error checking elided */
    if (l1->count == 0) {
        l1->First = l2->First;
        l1->Last = l2->Last;
    }
    else if (l2->count > 0) {
        if (l2->First)
            l1->Last->Next = l2->First;
        if (l2->Last)
            l1->Last = l2->Last;
    }
    l1->count += l2->count;
    l1->timestamp++;
    l2->Allocator->free(l2);
    return 1;
}
```

This function adds the second argument list to the first one. The second list is destroyed because all its elements are inserted into the first one. The result is obtained by pointer manipulation: no data is moved at all, and any pointers to the objects in the second list remain valid.

Apply ---

```
static int Apply(List *L,int (Applyfn)(void *,void *),void *arg)
{
    list_element *le;
    void *pElem=NULL;

    /* Null error checking ellided */
    le = L->First;
    if (L->Flags&CONTAINER_LIST_READONLY) {
        pElem = malloc(L->ElementSize);
        if (pElem == NULL) {
            L->RaiseError("iList.Apply",CONTAINER_ERROR_NOMEMORY);
        }
    }
}
```

```
        return CONTAINER_ERROR_NOMEMORY;
    }
}
while (le) {
    if (pElem) {
        memcpy(pElem, le->Data, L->ElementSize);
        Applyfn(pElem, arg);
    }
    else Applyfn(le->Data, arg);
    le = le->Next;
}
if (pElem)
    free(pElem);
return 1;
}
```

This function calls the given function for each element. If the container is read only, a copy of each element is passed to the called function. This copy is allocated with "malloc" because it is used for internal purposes, and the standard allocator for the list could be a heap based, i.e. one that doesn't really free any memory. That could be a problem if repeated calls to **Apply** are done.

This function does not pass any pointer to the called function to mark the list as changed if the data passed to it is rewritten. This means that there is no way to let the called function inform the rest of the software of any modifications. This can be justified by the fact that only the data, not the container itself can be modified, but this can be tricky in multi-threaded environments. Other implementations could pass some pointer or away to inform the rest of the software that a modification has been done.

Clear

```
static int Clear(List *l)
{
    if (l->Heap)
        iHeap.Destroy(l->Heap);
    else {
        list_element *rvp = l->First, *tmp;
        while (rvp) {
            tmp = rvp;
            rvp = rvp->Next;
            l->Allocator->free(tmp);
        }
    }
    l->count = 0;
    l->Heap = NULL;
}
```



```
l->First = l->Last = NULL;
l->Flags = 0;
l->timestamp = 0;
return 1;
}
```

This function should clear all stored elements and reset some fields of the header structure so that the resulting list header is almost the same as when it was created. The only difference is that any functions like the comparison function or the error function are not cleared. If they were changed by the user they still remain changed.

Copy

```
static List *Copy(List *l)
{
    List *result;
    list_element *elem,*newElem;

    /* Null error checking ellided */
    result = iList.CreateWithAllocator(l->ElementSize,l->Allocator);
    if (result == NULL) {
        l->RaiseError("iList.Copy",CONTAINER_ERROR_NOMEMORY);
        return NULL;
    }
    result->Flags = l->Flags; /* Same flags */
    result->VTable = l->VTable; /* Copy possibly subclassed methods */
    result->Compare = l->Compare; /* Copy compare function */
    result->RaiseError = l->RaiseError;
    elem = l->First;
    while (elem) {
        newElem = new_link(result,elem->Data,"iList.Copy");
        if (newElem == NULL) {
            l->RaiseError("iList.Copy",CONTAINER_ERROR_NOMEMORY);
            result->VTable->Finalize(result);
            return NULL;
        }
        if (elem == l->First) {
            result->First = newElem;
        }
        else {
            result->Last->Next = newElem;
        }
        result->Last = newElem;
        elem = elem->Next;
    }
}
```

```
        result->count++;
    }
    return result;
}
```

This function requires a non null list pointer. It creates a header structure, and fills some of its fields with the corresponding fields of the source list:

1. The allocator
2. The flags.
3. The table of functions. This is necessary in case some of those functions have been sub-classed.
4. The comparison function
5. The error function

If an error occurs during the copy, probably because of lack of memory, the new list is destroyed and the result is NULL. Otherwise elements are added at the growing end of the list.

Contains

```
static int Contains(List *l,void *data)
{
    size_t idx;
    return (IndexOf(l,data,NULL,&idx) < 0) ? 0 : 1;
}
```

The Contains function is just a cover function for IndexOf.

CopyElement

```
static int CopyElement(List *l,size_t position,void *outBuffer)
{
    list_element *rvp;

    /* Error checking ellided */
    rvp = l->First;
    while (position) {
        rvp = rvp->Next;
        position--;
    }
    memcpy(outBuffer,rvp->Data,l->ElementSize);
    return 1;
}
```

After the error checking, this function positions at the given element and copies its contents into the given buffer. Other designs are obviously possible.

- This function could return a newly allocated buffer. This poses other problems like the type of allocator to use. If we use the list allocator we could run into problems if it is a specialized allocator that is designed for allocating list elements from a pool where no 'free' operation exists. Another, more important problem with that solution is that it forces an allocation when none is necessary if the buffer you use is stack based.
- The function could require the buffer length to be sure there are no buffer overflows. This solution was discarded because it actually increases the chances of errors: you have to pass the size of the buffer, and if you pass the wrong one more problems arise. Is it an error if you pass more space than is actually needed? It could be an error if the passed size differs from the size of the elements stored or it could be just a consequence that you used the `sizeof(buffer)` expression with a bigger buffer than necessary.

Create

```
static List *Create(size_t elementsize)
{
    return CreateWithAllocator(elementsize, CurrentMemoryManager);
}
```

This function just calls `CreateWithAllocator` using the current memory manager.

CreateWithAllocator

```
static List *CreateWithAllocator(size_t elementsize,
                                ContainerMemoryManager *allocator)
{
    List *result;

    if (elementsiz == 0) {
        iError.RaiseError("iList.Create", CONTAINER_ERROR_BADARG);
        return NULL;
    }
    result = allocator->malloc(sizeof(List));
    if (result == NULL) {
        iError.RaiseError("iList.Create", CONTAINER_ERROR_NOMEMORY);
        return NULL;
    }
    memset(result, 0, sizeof(List));
    result->ElementSize = elementsize;
}
```

```
    result->VTable = &iList;
    result->Compare = DefaultListCompareFunction;
    result->RaiseError = iError.RaiseError;
    result->Allocator = allocator;
    return result;
}
```

After doing some error checking, the creation function allocates and initializes the new container with its default values.

A big question is the alignment problem for the given size. This can't be checked and could lead to problems if you pass to this function any argument that is not the product of a sizeof expression.

DefaultListCompareFunction

```
static int DefaultListCompareFunction(const void *left,
                                     const void *right,
                                     CompareInfo *ExtraArgs)
{
    size_t siz=((List *)ExtraArgs->Container)->ElementSize;
    return memcmp(left,right,siz);
}
```

The default element compare function is just a cover for `memcmp`. It is assumed that the user will replace it with a comparison function of its own if necessary.

DefaultListLoadFunction

```
static size_t DefaultLoadFunction(void *element,void *arg, FILE *Infile)
{
    size_t len = *(size_t *)arg;

    return fread(element,1,len,Infile);
}
```

This function just reads an element from the disk file. Returns the result value of `fread`, what is OK for our purposes.

DefaultSaveFunction

```
static size_t DefaultSaveFunction(const void *element,void *arg,
                                  FILE *Outfile)
{
    const unsigned char *str = element;
    size_t len = *(size_t *)arg;
```

```
    return fwrite(str,1,len,Outfile);
}
```

This function just writes the given element to the disk. Together with the default load function they allow for a very effective serialization package for containers. Obviously here we have a shallow copy, and all this will never work for recursive saves, i.e. for elements that contain pointers.

deleteIterator

```
static int deleteIterator(Iterator *it)
{
    struct ListIterator *li;
    List *L;

    if (it == NULL) {
        iError.RaiseError("deleteIterator",CONTAINER_ERROR_BADARG);
        return CONTAINER_ERROR_BADARG;
    }
    li = (struct ListIterator *)it;
    L = li->L;
    L->Allocator->free(it);
    return 1;
}
```

This routine retrieves the list header object from the hidden part of the iterator and uses its allocator object to free the memory used by the iterator.

The functions `newIterator` and `deleteIterator` should occur in pairs like many others in C: `malloc` and `free`, `fopen` and `fclose`, etc. It would be very easy to have in the header object a counter of iterators that should be zero when the list is destroyed or cleared.

Equal

```
static int Equal(List *l1,List *l2)
{
    list_element *link1,*link2;
    CompareFunction fn;
    CompareInfo ci;

    if (l1 == l2)
        return 1;
    if (l1 == NULL || l2 == NULL)
        return 0;
    if (l1->count != l2->count)
```

```
        return 0;
    if (l1->ElementSize != l2->ElementSize)
        return 0;
    if (l1->Compare != l2->Compare)
        return 0;
    if (l1->count == 0)
        return 1;
    fn = l1->Compare;
    link1 = l1->First;
    link2 = l2->First;
    ci.Container = l1;
    ci.ExtraArgs = NULL;
    while (link1 && link2) {
        if (fn(link1->Data,link2->Data,&ci))
            return 0;
        link1 = link1->Next;
        link2 = link2->Next;
    }
    if (link1 || link2)
        return 0;
    return 1;
}
```

If two null pointers are passed to the `Equal` function it returns true. This is a design decision: `Equal` doesn't have any error result. Either the two objects are equal or not.

A redundant test is done at the end of the function: if the lists have the same count and all elements are equal, `link1` and `link2` should be `NULL`. If they aren't that means there is a memory overwrite problem somewhere...

Erase

```
static int Erase(List *l,void *elem)
{
    size_t idx;
    int i;

    if (l == NULL) {
        iError.RaiseError("iList.Erase",CONTAINER_ERROR_BADARG);
        return CONTAINER_ERROR_BADARG;
    }
    if (elem == NULL) {
        l->RaiseError("iList.Erase",CONTAINER_ERROR_BADARG);
        return CONTAINER_ERROR_BADARG;
    }
}
```

```
    if (l->count == 0) {
        return CONTAINER_ERROR_NOTFOUND;
    }
    i = IndexOf(l,elem,NULL,&idx);
    if (i < 0)
        return i;
    return RemoveAt(l,idx);
}
```

This is a very inefficient implementation. The list will be traversed twice, the first by `IndexOf`, and the second by `RemoveAt`. The obvious solution is to merge both into one function.

EraseRange

```
static int EraseRange(List *l,size_t start,size_t end)
{
    list_element *rvp,*start_pos,*tmp;
    size_t toremove;
    if (end > l->count)
        end = l->count;
    if (start >= l->count)
        return 0;
    if (start >= end)
        return 0;
    toremove = end - start+1;
    rvp = l->First;
    while (rvp && start > 1) {
        rvp = rvp->Next;
        start--;
    }
    start_pos = rvp;
    rvp = rvp->Next;
    while (toremove > 1) {
        tmp = rvp->Next;
        if (l->Heap)
            iHeap.AddToFreeList(l->Heap,rvp);
        else {
            l->Allocator->free(rvp);
        }
        rvp = tmp;
        toremove--;
        l->count--;
    }
}
```

```
    start_pos->Next = rvp;
    return 1;
}
```

This function positions the cursor ³ at the element before the one where the range starts, and then erases until it reaches the end of the range.

Finalize

```
static int Finalize(List *l)
{
    int t=0;

    t = Clear(l);
    if (t < 0)
        return t;
    l->Allocator->free(l);
    return 1;
}
```

This function should free the memory used by the header object. It is fundamental that this will never be done with an object not allocated with that iterator in the first place, i.e. when the user has called **Init** instead of **Create**. This can't be tested in a portable manner since there is no function to verify that a given memory space belongs or not to a given allocator.⁴

GetCurrent

```
static void *GetCurrent(Iterator *it)
{
    struct ListIterator *li = (struct ListIterator *)it;

    if (li->L->count == 0)
        return NULL;
    if (li->index == (size_t)-1) {
        li->L->RaiseError("GetCurrent",CONTAINER_ERROR_BADARG);
        return NULL;
    }
    if (li->L->Flags & CONTAINER_LIST_READONLY) {
        return li->ElementBuffer;
    }
    return li->Current->Data;
}
```

³Very often I use the name "rvp" for **roving pointer**

⁴This has been discussed several times in the comp.lang.c discussion group, but the committee never followed any of those proposals


```
}
```

Returns the current object pointed by the given iterator. This function should be called only after `GetFirst` is called. It verifies this by testing if a correct value is stored in the `index` field. This value is stored by the `newIterator` function. This simple algorithm avoids the usage of an uninitialized iterator at the cost of one integer comparison per call.

GetFirst

```
static void *GetFirst(Iterator *it)
{
    struct ListIterator *li = (struct ListIterator *)it;
    List *L;

    L = li->L;
    if (L->count == 0)
        return NULL;
    if (li->timestamp != L->timestamp) {
        L->RaiseError("iList.GetFirst",CONTAINER_ERROR_OBJECT_CHANGED);
        return NULL;
    }
    li->index = 0;
    li->Current = L->First;
    if (L->Flags & CONTAINER_LIST_READONLY) {
        memcpy(li->ElementBuffer,L->First->Data,L->ElementSize);
        return li->ElementBuffer;
    }
    return L->First->Data;
}
```

This function should set the iteration at the first element of the container, ready to get the iteration started. After the error checking phase it returns a pointer to the data in the first element, or a pointer to a copy of that data if the container is read only.

GetFlags

```
static unsigned GetFlags(List *l)
{
    if (l == NULL) {
        iError.RaiseError("iList.GetFlags",CONTAINER_ERROR_BADARG);
        return (unsigned)CONTAINER_ERROR_BADARG;
    }
    return l->Flags;
}
```

```
}
```

Just returns the value of the flags.

GetNext ---

```
static void *GetNext(Iterator *it)
{
    struct ListIterator *li = (struct ListIterator *)it;
    List *L;
    void *result;

    if (li == NULL) {
        iError.RaiseError("iList.GetNext",CONTAINER_ERROR_BADARG);
        return NULL;
    }
    L = li->L;
    if (li->index >= (L->count-1) || li->Current == NULL)
        return NULL;
    if (li->L->count == 0)
        return NULL;
    if (li->timestamp != L->timestamp) {
        L->RaiseError("GetNext",CONTAINER_ERROR_OBJECT_CHANGED);
        return NULL;
    }
    li->Current = li->Current->Next;
    li->index++;
    if (L->Flags & CONTAINER_LIST_READONLY) {
        memcpy(li->ElementBuffer,li->Current->Data,L->ElementSize);
        return li->ElementBuffer;
    }
    result = li->Current->Data;
    return result;
}
```

Advances the cursor to the next element and returns either a pointer to it or a pointer to a copy if the list is read only. The test for the cursor being NULL avoids using `GetNext` with an uninitialized iterator.

GetPrevious ---

```
static void *GetPrevious(Iterator *it)
{
    struct ListIterator *li = (struct ListIterator *)it;
```

```
List *L;
list_element *rvp;
size_t i;

L = li->L;
if (li->index >= L->count || li->index == 0)
    return NULL;
if (li->timestamp != L->timestamp) {
    L->RaiseError("GetPrevious",CONTAINER_ERROR_OBJECT_CHANGED);
    return NULL;
}
rvp = L->First;
i=0;
li->index--;
if (li->index > 0) {
    while (rvp && i < li->index) {
        rvp = rvp->Next;
        i++;
    }
}
li->Current = rvp;
return rvp->Data;
}
```

There were heated discussions about this function. In single linked lists it is necessary to go through the whole list at each call to this function. This is extremely inefficient and its usage should be avoided, it is much better to use double linked lists if you are interested in bi-directional cursor positioning. In the other hand this should be a required iterator feature, and rather than filling this function pointer with a function that just returns an error, the user is better served with a function that actually returns the previous item. Besides for short lists the performance lost is quite small, and would justify using lists with smaller overhead per item.⁵.

GetRange

```
static List *GetRange(List *l,size_t start,size_t end)
{
    size_t counter;
    List *result;
    list_element *rvp;;

    result = iList.Create(l->ElementSize);
```

⁵But then, if the lists are small, the greater overhead of the double linked lists is small too. You see, there were a lot of good arguments from both sides

```
result->VTable = l->VTable;
if (l->count == 0)
    return result;
if (end >= l->count)
    end = l->count;
if (start > end || start > l->count)
    return NULL;
if (start == l->count-1)
    rvp = l->Last;
else {
    rvp = l->First;
    counter = 0;
    while (counter < start) {
        rvp = rvp->Next;
        counter++;
    }
}
while (start < end && rvp != NULL) {
    int r = result->VTable->Add(result,&rvp->Data);
    if (r < 0) {
        Finalize(result);
        result = NULL;
        break;
    }
    rvp = rvp->Next;
    start++;
}
return result;
}
```

A new list is constructed from the given range of elements. The elements are copied. Any error during the construction of the new list provokes a NULL result: the copied elements are destroyed. Only correctly constructed ranges are returned. A recurring problem arises because it is impossible to report any details about the error that stops the copy. The result is actually boolean, either everything worked and there is a non NULL result, or something didn't.

An alternative design would have an integer return code, and a pointer to a result. This option was discarded because it is cumbersome and the most likely reason for `Add` to fail is lack of memory.

IndexOf

```
static int IndexOf(List *l,void *ElementToFind,
                  void *ExtraArgs,size_t *result)
```

```

{
    list_element *rvp;
    int r,i=0;
    CompareFunction fn;
    CompareInfo ci;

    if (l == NULL || ElementToFind == NULL) {
        if (l)
            l->RaiseError("iList.IndexOf",CONTAINER_ERROR_BADARG);
        else
            iError.RaiseError("iList.IndexOf",CONTAINER_ERROR_BADARG);
        return CONTAINER_ERROR_BADARG;
    }
    rvp = l->First;
    fn = l->Compare;
    ci.Container = l;
    ci.ExtraArgs = ExtraArgs;
    while (rvp) {
        r = fn(&rvp->Data,ElementToFind,&ci);
        if (r == 0) {
            *result = i;
            return 1;
        }
        rvp = rvp->Next;
        i++;
    }
    return CONTAINER_ERROR_NOTFOUND;
}

```

The design of this function went through several iterations. The big problem was the result type: a `size_t`, that in most cases is an unsigned quantity. A negative error result then was out of the question. But then, how would you indicate an error? ⁶

A first solution was to return a 1 based index and reserve zero for the 'not found' value. That could work, but was the source of many bugs in the rest of the software when the value was used without decrementing it first.

A second solution was to reserve a value within the `size_t` range to represent the 'not found' result. That works, and it is doable, but produced other, more subtle, problems in the rest of the software since in all checks of a `size_t`, it could be that *this* `size_t` has a value that is actually the sentinel value of `IndexOf`: the tests tended to multiply and the handling of those tests started to become a problem.

Here you see the third iteration: the function receives a pointer to a `size_t` that will be set if the function returns with a result greater than zero.

⁶The function `Contains` started its life as a way of avoiding all this problems

Another, completely different issue is the fact that in lists, this function is inefficient since it forces the function that uses the result to restart a list traversal to access the *nth* element. Much more efficient would be to do something immediately with the result, or to return a list element that allows the calling software to use it without going again through the list.

Problems with those solutions is that they are not portable, and that they would expose the inner workings of the list container to the users. The `list_element` structure is not even mentioned in the public `containers.h`.

InitWithAllocator

```
static List *InitWithAllocator(List *result, size_t elementsize,
                               ContainerMemoryManager *allocator)
{
    if (elementsize == 0) {
        iError.RaiseError("iList.Init", CONTAINER_ERROR_BADARG);
        return NULL;
    }
    memset(result, 0, sizeof(List));
    result->ElementSize = elementsize;
    result->VTable = &iList;
    result->Compare = DefaultListCompareFunction;
    result->RaiseError = iError.RaiseError;
    result->Allocator = allocator;
    return result;
}
```

This function initializes a piece of storage to a list container. This allows the user to use stack storage for the list container, saving an allocation from the heap, and the corresponding need to free that storage.

Init

```
static List *Init(List *result, size_t elementsize)
{
    return InitWithAllocator(result, elementsize, CurrentMemoryManager);
}
```

Uses the current memory manager to call `InitWithAllocator`.

InsertAt

```
static int InsertAt(List *l, size_t pos, void *pdata)
{
    list_element *elem;
```

```

if (l == NULL || pdata == NULL) {
    if (l)
        l->RaiseError("iList.InsertAt",CONTAINER_ERROR_BADARG);
    else
        iError.RaiseError("iList.InsertAt",CONTAINER_ERROR_BADARG);
    return CONTAINER_ERROR_BADARG;
}
if (pos > l->count) {
    l->RaiseError("iList.InsertAt",CONTAINER_ERROR_INDEX);
    return CONTAINER_ERROR_INDEX;
}
if (l->Flags & CONTAINER_LIST_READONLY) {
    l->RaiseError("iList.InsertAt",CONTAINER_ERROR_READONLY);
    return CONTAINER_ERROR_READONLY;
}
if (pos == l->count) {
    return l->VTable->Add(l,pdata);
}

elem = new_link(l,pdata,"iList. InsertAt");
if (elem == NULL) {
    l->RaiseError("iList.InsertAt",CONTAINER_ERROR_NOMEMORY);
    return CONTAINER_ERROR_NOMEMORY;
}
if (pos == 0) {
    elem->Next = l->First;
    l->First = elem;
}
else {
    list_element *rvp = l->First;
    while (--pos > 0) {
        rvp = rvp->Next;
    }
    elem->Next = rvp->Next;
    rvp->Next = elem;
}
l->count++;
l->timestamp++;
return 1;
}

```

This inserts before the given index. It would have been equally possible to insert after, that is a more or less random decision.

InsertIn

```
static int InsertIn(List *l, size_t idx, List *newData)
{
    size_t newCount;
    list_element *le,*nle;

    if (idx > l->count) {
        l->RaiseError("iList.InsertIn",CONTAINER_ERROR_INDEX);
        return CONTAINER_ERROR_INDEX;
    }
    if (l->ElementSize != newData->ElementSize) {
        l->RaiseError("iList.InsertIn",CONTAINER_ERROR_INCOMPATIBLE);
        return CONTAINER_ERROR_INCOMPATIBLE;
    }
    if (newData->count == 0)
        return 1;
    newData = Copy(newData);
    if (newData == NULL) {
        l->RaiseError("iList.InsertIn",CONTAINER_ERROR_NOMEMORY);
        return CONTAINER_ERROR_NOMEMORY;
    }
    newCount = l->count + newData->count;
    if (l->count == 0) {
        l->First = newData->First;
        l->Last = newData->Last;
    }
    else {
        le = l->First;
        while (le && idx > 1) {
            le = le->Next;
            idx--;
        }
        nle = le->Next;
        le->Next = newData->First;
        newData->Last->Next = nle;
    }
    newData->Allocator->free(newData);
    l->timestamp++;
    l->count = newCount;
    return 1;
}
```

Inserts the given list at the specified position.

1. Error checking. First argument must be non NULL and read/write. Second must be non NULL .
2. If the position given is exactly the same as the length of the receiving list, the second list is just appended to the first one.
3. Otherwise search the position and insert a copy of the elements in the second list.

Load

```
static List *Load(FILE *stream, ReadFunction loadFn,void *arg)
{
    size_t i,elemSize;
    List *result,L;
    char *buf;
    int r;
    guid Guid;

    if (loadFn == NULL) {
        loadFn = DefaultLoadFunction;
        arg = &elemSize;
    }
    if (fread(&Guid,sizeof(guid),1,stream) <= 0) {
        iError.RaiseError("iList.Load",CONTAINER_ERROR_FILE_READ);
        return NULL;
    }
    if (memcmp(&Guid,&ListGuid,sizeof(guid))) {
        iError.RaiseError("iList.Load",CONTAINER_ERROR_WRONGFILE);
        return NULL;
    }
    if (fread(&L,1,sizeof(List),stream) <= 0) {
        iError.RaiseError("iList.Load",CONTAINER_ERROR_FILE_READ);
        return NULL;
    }
    elemSize = L.ElementSize;
    buf = malloc(L.ElementSize);
    if (buf == NULL) {
        iError.RaiseError("iList.Load",CONTAINER_ERROR_NOMEMORY);
        return NULL;
    }
    result = iList.Create(L.ElementSize);
    if (result == NULL) {
        iError.RaiseError("iList.Load",CONTAINER_ERROR_NOMEMORY);
        return NULL;
    }
}
```

```
    }
    result->Flags = L.Flags;
    r = 1;
    for (i=0; i < L.count; i++) {
        if (loadFn(buf,arg,stream) <= 0) {
            r = CONTAINER_ERROR_FILE_READ;
            break;
        }
        if ((r=Add(result,buf)) < 0) {
            break;
        }
    }
    free(buf);
    if (r < 0) {
        iError.RaiseError("iList.Load",r);
        iList.Finalize(result);
        result = NULL;
    }
    return result;
}
```

The load function is long and complex. As always, the process starts with error checking. All streams written to by its counterpart **Save** are marked with a container specific globally unique identifier (GUID). This ensures that a load function from the list container will not crash if passed a file that belongs to an array or a dictionary.

Then, the header object is read, what gives the data to continue the process, since we now know the number of elements and the size of each element.

A new list is created with the given element size, and we start reading *count* elements from the stream. Any error provokes the destruction of the elements read so far and a result of NULL.

newIterator

```
static Iterator *newIterator(List *L)
{
    struct ListIterator *result;

    if (L == NULL) {
        iError.RaiseError("iList.newIterator",CONTAINER_ERROR_BADARG);
        return NULL;
    }
    result = L->Allocator->malloc(sizeof(struct ListIterator));
    if (result == NULL) {
        L->RaiseError("iList.newIterator",CONTAINER_ERROR_NOMEMORY);
    }
}
```

```
        return NULL;
    }
    result->it.GetNext = GetNext;
    result->it.GetPrevious = GetPrevious;
    result->it.GetFirst = GetFirst;
    result->it.GetCurrent = GetCurrent;
    result->L = L;
    result->timestamp = L->timestamp;
    result->index = (size_t)-1;
    result->Current = NULL;
    return &result->it;
}
```

The creation of a new iterator involves just allocating and initializing values to their defaults.

PopFront

```
static int PopFront(List *l, void *result)
{
    list_element *le;

    if (l->count == 0)
        return 0;
    le = l->First;
    if (l->count == 1) {
        l->First = l->Last = NULL;
    }
    else l->First = l->First->Next;
    l->count--;
    if (result)
        memcpy(result, &le->Data, l->ElementSize);
    if (l->Heap) {
        iHeap.AddToFreeList(l->Heap, le);
    }
    else l->Allocator->free(le);
    l->timestamp++;
    return 1;
}
```

Contrary to most versions of this function, **PopFront** does not return the data of the element but stores it in a pointer that it receives. If the pointer is `NULL`, the data is just discarded.

The problem with returning a pointer to the first element, is that the user code should remember to discard it when no longer needed, and it should discard it using

the same allocator that the list used to allocate it. That would be a very error prone interface.

PushFront

```
static int PushFront(List *l,void *pdata)
{
    list_element *rvp;

    rvp = new_link(l,pdata,"Insert");
    if (rvp == NULL)
        return CONTAINER_ERROR_NOMEMORY;
    rvp->Next = l->First;
    l->First = rvp;
    if (l->Last == NULL)
        l->Last = rvp;
    l->count++;
    l->timestamp++;
    return 1;
}
```

Lists are a good base to implement a stack. PushFront and PopFront take a constant and small time to complete and they would be much smaller if we would eliminate the error checking.

RemoveAt

```
static int RemoveAt(List *l,size_t position)
{
    list_element *rvp,*last,*removed;

    rvp = l->First;
    if (position == 0) {
        removed = l->First;
        if (l->count == 1) {
            l->First = l->Last = NULL;
        }
        else {
            l->First = l->First->Next;
        }
    }
    else if (position == l->count - 1) {
        while (rvp->Next != l->Last)
            rvp = rvp->Next;
    }
}
```

```
        removed = rvp->Next;
        rvp->Next = NULL;
        l->Last = rvp;
    }
    else {
        last = rvp;
        while (position > 0) {
            last = rvp;
            rvp = rvp->Next;
            position --;
        }
        removed = rvp;
        last->Next = rvp->Next;
    }
    if (l->Heap) {
        iHeap.AddToFreeList(l->Heap,removed);
    }
    else l->Allocator->free(removed);
    l->timestamp++;
    --l->count;
    return 1;
}
```

The operation when `RemoveAt` is called with the index of the last element is equivalent to the `PopBack` function, that is absent in the single linked list interface. After much discussions, we decided that the generic interface would have only `Push` and `Pop`, and that each container would fill those functions with the most efficient implementation available for it. For lists, the most efficient implementation is `PopFront` and `PushFront`. For arrays, the most efficient is `PushBack` and `PopBack`. For double linked lists is either.

ReplaceAt

```
static int ReplaceAt(List *l,size_t position,void *data)
{
    list_element *rvp;

    if (position == l->count-1)
        rvp = l->Last;
    else {
        rvp = l->First;
        while (position) {
            rvp = rvp->Next;
            position--;
        }
    }
    *rvp = *data;
}
```

```

    }
}
memcpy(&rvp->Data , data,l->ElementSize);
l->timestamp++;
return 1;
}

```

After error checking (not shown), position the cursor at the right item, then copy from the given data pointer the element size bytes needed.

An open issue is whether the "timestamp" field should be changed. Nothing in the list structure has been changed, only the data stored in the container. Any iterators will go on working as advertised even if this function is called to replace many items in the list. In the other hand, if user programs were making assumptions about the data (for instance a search function doesn't always look again at past items to see if they have been changed) this could bad consequences. As a rule, any change will provoke the incrementing of the "timestamp" counter.

Reverse

```

static int Reverse(List *l)
{
    list_element *New,*current,*old;

    if (l->count < 2)
        return 1;
    old = l->First;
    l->Last = l->First;
    New = NULL;
    while (old) {
        current = old;
        old = old->Next;
        current->Next = New;
        New = current;
    }
    l->First = New;
    l->Last->Next = NULL;
    l->timestamp++;
    return 1;
}

```

After the error checking, the list is reversed in place if the count of its element is bigger than 1.⁷

⁷Looks easy isn't it? It isn't. It took me a while to arrive at the code above. Even worst is the reversing of a double linked list

Save

```
static int Save(List *L, FILE *stream, SaveFunction saveFn, void *arg)
{
    size_t i;
    list_element *rvp;

    if (saveFn == NULL) {
        saveFn = DefaultSaveFunction;
        arg = &L->ElementSize;
    }

    if (fwrite(&ListGuid, sizeof(guid), 1, stream) <= 0)
        return EOF;

    if (fwrite(L, 1, sizeof(List), stream) <= 0)
        return EOF;
    rvp = L->First;
    for (i=0; i< L->count; i++) {
        char *p = rvp->Data;

        if (saveFn(p, arg, stream) <= 0)
            return EOF;
        rvp = rvp->Next;
    }
    return 1;
}
```

The format of the saved list container is:

1. The GUID of the list container: 128 bytes
2. The Header object
3. The data for all the elements of the list

Seek

```
static void *Seek(Iterator *it, size_t idx)
{
    struct ListIterator *li = (struct ListIterator *)it;
    list_element *rvp;

    if (li->L->count == 0)
```

```
        return NULL;

    rvp = li->L->First;
    if (idx >= li->L->count-1) {
        li->index = li->L->count-1;
        li->Current = li->L->Last;
    }
    else if (idx == 0) {
        li->index = 0;
        li->Current = li->L->First;
    }
    else {
        li->index = idx;
        while (idx > 0) {
            rvp = rvp->Next;
            idx--;
        }
        li->Current = rvp;
    }
    return li->Current;
}
```

This function positions the given iterator at the desired position. Several alternatives are possible, for instance position the iterator at a given item. This can be obtained now only by calling first `IndexOf`, then `Seek`, what forces to go through the list twice.

SetCompareFunction

```
static CompareFunction SetCompareFunction(List *l, CompareFunction fn)
{
    CompareFunction oldfn = l->Compare;

    if (l == NULL) {
        iError.RaiseError("iList.SetCompareFunction",
                          CONTAINER_ERROR_BADARG);
        return NULL;
    }
    if (fn != NULL) {
        if (l->Flags & CONTAINER_LIST_READONLY) {
            l->RaiseError("iList.SetCompareFunction",
                          CONTAINER_LIST_READONLY);
        }
        else l->Compare = fn;
    }
}
```



```
    return oldfn;
}
```

This function returns the old value of the comparison function and sets it to the new one, if the new one is not NULL . This allows to query the comparison function without changing it, avoiding yet another trivial function like `GetComparisonFunction`. This is just what in other languages like Objective C or others is called a *property* of the `iList` object. Objective C makes all this automatic with its `synthesize` directive.

In C there isn't any such hand holding and you have to write that code yourself. There are several other functions in the same style like `SetErrorFunction`, `Size` (that returns the `count` field) and `SetFlags`. They aren't listed here but you can look at the code by browsing through the `list.c` file distributed with this software.

Sizeof

```
static size_t Sizeof(List *l)
{
    if (l == NULL) {
        return sizeof(List);
    }

    return sizeof(List) +
        l->ElementSize * l->count +
        l->count *sizeof(list_element);
}
```

Returns the number of bytes used by the given list, including the data, and all overhead. For lists, tghis is the size of the header object, and for each element the overhead of a pointer to the next element and the size of each stored object. With a NULL list pointer returns the size of the list header object, what allows you to allocate buffers containing a header object and use the `Init` function.

Sort

```
static int Sort(List *l)
{
    list_element **tab;
    size_t i;
    list_element *rvp;
    CompareInfo ci;

    if (l == NULL) {
        iError.RaiseError("iList.Sort",CONTAINER_ERROR_BADARG);
        return CONTAINER_ERROR_BADARG;
    }
}
```

```
    if (l->count < 2)
        return 1;
    if (l->Flags&CONTAINER_LIST_READONLY) {
        l->RaiseError("iList.Sort",CONTAINER_ERROR_READONLY);
        return CONTAINER_ERROR_READONLY;
    }
    tab = l->Allocator->malloc(l->count * sizeof(list_element *));
    if (tab == NULL) {
        l->RaiseError("iList.Sort",CONTAINER_ERROR_NOMEMORY);
        return CONTAINER_ERROR_NOMEMORY;
    }
    rvp = l->First;
    for (i=0; i<l->count;i++) {
        tab[i] = rvp;
        rvp = rvp->Next;
    }
    ci.Container = l;
    ci.ExtraArgs = NULL;
    qsortEx(tab,l->count,sizeof(list_element *),lcompar,&ci);
    for (i=0; i<l->count-1;i++) {
        tab[i]->Next = tab[i+1];
    }
    tab[l->count-1]->Next = NULL;
    l->Last = tab[l->count-1];
    l->First = tab[0];
    l->Allocator->free(tab);
    return 1;
}
```

This function basically builds an array and calls quicksort, nothing really fancy. Note that it calls a modified version of the library function quicksort, since it needs to pass a context to it for the comparison function. The default comparison function is listed below:

```
static bool lcompar (const void *elem1, const void *elem2,
                    CompareInfo *ExtraArgs)
{
    list_element *Elem1 = *(list_element **)elem1;
    list_element *Elem2 = *(list_element **)elem2;
    List *l = (List *)ExtraArgs->Container;
    CompareFunction fn = l->Compare;
    return fn(Elem1->Data,Elem2->Data,ExtraArgs);
}
```

The default comparison function pulls the list compare function and calls it with the extra arguments needed to pass a context to it.

UseHeap

```
static int UseHeap(List *L, ContainerMemoryManager *m)
{
    if (L == NULL) {
        iError.RaiseError("iList.UseHeap",CONTAINER_ERROR_BADARG);
        return CONTAINER_ERROR_BADARG;
    }
    if (L->Heap || L->count) {
        L->RaiseError("UseHeap",CONTAINER_ERROR_NOT_EMPTY);
        return CONTAINER_ERROR_NOT_EMPTY;
    }
    if (m == NULL)
        m = CurrentMemoryManager;
    L->Heap = iHeap.Create(L->ElementSize+sizeof(list_element), m);
    return 1;
}
```

This function installs a heap to be used by the list. This is very important for huge lists, since performance goes quickly down if you call malloc for each element you add to the list. Basically, the heap is just a way to allocate memory in blocks so that malloc calls are reduced.

7.2.2 Queues

Queues are, to use the C++ terminology, *adaptor* containers, i.e. containers based on other containers, in this case a list. We describe here an implementation with the objective to show how those adaptors can be implemented, and how you can restrain the interface of the underlying container with a small cost.

The data structure used is very simple:

```
typedef struct _Queue {
    QueueInterface *VTable;
    List *Items;
} _Queue;
```

Just two fields: the interface and the underlying list. We do not document here some functions of the queue interface that trivially call the corresponding List functions.

Back

```
static int Back(Queue *Q,void *result)
{
```

```

size_t idx;
if (Q == NULL) {
    iError.RaiseError("iQueue.Front",CONTAINER_ERROR_BADARG);
    return CONTAINER_ERROR_BADARG;
}
idx = iList.Size(Q->Items);
if (idx == 0)
    return 0;
return iList.CopyElement(Q->Items,idx-1,result);
}

```

Returns the last element of the queue. We do not want to have any errors issued by the underlying list, so we test for NULL . We use the size as an index, except of course when the queue is empty.

CreateWithAllocator

```

static Queue *CreateWithAllocator(size_t ElementSize,
                                   ContainerMemoryManager *allocator)
{
    Queue *result = allocator->malloc(sizeof(Queue));

    if (result == NULL)
        return NULL;
    result->Items = iList.CreateWithAllocator(ElementSize,allocator);
    if (result->Items == NULL) {
        allocator->free(result);
        return NULL;
    }
    result->VTable = &iQueue;
    return result;
}

```

Using the given allocator, we get memory for the Queue object, then for the list using the given allocator.

Finalize

```

static int Finalize(Queue *Q)
{
    ContainerMemoryManager *allocator = iList.GetAllocator(Q->Items);
    iList.Finalize(Q->Items);
    allocator->free(Q);
    return 1;
}

```

We should free the queue header object with the same allocator we used for the list. We obtain it first, before we free the list.

Front

```
static int Front(Queue *Q,void *result)
{
    size_t idx;
    if (Q == NULL) {
        iError.RaiseError("iQueue.Front",CONTAINER_ERROR_BADARG);
        return CONTAINER_ERROR_BADARG;
    }
    idx = iList.Size(Q->Items);
    if (idx == 0)
        return 0;
    return iList.CopyElement(Q->Items,0,result);
}
```

Same as Back. We make the error checking to avoid errors when accessing the list.

Sizeof

```
static size_t Sizeof(Queue *q)
{
    if (q == NULL) return sizeof(Queue);
    return sizeof(*q) + iList.Sizeof(q->Items);
}
```

If passed a NULL queue, we return the size of the Queue header object. Note that we do not return the size of the underlying list even if it has been allocated and uses up space. An alternative design would have required to take into account the list header as it would have been part of the overhead of the Queue object. But in that case we could never know the size of the Queue itself...

7.2.3 The dictionary

Dictionary is an instance of a hash table where the key is supposed to contain character strings (names) that are associated with some data. Hash tables are normal tables that are indexed by a hash function, i.e. a function that maps character strings into some integer that is used to index the table. At each slot of the table we find a linked list of elements that were classified by the hash function into the same slot. If we have a good hash function, i.e. one that spreads evenly the elements across the table, we can have a speed up for searching an element of the order of the table size, in the best case.

Hashing

One of the important aspects of a dictionary implementation is to use a good hash function, i.e. one that distributes evenly the keys. I have picked up for this work one of the most used functions of this type. Here is the documentation I found for this function in the Apache runtime:

This is the popular ‘times 33’ hash algorithm which is used by perl and that also appears in Berkeley DB. This is one of the best known hash functions for strings because it is both computed very fast and distributes very well.

The originator may be Dan Bernstein but the code in Berkeley DB cites Chris Torek as the source. The best citation I have found is “Chris Torek, Hash function for text in C, Usenet message <27038@mimsy.umd.edu> in comp.lang.c , October, 1990.” in Rich Salz’s USENIX 1992 paper about INN which can be found at <http://citeseer.nj.nec.com/salz92internetnews.html>.

The magic of number 33, i.e. why it works better than many other constants, prime or not, has never been adequately explained by anyone. So I try an explanation: if one experimentally tests all multipliers between 1 and 256 (as I did while writing a low-level data structure library some time ago) one detects that even numbers are not useable at all. The remaining 128 odd numbers (except for the number 1) work more or less all equally well. They all distribute in an acceptable way and this way fill a hash table with an average percent of approx. 86%.

If one compares the χ^2 values of the variants (see Bob Jenkins “Hashing FAQ” at <http://burtleburtle.net/bob/hash/hashfaq.html> for a description of χ^2), the number 33 not even has the best value.

But the number 33 and a few other equally good numbers like 17, 31, 63, 127 and 129 have nevertheless a great advantage to the remaining numbers in the large set of possible multipliers: their multiply operation can be replaced by a faster operation based on just one shift plus either a single addition or subtraction operation. And because a hash function has to both distribute good *and* has to be very fast to compute, those few numbers should be preferred.

– Ralf S. Engelschall <rse@engelschall.com>

Julienne Walker has another twist to this story. She says:⁸

Bernstein hash

⁸In the very interesting web page http://eternallyconfuzzled.com/tuts/algorithms/jsw_tut.hashing.aspx In that page she also proposes to replace the addition operation with an XOR operations. She says that that improves the algorithm.

Dan Bernstein created this algorithm and posted it in a newsgroup. It is known by many as the Chris Torek hash because Chris went a long way toward popularizing it. Since then it has been used successfully by many, but despite that the algorithm itself is not very sound when it comes to avalanche and permutation of the internal state. It has proven very good for small character keys, where it can outperform algorithms that result in a more random distribution.

Bernstein's hash should be used with caution. It performs very well in practice, for no apparently known reasons (much like how the constant 33 does better than more logical constants for no apparent reason), but in theory it is not up to snuff. Always test this function with sample data for every application to ensure that it does not encounter a degenerate case and cause excessive collisions.

hash

```
static unsigned int hash(const unsigned char *key)
{
    unsigned int Hash = 0;
    const unsigned char *p;

    for (p = key; *p; p++) {
        Hash = Hash * 33 + scatter[*p];
    }
    return Hash;
}
```

Note that I have slightly modified the algorithm by using a scatter table of 256 positions filled with random numbers. The objective is to avoid that letters that appear frequently in the text would tend to cluster the keys in the same position.

This default function may not be the best for the data in the user's application. The library has reserved a field in the dictionary header object for a pointer to a hash function that can be changed by the user.

Creation

Another important aspect of the dictionary implementation is the decision of how many slots the table should have. I have followed the recommendations of Dave Hanson in his Book "C interfaces and Implementations"⁹, and I use a small table of primes to decide what size the table should have:

Init

⁹C Interfaces and Implementations, David R. Hanson, Addison Wesley. ISBN 0-201-49841-3 3rd printing June 2001 page 149

```
static Dictionary *Init(Dictionary *Dict,
                        size_t elementszize,size_t hint)
{
    size_t i,allocSiz;
    static unsigned primes[] = { 509, 509, 1021, 2053, 4093, 8191,
                                16381, 32771, 65521, 131071, 0 };
    for (i = 1; primes[i] < hint && primes[i] > 0; i++)
        ;
    allocSiz = sizeof (Dictionary);
    memset(Dict,0,allocSiz);
    allocSiz = primes[i-1]*sizeof (Dict->buckets[0]);
    Dict->buckets = CurrentMemoryManager->malloc(allocSiz);
    if (Dict->buckets == NULL) {
        return NULL;
    }
    memset(Dict->buckets,0,allocSiz);
    Dict->size = primes[i-1];
    Dict->hash = hash;
    Dict->VTable = &iDictionary;
    Dict->ElementSize = elementszize;
    Dict->Allocator = CurrentMemoryManager;
    Dict->RaiseError = iError.RaiseError;
    return Dict;
}
```

The primes in the table are the nearest primes to the regular powers of two. Table sizes can range from 509 to more than 130000, what gives a really wide range of table sizes. Obviously, bigger tables could be necessary, and other specialized implementations could use the *hint* parameter to extend this algorithm or to use a completely different algorithm altogether.

Adding elements

This operation consists of:

- hash the key to find a slot
- go through the list at that slot to see if the key is already there
- if key is already there replace
- if key is absent add it in a new list item

Add _____

```

static int Add(Dictionary *Dict, const unsigned char *Key, void *Value)
{
    size_t i;
    struct DataList *p;
    unsigned char *tmp;

    if (Dict == NULL)
        return NullPtrError("Add");
    if (Dict->Flags & CONTAINER_READONLY)
        return ReadOnlyError(Dict, "Add");
    if (Key == NULL || Value == NULL)
        return BadArgError(Dict, "Add");
    i = (*Dict->hash)(Key) % Dict->size;
    for (p = Dict->buckets[i]; p; p = p->Next) {
        if (strcmp(Key, p->Key) == 0)
            break;
    }
    Dict->timestamp++;
    if (p == NULL) {
        p = Dict->Allocator->malloc(sizeof(*p)+Dict->ElementSize);
        tmp = Dict->Allocator->malloc(1+strlen((char *)Key));
        if (p == NULL || tmp == NULL) {
            if (p) Dict->Allocator->free(p);
            if (tmp) Dict->Allocator->free(tmp);
            return NoMemoryError(Dict, "Add");
        }
        p->Value = (void *) (p+1);
        strcpy(tmp, Key);
        p->Key = tmp;
        p->Next = Dict->buckets[i];
        Dict->buckets[i] = p;
        Dict->count++;
    }
    memcpy((void *) p->Value, Value, Dict->ElementSize);
    return 0;
}

```

Following the logical steps outlined above, we:

1. Call the hash function and use its result modulo the size of the slot table to fetch the list at the indicated slot.
2. See if the key was absent. If that is the case, we need to add a new key. We copy the key and allocate memory for a new list element that is initialized afterwards with the copied value of the key and inserted into the list.

3. Copy in the value. If it was a new key, its value is initialized, if the key was already present we overwrite the old contents.

This function uses `strcmp` for comparing keys. This has the advantage of simplicity and speed, but in many other contexts a key comparison function would be necessary, to allow for keys in Unicode for instance, or for binary keys, for instance a GUID or similar binary data.

An important design decision was to replace the data associated with a key if the key is already there. This is a decision that has consequences for all associative containers, since it must be coherent in all of them. Since the "Insert" function allows for non-destructive insertions, Add was allowed to replace contents since this is a very common operation for instance in some symbol tables, where "Insert if absent or replace if present" is used to ensure that a symbol is associated with a certain value.¹⁰ At the same time we need a **Replace** function since we want to get an error if the element we want to replace was **not** found. A small table makes this clearer

Add	Insert or replace an item for a key
Insert	Insert, error if the key was present
Replace	Replace, error if key was absent

Implementing iterators

Iterators in sequential containers are conceptually easy: just start at the first and stop at the last. In associative containers however things are more complicated since there is no obvious way to order them. The solution retained in the sample implementation involves going through all elements starting at the first element of the slots table, and for each slot go through the linked list of items if any. This guarantees to visit all elements in a fixed order. As an example of this here is the **Apply** function that should go through all elements calling the given function for each one of them.

Apply

```
static int Apply(Dictionary *Dict,
                  int (*apply)(const unsigned char *Key,
                               const void *Value,
                               void *ExtraArgs),
                  void *ExtraArgs)
{
    size_t i;
    unsigned stamp;
    struct DataList *p;

    if (Dict == NULL) {
```

¹⁰Note that the C++ `map::insert` does not replace an element

```

        return NullPtrError("Apply");
    }
    if (apply == NULL)
        return BadArgError(Dict,"Apply");
    stamp = Dict->timestamp;
    for (i = 0; i < Dict->size; i++) {
        for (p = Dict->buckets[i]; p; p = p->Next) {
            apply(p->Key,p->Value, ExtraArgs);
            if (Dict->timestamp != stamp)
                return 0;
        }
    }
    return 1;
}

```

As we outlined above, we start at slot zero, going upwards. If we find a non-empty slot, we go through the linked list of items.

Iterators are implemented using the same algorithm, and need conceptually two indexes to remember their position: a first index for the slots table, and another for the position in the list of items at that slot.

The implementation of the dictionary iterator is as follows:

```

struct DictionaryIterator {
    Iterator it;
    Dictionary *Dict;
    size_t index;
    struct DataList *dl;
    size_t timestamp;
    unsigned long Flags;
};

```

The `index` field remembers the position in the slot table, and the `dl` field is just a small structure that contains a link to the next item in the linked list and a pointer to the key. Storing the list element itself spare us the work of going through all the list to position ourselves at each advance of the cursor in the list.

7.2.4 The bloom filter

This container is a completely different beast as all other ones we have in the library. It is a probabilistic data structure. It was conceived by Mr Burton Howard Bloom in 1970 according to D. E Knuth in his Art of Computer Programming.

Bloom filters are designed to cheaply test if a given element is in a large set. It is possible that the filter says that an element is there when in fact, it is not. But if the filter says it is *not* there you can be ceratin that the element is not in the set.

You can add elements to the set but not remove them. The more elements you add to the filter, the larger the possibility of getting false positives, i.e. getting an answer of "yes, the element is there" when in fact it is not.

Debugging malloc

The library provides a sample of how a malloc used for debugging allocation problems could look like. It is designed to be enhanced and even if it has several important features like detection of double free and buffer overflows, it is not a competitor for the professional versions you can find in the market like valgrind or similar.

Malloc

```
static void *Malloc(size_t size)
{
    register char *r;
    register size_t *ip = NULL;

    size = ALIGN_DEFAULT(size);
    size += 3 * sizeof(size_t);
    r = malloc(size);
    if (r == NULL)
        return NULL;
    AllocatedMemory += size;
    ip = (size_t *) r;
    *ip++ = SIGNATURE;
    *ip++ = size;
    memset(ip, 0, size - 3*sizeof(size_t));
    ip = (size_t *) (&r[size - sizeof(size_t)]);
    *ip = MAGIC;
    return (r + 2 * sizeof(size_t));
}
```

The algorithm is as follows:

- The given size will be aligned to a multiple of `size_t`. It is assumed that this size is the size of a register, and will be good for any type of allocation. In some machines this may be completely wrong, for instance for some quantities the Intel processors need an alignment of 16 bytes, and there is no implementation of `size_t` with that size.
- We reserve three words more than the requested size to store:
 1. The "magic number". This is just an integer that will enable us to ensure that we are dealing with a valid block. Blocks that have this number two words below the address passed to our `Free` function will be assumed to be

real blocks. There is of course a chance that the memory could contain that number for other reasons, but choosing a value that can't be a pointer and that is high above 100 millions give us a fighting chance that the probability of hitting a bad positive is fairly low.

2. The length of the block. This will allow us to verify that nothing was written beyond the required length of the block.
 3. A guard at the end of the block. We will ensure that we can read this quantity when freeing the block.
- We obtain memory using `malloc`. If not available we just return `NULL`.
 - We keep a counter of all memory allocated so far. This counter should be zero at program exit. It helps to detect the leaks between two operations: it suffices to note the value of the counter before some part of the software and then see if the counter returns to the same value after the module has finished.
 - We write the two different integers at the start and at the end of the block, together with its size.
 - We set to zero all memory even if the program didn't ask us. This ensures that any error that accesses uninitialized memory will always have the same consequences.

The other functions that complete this memory manager (`free`, `realloc` `calloc`) are not shown here (they are available in the source code of the library). They just undo what `Malloc` has built, calling the error functions if they detect a problem.

This simple system has several drawbacks.

- If a buffer "underflow" happens, i.e. something is written to memory *before* the start of the block, our field "length" could be wrong. Depending on the resulting contents of the length field after the overwrite we could have a bogus length and access some invalid memory.
- Memory overwrites *after* the magic number that guards the end of the block are not detected. This is obviously impossible to detect unless we would just inspect each memory write, but a few words more after the end of the block could give us some extra security.

8 Building generic components

If you take the source code of a container like “arraylist”, for instance, you will notice that all those “void *” are actually a single type, i.e. the type of the objects being stored in the container. All generic containers use “void *” as the type under which the objects are stored so that the same code works with many different types.

Obviously another way is possible. You could actually replace the object type within that code and build a family of functions and types that can be specialized by its type parameter. For instance:

```
struct tag$(TYPE)ArrayInterface;
typedef struct _$(TYPE)Array {
    struct tag$(TYPE)ArrayInterface *VTable;
    size_t count;
    unsigned int Flags;
    $(TYPE) *contents;
    size_t capacity;
    size_t ElementSize;
    unsigned timestamp;
    CompareFunction CompareFn;
    ErrorFunction RaiseError;
} $(TYPE)_Array ;
```

Now, if we just substitute `$(TYPE)` with “**double**” in the code above, we obtain:

```
struct tagdoubleArrayInterface;
typedef struct _doubleArray {
    struct tagdoubleArrayInterface *VTable;
    size_t count;
    unsigned int Flags;
    double *contents;
    size_t capacity;
    size_t ElementSize;
    unsigned timestamp;
    CompareFunction CompareFn;
    ErrorFunction RaiseError;
} double_Array ;
```

We use the name of the parameter to build a family of names, and we use the name of the type parameter to declare an array of elements of that specific type as the contents of the array. This double usage allows us to build different name spaces for each different array type, so that we can declare arrays of different types without problems.

Using the same pattern, we can build a family of functions for this container that is specialized to a concrete type of element. For instance we can write:

```
static int EraseAt($(TYPE)_Array *AL,size_t idx)
{
    $(TYPE) *p;
    if (idx >= AL->count)
        return CONTAINER_ERROR_INDEX;
    if (AL->Flags & AL_READONLY)
        return CONTAINER_ERROR_READONLY;
    if (AL->count == 0)
        return -2;
    p = AL->contents+idx;
    if (idx < (AL->count-1)) {
        memmove(p,p+1,(AL->count-idx)*sizeof($(TYPE)));
    }
    AL->count--;
    AL->timestamp++;
    return AL->count;
}
```

when transformed, the function above becomes:

```
static int EraseAt(double_Array *AL,size_t idx)
{
    double *p;
    if (idx >= AL->count)
        return CONTAINER_ERROR_INDEX;
    if (AL->Flags & AL_READONLY)
        return CONTAINER_ERROR_READONLY;
    if (AL->count == 0)
        return -2;
    p = AL->contents+idx;
    if (idx < (AL->count-1)) {
        memmove(p,p+1,(AL->count-idx)*sizeof(double));
    }
    AL->count--;
    AL->timestamp++;
    return AL->count;
}
```

Now we can build a simple program in C that will do the substitution work for us. To make things easier, that program should build two files:

- The header file, that will contain the type definitions for our array.
- The C source file, containing all the parametrized function definitions.

We separate the commands to change the name of the file from the rest of the text by introducing in the first positions of a line a sequence of three or more @ signs. Normally we will have two of those “commands”: one for the header file, another for the c file.

Besides that, our program is just a plain text substitution. No parsing, nor anything else is required. If we write “\$(TYPE)” within a comment or a character string, it will be changed too.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define MAXLINE_LEN      2048
#define MAX_FNAME        1024
#define EXPANSION_LENGTH 256

int main(int argc, char *argv[])
{
    FILE *input, *output=NULL;
    char buf[MAXLINE_LEN],
        tmpLine[MAXLINE_LEN+EXPANSION_LENGTH];
    char tmpBuf[MAX_FNAME];
    char outputFile[MAX_FNAME];
    char *TypeDefinition;
    unsigned lineno = 1;

    if (argc < 3) {
        fprintf(stderr,
            "Usage: %s <template file to expand> <type name>\n",
            argv[0]);
        return EXIT_FAILURE;
    }
    input = fopen(argv[1], "r");
    if (input == NULL) {
        fprintf(stderr, "Unable to open file '%s'\n", argv[1]);
        return EXIT_FAILURE;
    }
    TypeDefinition = argv[2];
    while (fgets(buf, sizeof(buf)-1, input)) {
```

```
if (buf[0]=='@' && buf[1] == '@' && buf[2] == '@') {
    int i=0,j=0;
    while (buf[i] == '@')
        i++;
    while (buf[i] != 0 &&
        buf[i] != '\n' &&
        i < MAX_FNAME-1) {
        tmpBuf[j++] = buf[i];
        i++;
    }
    tmpBuf[j] = 0;
    if (strrepl(tmpBuf,"$(TYPE)",TypeDefinition,NULL)) {
        fprintf(stderr,"File name '%s' too long\n",
            tmpBuf);
        return EXIT_FAILURE;
    }
    strrepl(tmpBuf,"$(TYPE)",TypeDefinition,outputFile);
    if (output != NULL)
        fclose(output);
    output = fopen(outputFile,"w");
    if (output == NULL) {
        fprintf(stderr,
            "Impossible to open '%s'\n",outputFile);
        return(EXIT_FAILURE);
    }
}
else if (lineno == 1) {
    fprintf(stderr,
        "Error: First line should contain the file name\n");
    exit(EXIT_FAILURE);
}
else {
    /* Normal lines here */
    if (strrepl(buf,"$(TYPE)",TypeDefinition,NULL)
        >= sizeof(tmpLine)) {
        fprintf(stderr,
            "Line buffer overflow line %d\n",lineno);
        break;
    }
    strrepl(buf,"$(TYPE)",TypeDefinition,tmpLine);
    fwrite(tmpLine,1,strlen(tmpLine),output);
}
lineno++;
}
```

```

        fclose(input);
        fclose(output);
        return EXIT_SUCCESS;
}

```

The heart of this program is the “strrepl” function that replaces a given character string in a piece of text. If you call it with a NULL output parameter, it will return the number of characters that the replacement would need if any. For completeness, here is the code for strrepl:

```

int strrepl(const char *InputString, const char *StringToFind,
            const char *StringToReplace, char *output)
{
    char *offset = NULL, *CurrentPointer = NULL;
    int insertlen;
    int findlen = strlen(StringToFind);
    int result = 0;

    if (StringToReplace)
        insertlen = strlen(StringToReplace);
    else
        insertlen = 0;
    if (output) {
        if (output != InputString)
            memmove(output, InputString, strlen(InputString)+1);
        InputString = output;
    }
    else
        result = strlen(InputString)+1;

    while (*InputString) {
        offset = strstr (!offset ? InputString : CurrentPointer,
                        StringToFind);
        if (offset == NULL)
            break;
        CurrentPointer = (offset + (output ? insertlen : findlen));
        if (output) {
            strcpy (offset, (offset + findlen));
            memmove (offset + insertlen,
                    offset, strlen (offset) + 1);
            if (insertlen)
                memcpy (offset, StringToReplace, insertlen);
            result++;
        }
    }
}

```

```
        else {
            result -= findlen;
            result += insertlen;
        }
    }
    return result;
}
```

And now we are done. The usage of this program is very simple:

```
expand <template file> <type name>
```

For instance to substitute by “double” in the template file “arraylist.tpl” we would use:

```
expand arraylist.tpl double
```

We would obtain `doublearray.h` and `doublearray.c`

BUG: Obviously, this supposes that the type name does NOT contain any spaces or other characters like `*` or `[]`. If you want to use types with those characters you should substitute them with a `”_”` for instance, and make a typedef:

```
typedef long double long_double;
```

And use that type (`”long_double”`) as the substitution type.

Index

- Add, 48, 74, 91, 110, 122, 138, 145
 - code for hash, 198
 - code for list, 163
- AddRange, 49, 74, 102
 - code for list, 164
- AddToFreeList, 37
- Alloc, 39
- And, 92
- AndAssign, 93
- Append, 50, 75
 - code for list, 164
- Apply, 51, 75, 110, 123, 131
 - code for hash, 200
 - code for list, 165
- Back, 130, 132
 - code for list, 193
- BitBlockCount, 94
- BitString, 89
- Bloomfilter, 137
- Buffers, 140
- CalculateSpace, 138
- Calloc, 39
- CastToArray, 103
- Clear, 40, 52, 76, 111, 124, 132, 139, 141, 145
 - code for list, 166
- Contains, 52, 76, 78, 112, 132
 - code for list, 168
- Copy, 52, 77, 112, 124, 132
 - code for list, 167
- CopyBits, 95
- CopyElement, 53
 - code for list, 168
- CopyTo, 78
- Create, 37, 39, 53, 77, 112, 124, 132, 138, 141, 146
 - code for list, 169
- CreateFromFile, 103
- CreateWithAllocator, 54, 77, 112, 141, 145
 - code for list, 169, 194
- DefaultListCompareFunction
 - code for list, 170
- DefaultListLoadFunction
 - code for list, 170
- DefaultSaveFunction
 - code for list, 170
- deleteIterator, 54, 78, 112, 124
 - code for list, 170
- Deque, 130
- DestroyFreeList, 38
- Dictionary, 108
 - structure, 161
- Dlist, 68
 - structure, 159
- EmptyErrorFunction, 41
- Equal, 55, 78, 113, 133
 - code for list, 171
- Erase, 55, 79, 113, 124, 133
 - code for list, 172
- EraseAt, 55, 79
- EraseRange, 56
 - code for list, 173
- error-codes, 16
- Finalize, 38, 40, 57, 80, 113, 134, 139, 141, 146

- code for list, 173, 194
- Find, 139
- FindFirstText, 103
- FindNextText, 103
- FindTextPositions, 104
- Front, 129, 133
 - code for list, 194
- Generic Container
 - structure, 157
- GetAllocator, 57
- GetBits, 95
- GetCapacity, 80
- GetCurrent, 42, 43
 - code for list, 174
- GetData, 141
- GetElement, 57, 80, 114, 125
- GetElementSize, 57, 80, 114
- GetFirst, 42
 - code for list, 174
- GetFlags, 125, 134
 - code for list, 175
- GetFlags / SetFlags, 58, 81
- GetLast, 44
- GetList, 130
- GetNext, 43
 - code for list, 175
- GetPosition, 142
- GetPrevious, 43
 - code for list, 176
- GetRange, 58, 81, 95
 - code for list, 177
- hash
 - code for hash, 197
- HashTable, 121
- iAssociativeContainer, 150
- iBitString, 90
- iDeque, 131
- iDictionary, 109
- iDlist, 69
- iError, 40
- iGenericContainer, 148
- iHashTable, 121
- iHeap, 36
- iList, 45
- IndexIn, 82
- IndexOf, 59, 82
 - code for list, 178
- Init, 59, 104, 115
 - code for hash, 197
 - code for list, 180
- InitHeap, 37
- InitWithAllocator, 59, 104, 115
 - code for list, 179
- Insert, 115
- InsertAt, 60, 82
 - code for list, 180
- InsertIn, 60, 83, 104
 - code for list, 181
- iPool, 39
- iSequentiaContainer, 149
- iStringCollection, 101
- Iterator
 - structure, 162
- iterator
 - Dictionary, 201
- iTreeMap, 119
- iVector, 73
- LeftShift, 95
- List, 45
- ListIterator, 162
- lists
 - code, 163
 - double linked, 69
 - single linked, 45
 - structure, 158
- Load, 62, 84, 115, 125, 134
 - code for list, 182
- Malloc
 - code for debugMalloc, 201
- mapcar, 153
- mapcon, 155
- Merge, 125
- Mismatch, 85, 106
- newIterator, 62, 84, 116, 126

code for list, 184
newObject, 37
Not, 96
NotAssign, 96

ObjectToBitString, 96
Or, 96
OrAssign, 97
Overlay, 127

PeekFront, 146
PopBack, 70, 85, 107, 134
PopFront, 62, 135, 146
code for list, 185
PopulationCount, 97
Print, 97
PushBack, 135
PushFront, 63, 135
code for list, 185

Queue, 129

RaiseError, 40
Read, 142
RemoveAt, 98
code for list, 186
Replace, 127
ReplaceAt, 63, 86
code for list, 187
Resize, 127
Reverse, 64, 86, 97
code for list, 188

Save, 65, 86, 116, 127, 136
code for list, 188
Seek, 64
code for list, 189
Set, 99
SetAllocator, 65
SetCapacity, 87
SetCompareFunction, 65, 87
code for list, 190
SetErrorFunction, 41, 66, 87, 117, 128
SetPosition, 142
Size, 66, 88, 116, 117, 128, 142, 147
Sizeof, 38, 66, 88, 117, 128, 147
code for list, 191, 195
Sort, 66, 88
code for list, 191
Splice, 71
StrError, 41
StringCollection, 101
structure, 161
StringToBitString, 99

TreeMap, 119

UseHeap, 67
code for list, 192

Vector, 72
structure, 160

Write, 143
WriteToFile, 107

Xor, 99
XorAssign, 100