Spirick Tuning

A C++ Class and Template Library

for Performance Critical Applications

Reference Manual

SPIRICK

Version 1.48

November 2021

Important notice: Some parts of the documentation are under construction and incomplete.

Table of Contents

1 MEMORY MANAGEMENT

1.1 System Interface

1.1.1 Global Definitions (tuning/defs.hpp)

In the file **'tuning/defs.hpp'** compiler specific macros are evaluated and global data types and macros are defined. This file is included from all other header files of the library. At the end of the file optionally the file **'tl_user.hpp'** is included. That way the behavior of the library can be changed without changing the source code, e.g. the macro `TL_ASSERT` may be redefined.

Data Types

```
typedef ... t_Int;
typedef ... t_UInt;
typedef ... t_Int8;
typedef ... t_UInt8;
typedef ... t_Int16;
typedef ... t_UInt16;
typedef ... t_Int32;
typedef ... t_UInt32;
```

Numeric data types with a well-defined number of bits, signed or unsigned. The size of `t_Int` and `t_UInt` depends on the environment (32 or 64 bit).

1.1.2 Reserve Memory (tuning/sys/calloc.hpp)

With reserve memory the program can continue elementary operations in case of memory overflow. By using the reserve memory, there is no need to test each memory allocation for success. Reserve memory shall be allocated on program startup. If C standard library can't allocate any more memory, reserve memory will be released by `tl_Alloc` and `tl_Realloc`. Afterwards, `tl_HasReserve` returns `false`. Reserve memory management is protected against multiple thread access.

Memory Overflow

Many functions in the **Spirick Tuning** library allocate or reallocate memory. Within any function a memory overflow can occur. Handling each occurrence will increase program code and computing time. However memory overflows are very rare. The **Spirick Tuning** library is optimized for performance. Hence, memory overflow is handled exclusively in the `tl_Alloc` and `tl_Realloc` functions. **All other parts of the library assume success on memory allocations.**

A memory allocation or reallocation consists of the following steps: Try to allocate memory with C standard library (`malloc`, `realloc`). If it fails free reserve memory and call C standard library again. If it fails call overflow handler and call C standard library again. If it fails terminate the program with the function `tl_EndProcess`. In the last case it makes no sense to continue. Every following operation will probably fail because of lack of memory.

Data Types

`typedef void (* tpf_AllocHandler) ();`

Pointer to a gobal function taking no parameters and returning no value.

Functions

`tpf_AllocHandler tl_SetReserveHandler (tpf_AllocHandler pf_allocHandler);`

Sets new reserve handler and returns previous. Reserve handler is called if reserve memory is allocated, reallocated or released.

`tpf_AllocHandler tl_SetOverflowHandler (tpf_AllocHandler pf_allocHandler);`

Sets new overflow handler and returns previous. Overflow handler will be called if reserve memory is released and C standard library can't allocate any more memory. Within the **Spirick Tuning** library memory overflow is handled exclusively in the `tl_Alloc` and `tl_Realloc` functions. **All other parts of the library assume success on memory allocations.** Hence, overflow handler must not throw C++ exceptions. Exceptions from overflow handler are not handled by the library and lead to inconsistent objects.

`void tl_SetReserveSize (t_UInt u_resSize);`

Sets the size of the reserve memory to `u_resSize`. Afterwards, `tl_HasReserve` returns `true` on success.

`t_UInt tl_GetReserveSize ();`

Returns the size of the reserve memory, even if it is not allocated.

`bool tl_HasReserve ();`

Returns `true` if reserve memory is allocated.

`void tl_FreeReserve ();`

Frees reserve memory. Afterwards, `tl_HasReserve` returns `false`.

`void tl_AllocReserve ();`

Tries to allocate reserve memory. Afterwards, `tl_HasReserve` returns `true` on success.

1.1.3 Dynamic Memory (tuning/sys/calloc.hpp)

The system interface for memory allocations relies directly on C standard library. The global functions `malloc`, `realloc` and `free` are used. Debugging tools and heap walkers of the C standard library can be used together with the **Spirick Tuning** library. The functions `tl_Alloc` and `tl_Realloc` extend the C standard library with reserve memory.

Functions

`t_UInt tl_StoreInfoSize ();`

Returns the number of bytes for memory management per block. The value is used while calculating rounded block sizes.

`t_UInt tl_MaxAlloc ();`

Returns the maximum size of a contiguous memory block.

`void * tl_Alloc (t_UInt u_size);`

Allocates a contiguous memory block of size `u_size`. Returns null pointer if `u_size` is zero. On memory overflow reserve handler and overflow handler are called.

`void * tl_Realloc (void * pv_ptr, t_UInt u_size);`

Reallocates memory block pointed to by `pv_ptr` to size `u_size`. If `pv_ptr` is the null pointer, `tl_Realloc` is identical to `tl_Alloc`. If `u_size` is zero, `tl_Realloc` is identical to `tl_Free`. On memory overflow reserve handler and overflow handler are called.

`void tl_Free (void * pv_ptr);`

Frees memory block pointed to by `pv_ptr`. `pv_ptr` may be the null pointer.

Appropriate Classes

The classes `ct_StdStore`, `ct_RndStore` and `ct_ChnStore` rely on the global functions of this section.

1.1.4 Heap Operations (tuning/sys/calloc.hpp)

Debugging tools and heap walkers are not standardized. Hence, the system interface contains selected heap information only. The structure `st_HeapInfo` contains information about the number and the size of used and unused memory blocks. The number of unused memory blocks is a hint to memory fragmentation. Note that some C++ compilers don't publish heap information, especially in release mode.

Structure Declaration

```
struct st_HeapInfo
  {
  unsigned long       u_AllocEntries;
  unsigned long       u_FreeEntries;
  unsigned long       u_AllocSize;
  unsigned long       u_FreeSize;
  unsigned long       u_HeapSize;
  };
```

Functions

`bool tl_QueryHeapInfo (st_HeapInfo * pso_info);`

Stores information about the actual heap state in the structure pointed to by `pso_info`. Return value `false` is a hint to heap corruption.

`bool tl_FreeUnused ();`

Tries to free unused memory blocks. Return value `false` is a hint to heap corruption.

1.1.5 Memory Operations (tuning/sys/cmemory.hpp)

The system interface for memory operations relies directly on C standard library. Global functions like `memcpy` and `memcmp` are used. In addition, some special cases are handled, e.g. zero length parameters and null pointers. All parameters are checked by `ASSERT` macros. Length parameters refer to the number of characters, not to the size in bytes.

Functions

```
void tl_CopyMemory (char * pc_dst, const char * pc_src, t_UInt u_len);
void tl_CopyMemory (wchar_t * pc_dst, const wchar_t * pc_src, t_UInt u_len);
```

Copies u_len characters from pc_src to pc_dst. This function must not be used for overlapping memory blocks.

```
void tl_MoveMemory (char * pc_dst, const char * pc_src, t_UInt u_len);
void tl_MoveMemory (wchar_t * pc_dst, const wchar_t * pc_src, t_UInt u_len);
```

Copies u_len characters from pc_src to pc_dst. This function may be used for overlapping memory blocks.

```
char * tl_FillMemory (char * pc_dst, t_UInt u_len, char c_fill);
wchar_t * tl_FillMemory (wchar_t * pc_dst, t_UInt u_len, wchar_t c_fill);
```

Sets the first u_len characters of pc_dst to the character c_fill.

```
int tl_CompareChar (char c1, char c2);
int tl_CompareChar (wchar_t c1, wchar_t c2);
```

Compares the characters c1 and c2 and returns a value indicating their relationship. The return value is less than zero if c1 < c2, equal to zero if c1 == c2, and greater than zero if c1 > c2. The characters are compared as unsigned values.

```
int tl_CompareMemory (const char * pc1, const char * pc2, t_UInt u_len);
int tl_CompareMemory (const wchar_t * pc1, const wchar_t * pc2, t_UInt u_len);
```

Compares the first u_len characters of pc1 and pc2 and returns a value indicating their relationship. The return value is less than zero if *pc1 < *pc2, equal to zero if *pc1 == *pc2, and greater than zero if *pc1 > *pc2. The characters are compared as unsigned values.

```
const char * tl_FirstChar (const char * pc_mem, t_UInt u_len, char c_search);
const wchar_t * tl_FirstChar (const wchar_t * pc_mem, t_UInt u_len, wchar_t c_search);
```

If successful, it returns a pointer to the first occurrence of c_search in the first u_len characters of pc_mem. Otherwise it returns the null pointer.

```
const char * tl_FirstMemory (const char * pc_mem, t_UInt u_len, const char * pc_search, t_UInt u_searchLen);
const wchar_t * tl_FirstMemory (const wchar_t * pc_mem, t_UInt u_len, const wchar_t * pc_search, t_UInt
u_searchLen);
```

If successful, it returns a pointer to the first occurrence of the first u_searchLen characters of pc_search in the first u_len characters of pc_mem. Otherwise it returns the null pointer.

```
const char * tl_LastChar (const char * pc_mem, t_UInt u_len, char c_search);
const wchar_t * tl_LastChar (const wchar_t * pc_mem, t_UInt u_len, wchar_t c_search);
```

If successful, it returns a pointer to the last occurrence of c_search in the first u_len characters of pc_mem. Otherwise it returns the null pointer.

```
const char * tl_LastMemory (const char * pc_mem, t_UInt u_len, const char * pc_search, t_UInt u_searchLen);
const wchar_t * tl_LastMemory (const wchar_t * pc_mem, t_UInt u_len, const wchar_t * pc_search, t_UInt
u_searchLen);
```

If successful, it returns a pointer to the last occurrence of the first u_searchLen characters of pc_search in the first u_len characters of pc_mem. Otherwise it returns the null pointer.

```
template <t_UInt u_len>
  void tl_SwapMemory (void * pv1, void * pv2);
```

Swap the contents of the two memory blocks pv1 and pv2 with size u_len bytes.

```
template <class t_obj>
  void tl_SwapObj (t_obj & o1, t_obj & o2);
```

Swap the values of the two objects o1 and o2 using operator =. A third local object is used.

Appropriate Classes

The templates gct_CharBlock and gct_String rely on the global functions of this section.

1.2 Store

1.2.1 Store Interface

Stores are memory management objects. To increase performance there is no common base class with virtual functions. However, all store classes share a common interface. So it's easy to switch between multiple store implementations. To avoid compiler errors, all store classes contain all methods of the common interface. Methods not supported by a specific store class contain the statement ASSERT (false).

Class Declaration

```
class ct_AnyStore
  {
public:
  typedef t_UInt       t_Size;
  typedef void *       t_Position;

  void                 Swap (ct_AnyStore & co_swap);
  t_UInt               StoreInfoSize ();
  t_UInt               MaxAlloc ();

  t_Position           Alloc (t_Size o_size);
  t_Position           Realloc (t_Position o_pos, t_Size o_size);
  void                 Free (t_Position o_pos);

  void *               AddrOf (t_Position o_pos);
  t_Position           PosOf (void * pv_adr);

  t_Size               SizeOf (t_Position o_pos);
  t_Size               RoundedSizeOf (t_Position o_pos);

  bool                 CanFreeAll ();
  void                 FreeAll ();
  };
```

Data Types

typedef t_UInt **t_Size**;

The nested type t_Size describes the size of memory blocks, examples are t_UInt, t_UInt8, t_UInt16 and t_UInt32. If t_Size is defined as t_UInt8, the maximum size of a memory block will be 255 bytes and objects containing size information will require less space.

typedef void * **t_Position**;

Store objects use position values to manage their memory blocks, examples are void *, t_UInt, t_UInt8, t_UInt16 and t_UInt32. <u>The position value zero is invalid per definition.</u> The method AddrOf returns the memory address of a position value. If the position type is void *, the position value may (or may not) be equal to the memory address. Hence, always use the method AddrOf for memory access and do not use the position value itself.

Methods

`void Swap (ct_AnyStore & co_swap);`

Swaps the values of the two objects.

`t_UInt StoreInfoSize ();`

Returns the number of bytes for memory management per block. This method is not supported by all store classes.

`t_UInt MaxAlloc ();`

Returns the maximum size of a contiguous memory block.

`t_Position Alloc (t_Size o_size);`

Allocates a contiguous memory block of size `u_size`. Returns zero if `u_size` is zero. On memory overflow reserve handler and overflow handler are called.

`t_Position Realloc (t_Position o_pos, t_Size o_size);`

Reallocates memory block pointed to by `o_pos` to size `u_size`. If `o_pos` is zero, `Realloc` is identical to `Alloc`. If `u_size` is zero, `Realloc` is identical to `Free`. On memory overflow reserve handler and overflow handler are called.

`void Free (t_Position o_pos);`

Frees memory block pointed to by `o_pos`. `o_pos` may be zero.

`void * AddrOf (t_Position o_pos);`

Returns the memory address of position value `o_pos`. If `o_pos` is zero it returns the null pointer.

`t_Position PosOf (void * pv_adr);`

Returns the position value of memory address `pv_adr`. This method is not supported by all store classes.

`t_Size SizeOf (t_Position o_pos);`

Returns exactly the size of the memory block pointed to by `o_pos`. This method is not supported by all store classes.

`t_Size RoundedSizeOf (t_Position o_pos);`

Returns the rounded size of the memory block pointed to by `o_pos`. This method is not supported by all store classes.

`bool CanFreeAll ();`

Returns `true` if the store class can free all allocated memory blocks.

`void FreeAll ();`

Frees all allocated memory blocks. This method is not supported by all store classes.

1.2.2 Global Stores (tuning/defs.hpp)

Stores are used very differently within the **Spirick Tuning** library. The three dynamic stores (see following sections) are accessed by generated global wrapper classes (using a global store object). For example, in most cases there is no need to create multiple round stores. The parameters of one global round store object may be applied to the entire program.

Numerous class templates take a store class as parameter and create a store instance. For example, every list container allocates the node memory by its own store object. A block list container has a local block store. A normal list container uses a wrapper class to access a global store object.

There are four wrapper classes for each global store object. Each wrapper class has its own `t_Size` data type. All methods of a wrapper class are declared static. They can be called directly (class::method, e.g. in `gct_Block`) or by a wrapper object (object.method, e.g. in `gct_DList`).

A method of a wrapper class calls the appropriate method of the global store object. If the position value is equal to the memory address, then the `AddrOf` method is implemented inline in the wrapper class.

Each global store object has its own global access function. The global object is created in the first call of the access function. This technique ensures safe access to store objects from constructors of global C++ objects. A global store object may be created directly by a global `Create` function.

Global store objects are not destroyed automatically during program termination. This technique ensures safe access to store objects from destructors of global C++ objects. The destruction of global store objects is not necessary. They manage raw memory blocks, and this memory is released by the OS automatically. A global store object may be destroyed directly by a global `Delete` function.

Note that a heap walker may report the global store objects as memory leaks at the end of the program. This problem can be avoided by explicitly deleting these objects. Please ensure that a global store object is not used after deleting it.

GLOBAL_STORE_DCLS`(t_store, Obj, inl_or_stat)`

This macro appears at the end of the store class definition. `t_store` is the original store class. `Obj` is a small identifier for name generation. Multiple wrapper classes are generated. `inl_or_stat` determines whether the `AddrOf` and `PosOf` methods are implemented inline or static. The macro usage

```
GLOBAL_STORE_DCLS (ct_AnyStore, My, INLINE)
```

contains the following declarations:

```
void CreateMyStore ();
void DeleteMyStore ();
ct_AnyStore * GetMyStore ();
class ct_My_Store;
class ct_My8Store;
class ct_My16Store;
class ct_My32Store;
```

GLOBAL_STORE_DEFS`(t_store, Obj, inl_or_stat)`

This macro appears in the store class implementation file and contains the same parameters as `GLOBAL_STORE_DCLS`. The generated code contains the implementation of the wrapper class methods.

1.2.3 Wrapper Class Example

The entire declaration of the wrapper class `ct_My16Store` read as follows:

```
class ct_My16Store
  {
public:
  typedef t_UInt16                t_Size;
  typedef ct_AnyStore::t_Position t_Position;
  typedef ct_AnyStore             t_Store;

  static void             Swap (ct_My16Store &);
  static t_UInt           StoreInfoSize ();
```

```
  static t_UInt           MaxAlloc ();
  static t_Position       Alloc (t_Size o_size);
  static t_Position       Realloc (t_Position o_pos, t_Size o_size);
  static void             Free (t_Position o_pos);
  static inline void *    AddrOf (t_Position o_pos) { return o_pos; }
  static inline t_Position PosOf (void * pv_adr) { return pv_adr; }
  static t_Size           SizeOf (t_Position o_pos);
  static t_Size           RoundedSizeOf (t_Position o_pos);
  static bool             CanFreeAll ();
  static void             FreeAll ();
  static ct_AnyStore *    GetStore ();
  };
```

The macro GLOBAL_STORE_DEFS generates three global access functions. For performance reasons, the construction and destruction of global store objects are not thread-safe. These actions should be done at program startup/termination in single-thread mode.

```
static ct_AnyStore * pco_MyStore;
void CreateMyStore ()
  {
  if (pco_MyStore == 0)
    pco_MyStore = new ct_AnyStore;
  }
void DeleteMyStore ()
  {
  if (pco_MyStore != 0)
    {
    delete pco_MyStore;
    pco_MyStore = 0;
    }
  }
ct_AnyStore * GetMyStore ()
  {
  if (pco_MyStore == 0)
    CreateMyStore ();
  return pco_MyStore;
  }
```

The generated definition of ct_My16Store:: Alloc read as follows:

```
ct_My16Store::t_Position
ct_My16Store::Alloc (t_Size o_size)
  { return GetMyStore ()-> Alloc (o_size); }
```

1.3 Dynamic Stores

1.3.1 Standard Store (tuning/std/store.hpp)

ct_StdStore is the simplest store class. The global C functions of the system interface are mapped to the C++ class interface. For example, the Alloc method calls the global tl_Alloc function.

Class Declaration

```
class ct_StdStore
  {
public:
  typedef t_UInt          t_Size;
  typedef void *          t_Position;
  static inline void      Swap (ct_StdStore & co_swap);
```

```
  static inline t_UInt     StoreInfoSize ();
  static inline t_UInt     MaxAlloc ();

  static inline t_Position Alloc (t_Size o_size);
  static inline t_Position Realloc (t_Position o_pos, t_Size o_size);
  static inline void       Free (t_Position o_pos);

  static inline void *     AddrOf (t_Position o_pos);
  static inline t_Position PosOf (void * pv_adr);

  static inline t_Size     SizeOf (t_Position o_pos);
  static inline t_Size     RoundedSizeOf (t_Position o_pos);

  static inline bool       CanFreeAll ();
  static inline void       FreeAll ();
  };

inline ct_StdStore::t_Position ct_StdStore::Alloc (t_Size o_size)
  { return tl_Alloc (o_size); }
```

Special Cases, Wrapper Classes

The following methods are not supported by standard store: `SizeOf`, `RoundedSizeOf` and `FreeAll`. The class `ct_StdStore` relies on the system interface and uses reserve memory. Debugging tools and heap walkers of the C standard library can be used together with `ct_StdStore`.

The following declarations of access functions and wrapper classes are generated in the standard store header file:

```
void CreateStdStore ();
void DeleteStdStore ();
ct_StdStore * GetStdStore ();
class ct_Std_Store;
class ct_Std8Store;
class ct_Std16Store;
class ct_Std32Store;
```

1.3.2 Round Store (tuning/rnd/store.hpp)

`ct_RndStore` uses the system interface like `ct_StdStore`. Additionally, it rounds block sizes before calling global functions. The private method `Round` calculates rounded values.

Class Declaration

```
class ct_RndStore
  {
public:
  typedef t_UInt           t_Size;
  typedef void *           t_Position;

                           ct_RndStore ();
  void                     Swap (ct_RndStore & co_swap);

  static inline t_UInt     StoreInfoSize ();
  static inline t_UInt     MaxAlloc ();

  inline t_Position        Alloc (t_Size o_size);
  inline t_Position        Realloc (t_Position o_pos, t_Size o_size);
  static inline void       Free (t_Position o_pos);

  static inline void *     AddrOf (t_Position o_pos);
```

```
  static inline t_Position PosOf (void * pv_adr);

  static inline t_Size     SizeOf (t_Position o_pos);
  static inline t_Size     RoundedSizeOf (t_Position o_pos);

  static inline bool       CanFreeAll ();
  static inline void       FreeAll ();

  inline t_Size            GetMinSize () const;
  void                     SetMinSize (t_Size o_minSize);
  inline t_Size            GetStepDiv () const;
  void                     SetStepDiv (t_Size u_stepDiv);
  };

inline ct_RndStore::t_Position ct_RndStore::Alloc (t_Size o_size)
  { return tl_Alloc (Round (o_size)); }
```

Block size rounding minimizes the number of reallocations and prevents memory fragmentation. The calculation of rounded values is controlled by two parameters: the minimum size and the step divider. The step divider affects the rounding granularity. The smaller it is, the less rounding values exist. If step divider is one, then solely powers of two are used. Step divider n allowes n values between two successive powers of two (inclusive upper bound).

The higher the heap utilization is, the smaller the step divider should be. If the number of memory allocations and reallocations is low, then fragmentation is no problem and the step divider may be set to four or eight. If the heap utilization increases, then the memory fragmentation increases the total amount of memory and the heap management time. The step divider should be set to two or one. If the heap utilization is very high, then the chain store should be used.

The efficiency of the round store depends on the C standard library implementation. A rule of thumb is: The round store increases performance in older compiler environments. Newer compilers have their own heap optimizations and will disturb the round store. The chain store always increases the memory management performance.

`ct_RndStore` contains additional methods for rounding parameters. The rounding parameters are protected against multiple thread access.

Additional Methods

`t_Size` **`GetMinSize`** `() const;`
`void` **`SetMinSize`** `(t_Size o_minSize);`

Access to the minimum block size. `o_minSize` must be a power of two.

`t_Size` **`GetStepDiv`** `() const;`
`void` **`SetStepDiv`** `(t_Size u_stepDiv);`

Access to the step divider. `u_stepDiv` must be a power of two.

Special Cases, Wrapper Classes

The following methods are not supported by round store: `SizeOf`, `RoundedSizeOf` and `FreeAll`. The class `ct_RndStore` relies on the system interface and uses reserve memory. Debugging tools and heap walkers of the C standard library can be used together with `ct_RndStore`.

The following declarations of access functions and wrapper classes are generated in the round store header file:

```
void CreateRndStore ();
void DeleteRndStore ();
ct_RndStore * GetRndStore ();
class ct_Rnd_Store;
class ct_Rnd8Store;
```

```
class ct_Rnd16Store;
class ct_Rnd32Store;
```

1.3.3 Chain Store (tuning/chn/store.hpp)

The chain store is a significant improvement over the round store. The focus is on programs with heavy heap utilization. `ct_ChnStore` has several optimization techniques to improve performance. The chain store prevents memory fragmentation. In most cases, the total amount of memory will decrease. Furthermore, there are no disadvantages for programs with low heap utilization.

Class Declaration

```
class ct_ChnStore
  {
public:
  typedef t_UInt          t_Size;
  typedef void *          t_Position;

                          ct_ChnStore ();
                          ~ct_ChnStore ();
  void                    Swap (ct_ChnStore & co_swap);

  static inline t_UInt    StoreInfoSize ();
  static inline t_UInt    MaxAlloc ();

  t_Position              Alloc (t_Size o_size);
  t_Position              Realloc (t_Position o_pos, t_Size o_size);
  void                    Free (t_Position o_pos);

  static inline void *    AddrOf (t_Position o_pos);
  static inline t_Position PosOf (void * pv_adr);

  static inline t_Size    SizeOf (t_Position o_pos);
  inline t_Size           RoundedSizeOf (t_Position o_pos);

  static bool             CanFreeAll ();
  static void             FreeAll ();

  unsigned                GetMaxChainExp ();
  void                    SetMaxChainExp (unsigned u_exp);
  t_UInt                  GetEntries ();
  t_UInt                  GetSize ();
  t_UInt                  QueryAllocEntries ();
  t_UInt                  QueryAllocSize ();
  t_UInt                  QueryFreeEntries ();
  t_UInt                  QueryFreeSize ();
  void                    FreeUnused ();
  };
```

Chain store rounds block sizes like round store (with step divider one) to the next power of two. Additionally, `ct_ChnStore` has its own memory management. For each of the few block sizes chain store contains a chain of free memory blocks. If `ct_ChnStore` allocates a new memory block, then it looks into the appropriate chain for a free block. If `ct_ChnStore` frees a memory block, then it puts the block into the appropriate chain.

Chain store uses the first `sizeof (t_UInt)` bytes of the memory block for management information. The methods `SizeOf` and `RoundedSizeOf` are implemented. Furthermore, it is possible to calculate memory usage statistics.

If the application allocates and frees nearly the same amount of memory, then the chain store is very efficient. When a large number of memory blocks are freed, the chain store will contain a large amount of unused memory. In this case, the FreeUnused method will give the memory back to the C standard library.

With increasing block sizes the probability of memory fragmentation decreases. Therefore the free chains may be limited by a maximum value. Above this value chain store works like a round store with step divider one (no free chains are used).

ct_ChnStore contains additional methods for memory usage statistics. The private attributes are protected against multiple thread access.

Additional Methods

unsigned **GetMaxChainExp** ();

Returns the max. exponent for free chains.

void **SetMaxChainExp** (unsigned u_exp);

Sets the max. exponent for free chains. Default value is 22 (2^22 = 4 MB).

t_UInt **GetEntries** ();

Returns the number of used and unused memory blocks.

t_UInt **GetSize** ();

Returns the total size of used and unused memory blocks.

t_UInt **QueryAllocEntries** ();

Calculates the number of used memory blocks.

t_UInt **QueryAllocSize** ();

Calculates the total size of used memory blocks.

t_UInt **QueryFreeEntries** ();

Calculates the number of unused memory blocks.

t_UInt **QueryFreeSize** ();

Calculates the total size of unused memory blocks.

void **FreeUnused** ();

Gives all unused memory blocks back to the C standard library.

Special Cases, Wrapper Classes

The FreeAll method is not supported by chain store. The class ct_ChnStore relies on the system interface and uses reserve memory. Debugging tools and heap walkers of the C standard library can be used together with ct_ChnStore. Notice that free chain blocks appear as used memory and that the first four or eight bytes of memory blocks are used by chain store.

The following declarations of access functions and wrapper classes are generated in the chain store header file:

```
void CreateChnStore ();
void DeleteChnStore ();
ct_ChnStore * GetChnStore ();
class ct_Chn_Store;
class ct_Chn8Store;
class ct_Chn16Store;
class ct_Chn32Store;
```

1.3.4 Global new and delete operators (tuning/newdel.cpp)

The file **'tuning/newdel.cpp'** contains implementations of the global `new` and `delete` operators using the chain store. Sometimes this feature has side effects with other libraries. Therefore it must be explicitly enabled with the `TL_NEWDEL` macro.

```
void * operator new (size_t u_size)
  {
  return GetChnStore ()-> Alloc (u_size);
  }

void operator delete (void * pv)
  {
  GetChnStore ()-> Free (pv);
  }

void * operator new [] (size_t u_size)
  {
  return GetChnStore ()-> Alloc (u_size);
  }

void operator delete [] (void * pv)
  {
  GetChnStore ()-> Free (pv);
  }
```

1.4 Block

1.4.1 Block Interface

Numerous classes within the **Spirick Tuning** library use dynamic memory blocks to store their data. The block interface is a simple object oriented concept of managing a single memory block. To increase performance there is no common base class with virtual functions. However, all block classes share a common interface. So it's easy to switch between multiple block implementations. Block classes are used as template parameters of strings, arrays and block stores.

Class Declaration

```
class ct_AnyBlock
  {
public:
  typedef t_UInt       t_Size;

                       ct_AnyBlock ();
                       ct_AnyBlock (const ct_AnyBlock & co_init);
                       ~ct_AnyBlock ();
  ct_AnyBlock &        operator = (const ct_AnyBlock & co_asgn);
  void                 Swap (ct_AnyBlock & co_swap);

  static t_UInt        GetMaxByteSize ();
  t_Size               GetByteSize () const;
  void                 SetByteSize (t_Size o_newSize);
  void *               GetAddr () const;
  };
```

Data Types

typedef t_UInt **t_Size**;

The nested type t_Size describes the size of the memory block, examples are t_UInt, t_UInt8, t_UInt16 and t_UInt32. If t_Size is defined as t_UInt8, the maximum size of the memory block will be 255 bytes. An attribute of type t_Size will consume one byte.

Constructors, Destructor, Assignment, Swap

Every block class contains a constructor, a copy constructor, a destructor and an assignment operator.

ct_AnyBlock ();

Initializes an empty block object.

ct_AnyBlock (const ct_AnyBlock & co_init);

Initializes a block object and copies the input data into its own memory block (deep copy).

~ct_AnyBlock ();

Releases the allocated memory.

ct_AnyBlock & **operator =** (const ct_AnyBlock & co_asgn);

Copies the input data into its own memory block (deep copy).

void **Swap** (ct_AnyBlock & co_swap);

Swaps the values of the two objects.

Additional Methods

static t_UInt **GetMaxByteSize** ();

Returns the maximum size of the memory block.

t_Size **GetByteSize** () const;

Returns the current size of the memory block.

void **SetByteSize** (t_Size o_newSize);

Reallocates the memory block to size o_newSize.

void * **GetAddr** () const;

Returns the memory address of the block or the null pointer if size is zero.

The following sections describe different implementations of the block interface.

1.4.2 Simple Block (tuning/block.h)

The class template gct_Block is the standard implementation of the block interface. The implementation consists of the base class gct_BlockBase, the block class gct_Block and the helper classes gct_EmptyBaseBlock and gct_ObjectBaseBlock.

Base Class

The block base class contains attributes of the t_Position and t_Size data types of the corresponding store class. The size of the object depends on these data types. t_staticStore must have the common store interface. All methods of t_staticStore must be declared static, examples are ct_Rnd16Store and ct_Chn32Store. The block base class can be used for different purposes:

1. If the t_Position and t_Size data types have different sizes (e.g. void * and t_UInt16), then the compiler will insert padding bytes. Note that it is not possible to use padding bytes of a base class in a derived class. For optimal memory utilization base classes should be designed without padding bytes. The sample program TBlock contains a modified base class.

2. Sometimes a block class should be derived from a special base class. Therefore the gct_BlockBase template contains a t_base parameter.

Note that the Swap method is declared in the block base class and not in the block class.

Template Declaration

```
template <class t_staticStore, class t_base>
  class gct_BlockBase: public t_base
    {
  public:
    typedef t_staticStore t_StaticStore;
    typedef t_StaticStore::t_Size t_Size;

  protected:
    t_StaticStore::t_Position o_Pos;
    t_Size               o_Size;

  public:
    inline void          Swap (gct_BlockBase & co_swap);
    inline t_StaticStore::t_Store * GetStore () const;
    };
```

Block Class

The template parameter t_blockBase must at least contain the same data types, attributes and methods as the gct_BlockBase template.

Template Declaration

```
template < class t_blockBase>
  class gct_Block: public t_blockBase
    {
  public:
    typedef t_blockBase::t_Size t_Size;
    typedef t_blockBase::t_StaticStore t_StaticStore;

    inline               gct_Block ();
    inline               gct_Block (const gct_Block & co_init);
    inline               ~gct_Block ();
    inline gct_Block &   operator = (const gct_Block & co_asgn);

    static inline t_UInt GetMaxByteSize ();
    inline t_Size        GetByteSize () const;
    inline void          SetByteSize (t_Size o_newSize);
    inline void *        GetAddr () const;
    };
```

The methods of gct_Block are very simple. The store methods are called directly.

```
template <class t_staticStore>
  inline void gct_Block <t_staticStore>::SetByteSize (t_Size o_newSize)
    {
    o_Size = o_newSize;
```

```
    o_Pos = t_staticStore::Realloc (o_Pos, o_Size);
    }
```

Helper Classes

The top-level base class may be `ct_Empty` or `ct_Object`. Two class templates are predefined.

Template Declaration

```
template <class t_staticStore>
  class gct_EmptyBaseBlock:
    public gct_Block <gct_BlockBase <t_staticStore, ct_Empty> >
    {
    };
```

Template Declaration

```
template <class t_staticStore>
  class gct_ObjectBaseBlock:
    public gct_Block <gct_BlockBase <t_staticStore, ct_Object> >
    {
    };
```

1.4.3 Mini Block (tuning/miniblock.h)

A `gct_Block` object contains a size and a position attribute. If the store class supports the `SizeOf` method, then the size attribute is redundant. The `gct_MiniBlock` template uses the `SizeOf` method instead of a size attribute. The implementation consists of the base class `gct_MiniBlockBase`, the block class `gct_MiniBlock` and the helper classes `gct_EmptyBaseMiniBlock` and `gct_ObjectBaseMiniBlock`.

Base Class

The class template `gct_MiniBlockBase` is similar to `gct_BlockBase` (see above).

Template Declaration

```
template <class t_staticStore, class t_base>
  class gct_MiniBlockBase: public t_base
    {
  public:
    typedef t_staticStore t_StaticStore;
    typedef t_StaticStore::t_Size t_Size;

  protected:
    t_StaticStore::t_Position o_Pos;

  public:
    inline void          Swap (gct_MiniBlockBase & co_swap);
    inline t_StaticStore::t_Store * GetStore () const;
    };
```

Block Class

The template parameter `t_blockBase` must at least contain the same data types, attributes and methods as the `gct_MiniBlockBase` template.

Template Declaration

```
template <class t_blockBase>
  class gct_MiniBlock: public t_blockBase
    {
  public:
    typedef t_blockBase::t_Size t_Size;
    typedef t_blockBase::t_StaticStore t_StaticStore;

    inline                gct_MiniBlock ();
    inline                gct_MiniBlock (const gct_MiniBlock & co_init);
    inline                ~gct_MiniBlock ();
    inline gct_MiniBlock & operator = (const gct_MiniBlock & co_asgn);

    static inline t_UInt GetMaxByteSize ();
    inline t_Size        GetByteSize () const;
    inline void          SetByteSize (t_Size o_newSize);
    inline void *        GetAddr () const;
    };
```

A mini block object consumes less memory than a block object. Note that some methods are slightly slower than the corresponding block methods.

```
template <class t_blockBase>
  inline gct_MiniBlock <t_blockBase>::t_Size
  gct_MiniBlock <t_blockBase>::GetByteSize () const
    {
    return (t_Size) t_staticStore::SizeOf (o_Pos);
    }
```

Helper Classes

The top-level base class may be `ct_Empty` or `ct_Object`. Two class templates are predefined.

Template Declaration

```
template <class t_staticStore>
  class gct_EmptyBaseMiniBlock:
    public gct_MiniBlock <gct_MiniBlockBase <t_staticStore, ct_Empty> >
    {
    };
```

Template Declaration

```
template <class t_staticStore>
  class gct_ObjectBaseMiniBlock:
    public gct_MiniBlock <gct_MiniBlockBase <t_staticStore, ct_Object> >
    {
    };
```

1.4.4 Reserve Block (tuning/resblock.h)

The class template `gct_ResBlock` is similar to `gct_Block`. In addition to the current size of the block, a reserve block contains a minimum size parameter. In some use cases the number of reallocations can be reduced by using the minimum size. The implementation consists of the base class `gct_ResBlockBase`, the block class `gct_ResBlock` and the helper classes `gct_EmptyBaseResBlock` and `gct_ObjectBaseResBlock`.

Base Class

The class template `gct_ResBlockBase` is similar to `gct_BlockBase` (see above).

Template Declaration

```
template <class t_staticStore, class t_base>
  class gct_ResBlockBase: public t_base
    {
  public:
    typedef t_staticStore t_StaticStore;
    typedef t_StaticStore::t_Size t_Size;

  protected:
    t_StaticStore::t_Position o_Pos;
    t_Size              o_Size;
    t_Size              o_MinSize;

  public:
    inline void         Swap (gct_ResBlockBase & co_swap);
    inline t_StaticStore::t_Store * GetStore () const;
    };
```

Block Class

The template parameter `t_blockBase` must at least contain the same data types, attributes and methods as the `gct_ResBlockBase` template.

Template Declaration

```
template <class t_blockBase>
  class gct_ResBlock: public t_blockBase
    {
  public:
    typedef t_blockBase::t_Size t_Size;
    typedef t_blockBase::t_StaticStore t_StaticStore;

    inline              gct_ResBlock ();
    inline              gct_ResBlock (const gct_ResBlock & co_init);
    inline              ~gct_ResBlock ();
    inline gct_ResBlock & operator = (const gct_ResBlock & co_asgn);

    static inline t_UInt GetMaxByteSize ();
    inline t_Size       GetByteSize () const;
    inline void         SetByteSize (t_Size o_newSize);
    inline void *       GetAddr () const;

    inline t_Size       GetMinByteSize () const;
    inline t_Size       GetAllocByteSize () const;
    inline void         SetMinByteSize (t_Size o_newSize);
    };
```

Additional Methods

`t_Size` **`GetMinByteSize`** `() const;`

Returns the minimum size of the block.

`t_Size` **`GetAllocByteSize`** `() const;`

Returns the currently allocated size of the block.

`void` **`SetMinByteSize`** `(t_Size o_newSize);`

Sets the minimum size of the block to `o_newSize`.

Helper Classes

The top-level base class may be `ct_Empty` or `ct_Object`. Two class templates are predefined.

Template Declaration

```
template <class t_staticStore>
  class gct_EmptyBaseResBlock:
    public gct_ResBlock <gct_ResBlockBase <t_staticStore, ct_Empty> >
    {
    };
```

Template Declaration

```
template <class t_staticStore>
  class gct_ObjectBaseResBlock:
    public gct_ResBlock <gct_ResBlockBase <t_staticStore, ct_Object> >
    {
    };
```

1.4.5 Fixed Sized Block (tuning/fixblock.h)

The `gct_FixBlock` template eliminates the overhead of dynamic memory management. It is useful for block sizes from zero to 50 bytes. The block size is limited to a constant value. A `gct_FixBlock` object does not allocate dynamic memory. It contains a fixed sized byte array.

Template Declaration

```
template <class t_size, t_UInt u_fixSize>
  class gct_FixBlock
    {
  public:
    typedef t_size       t_Size;

  protected:
    t_Size               o_Size;
    char                 ac_Block [u_fixSize];

  public:
    inline               gct_FixBlock ();
    inline               gct_FixBlock (const gct_FixBlock & co_init);
    inline gct_FixBlock & operator = (const gct_FixBlock & co_asgn);
    void                 Swap (gct_FixBlock & co_swap);

    static inline t_UInt GetMaxByteSize ();
    inline t_Size        GetByteSize () const;
    inline void          SetByteSize (t_Size o_newSize);
    inline void *        GetAddr () const;
    };
```

Note that the alignment of the internal char array depends on the `t_size` parameter.

1.4.6 Null Data Block (tuning/nulldatablock.h)

A null-terminated string consumes memory even if it is empty (for the null character). Due to rounding of block sizes and memory management overhead, 8 or 16 bytes are consumed. In some use cases this may lead to a significant amount of memory. The class template `gct_NullDataBlock` uses a static allocated null-value object. If the block size is 1, then no dynamic memory ist allocated.

The template parameter t_block must contain the block interface.

Template Declaration

```
template <class t_block, class t_null>
  class gct_NullDataBlock: public t_block
    {
  public:
    typedef t_block::t_Size t_Size;

  private:
    static t_null       o_NullData;

  public:
    inline t_Size       GetByteSize () const;
    inline void         SetByteSize (t_Size o_newSize);
    inline void *       GetAddr () const;
    };
```

Note that the last character of the block must contain the null value, no other values are allowed.

1.4.7 Character Block (tuning/charblock.h)

The class template gct_CharBlock is an extension of the common block interface. It contains several useful methods. The common block is the base class of the character block. The template parameter t_char may be char or wchar_t. To avoid any possibility of confusion, byte-oriented methods are declared private.

Base Class

ct_AnyBlock (see above 'Block Interface')

Template Declaration

```
template <class t_block, class t_char>
  class gct_CharBlock: public t_block
    {
  public:
    inline t_Size       GetMaxCharSize () const;
    inline t_Size       GetCharSize () const;
    inline void         SetCharSize (t_Size o_size);
    inline void         IncCharSize (t_Size o_inc);
    inline void         DecCharSize (t_Size o_dec);
    inline t_char *     GetRawAddr () const;
    inline t_char *     GetRawAddr (t_Size o_pos) const;
    inline t_char *     GetCharAddr () const;
    inline t_char *     GetCharAddr (t_Size o_pos) const;

    t_char *            AppendChars (t_Size o_len);
    t_char *            InsertChars (t_Size o_pos, t_Size o_count);
    t_char *            DeleteChars (t_Size o_pos, t_Size o_count);
    inline t_char *     FillChars (t_Size o_pos, t_Size o_count, t_char c_fill = (t_char) 0);

    inline void         AssignChars (const t_char * pc_asgn, t_Size o_len);
    inline void         AppendChars (const t_char * pc_app, t_Size o_len);
    inline void         InsertChars (t_Size o_pos, const t_char * pc_ins, t_Size o_len);
    void                ReplaceChars (t_Size o_pos, t_Size o_delLen,
                          const t_char * pc_ins, t_Size o_insLen);

    inline t_Size       GetDefaultPageSize () const;
```

```
    inline void          AlignPageSize (t_Size o_itemSize, t_Size o_pageSize);
    };
```

Methods

t_Size **GetMaxCharSize** ();

Returns the maximum character size of the memory block.

t_Size **GetCharSize** () const;

Returns the current character size of the memory block.

void **SetCharSize** (t_Size o_size);

Reallocates the memory block to o_size characters.

void **IncCharSize** (t_Size o_inc);

Increases block size by o_inc characters.

void **DecCharSize** (t_Size o_dec);

Decreases block size by o_dec characters. o_dec must be less than or equal to GetCharSize ().

t_char * **GetRawAddr** () const;

Returns the memory address of the block or the null pointer if size is zero.

t_char * **GetRawAddr** (t_Size o_pos) const;

Returns the memory address of the character at position o_pos. o_pos must be less than or equal to GetCharSize ().

t_char * **GetCharAddr** () const;

Returns the memory address of the block. Size must be greater than zero.

t_char * **GetCharAddr** (t_Size o_pos) const;

Returns the memory address of the character at position o_pos. o_pos must be less than GetCharSize ().

t_char * **AppendChars** (t_Size o_len);

Increases block size by o_len characters. Returns the memory address of the character at position GetCharSize () - o_len.

t_char * **InsertChars** (t_Size o_pos, t_Size o_len);

Increases block size by o_len characters and moves memory from position o_pos to position o_pos + o_len. Returns the memory address of the character at position o_pos.

t_char * **DeleteChars** (t_Size o_pos, t_Size o_len);

Moves memory from position o_pos + o_len to position o_pos and decreases block size by o_len characters. Returns the memory address of the character at position o_pos.

t_char * **FillChars** (t_Size o_pos, t_Size o_len, t_char c_fill = (t_char) 0);

Sets o_len characters at position o_pos to the character c_fill. Returns the memory address of the character at position o_pos.

void **AssignChars** (const t_char * pc_asgn, t_Size o_len);

Reallocates the memory block to o_len characters and copies the first o_len characters from pc_asgn to the memory block.

void **AppendChars** (const t_char * pc_app, t_Size o_len);

Increases block size by o_len characters and copies the first o_len characters from pc_app to position GetCharSize () - o_len.

void **InsertChars** (t_Size o_pos, const t_char * pc_ins, t_Size o_len);

Increases block size by o_len characters, moves memory from position o_pos to position o_pos + o_len and copies the first o_len characters from pc_ins to position o_pos.

void **ReplaceChars** (t_Size o_pos, t_Size o_delLen, const t_char * pc_ins, t_Size o_insLen);

Replaces o_delLen characters at position o_pos by the first o_insLen characters from pc_ins. Block size may be changed.

t_Size **GetDefaultPageSize** () const;
void **AlignPageSize** (t_Size o_itemSize, t_Size o_pageSize);

These methods make gct_CharBlock compatible with the page block interface.

1.4.8 Item Block (tuning/itemblock.h)

The class template gct_ItemBlock is smilar to gct_CharBlock, but instead of a char type parameter, an arbitrary item size parameter is used. The implementation consists of the item block class gct_ItemBlock and the helper classes gct_VarItemBlock and gct_FixItemBlock. To avoid any possibility of confusion, byte-oriented methods are declared private.

Base Class

ct_AnyBlock (see above 'Block Interface')

Template Declaration

```
template <class t_block>
  class gct_ItemBlock: public t_block
    {
  public:
    inline t_Size       GetFixSize () const;
    inline t_Size       GetMaxItemSize () const;
    inline t_Size       GetItemSize () const;
    inline void         SetItemSize (t_Size o_size);
    inline void         IncItemSize1 ();
    inline void         DecItemSize1 ();
    inline void         IncItemSize (t_Size o_inc);
    inline void         DecItemSize (t_Size o_dec);
    inline void *       GetItemAddr (t_Size o_pos) const;

    void *              InsertItems (t_Size o_pos, t_Size o_count);
    void *              DeleteItems (t_Size o_pos, t_Size o_count);

    inline t_Size       GetDefaultPageSize () const;
    inline void         AlignPageSize (t_Size o_fixSize, t_Size o_pageSize);
    };
```

Methods

t_Size **GetFixSize** () const;

Returns the byte size of a single item.

t_Size **GetMaxItemSize** () const;

Returns the maximum item size of the memory block.

t_Size **GetItemSize** () const;

Returns the current item size of the memory block.

void **SetItemSize** (t_Size o_size) const;

Reallocates the memory block to o_size items.

void **IncItemSize1** ();

Increases block size by 1 item.

void **DecItemSize1** ();

Decreases block size by 1 item.

void **IncItemSize** (t_Size o_inc);

Increases block size by o_inc items.

void **DecItemSize** (t_Size o_dec);

Decreases block size by o_dec items. o_dec must be less than or equal to GetItemSize ().

void * **GetItemAddr** (t_Size o_pos) const;

Returns the memory address of the item at position o_pos. o_pos must be less than GetItemSize ().

void * **InsertItems** (t_Size o_pos, t_Size o_count);

Increases block size by o_count items and moves memory from position o_pos to position o_pos + o_count. Returns the memory address of the item at position o_pos.

void * **DeleteItems** (t_Size o_pos, t_Size o_count);

Moves memory from position o_pos + o_count to position o_pos and decreases block size by o_count items. Returns the memory address of the item at position o_pos.

t_Size **GetDefaultPageSize** () const;
void **AlignPageSize** (t_Size o_itemSize, t_Size o_pageSize);

These methods make gct_ItemBlock compatible with the page block interface.

Helper Classes

The item size can be configured at compile time or at runtime. The class template gct_VarItemBlock enables runtime configuration by using the method AlignPageSize. A typical use case is the block store.

Template Declaration

```
template <class t_block>
  class gct_VarItemBlock:
    public gct_ItemBlock <gct_VarItemBlockBase <t_block> >
    {
    };
```

The class template gct_FixItemBlock contains the parameter o_itemSize for compile time configuration. A typical use case is the array container.

Template Declaration

```
template <class t_block, t_UInt o_itemSize>
```

```
class gct_FixItemBlock:
  public gct_ItemBlock <gct_FixItemBlockBase <t_block, o_itemSize> >
  {
  };
```

1.4.9 Page Block (tuning/pageblock.hpp)

A page block uses equal-sized memory pages instead of a continuous memory block. This concept provides the following advantages:

1. Lower number of memory allocations and releases.
2. Lower memory fragmentation.
3. No memory copying while changing the block size.
4. All memory addresses remain valid while changing the block size.

This special implementation uses a class with virtual functions instead of template parameters. The page block uses a helper block for managing pointers to the pages. Different store classes can be used for the management block and the data pages.

The size of the pointer management block may be fixed or variable. If the size is fixed, then no mutex is required for the methods GetCharAddr and GetItemAddr in a multi-threaded environment. Note that a fixed sized management block leads to a maximum size of the entire page block.

The implementation of the page block consists of the base class gct_PageBlockBase with some virtual methods and the derived class ct_PageBlock with access to two store objects. The page block class contains some common block methods and additionally also the methods of gct_CharBlock and gct_ItemBlock.

Note that the memory location of a single item must not overlap a page boundary. Therefore the page block must be initialized with the method AlignPageSize while the size is zero.

Class Declaration

```
class ct_PageBlockBase
  {
public:
  typedef t_UInt       t_Size;

protected:
  void                 SetByteSize0 ();
  virtual void *       AllocPtr (t_Size o_size) = 0;
  virtual void *       ReallocPtr (void * pv_mem, t_Size o_size) = 0;
  virtual void *       AllocData (t_Size o_size) = 0;
  virtual void         FreeData (void * pv_mem) = 0;
  virtual void         LastPageWarning () { }
  virtual void         LastPageError () { }

public:
  // Block
                       ct_PageBlockBase ();
  inline               ct_PageBlockBase (const ct_PageBlockBase & co_init);
  virtual              ~ct_PageBlockBase () { }
  inline ct_PageBlockBase & operator = (const ct_PageBlockBase & co_asgn);
  void                 Swap (ct_PageBlockBase & co_swap);

  // CharBlock
  inline t_Size        GetMaxCharSize () const;
  inline t_Size        GetCharSize () const;
  inline void          SetCharSize (t_Size o_size);
  inline void          IncCharSize (t_Size o_inc);
  inline void          DecCharSize (t_Size o_dec);
```

```
inline char *       GetRawAddr () const;
inline char *       GetRawAddr (t_Size o_pos) const;
inline char *       GetCharAddr () const;
inline char *       GetCharAddr (t_Size o_pos) const;

char *              InsertChars (t_Size o_pos, t_Size o_count);
char *              DeleteChars (t_Size o_pos, t_Size o_count);
char *              FillChars (t_Size o_pos, t_Size o_count,
                      char c_fill = '\0');

// ItemBlock
inline t_Size       GetFixSize () const;
inline t_Size       GetMaxItemSize () const;
inline t_Size       GetItemSize () const;
inline void         SetItemSize (t_Size o_size);
inline void         IncItemSize1 ();
inline void         DecItemSize1 ();
inline void         IncItemSize (t_Size o_inc);
inline void         DecItemSize (t_Size o_dec);
inline void *       GetItemAddr (t_Size o_pos) const;

inline void *       InsertItems (t_Size o_pos, t_Size o_count);
inline void *       DeleteItems (t_Size o_pos, t_Size o_count);

// PageBlock only Methods
inline t_Size       GetDefaultPageSize () const;
inline t_Size       GetFixPagePtrs () const;
void                SetFixPagePtrs (t_Size o_ptrs);
void                AlignPageSize (t_Size o_fixSize, t_Size o_pageSize);
inline t_Size       GetPageSize () const;
inline t_Size       GetRoundedSize () const;
};
```

Additional Methods

`void` **`LastPageWarning`** `();`

This virtual method will be called if the pointer management block is fixed sized and the last data page was allocated. This implies that only a single data page is available.

`void` **`LastPageError`** `();`

This virtual method will be called if the pointer management block is fixed sized and the last data page does not contain any more free space.

The behaviour of this method is similar to the overflow handler (see above `tl_SetOverflowHandler`). This method must not throw C++ exceptions. Exceptions from `LastPageError` are not handled by the library and lead to inconsistent objects. Afterwards the program is terminated by the function `tl_EndProcess`.

`t_Size` **`GetDefaultPageSize`** `() const;`

Returns a default value for the size of a data page.

`t_Size` **`GetFixPagePtrs`** `() const;`

Returns the number of pointers in the management block (i.e. the max. number of data pages). The return value zero means that the size of the management block is variable.

`void` **`SetFixPagePtrs`** `(t_Size o_ptrs);`

Sets the number of pointers in the management block (i.e. the max. number of data pages) to `o_ptrs`. While calling this method, the block size must be zero.

`void AlignPageSize (t_Size o_fixSize, t_Size o_pageSize);`

The size of data pages is calculated so that it is a multiple of `o_fixSize` and greater than or equal to `o_pageSize`. While calling this method, the block size must be zero.

`t_Size GetPageSize () const;`

Returns the size of a data page.

`t_Size GetRoundedSize () const;`

Returns the product of the page size and the number of pages.

Class Declaration

```
class ct_PageBlock: public ct_PageBlockBase
  {
protected:
  virtual void *      AllocPtr (t_Size o_size);
  virtual void *      ReallocPtr (void * pv_mem, t_Size o_size);
  virtual void *      AllocData (t_Size o_size);
  virtual void        FreeData (void * pv_mem);

public:
                      ~ct_PageBlock ();
  };
```

Methods

`void * AllocPtr (t_Size o_size);`

Allocate memory for the pointer management block.

`void * ReallocPtr (void * pv_mem, t_Size o_size);`

Reallocate memory for the pointer management block.

`void * AllocData (t_Size o_size);`

Allocate a single data page.

`void FreeData (void * pv_mem);`

Release a single data page.

`~ct_PageBlock ();`

Within the destructor of the derived class all memory must be released. The destructor of the base class has no access to the virtual methods implemented in the derived class.

1.4.10 Block Instances (tuning/xxx/block.h)

Some template instances are predefined to easily use the block interface. The macro `BLOCK_DCLS(Obj)` generates for each wrapper class of a global store one block class.

The macro

```
BLOCK_DCLS (Any)
```

expands to:

```
class ct_Any_Block:
  public gct_EmptyBaseBlock <ct_Any_Store> { };
```

```
class ct_Any8Block:
  public gct_EmptyBaseBlock <ct_Any8Store> { };
class ct_Any16Block:
  public gct_EmptyBaseBlock <ct_Any16Store> { };
class ct_Any32Block:
  public gct_EmptyBaseBlock <ct_Any32Store> { };
```

Every directory of a global store contains a file **'block.h'.**

The file 'tuning/std/block.h' contains the following declarations:

```
class ct_Std_Block;
class ct_Std8Block;
class ct_Std16Block;
class ct_Std32Block;
```

The file 'tuning/rnd/block.h' contains the following declarations:

```
class ct_Rnd_Block;
class ct_Rnd8Block;
class ct_Rnd16Block;
class ct_Rnd32Block;
```

The file 'tuning/chn/block.h' contains the following declarations:

```
class ct_Chn_Block;
class ct_Chn8Block;
class ct_Chn16Block;
class ct_Chn32Block;
```

1.5 Special Stores

1.5.1 Block Store (tuning/blockstore.h)

A block store uses an item block (see above 'Item Block') for compact storage of smaller, equal-sized memory blocks. The rounding and management overhead of a dynamic memory management is significantly reduced. Typical use cases are list containers. All nodes of a list container have the same size.

The first template parameter `t_itemBlock` must at least contain the item block interface, e.g. `gct_VarItemBlock <ct_Chn16Block>` or `ct_PageBlock`. The second template parameter `t_charBlock` must at least contain the character block interface, e.g. `gct_CharBlock <ct_Chn32Block, char>`. It is used for temporary data inside of the method `FreeUnused`.

Base Class

`t_itemBlock` (see above 'Item Block')

Template Declaration

```
template <class t_itemBlock, class t_charBlock>
  class gct_BlockStore: public t_itemBlock
    {
  public:
    typedef t_itemBlock::t_Size t_Size;
    typedef t_itemBlock::t_Size t_Position;

    inline              gct_BlockStore ();
```

```
inline t_UInt       StoreInfoSize () const;
inline t_UInt       MaxAlloc () const;

t_Position          Alloc (t_Size o_size);
t_Position          Realloc (t_Position o_pos, t_Size o_size);
void                Free (t_Position o_pos);

inline void *       AddrOf (t_Position o_pos) const;
inline t_Position   PosOf (void * pv_adr) const;

inline t_Size       SizeOf (t_Position o_pos) const;
inline t_Size       RoundedSizeOf (t_Position o_pos) const;

inline bool         CanFreeAll () const;
inline void         FreeAll ();

void                SetSortedFree (bool b);
void                SetPageSize (t_Size o_size);
inline t_Position   LastIdx () const;
inline bool         HasFree () const;
void                FreeUnused ();
};
```

Size and position data types of a block store are the same as in the base class. Position values are indices beginning with 1, 2, 3 etc. The position value zero is invalid per definition (see above 'Store Interface').

Note that the memory addresses of block store entries can change if the size of the underlying item block changes i.e. if the block store methods `Alloc`, `Realloc` or `Free` are called. Note also that the memory addresses of block store entries remain valid if the parameter `t_itemBlock` equals `ct_PageBlock`.

The block store implementation uses two different algorithms to manage the internal list of free blocks. Algorithm 1 is optimized for speed, it uses an unsorted list. Algorithm 2 is optimized for size, it uses an sorted list. By default, algorithm 1 is active. The method `FreeUnused` sorts the list of free blocks and tries to reduce the size of the underlying item block. The method `SetSortedFree` can be used to switch between algorithm 1 and 2.

The class template `gct_BlockStore` does not support the `SizeOf` method. The item size is calculated in the first call of `Alloc` or `Realloc`. In subsequent calls of `Alloc` or `Realloc`, the requested size must be less than or equal to the item size.

Additional Methods

`void` **`SetSortedFree`** `(bool b);`

Select an algorithm for internal free list management.

`void` **`SetPageSize`** `(t_Size o_size);`

If the parameter `t_itemBlock` equals `ct_PageBlock`, then this method sets the page size of the underlying page block.

`t_Position` **`LastIdx`** `() const;`

Returns the maximun position value (allocated or free) or zero, if the block store is empty.

`bool` **`HasFree`** `() const;`

Returns `true`, if the internal free list contains at least one element.

`void` **`FreeUnused`** `();`

Sorts the list of free blocks and tries to reduce the size of the underlying item block.

1.5.2 Block Store Instances (tuning/xxx/blockstore.h)

Some template instances are predefined to easily use the block store interface. The macro `BLOCK_STORE_DCLS(Obj)` generates for each wrapper class of a global store one block store class.

The macro

```
BLOCK_STORE_DCLS (Any)
```

expands to:

```
class ct_Any_BlockStore:
  public gct_BlockStore <gct_VarItemBlock <ct_Any_Block>, gct_CharBlock <ct_Any_Block, char> > { };
class ct_Any8BlockStore:
  public gct_BlockStore <gct_VarItemBlock <ct_Any8Block>, gct_CharBlock <ct_Any8Block, char> > { };
class ct_Any16BlockStore:
  public gct_BlockStore <gct_VarItemBlock <ct_Any16Block>, gct_CharBlock <ct_Any16Block, char> > { };
class ct_Any32BlockStore:
  public gct_BlockStore <gct_VarItemBlock <ct_Any32Block>, gct_CharBlock <ct_Any32Block, char> > { };
```

Every directory of a global store contains a file **'blockstore.h'**.

The file 'tuning/std/blockstore.h' contains the following declarations:

```
class ct_Std_BlockStore;
class ct_Std8BlockStore;
class ct_Std16BlockStore;
class ct_Std32BlockStore;
```

The file 'tuning/rnd/blockstore.h' contains the following declarations:

```
class ct_Rnd_BlockStore;
class ct_Rnd8BlockStore;
class ct_Rnd16BlockStore;
class ct_Rnd32BlockStore;
```

The file 'tuning/chn/blockstore.h' contains the following declarations:

```
class ct_Chn_BlockStore;
class ct_Chn8BlockStore;
class ct_Chn16BlockStore;
class ct_Chn32BlockStore;
```

1.5.3 Reference Counter (tuning/refcount.hpp)

`ct_RefCount` is a class containing a reference counter and a boolean value. It is used by ref-stores.

Class Declaration

```
typedef t_UInt32 t_RefCount;

class ct_RefCount
  {
public:
  inline              ct_RefCount ();
  inline void         Initialize ();

  inline t_RefCount   GetRef () const;
  inline void         IncRef ();
  inline void         DecRef ();
```

```
inline bool          IsAlloc () const;
inline void          SetAlloc ();
inline bool          IsFree () const;
inline void          SetFree ();
inline bool          IsNull () const;
};
```

Data Types

`typedef t_UInt32 t_RefCount;`

This is the numeric reference counter type.

Methods

`ct_RefCount ();`

Sets the reference counter to zero and the alloc flag to `true`.

`void Initialize ();`

Sets the reference counter to zero and the alloc flag to `true`.

`t_RefCount GetRef () const;`

Returns the numeric reference counter.

`void IncRef ();`

Increases the reference counter by 1.

`void DecRef ();`

Decreases the reference counter by 1.

`bool IsAlloc () const;`

Returns the alloc flag.

`void SetAlloc ();`

Sets the alloc flag.

`bool IsFree () const;`

Returns `true`, if the alloc flag is not set.

`void SetFree ();`

Clears the alloc flag.

`bool IsNull () const;`

Returns `true`, if the reference counter equals zero and the alloc flag is not set.

1.5.4 Ref-Store (tuning/refstore.h)

A ref-store enhances an existing store class with reference counting. Each single memory block is associated with a reference counter. The reference counters can be used directly or indirectly by special classes, e.g. smart pointers.

Note that the reference counter is associated with the memory block and not with its contents, e.g. a C++ object. Deleting a C++ object and releasing the corresponding memory are two distinct steps. The

C++ object can be deleted by its owner, and the corresponding memory block can be released by the reference counter. If a C++ object is deleted and the reference counter is greater than zero, then all smart pointers remain valid, but access to the C++ object is not allowed. In this way isolated islands in complex, reference counting based data structures can be avoided.

Template Declaration

```
template <class t_store>
  class gct_RefStore
    {
  public:
    typedef t_store::t_Size     t_Size;
    typedef t_store::t_Position t_Position;

    void                Swap (gct_RefStore & co_swap);
    inline t_UInt       StoreInfoSize () const;
    inline t_UInt       MaxAlloc () const;

    t_Position          Alloc (t_Size o_size);
    t_Position          Realloc (t_Position o_pos, t_Size o_size);
    inline void         Free (t_Position o_pos);

    inline void *       AddrOf (t_Position o_pos) const;
    inline t_Position   PosOf (void * pv_adr) const;

    inline t_Size       SizeOf (t_Position o_pos) const;
    inline t_Size       RoundedSizeOf (t_Position o_pos) const;

    inline bool         CanFreeAll () const;
    inline void         FreeAll ();

    inline void         IncRef (t_Position o_pos);
    inline void         DecRef (t_Position o_pos);
    inline t_RefCount   GetRef (t_Position o_pos) const;
    inline bool         IsAlloc (t_Position o_pos) const;
    inline bool         IsFree (t_Position o_pos) const;

    inline t_store *    GetStore ();
    };
```

A ref-store passes allocation requests to the underlying store object. The block size is increased by the size of the `ct_RefCount` object, and the `ct_RefCount` object is initialized. The reference counter can be changed by the ref-store methods `IncRef` and `DecRef`.

If a memory block is released by the ref-store method `Free`, then the alloc flag of the corresponding `ct_RefCount` object is cleared. If additionally the reference counter equals zero, the block is released by the underlying store object. Otherwise the reference counter can be changed by the ref-store methods `IncRef` and `DecRef`, but access to the memory by calling the method `AddrOf` is not allowed. If the reference counter becomes zero, the block is released by the underlying store object.

The class template `gct_RefStore` does not support the `FreeAll` method.

Additional Methods

`void` **`IncRef`** `(t_Position o_pos);`

Increases the reference counter at position `o_pos` by 1.

`void` **`DecRef`** `(t_Position o_pos);`

Decreases the reference counter at position `o_pos` by 1.

`t_RefCount GetRef (t_Position o_pos) const;`

Returns the numeric reference counter at position `o_pos`.

`bool IsAlloc (t_Position o_pos) const;`

Returns the alloc flag of position value `o_pos`.

`bool IsFree (t_Position o_pos) const;`

Returns `true`, if the alloc flag of position value `o_pos` is not set.

`t_store * GetStore ();`

Returns a pointer to the underlaying store object.

1.5.5 Ref-Store Instances (tuning/xxx/refstore.h)

Some template instances are predefined to easily use the ref-store interface. The macro `REF_STORE_DCLS(Obj)` generates for each wrapper class of a global store one ref-store class.

The macro

```
REF_STORE_DCLS (Any)
```

expands to:

```
class ct_Any_RefStore:
  public gct_RefStore <ct_Any_Store> { };
class ct_Any8RefStore:
  public gct_RefStore <ct_Any8Store> { };
class ct_Any16RefStore:
  public gct_RefStore <ct_Any16Store> { };
class ct_Any32RefStore:
  public gct_RefStore <ct_Any32Store> { };
```

Every directory of a global store contains a file '**refstore.h**'.

The file 'tuning/std/refstore.h' contains the following declarations:

```
class ct_Std_RefStore;
class ct_Std8RefStore;
class ct_Std16RefStore;
class ct_Std32RefStore;
```

The file 'tuning/rnd/refstore.h' contains the following declarations:

```
class ct_Rnd_RefStore;
class ct_Rnd8RefStore;
class ct_Rnd16RefStore;
class ct_Rnd32RefStore;
```

The file 'tuning/chn/refstore.h' contains the following declarations:

```
class ct_Chn_RefStore;
class ct_Chn8RefStore;
class ct_Chn16RefStore;
class ct_Chn32RefStore;
```

1.5.6 Block-Ref-Store Instances (tuning/xxx/blockrefstore.h)

A block-ref-store is a ref-store enhancement of a block store.
Some template instances are predefined to easily use block-ref-stores. The macro `BLOCKREF_STORE_DCLS(Obj)` generates for each wrapper class of a global store one block-ref-store class.

The macro

```
BLOCKREF_STORE_DCLS (Any)
```

expands to:

```
class ct_Any_BlockRefStore:
  public gct_RefStore <ct_Any_BlockStore> { };
class ct_Any8BlockRefStore:
  public gct_RefStore <ct_Any8BlockStore> { };
class ct_Any16BlockRefStore:
  public gct_RefStore <ct_Any16BlockStore> { };
class ct_Any32BlockRefStore:
  public gct_RefStore <ct_Any32BlockStore> { };
```

Every directory of a global store contains a file **'blockrefstore.h'.**

The file 'tuning/std/blockrefstore.h' contains the following declarations:

```
class ct_Std_BlockRefStore;
class ct_Std8BlockRefStore;
class ct_Std16BlockRefStore;
class ct_Std32BlockRefStore;
```

The file 'tuning/rnd/blockrefstore.h' contains the following declarations:

```
class ct_Rnd_BlockRefStore;
class ct_Rnd8BlockRefStore;
class ct_Rnd16BlockRefStore;
class ct_Rnd32BlockRefStore;
```

The file 'tuning/chn/blockrefstore.h' contains the following declarations:

```
class ct_Chn_BlockRefStore;
class ct_Chn8BlockRefStore;
class ct_Chn16BlockRefStore;
class ct_Chn32BlockRefStore;
```

1.5.7 Pack Store (tuning/packstore.hpp)

A pack store is optimized for many successive memory allocations which can be released in a single step. Typical use cases are temporary data inside of a complex calculation.

The internal memory layout algorithm is very simple. A pack store uses successively the space of a data page. Memory requests may have an arbitrary size. If the remaining space of the data page is too small for a new memory request, a new data page is used. If the size of a memory request is greater than a configurable minimum size, the new memory block uses its own data page.

Reallocation and release of single memory blocks are not implemented. However, a pack store can release the entire memory by calling the method `FreeAll`.

This special implementation uses a class with virtual functions instead of template parameters. The pack store uses a helper block for managing pointers to the pages. Different store classes can be used for the management block and the data pages.

The implementation of the pack store consists of the base class `ct_PackStoreBase` with some virtual methods and the derived class `ct_PackStore` with access to two store objects.

Class Declaration

```
class ct_PackStoreBase
  {
public:
  typedef t_UInt           t_Size;
  typedef void *           t_Position;

protected:
  virtual void *           ReallocPtr (void * pv_mem, t_Size o_size) = 0;
  virtual t_UInt           MaxDataAlloc () const = 0;
  virtual void *           AllocData (t_Size o_size) = 0;
  virtual void             FreeData (void * pv_mem) = 0;

public:
                           ct_PackStoreBase ();
  virtual                  ~ct_PackStoreBase () { }
  void                     Swap (ct_PackStoreBase & co_swap);

  static inline t_UInt     StoreInfoSize ();
  inline t_UInt            MaxAlloc ();

  t_Position               Alloc (t_Size o_size);
  t_Position               Realloc (t_Position o_pos, t_Size o_size);
  void                     Free (t_Position o_pos);

  static inline void *     AddrOf (t_Position o_pos);
  static inline t_Position PosOf (void * pv_adr);

  t_Size                   SizeOf (t_Position o_pos);
  t_Size                   RoundedSizeOf (t_Position o_pos);

  bool                     CanFreeAll ();
  void                     FreeAll (bool b_keepPage = false);

  bool                     Init (t_Size o_align, t_Size o_pageSize,
                             t_Size o_ownPageSize = 0);
```

Additional Methods

`bool Init (t_Size o_align, t_Size o_pageSize, t_Size o_ownPageSize = 0);`

Initializes an empty pack store. The parameter `o_align` determines the alignment of memory blocks (1, 2, 4, 8 or 16 bytes). The parameter `o_pageSize` determines the size of data pages. The optional parameter `o_ownPageSize` determines the minimum size of own data pages (default: `o_pageSize / 4`). If the size of a memory request is greater than this minimum size, the new memory block uses its own data page.

Class Declaration

```
class ct_PackStore: public ct_PackStoreBase
  {
protected:
  virtual void *       ReallocPtr (void * pv_mem, t_Size o_size);
  virtual t_UInt       MaxDataAlloc () const;
  virtual void *       AllocData (t_Size o_size);
  virtual void         FreeData (void * pv_mem);
```

```
public:
                        ~ct_PackStore ();
  };
```

Methods

`void * ReallocPtr (void * pv_mem, t_Size o_size);`

Reallocate memory for the pointer management block.

`t_UInt MaxDataAlloc () const;`

Returns the maximum size of a contiguous data block.

`void * AllocData (t_Size o_size);`

Allocate a single data page.

`void FreeData (void * pv_mem);`

Release a single data page.

`~ct_PackStore ();`

Within the destructor of the derived class all memory must be released. The destructor of the base class has no access to the virtual methods implemented in the derived class.

2 OBJECT MANAGEMENT

2.1 Container

2.1.1 Container Interface

Containers and collections are two different concepts to manage sets of C++ objects. A collection can manage a polymorphic set of objects which are derived from a common base class. A container manages a uniform set of objects. It also contains the objects itself, i.e. the underlying memory. A container can optimize memory usage in many different ways.

Like store classes, all container classes share a common interface. So it's easy to switch between multiple container implementations.

Template Declaration

```
template <class t_obj>
  class gct_AnyContainer
    {
  public:
    typedef t_UInt       t_Length;
    typedef void *       t_Position;
    typedef t_obj        t_Object;

                         gct_AnyContainer ();
                         gct_AnyContainer (const gct_AnyContainer & co);
                         ~gct_AnyContainer ();
    gct_AnyContainer &   operator = (const gct_AnyContainer & co_asgn);
    void                 Swap (gct_AnyContainer & co_swap);

    bool                 IsEmpty () const;
    t_Length             GetLen () const;

    t_Position           First () const;
    t_Position           Last () const;
    t_Position           Next (t_Position o_pos) const;
    t_Position           Prev (t_Position o_pos) const;
    t_Position           Nth (t_Length u_idx) const;

    t_Object *           GetObj (t_Position o_pos) const;
    t_Position           AddObj (const t_Object * po_obj = 0);
    t_Position           AddObjBefore (t_Position o_pos, const t_Object * po_obj = 0);
    t_Position           AddObjAfter (t_Position o_pos, const t_Object * po_obj = 0);

    void                 AppendObj (const t_Object * po_obj = 0, t_Length o_count = 1);
    void                 TruncateObj (t_Length o_count = 1);

    t_Position           DelObj (t_Position o_pos);
    void                 DelAll ();
    t_Position           FreeObj (t_Position o_pos);
    void                 FreeAll ();
    };
```

Object Type Requirements

The **Spirick** container interface consists of a basic interface (described in this section) and various enhancements (e.g. the comp-container interface). The object type requirements of the basic interface are very simple. A class type must contain a default and a copy constructor, no other requirements have to be fulfilled. Numeric and pointer types can also be used.

Object Constructor, Destructor

A container contains the objects itself, i.e. the underlying memory, and it calls the constructors and destructors of the managed objects. If a new object is added to a container, the default constructor is called. If an existing object is added to a container, the copy constructor of a new object is called and the existing object remains unchanged. If an object is deleted from a container, the destructor is called and the memory is released to the underlying store object.

Copy/Move Object Memory

The C++ standard (ISO/IEC 14882) states that only "trivially copyable" objects may be copied or moved by `memcpy` and `memmove`. However, in almost all cases C++ objects can be copied or moved by `memcpy` and `memmove` without any side effects. Another possibility is to copy the objects by copy constructors and assignment operators. In this case the performance would significantly drop. That's why some **Spirick** containers copy and move objects by `memcpy` and `memmove`. Note that there are some rare cases where objects must not be copied by `memcpy` and `memmove`, e.g. lowlevel mutex objects.

Stores and Containers

There are some similarities between **Spirick** stores and containers. Like stores the containers use position values to manage their contents. The store method `Alloc` is similar to the container method `AddObj`. The store method `AddrOf` is similar to the container method `GetObj`. The store method `Free` is similar to the container method `DelObj` etc.

Validity of Position Values

Spirick stores ensure the validity of position values until the method `Free` is called. In contrast, some **Spirick** containers ensure the validity of position values and some do not. For example, list containers (like store objects) ensure the validity of position values. But, if an array container was modified by adding or deleting an object, the position values of all subsequent entries become invalid.

Data Types

`typedef t_UInt t_Length;`

The nested type `t_Length` describes the number of contained objects, examples are `t_UInt`, `t_UInt8`, `t_UInt16` and `t_UInt32`. If `t_Length` is defined as `t_UInt8`, the maximum number of entries will be 255. The size of the container object can be reduced in some cases.

`typedef void * t_Position;`

Like store classes, container classes use position values to manage their objects, examples are `void *`, `t_UInt`, `t_UInt8`, `t_UInt16` and `t_UInt32`. The position value zero is invalid per definition. The method `GetObj` returns a pointer to the object at a specific position. If the position type is `void *`, the position value may (or may not) be equal to the object pointer. Hence, always use the method `GetObj` to access objects and do not use the position value itself.

`typedef t_obj t_Object;`

The nested type `t_Object` corresponds to the template parameter `t_obj`. It can be used by derived classes.

Constructors, Destructor, Assignment, Swap

`gct_AnyContainer ();`

Initializes an empty container object.

gct_AnyContainer (const gct_AnyContainer & co_init);

The copy constructor copies the contents of an existing container by using the copy constructors of the contained objects.

~gct_AnyContainer ();

The destructor clears the container by calling the method `DelAll`.

gct_AnyContainer & **operator =** (const gct_AnyContainer & co_asgn);

The assignment operator copies the contents of an existing container by using the copy constructors of the contained objects.

void **Swap** (gct_AnyContainer & co_swap);

Swaps the contents of the two container objects.

Number of Objects

bool **IsEmpty** () const;

Returns `true` if the container is empty.

t_Length **GetLen** () const;

Returns the number of contained objects.

Iterate over Objects

t_Position **First** () const;

Returns the position of the first object or zero if the container is empty.

t_Position **Last** () const;

Returns the position of the last object or zero if the container is empty.

t_Position **Next** (t_Position o_pos) const;

Returns the position of the next object or zero if `o_pos` is the position of the last object. `o_pos` must be a valid position value.

t_Position **Prev** (t_Position o_pos) const;

Returns the position of the previous object or zero if `o_pos` is the position of the first object. `o_pos` must be a valid position value.

t_Position **Nth** (t_Length u_idx) const;

Returns the position of the nth object (`0 < u_idx <= GetLen`).
Note that there is no zeroth object. The first object has index 1.

Access to Objects

t_Object * **GetObj** (t_Position o_pos) const;

Returns a pointer to the object at position `o_pos`. `o_pos` must be a valid position value.

Add Objects

`t_Position AddObj (const t_Object * po_obj = 0);`

Adds an object and returns the position of the new object. The logical position of the new object depends on the container implementation. If `po_obj` equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

`t_Position AddObjBefore (t_Position o_pos, const t_Object * po_obj = 0);`

Adds an object before a specific position and returns the position of the new object. If `o_pos` equals zero, the new object is appended after the last object, i.e. it will be the new last object. If `po_obj` equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

`t_Position AddObjAfter (t_Position o_pos, const t_Object * po_obj = 0);`

Adds an object after a specific position and returns the position of the new object. If `o_pos` equals zero, the new object is inserted before the first object, i.e. it will be the new first object. If `po_obj` equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

Append/Truncate Multiple Objects

`void AppendObj (const t_Object * po_obj = 0, t_Length o_count = 1);`

Adds `o_count` objects at the end of the container. If `po_obj` equals zero, the new objects are created by the default constructor, otherwise the copy constructor is used.

`void TruncateObj (t_Length o_count = 1);`

Deletes `o_count` objects at the end of the container.

Return Value of Delete Methods

Delete methods always return the position of the successor of the deleted entry. With this technique, a container can be iterated and modified at the same time. If the last object was deleted, the return value equals zero.

Delete Objects

`t_Position DelObj (t_Position o_pos);`

Deletes the object at position `o_pos`. Calls the destructor of the object and releases the corresponding memory. `o_pos` must be a valid position value. The method returns `Next (o_pos)`, i.e. the position of the next object or zero, if the last object was deleted.

`void DelAll ();`

Deletes all contained objects. Calls the destructor of the objects and releases the corresponding memory.

`t_Position FreeObj (t_Position o_pos);`

Deletes the object at position `o_pos` <u>without calling the destructor</u>. This method is slightly faster than `DelObj`. `o_pos` must be a valid position value. The method returns `Next (o_pos)`, i.e. the position of the next object or zero, if the last object was deleted.

`void FreeAll ();`

Releases the entire memory <u>without calling the destructor</u> of the contained objects.

Exception Handling

While working with containers, exceptions may occur inside of constructors and destructors of contained objects. **Spirick** container classes contain minimal exception handlers. These handlers ensure the consistency of the container object and pass the exception unchanged to a higher-level handler.

The following rules apply:

If the exception occurs inside of the constructor while adding a new object (`AddObj`), the container remains unchanged (no new object will be added).
If the exception occurs inside of the destructor while deleting an object (`DelObj`), the object will be deleted anyway.
If the exception occurs inside of a constructor while adding several objects (`AppendObj`), the insertion is aborted. All previously added objects remain unchanged.
If the exception occurs inside of a destructor while deleting several objects (`TruncateObj`), the deletion is aborted. The object causing the exception will be deleted anyway.
If the exception occurs inside of a destructor while deleting all objects (`DelAll`), the deletion will be continued. Afterwards the container will be empty.
If the exception occurs inside of the container copy constructor or assignment operator, the method `DelAll` will be called.

2.1.2 Container Operations

Insert, Copy and Delete Objects

The following sample code demonstrates some simple container operations. The class `ct_Int` is described in the section 'Sample Programs'.

```
ct_Int co_int = 1;
ct_Int * pco_int;
gct_AnyContainer <ct_Int> co_container;
gct_AnyContainer <ct_Int>::t_Position o_pos;

// Add a new object by calling the default constructor
o_pos = co_container. AddObj ();

// Access the object and initialize it
pco_int = co_container. GetObj (o_pos);
(* pco_int) = 2;

// Copy an existing object into the container
o_pos = co_container. AddObj (& co_int);

// Delete a single object
co_container. DelObj (o_pos);
```

Iterate Forward

The following sample code demonstrates a forward iteration over a container.

```
gct_AnyContainer <float> co_container;
gct_AnyContainer <float>::t_Position o_pos;

for (o_pos = co_container. First ();
     o_pos != 0;
     o_pos = co_container. Next (o_pos))
  {
  float * pf = co_container. GetObj (o_pos);
  // ...
  }
```

Iterate Backward

The following sample code demonstrates a backward iteration over a container.

```
gct_AnyContainer <float> co_container;
gct_AnyContainer <float>::t_Position o_pos;

for (o_pos = co_container. Last ();
     o_pos != 0;
     o_pos = co_container. Prev (o_pos))
  {
  float * pf = co_container. GetObj (o_pos);
  // ...
  }
```

Iterate and Modify

The following sample code demonstrates how to iterate and modify a container.

```
gct_AnyContainer <float> co_container;
gct_AnyContainer <float>::t_Position o_pos;

for (o_pos = co_container. First ();
     o_pos != 0;
     o_pos = /* delete entry ? */ ?
             co_container. DelObj (o_pos) :
             co_container. Next (o_pos))
  {
  float * pf = co_container. GetObj (o_pos);
  // ...
  }
```

Alternatively a `while` loop can be used.

```
gct_AnyContainer <float> co_container;
gct_AnyContainer <float>::t_Position o_pos;

o_pos = co_container. First ();

while (o_pos != 0)
  {
  float * pf = co_container. GetObj (o_pos);
  // ...
  if ( /* delete entry ? */ )
    o_pos = co_container. DelObj (o_pos);
  else
    o_pos = co_container. Next (o_pos);
  }
```

2.1.3 Extended Container (tuning/extcont.h)

The class template `gct_ExtContainer` enhances the usability of the basic container interface. Example: To access the nth object of a container, two methods must be called.

```
gct_AnyContainer <float> co_floats;
// ...
float f = co_floats. GetObj (co_floats. Nth (5));
```

For such a case the class template `gct_ExtContainer` provides the method `GetNthObj`.

The template parameter t_container must comply with the basic container interface. It is used as the base class of the extended container.

Base Class

gct_AnyContainer (see above 'Container Interface')

Template Declaration

```
template <class t_container>
  class gct_ExtContainer: public t_container
    {
  public:
    inline t_Object *    GetFirstObj () const;
    inline t_Object *    GetLastObj () const;
    inline t_Object *    GetNextObj (t_Position o_pos) const;
    inline t_Object *    GetPrevObj (t_Position o_pos) const;
    inline t_Object *    GetNthObj (t_Length u_idx) const;

    inline t_Position    AddObjBeforeFirst (const t_Object * po_obj = 0);
    inline t_Position    AddObjAfterLast (const t_Object * po_obj = 0);
    inline t_Position    AddObjBeforeNth (t_Length u_idx, const t_Object * po_obj = 0);
    inline t_Position    AddObjAfterNth (t_Length u_idx, const t_Object * po_obj = 0);

    t_Object *           GetNewObj (const t_Object * po_obj = 0);
    t_Object *           GetNewFirstObj (const t_Object * po_obj = 0);
    t_Object *           GetNewLastObj (const t_Object * po_obj = 0);
    t_Object *           GetNewObjBefore (t_Position o_pos, const t_Object * po_obj = 0);
    t_Object *           GetNewObjAfter (t_Position o_pos, const t_Object * po_obj = 0);
    t_Object *           GetNewObjBeforeNth (t_Length u_idx, const t_Object * po_obj = 0);
    t_Object *           GetNewObjAfterNth (t_Length u_idx, const t_Object * po_obj = 0);

    inline t_Position    DelFirstObj ();
    inline t_Position    DelLastObj ();
    inline t_Position    DelNextObj (t_Position o_pos);
    inline t_Position    DelPrevObj (t_Position o_pos);
    inline t_Position    DelNthObj (t_Length u_idx);

    inline t_Position    FreeFirstObj ();
    inline t_Position    FreeLastObj ();
    inline t_Position    FreeNextObj (t_Position o_pos);
    inline t_Position    FreePrevObj (t_Position o_pos);
    inline t_Position    FreeNthObj (t_Length u_idx);
    };

// Example of an implementation
template <class t_container>
  inline gct_ExtContainer <t_container>:: t_Object *
  gct_ExtContainer <t_container>:: GetNthObj (t_Length u_idx) const
    {
    return GetObj (Nth (u_idx));
    }
```

Access to Objects

t_Object * **GetFirstObj** () const;

Returns a pointer to the first object. The container must contain at least one object.

t_Object * **GetLastObj** () const;

Returns a pointer to the last object. The container must contain at least one object.

t_Object * **GetNextObj** (t_Position o_pos) const;

Returns a pointer to the next object. o_pos and Next (o_pos) must be valid position values.

t_Object * **GetPrevObj** (t_Position o_pos) const;

Returns a pointer to the previous object. o_pos and Prev (o_pos) must be valid position values.

t_Object * **GetNthObj** (t_Length u_idx) const;

Returns a pointer to the nth object (0 < u_idx <= GetLen).

Add Objects

t_Position **AddObjBeforeFirst** (const t_Object * po_obj = 0);

Adds an object before the first object and returns the position of the new object. The new object will be the new first object. If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

t_Position **AddObjAfterLast** (const t_Object * po_obj = 0);

Adds an object after the last object and returns the position of the new object. The new object will be the new last object. If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

t_Position **AddObjBeforeNth** (t_Length u_idx, const t_Object * po_obj = 0);

Adds an object before the nth object and returns the position of the new object (0 < u_idx <= GetLen). If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

t_Position **AddObjAfterNth** (t_Length u_idx, const t_Object * po_obj = 0);

Adds an object after the nth object and returns the position of the new object (0 < u_idx <= GetLen). If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

Access to New Objects

t_Object * **GetNewObj** (const t_Object * po_obj = 0);

Adds an object and returns a pointer to the new object. The logical position of the new object depends on the container implementation. If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

t_Object * **GetNewFirstObj** (const t_Object * po_obj = 0);

Adds an object before the first object and returns a pointer to the new object. The new object will be the new first object. If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

t_Object * **GetNewLastObj** (const t_Object * po_obj = 0);

Adds an object after the last object and returns a pointer to the new object. The new object will be the new last object. If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

t_Object * **GetNewObjBefore** (t_Position o_pos, const t_Object * po_obj = 0);

Adds an object before a specific position and returns a pointer to the new object. If o_pos equals zero, the new object is appended after the last object, i.e. it will be the new last object. If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

t_Object * **GetNewObjAfter** (t_Position o_pos, const t_Object * po_obj = 0);

Adds an object after a specific position and returns a pointer to the new object. If o_pos equals zero, the new object is inserted before the first object, i.e. it will be the new first object. If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

t_Object * **GetNewObjBeforeNth** (t_Length u_idx, const t_Object * po_obj = 0);

Adds an object before the nth object and returns a pointer to the new object (0 < u_idx <= GetLen). If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

t_Object * **GetNewObjAfterNth** (t_Length u_idx, const t_Object * po_obj = 0);

Adds an object after the nth object and returns a pointer to the new object (0 < u_idx <= GetLen). If po_obj equals zero, the new object is created by the default constructor, otherwise the copy constructor is used.

Return Value of Delete Methods

Delete methods always return the position of the successor of the deleted entry. With this technique, a container can be iterated and modified at the same time. If the last object was deleted, the return value equals zero.

Delete Objects

t_Position **DelFirstObj** ();

Deletes the first object. Calls the destructor of the object and releases the corresponding memory. The container must contain at least one object. The method returns the position of the new first object or zero, if the last object was deleted.

t_Position **DelLastObj** ();

Deletes the last object. Calls the destructor of the object and releases the corresponding memory. The container must contain at least one object. The method always returns zero, because the last object was deleted.

t_Position **DelNextObj** (t_Position o_pos);

Deletes the object at position Next (o_pos). Calls the destructor of the object and releases the corresponding memory. o_pos and Next (o_pos) must be valid position values. The method returns Next (Next (o_pos)), i.e. the position of the next object of the deleted object or zero, if the last object was deleted.

t_Position **DelPrevObj** (t_Position o_pos);

Deletes the object at position Prev (o_pos). Calls the destructor of the object and releases the corresponding memory. o_pos and Prev (o_pos) must be valid position values. The method returns o_pos, because it is the position of the next object of the deleted object.

t_Position **DelNthObj** (t_Length u_idx);

Deletes the nth object (0 < u_idx <= GetLen). Calls the destructor of the object and releases the corresponding memory. The method returns Next (Nth (u_idx)), i.e. the position of the next object of the deleted object or zero, if the last object was deleted.

t_Position **FreeFirstObj** ();

Deletes the first object without calling the destructor. The container must contain at least one object. The method returns the position of the new first object or zero, if the last object was deleted.

t_Position **FreeLastObj** ();

Deletes the last object without calling the destructor. The container must contain at least one object. The method always returns zero, because the last object was deleted.

t_Position **FreeNextObj** (t_Position o_pos);

Deletes the object at position Next (o_pos) without calling the destructor. o_pos and Next (o_pos) must be valid position values. The method returns Next (Next (o_pos)), i.e. the position of the next object of the deleted object or zero, if the last object was deleted.

t_Position **FreePrevObj** (t_Position o_pos);

Deletes the object at position Prev (o_pos) without calling the destructor. o_pos and Prev (o_pos) must be valid position values. The method returns o_pos, because it is the position of the next object of the deleted object.

t_Position **FreeNthObj** (t_Length u_idx);

Deletes the nth object without calling the destructor (0 < u_idx <= GetLen). The method returns Next (Nth (u_idx)), i.e. the position of the next object of the deleted object or zero, if the last object was deleted.

2.2 Array and List Containers

2.2.1 Array Containers (tuning/array.h)

Array containers are optimized for size. Like static arrays, array containers store objects contiguous, without any management overhead. If an array container was modified by adding or deleting an object, all subsequent entries are moved by memmove and the position values of these objects become invalid. The validity of memory addresses depends on the implementation of the underlying block class. Array containers provide direct access to the nth object.

The first template parameter t_obj is the type of the contained objects. The second template parameter t_block must at least contain the item block interface. It is used as the base class of the array container. The helper class template gct_FixItemArray passes the size of an object to the class template gct_FixItemBlock.

Base Class

gct_...ItemBlock (see above 'Item Block')

Template Declaration

```
template <class t_obj, class t_block>
  class gct_Array: public t_block
    {
  public:
    typedef t_block::t_Size t_Length;
    typedef t_block::t_Size t_Position;
    typedef t_obj           t_Object;

    inline              gct_Array ();
    inline              gct_Array (const gct_Array & co_init);
    inline              ~gct_Array ();
    inline gct_Array &  operator = (const gct_Array & co_asgn);

    inline bool         IsEmpty () const;
    inline t_Length     GetMaxLen () const;
    inline t_Length     GetLen () const;
```

```
    inline t_Position    First () const;
    inline t_Position    Last () const;
    inline t_Position    Next (t_Position o_pos) const;
    inline t_Position    Prev (t_Position o_pos) const;
    inline t_Position    Nth (t_Length u_idx) const;

    inline t_Object *    GetObj (t_Position o_pos) const;
    inline t_Position    AddObj (const t_Object * po_obj = 0);
    inline t_Position    AddObjBefore (t_Position o_pos, const t_Object * po_obj = 0);
    t_Position           AddObjAfter (t_Position o_pos, const t_Object * po_obj = 0);

    void                 AppendObj (const t_Object * po_obj = 0, t_Length o_count = 1);
    void                 TruncateObj (t_Length o_count = 1);

    t_Position           DelObj (t_Position o_pos);
    void                 DelAll ();
    inline t_Position    FreeObj (t_Position o_pos);
    inline void          FreeAll ();

    inline void          SetPageSize (t_Size o_size);
    };
```

Additional Methods

t_Length **GetMaxLen** () const;

Returns the maximum number of contained objects.

void **SetPageSize** (t_Size o_size);

Sets the page size, if ct_PageBlock is used as template parameter t_block.

Template Declaration

```
template <class t_obj, class t_block>
  class gct_FixItemArray:
    public gct_Array <t_obj, gct_FixItemBlock <t_block, sizeof (gct_ArrayNode <t_obj>)> >
    {
    };
```

2.2.2 Array Instances (tuning/xxx/array.h)

Some template instances are predefined to easily use array containers. The macro ARRAY_DCLS(Obj) generates for each wrapper class of a global store one array template.

The macro

```
ARRAY_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_Array:
  public gct_ExtContainer <gct_FixItemArray <t_obj, ct_Any_Block> > { };
template <class t_obj> class gct_Any8Array:
  public gct_ExtContainer <gct_FixItemArray <t_obj, ct_Any8Block> > { };
template <class t_obj> class gct_Any16Array:
  public gct_ExtContainer <gct_FixItemArray <t_obj, ct_Any16Block> > { };
template <class t_obj> class gct_Any32Array:
  public gct_ExtContainer <gct_FixItemArray <t_obj, ct_Any32Block> > { };
```

Every directory of a global store contains a file '**array.h**'.

The file 'tuning/std/array.h' contains the following declarations:

```
template <class t_obj> class gct_Std_Array;
template <class t_obj> class gct_Std8Array;
template <class t_obj> class gct_Std16Array;
template <class t_obj> class gct_Std32Array;
```

The file 'tuning/rnd/array.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_Array;
template <class t_obj> class gct_Rnd8Array;
template <class t_obj> class gct_Rnd16Array;
template <class t_obj> class gct_Rnd32Array;
```

The file 'tuning/chn/array.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_Array;
template <class t_obj> class gct_Chn8Array;
template <class t_obj> class gct_Chn16Array;
template <class t_obj> class gct_Chn32Array;
```

2.2.3 List Containers (tuning/dlist.h)

List containers are optimized for fast random modification and for validity of position values. If a list entry is added or deleted, only the direct neighbors are affected. All other list entries remain unchanged. The position value of a list entry remains valid until the entry is deleted. This feature is important if references (position values) to list entries are stored permanently.

The validity of memory addresses depends on the implementation of the underlying store class. If a predefined global store or a page-based block store is used, memory addresses of list entries remain valid. If a non-paged block store is used, memory addresses of list entries can change, if the size of the underlying block changes.

Note that every list node contains references (position values) to the direct neighbors. Note also that every list node is allocated separately. If a predefined global store is used, rounding and management overhead occurs at every single list node. This overhead can be avoided by using a block store.

The first template parameter `t_obj` is the type of the contained objects. The second template parameter `t_store` must at least contain the store interface. The list class contains a data member of type `t_store`. The additional method `GetStore` provides access to the store object.

Template Declaration

```
template <class t_obj, class t_store>
  class gct_DList
    {
  public:
    typedef t_store::t_Size     t_Length;
    typedef t_store::t_Position t_Position;
    typedef t_obj               t_Object;

    inline              gct_DList ();
    inline              gct_DList (const gct_DList & co_init);
    inline              ~gct_DList ();
    inline gct_DList &  operator = (const gct_DList & co_asgn);
    void                Swap (gct_DList & co_swap);

    inline bool         IsEmpty () const;
    inline t_Length     GetLen () const;
```

```
    inline t_Position    First () const;
    inline t_Position    Last () const;
    inline t_Position    Next (t_Position o_pos) const;
    inline t_Position    Prev (t_Position o_pos) const;
    t_Position           Nth (t_Length u_idx) const;

    inline t_Object *    GetObj (t_Position o_pos) const;
    inline t_Position    AddObj (const t_Object * po_obj = 0);
    inline t_Position    AddObjBefore (t_Position o_pos, const t_Object * po_obj = 0);
    t_Position           AddObjAfter (t_Position o_pos, const t_Object * po_obj = 0);

    void                 AppendObj (const t_Object * po_obj = 0, t_Length o_count = 1);
    void                 TruncateObj (t_Length o_count = 1);

    t_Position           DelObj (t_Position o_pos);
    void                 DelAll ();
    t_Position           FreeObj (t_Position o_pos);
    void                 FreeAll ();

    inline t_store *     GetStore ();
    };
```

2.2.4 List Instances (tuning/xxx/dlist.h)

Some template instances are predefined to easily use list containers. The macro `DLIST_DCLS(Obj)` generates for each wrapper class of a global store one list template.

The macro

```
DLIST_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_DList:
  public gct_ExtContainer <gct_DList <t_obj, ct_Any_Store> > { };
template <class t_obj> class gct_Any8DList:
  public gct_ExtContainer <gct_DList <t_obj, ct_Any8Store> > { };
template <class t_obj> class gct_Any16DList:
  public gct_ExtContainer <gct_DList <t_obj, ct_Any16Store> > { };
template <class t_obj> class gct_Any32DList:
  public gct_ExtContainer <gct_DList <t_obj, ct_Any32Store> > { };
```

Every directory of a global store contains a file **'dlist.h'.**

The file 'tuning/std/dlist.h' contains the following declarations:

```
template <class t_obj> class gct_Std_DList;
template <class t_obj> class gct_Std8DList;
template <class t_obj> class gct_Std16DList;
template <class t_obj> class gct_Std32DList;
```

The file 'tuning/rnd/dlist.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_DList;
template <class t_obj> class gct_Rnd8DList;
template <class t_obj> class gct_Rnd16DList;
template <class t_obj> class gct_Rnd32DList;
```

The file 'tuning/chn/dlist.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_DList;
```

```
template <class t_obj> class gct_Chn8DList;
template <class t_obj> class gct_Chn16DList;
template <class t_obj> class gct_Chn32DList;
```

2.3 Sorted Containers

2.3.1 Sorted Arrays (tuning/sortarr.h)

Sorted array containers are very similar to normal array containers. The main difference between these two concepts is the order in which objects are positioned. The object type of a sorted array container must provide a comparison function 'operator <'. New objects are added by AddObj. They are sorted automatically in ascending order. Adding multiple equal objects is possible. They are positioned in the order they have been added.

Note that using the methods AddObjBefore and AddObjAfter is allowed, if the position is correct with respect to 'operator <'. The method AppendObj is not supported.

If the object type additionally provides the comparison function 'operator ==', the sorted array can be extended by the comp-container interface (see below 'Comp-Container'). In this case, an efficient binary search is used.

The first template parameter t_obj is the type of the contained objects. The second template parameter t_block must at least contain the item block interface. It is used as the base class of the sorted array container. The helper class template gct_FixItemSortedArray passes the size of an object to the class template gct_FixItemBlock.

Base Class

gct_...ItemBlock (see above 'Item Block')

Template Declaration

```
template <class t_obj, class t_block >
  class gct_SortedArray: public t_block
    {
  public:
    typedef t_block::t_Size t_Length;
    typedef t_block::t_Size t_Position;
    typedef t_obj           t_Object;

    inline              gct_SortedArray ();
    inline              gct_SortedArray (const gct_SortedArray & co_init);
    inline              ~gct_SortedArray ();
    inline gct_SortedArray & operator = (const gct_SortedArray & co_asgn);

    inline bool         IsEmpty () const;
    inline t_Length     GetMaxLen () const;
    inline t_Length     GetLen () const;

    inline t_Position   First () const;
    inline t_Position   Last () const;
    inline t_Position   Next (t_Position o_pos) const;
    inline t_Position   Prev (t_Position o_pos) const;
    inline t_Position   Nth (t_Length u_idx) const;

    inline t_Object *   GetObj (t_Position o_pos) const;

    t_Position          AddObj (const t_Object * po_obj);
```

```
    inline t_Position   AddObjBefore (t_Position o_pos, const t_Object * po_obj);
    t_Position          AddObjAfter (t_Position o_pos, const t_Object * po_obj);

    void                AppendObj (const t_Object * po_obj = 0, t_Length o_count = 1);
    void                TruncateObj (t_Length o_count = 1);

    t_Position          DelObj (t_Position o_pos);
    void                DelAll ();

    inline t_Position   FreeObj (t_Position o_pos);
    inline void         FreeAll ();

    inline void         SetPageSize (t_Size o_size);
    t_Position          Before (const t_Object * po_obj) const;
    };
```

Additional Methods

t_Length **GetMaxLen** () const;

Returns the maximum number of contained objects.

void **SetPageSize** (t_Size o_size);

Sets the page size, if ct_PageBlock is used as template parameter t_block.

t_Position **Before** (const t_Object * po_obj) const;

Returns the position of the last object which is smaller than or equal to * po_obj. Returns zero if * po_obj is smaller than the first object. Returns Last () if * po_obj is greater than or equal to the last object.

Template Declaration

```
template <class t_obj, class t_block>
  class gct_FixItemSortedArray:
    public gct_SortedArray <t_obj, gct_FixItemBlock <t_block, sizeof (gct_SortedArrayNode <t_obj>)> >
    {
    };
```

2.3.2 Sorted Array Instances (tuning/xxx/sortedarray.h)

Some template instances are predefined to easily use sorted array containers. The macro SORTEDARRAY_DCLS(Obj) generates for each wrapper class of a global store one sorted array template.

The macro

```
SORTEDARRAY_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_AnySortedArray:
  public gct_ExtContainer <gct_FixItemSortedArray <t_obj, ct_Any_Block> > { };
template <class t_obj> class gct_Any8SortedArray:
  public gct_ExtContainer <gct_FixItemSortedArray <t_obj, ct_Any8Block> > { };
template <class t_obj> class gct_Any16SortedArray:
  public gct_ExtContainer <gct_FixItemSortedArray <t_obj, ct_Any16Block> > { };
template <class t_obj> class gct_Any32SortedArray:
  public gct_ExtContainer <gct_FixItemSortedArray <t_obj, ct_Any32Block> > { };
```

Every directory of a global store contains a file **'sortedarray.h'**.

The file 'tuning/std/sortedarray.h' contains the following declarations:

```
template <class t_obj> class gct_Std_SortedArray;
template <class t_obj> class gct_Std8SortedArray;
template <class t_obj> class gct_Std16SortedArray;
template <class t_obj> class gct_Std32SortedArray;
```

The file 'tuning/rnd/sortedarray.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_SortedArray;
template <class t_obj> class gct_Rnd8SortedArray;
template <class t_obj> class gct_Rnd16SortedArray;
template <class t_obj> class gct_Rnd32SortedArray;
```

The file 'tuning/chn/sortedarray.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_SortedArray;
template <class t_obj> class gct_Chn8SortedArray;
template <class t_obj> class gct_Chn16SortedArray;
template <class t_obj> class gct_Chn32SortedArray;
```

2.3.3 Hash Tables (tuning/hashtable.h)

Sorted arrays and hash tables are two different concepts for access acceleration in container classes. Sorted arrays are suitable for a small amount of data. If an array container becomes too large, modifications become time consuming. Hash tables are suitable for larger amounts of data. If a hash table contains only a few objects, the management overhead is relatively high.

The **Spirick** hash table container is a special implementation of the common hash table concept. It is implemented as an array of arrays. The outer array has a fixed size, the so-called 'hash size'. The result of 'hash value' modulo 'hash size' is an index for this array. An inner array contains all objects which have the same index value.

To reduce the number of collisions of index values, the hash size should be a prime number. The constants `u_HashPrime1` to `u_HashPrime16` are predefined. The hash size can be set by the method `SetHashSize` while the container is empty. The default value is `u_HashPrime4`.

The object type of a hash table must provide a hash function `GetHash` returning an unsigned integer value. New objects are added by `AddObj`. The methods `AddObjBefore`, `AddObjAfter`, `AppendObj` and `TruncateObj` are not supported. If the object type additionally provides the comparison function '`operator ==`', the hash table can be extended by the comp-container interface (see below 'Comp-Container'). In this case, an efficient hash search is used.

The first template parameter `t_obj` is the type of the contained objects. The second template parameter `t_block` must at least contain the block interface, e.g. `ct_Chn16Block`. It is used for inner and outer arrays.

Note that the position type of a hash table is a class containing two data members of type `t_block::t_Size`. Using `t_UInt16` or `t_UInt32` can improve performance. If a hash table container was modified by adding or deleting an object, the position values of other objects become invalid.

Template Declaration

```
const unsigned u_HashPrime1  =  1013;
const unsigned u_HashPrime2  =  2039;
const unsigned u_HashPrime4  =  4079;
const unsigned u_HashPrime8  =  8179;
const unsigned u_HashPrime16 = 16369;

template <class t_obj, class t_block>
  class gct_HashTable
```

```
  {
public:
  typedef t_block::t_Size                 t_Length;
  typedef gct_HashTablePosition <t_block> t_Position;
  typedef t_obj                           t_Object;

                     gct_HashTable ();
  void               Swap (gct_HashTable & co_swap);

  inline bool        IsEmpty () const;
  inline t_Length    GetLen () const;

  t_Position         First () const;
  t_Position         Last () const;
  t_Position         Next (t_Position o_pos) const;
  t_Position         Prev (t_Position o_pos) const;
  t_Position         Nth (t_Length u_idx) const;

  inline t_Object *  GetObj (t_Position o_pos) const;

  t_Position         AddObj (const t_Object * po_obj);
  t_Position         AddObjBefore (t_Position o_pos, const t_Object * po_obj);
  t_Position         AddObjAfter (t_Position o_pos, const t_Object * po_obj);

  void               AppendObj (const t_Object * po_obj = 0, t_Length o_count = 1);
  void               TruncateObj (t_Length o_count = 1);

  t_Position         DelObj (t_Position o_pos);
  void               DelAll ();

  t_Position         FreeObj (t_Position o_pos);
  void               FreeAll ();

  void               SetHashSize (t_Length o_size);
  inline t_Length    GetHashSize () const;
  };
```

Constants

const unsigned **cu_HashPrime1** = 1013;
const unsigned **cu_HashPrime2** = 2039;
const unsigned **cu_HashPrime4** = 4079;
const unsigned **cu_HashPrime8** = 8179;
const unsigned **cu_HashPrime16** = 16369;

These constants are recommended values for the hash size.

Additional Methods

void **SetHashSize** (t_Length o_size);

Sets the hash size while the container is empty.

t_Length **GetHashSize** () const;

Returns the hash size.

2.3.4 Hash Table Instances (tuning/xxx/hashtable.h)

Some template instances are predefined to easily use hash table containers. The macro HASHTABLE_DCLS(Obj) generates for each wrapper class of a global store one hash table template.

The macro

```
HASHTABLE_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_HashTable:
  public gct_ExtContainer <gct_HashTable <t_obj, ct_Any_Block> > { };
template <class t_obj> class gct_Any8HashTable:
  public gct_ExtContainer <gct_HashTable <t_obj, ct_Any8Block> > { };
template <class t_obj> class gct_Any16HashTable:
  public gct_ExtContainer <gct_HashTable <t_obj, ct_Any16Block> > { };
template <class t_obj> class gct_Any32HashTable:
  public gct_ExtContainer <gct_HashTable <t_obj, ct_Any32Block> > { };
```

Every directory of a global store contains a file **'hashtable.h'.**

The file 'tuning/std/hashtable.h' contains the following declarations:

```
template <class t_obj> class gct_Std_HashTable;
template <class t_obj> class gct_Std8HashTable;
template <class t_obj> class gct_Std16HashTable;
template <class t_obj> class gct_Std32HashTable;
```

The file 'tuning/rnd/hashtable.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_HashTable;
template <class t_obj> class gct_Rnd8HashTable;
template <class t_obj> class gct_Rnd16HashTable;
template <class t_obj> class gct_Rnd32HashTable;
```

The file 'tuning/chn/hashtable.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_HashTable;
template <class t_obj> class gct_Chn8HashTable;
template <class t_obj> class gct_Chn16HashTable;
template <class t_obj> class gct_Chn32HashTable;
```

2.4 Block and Ref Lists

2.4.1 Block Lists

Various store classes can be used to implement list containers. If a block store is used, the resulting container will be a 'block list'. Performance improvement: Every list node is allocated separately. If a predefined global store is used, rounding and management overhead occurs at every single list node. This overhead can be avoided by using a block store.

Note that every list node contains references (position values) to the direct neighbors. Using `t_UInt16` or `t_UInt32` can reduce the size of list nodes. Note also that if a non-paged block store is used, memory addresses of list entries can change, if the size of the underlying block changes.

2.4.2 Block List Instances (tuning/xxx/blockdlist.h)

Some template instances are predefined to easily use block list containers. The macro `BLOCK_DLIST_DCLS(Obj)` generates for each wrapper class of a global store one block list template.

The macro

```
BLOCK_DLIST_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_BlockDList:
  public gct_ExtContainer <gct_DList <t_obj, ct_Any_BlockStore> > { };
template <class t_obj> class gct_Any8BlockDList:
  public gct_ExtContainer <gct_DList <t_obj, ct_Any8BlockStore> > { };
template <class t_obj> class gct_Any16BlockDList:
  public gct_ExtContainer <gct_DList <t_obj, ct_Any16BlockStore> > { };
template <class t_obj> class gct_Any32BlockDList:
  public gct_ExtContainer <gct_DList <t_obj, ct_Any32BlockStore> > { };
```

Every directory of a global store contains a file **'blockdlist.h'.**

The file 'tuning/std/blockdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Std_BlockDList;
template <class t_obj> class gct_Std8BlockDList;
template <class t_obj> class gct_Std16BlockDList;
template <class t_obj> class gct_Std32BlockDList;
```

The file 'tuning/rnd/blockdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_BlockDList;
template <class t_obj> class gct_Rnd8BlockDList;
template <class t_obj> class gct_Rnd16BlockDList;
template <class t_obj> class gct_Rnd32BlockDList;
```

The file 'tuning/chn/blockdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_BlockDList;
template <class t_obj> class gct_Chn8BlockDList;
template <class t_obj> class gct_Chn16BlockDList;
template <class t_obj> class gct_Chn32BlockDList;
```

2.4.3 Ref-Lists (tuning/refdlist.h)

Various store classes can be used to implement list containers. If a ref-store is used, the resulting container will be a 'ref-list'. The class template `gct_RefDList` simplifies the access to the reference counters of the embedded store object.

Base Classes

`gct_DList` (see above 'List Containers')
  `gct_ExtContainer` (see above 'Extended Container')

Template Declaration

```
template <class t_obj, class t_store>
  class gct_RefDList:
    public gct_ExtContainer <gct_DList <t_obj, t_store> >
    {
  public:
    inline void          IncRef (t_Position o_pos);
    inline void          DecRef (t_Position o_pos);
    inline t_RefCount    GetRef (t_Position o_pos) const;
    inline bool          IsAlloc (t_Position o_pos) const;
    inline bool          IsFree (t_Position o_pos) const;
```

```
    };

// Example of an implementation
template <class t_obj, class t_store>
  inline void gct_RefDList <t_obj, t_store>::IncRef (t_Position o_pos)
    {
    o_Store. IncRef (o_pos);
    }
```

Each single entry of a ref-list is associated with a reference counter. The reference counters can be used directly or indirectly by special classes, e.g. smart pointers.

Note that the reference counter is associated with the memory of the ref-list entry and not with the C++ object. Deleting a ref-list entry and releasing the corresponding memory are two distinct steps. The ref-list entry can be deleted by its owner, and the corresponding memory can be released by the reference counter. If a ref-list entry is deleted and the reference counter is greater than zero, then all smart pointers remain valid, but access to the C++ object is not allowed. In this way isolated islands in complex, reference counting based data structures can be avoided.

If a ref-list entry is deleted (e.g. by `DelObj`), then the alloc flag of the corresponding `ct_RefCount` object is cleared. If additionally the reference counter equals zero, the memory of the ref-list entry is released by the underlying store object. Otherwise the reference counter can be changed by the ref-list methods `IncRef` and `DecRef`, but access to the C++ object by calling the method `GetObj` is not allowed. If the reference counter becomes zero, the memory is released by the underlying store object.

Methods

`void` **`IncRef`** `(t_Position o_pos);`

Increases the reference counter at position `o_pos` by 1.

`void` **`DecRef`** `(t_Position o_pos);`

Decreases the reference counter at position `o_pos` by 1.

`t_RefCount` **`GetRef`** `(t_Position o_pos) const;`

Returns the numeric reference counter at position `o_pos`.

`bool` **`IsAlloc`** `(t_Position o_pos) const;`

Returns the alloc flag of position value `o_pos`. If the method returns `true`, access by `GetObj` is allowed.

`bool` **`IsFree`** `(t_Position o_pos) const;`

Returns `true`, if the alloc flag of position value `o_pos` is not set.

2.4.4 Ref-List Instances (tuning/xxx/refdlist.h)

Some template instances are predefined to easily use ref-list containers. The macro `REF_DLIST_DCLS(Obj)` generates for each wrapper class of a global store one ref-list template.

The macro

```
REF_DLIST_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_RefDList:
  public gct_RefDList <t_obj, ct_Any_RefStore> { };
template <class t_obj> class gct_Any8RefDList:
  public gct_RefDList <t_obj, ct_Any8RefStore> { };
```

```
template <class t_obj> class gct_Any16RefDList:
  public gct_RefDList <t_obj, ct_Any16RefStore> { };
template <class t_obj> class gct_Any32RefDList:
  public gct_RefDList <t_obj, ct_Any32RefStore> { };
```

Every directory of a global store contains a file **'refdlist.h'**.

The file 'tuning/std/refdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Std_RefDList;
template <class t_obj> class gct_Std8RefDList;
template <class t_obj> class gct_Std16RefDList;
template <class t_obj> class gct_Std32RefDList;
```

The file 'tuning/rnd/refdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_RefDList;
template <class t_obj> class gct_Rnd8RefDList;
template <class t_obj> class gct_Rnd16RefDList;
template <class t_obj> class gct_Rnd32RefDList;
```

The file 'tuning/chn/refdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_RefDList;
template <class t_obj> class gct_Chn8RefDList;
template <class t_obj> class gct_Chn16RefDList;
template <class t_obj> class gct_Chn32RefDList;
```

2.4.5 Block-Ref-List Instances (tuning/xxx/blockrefdlist.h)

Various store classes can be used to implement list containers. If a block-ref-store is used, the resulting container will be a 'block-ref-list'.

Some template instances are predefined to easily use block-ref-list containers. The macro `BLOCKREF_DLIST_DCLS(Obj)` generates for each wrapper class of a global store one block-ref-list template.

The macro

```
BLOCKREF_DLIST_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_BlockRefDList:
  public gct_RefDList <t_obj, ct_Any_BlockRefStore> { };
template <class t_obj> class gct_Any8BlockRefDList:
  public gct_RefDList <t_obj, ct_Any8BlockRefStore> { };
template <class t_obj> class gct_Any16BlockRefDList:
  public gct_RefDList <t_obj, ct_Any16BlockRefStore> { };
template <class t_obj> class gct_Any32BlockRefDList:
  public gct_RefDList <t_obj, ct_Any32BlockRefStore> { };
```

Every directory of a global store contains a file **'blockrefdlist.h'**.

The file 'tuning/std/blockrefdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Std_BlockRefDList;
template <class t_obj> class gct_Std8BlockRefDList;
template <class t_obj> class gct_Std16BlockRefDList;
template <class t_obj> class gct_Std32BlockRefDList;
```

The file 'tuning/rnd/blockrefdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_BlockRefDList;
template <class t_obj> class gct_Rnd8BlockRefDList;
template <class t_obj> class gct_Rnd16BlockRefDList;
template <class t_obj> class gct_Rnd32BlockRefDList;
```

The file 'tuning/chn/blockrefdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_BlockRefDList;
template <class t_obj> class gct_Chn8BlockRefDList;
template <class t_obj> class gct_Chn16BlockRefDList;
template <class t_obj> class gct_Chn32BlockRefDList;
```

2.5 Comp, Pointer and Map Containers

2.5.1 Comp-Containers (tuning/compcontainer.h)

The **Spirick** container interface consists of a basic interface (see above) and various enhancements (e.g. the comp-container interface). The object type requirements of the basic interface are very simple. A class type must contain a default and a copy constructor, no other requirements have to be fulfilled. If the object type additionally provides the comparison function 'operator ==', the basic container can be extended by the comp-container interface. Numeric and pointer types can also be used.

The class template gct_CompContainer implements some count, search and conditional methods. If the base container is a normal (unsorted) array or a list, a linear search is used. Sorted arrays and hash tables provide accelerated algorithms for searching objects. The template parameter t_container must at least contain the basic container interface, e.g. gct_Std32Array <float>. It is used as the base class of the comp-container.

Base Classes

gct_AnyContainer (see above 'Container Interface')
[gct_ExtContainer (optional, see above 'Extended Container')]

Template Declaration

```
template <class t_container>
  class gct_CompContainer: public t_container
    {
  public:
    inline bool          ContainsObj (const t_Object * po_obj) const;
    t_Length             CountObjs (const t_Object * po_obj) const;

    t_Position           SearchFirstObj (const t_Object * po_obj) const;
    t_Position           SearchLastObj (const t_Object * po_obj) const;
    t_Position           SearchNextObj (t_Position o_pos, const t_Object * po_obj) const;
    t_Position           SearchPrevObj (t_Position o_pos, const t_Object * po_obj) const;

    inline t_Object *    GetFirstEqualObj (const t_Object * po_obj) const;
    inline t_Object *    GetLastEqualObj (const t_Object * po_obj) const;

    inline t_Position    AddObjCond (const t_Object * po_obj);
    inline t_Position    AddObjBeforeFirstCond (const t_Object * po_obj);
    inline t_Position    AddObjAfterLastCond (const t_Object * po_obj);

    inline t_Position    DelFirstEqualObj (const t_Object * po_obj);
    inline t_Position    DelLastEqualObj (const t_Object * po_obj);
    inline t_Position    DelFirstEqualObjCond (const t_Object * po_obj);
```

```
    inline t_Position    DelLastEqualObjCond (const t_Object * po_obj);
    };
```

Search for Objects

bool **ContainsObj** (const t_Object * po_obj) const;

Returns true, if a contained object is equal to * po_obj.

t_Length **CountObjs** (const t_Object * po_obj) const;

Returns the number of objects which are equal to * po_obj.

t_Position **SearchFirstObj** (const t_Object * po_obj) const;

Returns the position of the first object which is equal to * po_obj or zero if no object was found.

t_Position **SearchLastObj** (const t_Object * po_obj) const;

Returns the position of the last object which is equal to * po_obj or zero if no object was found.

t_Position **SearchNextObj** (t_Position o_pos, const t_Object * po_obj) const;

Returns the position of the next object which is equal to * po_obj or zero if no object was found. o_pos must be a valid position value.

t_Position **SearchPrevObj** (t_Position o_pos, const t_Object * po_obj) const;

Returns the position of the previous object which is equal to * po_obj or zero if no object was found. o_pos must be a valid position value.

Access to Found Objects

t_Object * **GetFirstEqualObj** (const t_Object * po_obj) const;

Returns a pointer to the first object which is equal to * po_obj. There must be at least one equal object.

t_Object * **GetLastEqualObj** (const t_Object * po_obj) const;

Returns a pointer to the last object which is equal to * po_obj. There must be at least one equal object.

Add Objects Conditionally

t_Position **AddObjCond** (const t_Object * po_obj);

Returns the position of the first object which is equal to * po_obj or the position of a new object (added by AddObj) if no equal object was found.

t_Position **AddObjBeforeFirstCond** (const t_Object * po_obj);

Returns the position of the first object which is equal to * po_obj or the position of a new object (added by AddObjBeforeFirst) if no equal object was found.

t_Position **AddObjAfterLastCond** (const t_Object * po_obj);

Returns the position of the first object which is equal to * po_obj or the position of a new object (added by AddObjAfterLast) if no equal object was found.

Return Value of Delete Methods

Delete methods always return the position of the successor of the deleted entry. With this technique, a container can be iterated and modified at the same time. If the last object was deleted, the return value equals zero.

Delete Found Objects

`t_Position` **`DelFirstEqualObj`** `(const t_Object * po_obj);`

Deletes the first object which is equal to `* po_obj`. There must be at least one equal object. The method returns the position of the next object of the deleted object or zero, if the last object was deleted.

`t_Position` **`DelLastEqualObj`** `(const t_Object * po_obj);`

Deletes the last object which is equal to `* po_obj`. There must be at least one equal object. The method returns the position of the next object of the deleted object or zero, if the last object was deleted.

Delete Found Objects Conditionally

`t_Position` **`DelFirstEqualObjCond`** `(const t_Object * po_obj);`

Deletes the first object which is equal to `* po_obj` or returns zero if no equal object was found. If an equal object was found the method returns the position of the next object of the deleted object or zero, if the last object was deleted.

`t_Position` **`DelLastEqualObjCond`** `(const t_Object * po_obj);`

Deletes the last object which is equal to `* po_obj` or returns zero if no equal object was found. If an equal object was found the method returns the position of the next object of the deleted object or zero, if the last object was deleted.

2.5.2 Pointer Containers (tuning/ptrcontainer.h)

A container can manage objects of many different types (e.g. `ct_String`, `int`, `float`). If the object type is a pointer type, some container methods are very unhandily. The method `GetObj` returns a pointer to a pointer, `AddObj` requires a parameter of type pointer to pointer etc.

```
gct_Rnd16Array <ct_String *> co_array;
gct_Rnd16Array <ct_String *>::t_Position o_pos;
ct_String * pco_str = new ct_String;
o_pos = co_array. AddObj (& pco_str);
pco_str = * co_array. GetObj (o_pos);
```

The class template `gct_PtrContainer` provides a comfortable interface for pointer containers. It maps many methods of the basic, extended and comp-container interface and provides some additional methods. To avoid confusions, method names contain the abbreviation `Ptr` (e.g. `GetPtr` instead of `GetObj`).

Note that a pointer container can be the owner of the referenced objects <u>or</u> it can manage pointers to objects which have a different owner. The method `DelPtrAndObj` deletes a pointer and the referenced object. The method `DelPtr` simply deletes the pointer, the referenced object remains unchanged.

Note also the difference between comparing the pointers and comparing the referenced objects. In C++ language pointers can be compared. That's why the pointer container interface provides methods of the comp-container interface. If the object type additionally provides the comparison function '`operator ==`', the pointer container can be extended by the pointer-comp-container interface (see next section).

C++ compilers generate binary code for each template instance. To reduce the size of the binary code the **Spirick** pointer containers are based on containers of object type `void *`. With this technique, many pointer container instances can share the same binary code.

The first template parameter `t_obj` is the type of the referenced objects. The second template parameter `t_container` must at least contain the extended container interface, e.g. `gct_Chn32DList <void *>`. It is extended by the comp-container interface and then used as the base class of the pointer container.

Base Classes

```
gct_AnyContainer          (see above 'Container Interface')
  gct_ExtContainer        (see above 'Extended Container')
    gct_CompContainer     (see above 'Comp-Container')
```

Template Declaration

```
template <class t_obj, class t_container>
  class gct_PtrContainer: public gct_CompContainer <t_container>
    {
  public:
    typedef t_obj        t_RefObject;

    inline               ~gct_PtrContainer ();

    inline t_obj *       GetPtr (t_Position o_pos) const;
    inline t_obj *       GetFirstPtr () const;
    inline t_obj *       GetLastPtr () const;
    inline t_obj *       GetNextPtr (t_Position o_pos) const;
    inline t_obj *       GetPrevPtr (t_Position o_pos) const;
    inline t_obj *       GetNthPtr (t_Length u_idx) const;

    inline t_Position    AddPtr (const t_obj * po_obj);
    inline t_Position    AddPtrBefore (t_Position o_pos, const t_obj * po_obj);
    inline t_Position    AddPtrAfter (t_Position o_pos, const t_obj * po_obj);
    inline t_Position    AddPtrBeforeFirst (const t_obj * po_obj);
    inline t_Position    AddPtrAfterLast (const t_obj * po_obj);
    inline t_Position    AddPtrBeforeNth (t_Length u_idx, const t_obj * po_obj);
    inline t_Position    AddPtrAfterNth (t_Length u_idx, const t_obj * po_obj);

    inline t_Position    DelPtr (t_Position o_pos);
    inline t_Position    DelFirstPtr ();
    inline t_Position    DelLastPtr ();
    inline t_Position    DelNextPtr (t_Position o_pos);
    inline t_Position    DelPrevPtr (t_Position o_pos);
    inline t_Position    DelNthPtr (t_Length u_idx);
    inline void          DelAllPtr ();

    inline t_Position    DelPtrAndObj (t_Position o_pos);
    inline t_Position    DelFirstPtrAndObj ();
    inline t_Position    DelLastPtrAndObj ();
    inline t_Position    DelNextPtrAndObj (t_Position o_pos);
    inline t_Position    DelPrevPtrAndObj (t_Position o_pos);
    inline t_Position    DelNthPtrAndObj (t_Length u_idx);
    inline void          DelAllPtrAndObj ();

    inline bool          ContainsPtr (const t_obj * po_obj) const;
    inline t_Length      CountPtrs (const t_obj * po_obj) const;

    inline t_Position    SearchFirstPtr (const t_obj * po_obj) const;
    inline t_Position    SearchLastPtr (const t_obj * po_obj) const;
    inline t_Position    SearchNextPtr (t_Position o_pos, const t_obj * po_obj) const;
    inline t_Position    SearchPrevPtr (t_Position o_pos, const t_obj * po_obj) const;

    inline t_Position    AddPtrCond (const t_obj * po_obj);
    inline t_Position    AddPtrBeforeFirstCond (const t_obj * po_obj);
    inline t_Position    AddPtrAfterLastCond (const t_obj * po_obj);

    inline t_Position    DelFirstEqualPtr (const t_obj * po_obj);
    inline t_Position    DelLastEqualPtr (const t_obj * po_obj);
    inline t_Position    DelFirstEqualPtrCond (const t_obj * po_obj);
    inline t_Position    DelLastEqualPtrCond (const t_obj * po_obj);

    inline t_Position    DelFirstEqualPtrAndObj (const t_obj * po_obj);
```

```
    inline t_Position    DelLastEqualPtrAndObj (const t_obj * po_obj);
    inline t_Position    DelFirstEqualPtrAndObjCond (const t_obj * po_obj);
    inline t_Position    DelLastEqualPtrAndObjCond (const t_obj * po_obj);
    };

// Example of an implementation
template <class t_obj, class t_container>
  inline t_obj * gct_PtrContainer <t_obj, t_container>::
  GetPtr (t_Position o_pos) const
    {
    return (t_obj *) * GetObj (o_pos);
    }

template <class t_obj, class t_container>
  inline gct_PtrContainer <t_obj, t_container>::t_Position
  gct_PtrContainer <t_obj, t_container>::
  DelPtrAndObj (t_Position o_pos)
    {
    delete GetPtr (o_pos);
    return FreeObj (o_pos);
    }
```

Data Types

typedef t_obj **t_RefObject**;

The nested type t_RefObject corresponds to the template parameter t_obj. It can be used by derived classes.

Destructor

~gct_PtrContainer ();

The destructor of a pointer container deletes all pointers by calling the method FreeAll, the referenced objects remain unchanged.

Access to Referenced Objects

t_obj * **GetPtr** (t_Position o_pos) const;

Returns a pointer to the object at position o_pos. o_pos must be a valid position value.

t_obj * **GetFirstPtr** () const;

Returns a pointer to the first object. The container must contain at least one pointer.

t_obj * **GetLastPtr** () const;

Returns a pointer to the last object. The container must contain at least one pointer.

t_obj * **GetNextPtr** (t_Position o_pos) const;

Returns a pointer to the next object. o_pos and Next (o_pos) must be valid position values.

t_obj * **GetPrevPtr** (t_Position o_pos) const;

Returns a pointer to the previous object. o_pos and Prev (o_pos) must be valid position values.

t_obj * **GetNthPtr** (t_Length u_idx) const;

Returns a pointer to the nth object (0 < u_idx <= GetLen).

Add Pointers

t_Position **AddPtr** (const t_obj * po_obj);

Adds a pointer and returns the position of the new pointer. The logical position of the new pointer depends on the container implementation.

t_Position **AddPtrBefore** (t_Position o_pos, const t_obj * po_obj);

Adds a pointer before a specific position and returns the position of the new pointer. If o_pos equals zero, the new pointer is appended after the last pointer, i.e. it will be the new last pointer.

t_Position **AddPtrAfter** (t_Position o_pos, const t_obj * po_obj);

Adds a pointer after a specific position and returns the position of the new pointer. If o_pos equals zero, the new pointer is inserted before the first pointer, i.e. it will be the new first pointer.

t_Position **AddPtrBeforeFirst** (const t_obj * po_obj);

Adds a pointer before the first pointer and returns the position of the new pointer. The new pointer will be the new first pointer.

t_Position **AddPtrAfterLast** (const t_obj * po_obj);

Adds a pointer after the last pointer and returns the position of the new pointer. The new pointer will be the new last pointer.

t_Position **AddPtrBeforeNth** (t_Length u_idx, const t_obj * po_obj);

Adds a pointer before the nth pointer and returns the position of the new pointer (0 < u_idx <= GetLen).

t_Position **AddPtrAfterNth** (t_Length u_idx, const t_obj * po_obj);

Adds a pointer after the nth pointer and returns the position of the new pointer (0 < u_idx <= GetLen).

Return Value of Delete Methods

Delete methods always return the position of the successor of the deleted entry. With this technique, a container can be iterated and modified at the same time. If the last object was deleted, the return value equals zero.

Delete Pointers

t_Position **DelPtr** (t_Position o_pos);

Deletes the pointer at position o_pos by calling the method FreeObj, the referenced object remains unchanged. o_pos must be a valid position value. The method returns Next (o_pos), i.e. the position of the next pointer or zero, if the last pointer was deleted.

t_Position **DelFirstPtr** ();

Deletes the first pointer by calling the method FreeFirstObj, the referenced object remains unchanged. The container must contain at least one pointer. The method returns the position of the new first pointer or zero, if the last pointer was deleted.

t_Position **DelLastPtr** ();

Deletes the last pointer by calling the method FreeLastObj, the referenced object remains unchanged. The container must contain at least one pointer. The method always returns zero, because the last pointer was deleted.

t_Position **DelNextPtr** (t_Position o_pos);

Deletes the pointer at position Next (o_pos) by calling the method FreeNextObj, the referenced object remains unchanged. o_pos and Next (o_pos) must be valid position values. The method returns Next (Next (o_pos)), i.e. the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

t_Position **DelPrevPtr** (t_Position o_pos);

Deletes the pointer at position Prev (o_pos) by calling the method FreePrevObj, the referenced object remains unchanged. o_pos and Prev (o_pos) must be valid position values. The method returns o_pos, because it is the position of the next pointer of the deleted pointer.

t_Position **DelNthPtr** (t_Length u_idx);

Deletes the nth pointer (0 < u_idx <= GetLen) by calling the method FreeNthObj, the referenced object remains unchanged. The method returns Next (Nth (u_idx)), i.e. the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

void **DelAllPtr** ();

Deletes all pointers by calling the method FreeAll, the referenced objects remain unchanged.

Delete Pointers and Referenced Objects

t_Position **DelPtrAndObj** (t_Position o_pos);

This method works like DelPtr and deletes the referenced object.

t_Position **DelFirstPtrAndObj** ();

This method works like DelFirstPtr and deletes the referenced object.

t_Position **DelLastPtrAndObj** ();

This method works like DelLastPtr and deletes the referenced object.

t_Position **DelNextPtrAndObj** (t_Position o_pos);

This method works like DelNextPtr and deletes the referenced object.

t_Position **DelPrevPtrAndObj** (t_Position o_pos);

This method works like DelPrevPtr and deletes the referenced object.

t_Position **DelNthPtrAndObj** (t_Length u_idx);

This method works like DelNthPtr and deletes the referenced object.

void **DelAllPtrAndObj** ();

This method works like DelAllPtr and deletes the referenced objects.

Compare Pointers

Note the difference between comparing the pointers and comparing the referenced objects. In C++ language pointers can be compared. That's why the pointer container interface provides methods of the comp-container interface. If the object type additionally provides the comparison function 'operator ==', the pointer container can be extended by the pointer-comp-container interface (see next section).

Search for Pointers

bool **ContainsPtr** (const t_obj * po_obj) const;

Returns true, if a contained pointer is equal to po_obj.

t_Length **CountPtrs** (const t_obj * po_obj) const;

Returns the number of pointers which are equal to po_obj.

t_Position **SearchFirstPtr** (const t_obj * po_obj) const;

Returns the position of the first pointer which is equal to po_obj or zero if no pointer was found.

t_Position **SearchLastPtr** (const t_obj * po_obj) const;

Returns the position of the last pointer which is equal to po_obj or zero if no pointer was found.

t_Position **SearchNextPtr** (t_Position o_pos, const t_obj * po_obj) const;

Returns the position of the next pointer which is equal to po_obj or zero if no pointer was found. o_pos must be a valid position value.

t_Position **SearchPrevPtr** (t_Position o_pos, const t_obj * po_obj) const;

Returns the position of the previous pointer which is equal to po_obj or zero if no pointer was found. o_pos must be a valid position value.

Add Pointers Conditionally

t_Position **AddPtrCond** (const t_obj * po_obj);

Returns the position of the first pointer which is equal to po_obj or the position of a new pointer (added by AddPtr) if no equal pointer was found.

t_Position **AddPtrBeforeFirstCond** (const t_obj * po_obj);

Returns the position of the first pointer which is equal to po_obj or the position of a new pointer (added by AddPtrBeforeFirst) if no equal pointer was found.

t_Position **AddPtrAfterLastCond** (const t_obj * po_obj);

Returns the position of the first pointer which is equal to po_obj or the position of a new pointer (added by AddPtrAfterLast) if no equal pointer was found.

Delete Found Pointers

t_Position **DelFirstEqualPtr** (const t_obj * po_obj);

Deletes the first pointer which is equal to po_obj. There must be at least one equal pointer. The method returns the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

t_Position **DelLastEqualPtr** (const t_obj * po_obj);

Deletes the last pointer which is equal to po_obj. There must be at least one equal pointer. The method returns the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

Delete Found Pointers Conditionally

t_Position **DelFirstEqualPtrCond** (const t_obj * po_obj);

Deletes the first pointer which is equal to po_obj or returns zero if no equal pointer was found. If an equal pointer was found the method returns the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

t_Position **DelLastEqualPtrCond** (const t_obj * po_obj);

Deletes the last pointer which is equal to po_obj or returns zero if no equal pointer was found. If an equal pointer was found the method returns the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

Delete Found Pointers and Referenced Objects

t_Position **DelFirstEqualPtrAndObj** (const t_obj * po_obj);

This method works like DelFirstEqualPtr and deletes the referenced object.

t_Position **DelLastEqualPtrAndObj** (const t_obj * po_obj);

This method works like DelLastEqualPtr and deletes the referenced object.

Delete Found Pointers and Referenced Objects Conditionally

t_Position **DelFirstEqualPtrAndObjCond** (const t_obj * po_obj);

This method works like DelFirstEqualPtrCond and deletes the referenced object.

t_Position **DelLastEqualPtrAndObjCond** (const t_obj * po_obj);

This method works like DelLastEqualPtrCond and deletes the referenced object.

2.5.3 Pointer Container Operations

Insert, Copy and Delete Objects

The following sample code demonstrates some simple pointer container operations. The class ct_Int is described in the section 'Sample Programs'.

```
ct_Int co_int = 1;
ct_Int * pco_int;
gct_AnyPtrContainer <ct_Int> co_ptrContainer;
gct_AnyPtrContainer <ct_Int>::t_Position o_pos;

// Add a new object by calling the default constructor
o_pos = co_ptrContainer. AddPtr (new ct_Int);

// Access the object and initialize it
pco_int = co_ptrContainer. GetPtr (o_pos);
(* pco_int) = 2;

// Copy an existing object into the pointer container
o_pos = co_ptrContainer. AddPtr (new ct_Int (co_int));

// Delete a single pointer and the referenced object
co_ptrContainer. DelPtrAndObj (o_pos);
```

Iterate Forward

The following sample code demonstrates a forward iteration over a pointer container.

```
gct_AnyPtrContainer <float> co_ptrContainer;
gct_AnyPtrContainer <float>::t_Position o_pos;

for (o_pos = co_ptrContainer. First ();
     o_pos != 0;
     o_pos = co_ptrContainer. Next (o_pos))
  {
  float * pf = co_ptrContainer. GetPtr (o_pos);
  // ...
  }
```

Iterate Backward

The following sample code demonstrates a backward iteration over a pointer container.

```
gct_AnyPtrContainer <float> co_ptrContainer;
gct_AnyPtrContainer <float>::t_Position o_pos;
```

```
for (o_pos = co_ptrContainer. Last ();
     o_pos != 0;
     o_pos = co_ptrContainer. Prev (o_pos))
  {
  float * pf = co_ptrContainer. GetPtr (o_pos);
  // ...
  }
```

Iterate and Modify

The following sample code demonstrates how to iterate and modify a pointer container.

```
gct_AnyPtrContainer <float> co_ptrContainer;
gct_AnyPtrContainer <float>::t_Position o_pos;

for (o_pos = co_ptrContainer. First ();
     o_pos != 0;
     o_pos = /* delete entry ? */ ?
             co_ptrContainer. DelPtrAndObj (o_pos) :
             co_ptrContainer. Next (o_pos))
  {
  float * pf = co_ptrContainer. GetPtr (o_pos);
  // ...
  }
```

Alternatively a `while` loop can be used.

```
gct_AnyPtrContainer <float> co_ptrContainer;
gct_AnyPtrContainer <float>::t_Position o_pos;

o_pos = co_ptrContainer. First ();

while (o_pos != 0)
  {
  float * pf = co_ptrContainer. GetPtr (o_pos);
  // ...
  if ( /* delete entry ? */ )
    o_pos = co_ptrContainer. DelPtrAndObj (o_pos);
  else
    o_pos = co_ptrContainer. Next (o_pos);
  }
```

2.5.4 Pointer-Comp-Containers (tuning/ptrcompcontainer.h)

If the object type of a pointer container provides the comparison function '`operator ==`', the pointer container can be extended by the pointer-comp-container interface. This interface is very similar to the comp-container interface (see above). The methods of a pointer-comp-container are based on the '`operator ==`' of referenced objects. To avoid confusions, method names contain the abbreviation `Ref` (e.g. `AddRefCond` instead of `AddObjCond` or `AddPtrCond`).

The template parameter `t_container` must at least contain the pointer container interface, e.g. `gct_Std32PtrArray <float>`. It is used as the base class of the pointer-comp-container.

Base Classes

```
gct_AnyContainer       (see above 'Container Interface')
  gct_ExtContainer     (see above 'Extended Container')
    gct_CompContainer  (see above 'Comp-Container')
      gct_PtrContainer (see above 'Pointer Container')
```

Template Declaration

```
template <class t_container>
  class gct_PtrCompContainer: public t_container
    {
  public:
    inline bool         ContainsRef (const t_RefObject * po_obj) const;
    t_Length            CountRefs (const t_RefObject * po_obj) const;

    t_Position          SearchFirstRef (const t_RefObject * po_obj) const;
    t_Position          SearchLastRef (const t_RefObject * po_obj) const;
    t_Position          SearchNextRef (t_Position o_pos, const t_RefObject * po_obj) const;
    t_Position          SearchPrevRef (t_Position o_pos, const t_RefObject * po_obj) const;

    inline t_RefObject * GetFirstEqualRef (const t_RefObject * po_obj) const;
    inline t_RefObject * GetLastEqualRef (const t_RefObject * po_obj) const;

    inline t_Position   AddRefCond (const t_RefObject * po_obj);
    inline t_Position   AddRefBeforeFirstCond (const t_RefObject * po_obj);
    inline t_Position   AddRefAfterLastCond (const t_RefObject * po_obj);

    inline t_Position   DelFirstEqualRef (const t_RefObject * po_obj);
    inline t_Position   DelLastEqualRef (const t_RefObject * po_obj);
    inline t_Position   DelFirstEqualRefCond (const t_RefObject * po_obj);
    inline t_Position   DelLastEqualRefCond (const t_RefObject * po_obj);

    inline t_Position   DelFirstEqualRefAndObj (const t_RefObject * po_obj);
    inline t_Position   DelLastEqualRefAndObj (const t_RefObject * po_obj);
    inline t_Position   DelFirstEqualRefAndObjCond (const t_RefObject * po_obj);
    inline t_Position   DelLastEqualRefAndObjCond (const t_RefObject * po_obj);
    };
```

Search for Referenced Objects

`bool` **`ContainsRef`** `(const t_RefObject * po_obj) const;`

Returns `true`, if a referenced object is equal to `* po_obj`.

`t_Length` **`CountRefs`** `(const t_RefObject * po_obj) const;`

Returns the number of referenced objects which are equal to `* po_obj`.

`t_Position` **`SearchFirstRef`** `(const t_RefObject * po_obj) const;`

Returns the position of the first referenced object which is equal to `* po_obj` or zero if no object was found.

`t_Position` **`SearchLastRef`** `(const t_RefObject * po_obj) const;`

Returns the position of the last referenced object which is equal to `* po_obj` or zero if no object was found.

`t_Position` **`SearchNextRef`** `(t_Position o_pos, const t_RefObject * po_obj) const;`

Returns the position of the next referenced object which is equal to `* po_obj` or zero if no object was found. `o_pos` must be a valid position value.

`t_Position` **`SearchPrevRef`** `(t_Position o_pos, const t_RefObject * po_obj) const;`

Returns the position of the previous referenced object which is equal to `* po_obj` or zero if no object was found. `o_pos` must be a valid position value.

Access to Found Objects

`t_RefObject * GetFirstEqualRef (const t_RefObject * po_obj) const;`

Returns a pointer to the first referenced object which is equal to `* po_obj`. There must be at least one equal object.

`t_RefObject * GetLastEqualRef (const t_RefObject * po_obj) const;`

Returns a pointer to the last referenced object which is equal to `* po_obj`. There must be at least one equal object.

Add Pointers Conditionally

`t_Position AddRefCond (const t_RefObject * po_obj);`

Returns the position of the first referenced object which is equal to `* po_obj` or the position of a new pointer (added by `AddPtr`) if no equal object was found.

`t_Position AddRefBeforeFirstCond (const t_RefObject * po_obj);`

Returns the position of the first referenced object which is equal to `* po_obj` or the position of a new pointer (added by `AddPtrBeforeFirst`) if no equal object was found.

`t_Position AddRefAfterLastCond (const t_RefObject * po_obj);`

Returns the position of the first referenced object which is equal to `* po_obj` or the position of a new pointer (added by `AddPtrAfterLast`) if no equal object was found.

Return Value of Delete Methods

Delete methods always return the position of the successor of the deleted entry. With this technique, a container can be iterated and modified at the same time. If the last object was deleted, the return value equals zero.

Delete Pointers of Found Objects

`t_Position DelFirstEqualRef (const t_RefObject * po_obj);`

Deletes the pointer of the first referenced object which is equal to `* po_obj`. There must be at least one equal object. The method returns the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

`t_Position DelLastEqualRef (const t_RefObject * po_obj);`

Deletes the pointer of the last referenced object which is equal to `* po_obj`. There must be at least one equal object. The method returns the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

Delete Pointers of Found Objects Conditionally

`t_Position DelFirstEqualRefCond (const t_RefObject * po_obj);`

Deletes the pointer of the first referenced object which is equal to `* po_obj` or returns zero if no equal object was found. If an equal object was found the method returns the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

`t_Position DelLastEqualRefCond (const t_RefObject * po_obj);`

Deletes the pointer of the last referenced object which is equal to `* po_obj` or returns zero if no equal object was found. If an equal object was found the method returns the position of the next pointer of the deleted pointer or zero, if the last pointer was deleted.

Delete Pointers and Objects of Found Objects

t_Position **DelFirstEqualRefAndObj** (const t_RefObject * po_obj);

This method works like DelFirstEqualRef and deletes the referenced object.

t_Position **DelLastEqualRefAndObj** (const t_RefObject * po_obj);

This method works like DelLastEqualRef and deletes the referenced object.

Delete Pointers and Objects of Found Objects Conditionally

t_Position **DelFirstEqualRefAndObjCond** (const t_RefObject * po_obj);

This method works like DelFirstEqualRefCond and deletes the referenced object.

t_Position **DelLastEqualRefAndObjCond** (const t_RefObject * po_obj);

This method works like DelLastEqualRefCond and deletes the referenced object.

2.5.5 Map Containers (tuning/map.h)

The map container interface is an extension of the basic container interface. A map container manages key-value pairs. The 'value' type requirements are very simple. A class type must contain a default and a copy constructor, no other requirements have to be fulfilled. Numeric and pointer types can also be used. The 'key' type must additionally provide the comparison function 'operator =='. So it's possible to search for a specific key.

A map container is based on a basic container which manages key-value pairs, e.g. gct_Std32Array <gct_MapEntry <ct_String, ct_Int> >. The basic container is used as the base class of the map container. Key-value type example: The type gct_MapEntry <ct_String, ct_Int> is based on the 'key' type ct_String and the 'value' type ct_Int. The 'key' type is used as the base class of the key-value type. Numeric data types, e.g. int or char, must be extended by the template gct_Key, e.g. gct_MapEntry <gct_Key <int>, ct_String>. If the base container of a map container is a sorted array, the 'key' type must provide the comparison function 'operator <'. If the base container is a hash table, the 'key' type must provide the method GetHash.

Base Classes

gct_AnyContainer (see above 'Container Interface')
[gct_ExtContainer (optional, see above 'Extended Container')]

Template Declaration

```
template <class t_container>
  class gct_Map: public t_container
    {
  public:
    typedef t_Object::t_Key         t_Key;
    typedef t_Object::t_Value       t_Value;

    inline bool          ContainsKey (t_Key o_key) const;
    t_Length             CountKeys (t_Key o_key) const;

    t_Position           SearchFirstKey (t_Key o_key) const;
    t_Position           SearchLastKey (t_Key o_key) const;
    t_Position           SearchNextKey (t_Position o_pos, t_Key o_key) const;
    t_Position           SearchPrevKey (t_Position o_pos, t_Key o_key) const;

    inline t_Key         GetKey (t_Position o_pos) const;
    inline t_Value *     GetValue (t_Position o_pos) const;
```

```
inline t_Value *     GetFirstValue (t_Key o_key) const;
inline t_Value *     GetLastValue (t_Key o_key) const;

t_Position           AddKeyAndValue (t_Key o_key, const t_Value * po_value = 0);
t_Position           AddKeyAndValueCond (t_Key o_key, const t_Value * po_value = 0);

inline t_Position    DelKeyAndValue (t_Position o_pos);
inline t_Position    DelFirstKeyAndValue (t_Key o_key);
inline t_Position    DelLastKeyAndValue (t_Key o_key);
inline t_Position    DelFirstKeyAndValueCond (t_Key o_key);
inline t_Position    DelLastKeyAndValueCond (t_Key o_key);
inline void          DelAllKeyAndValue ();
};
```

Data Types

typedef t_Object::t_Key **t_Key**;

The nested type t_Key describes the 'key' type of key-value pairs.

typedef t_Object::t_Value **t_Value**;

The nested type t_Value describes the 'value' type of key-value pairs.

Search for Pairs

bool **ContainsKey** (t_Key o_key) const;

Returns true, if a contained key is equal to o_key.

t_Length **CountKeys** (t_Key o_key) const;

Returns the number of keys which are equal to o_key.

t_Position **SearchFirstKey** (t_Key o_key) const;

Returns the position of the first key-value pair whose key is equal to o_key or zero if no key was found.

t_Position **SearchLastKey** (t_Key o_key) const;

Returns the position of the last key-value pair whose key is equal to o_key or zero if no key was found.

t_Position **SearchNextKey** (t_Position o_pos, t_Key o_key) const;

Returns the position of the next key-value pair whose key is equal to o_key or zero if no key was found. o_pos must be a valid position value.

t_Position **SearchPrevKey** (t_Position o_pos, t_Key o_key) const;

Returns the position of the previous key-value pair whose key is equal to o_key or zero if no key was found. o_pos must be a valid position value.

Access to Key and Value

t_Key **GetKey** (t_Position o_pos) const;

Returns the key of the key-value pair at position o_pos. o_pos must be a valid position value.

t_Value * **GetValue** (t_Position o_pos) const;

Returns a pointer to the value of the key-value pair at position o_pos. o_pos must be a valid position value.

Access to Found Values

t_Value * **GetFirstValue** (t_Key o_key) const;

Returns a pointer to the value of the first key-value pair whose key is equal to o_key. There must be at least one equal key.

t_Value * **GetLastValue** (t_Key o_key) const;

Returns a pointer to the value of the last key-value pair whose key is equal to o_key. There must be at least one equal key.

Add Key-Value Pairs (Conditionally)

t_Position **AddKeyAndValue** (t_Key o_key, const t_Value * po_value = 0);

Adds a key-value pair and returns the position of the new pair. The logical position of the new pair depends on the container implementation. If po_value equals zero, the new value is created by the default constructor, otherwise the copy constructor is used.

t_Position **AddKeyAndValueCond** (t_Key o_key, const t_Value * po_value = 0);

Returns the position of the first key-value pair whose key is equal to o_key or the position of a new pair if no equal key was found. The logical position of the new pair depends on the container implementation. If po_value equals zero, the new value is created by the default constructor, otherwise the copy constructor is used.

Return Value of Delete Methods

Delete methods always return the position of the successor of the deleted entry. With this technique, a container can be iterated and modified at the same time. If the last object was deleted, the return value equals zero.

Delete Pairs

t_Position **DelKeyAndValue** (t_Position o_pos);

Deletes the key-value pair at position o_pos. Calls the destructor of the key-value pair and releases the corresponding memory. o_pos must be a valid position value. The method returns Next (o_pos), i.e. the position of the next pair or zero, if the last pair was deleted.

void **DelAllKeyAndValue** ();

Deletes all contained key-value pairs. Calls the destructor of the pairs and releases the corresponding memory.

Delete Found Pairs

t_Position **DelFirstKeyAndValue** (t_Key o_key);

Deletes the first key-value pair whose key is equal to o_key. There must be at least one equal key. The method returns the position of the next pair of the deleted pair or zero, if the last pair was deleted.

t_Position **DelLastKeyAndValue** (t_Key o_key);

Deletes the last key-value pair whose key is equal to o_key. There must be at least one equal key. The method returns the position of the next pair of the deleted pair or zero, if the last pair was deleted.

Delete Found Pairs Conditionally

t_Position **DelFirstKeyAndValueCond** (t_Key o_key);

Deletes the first key-value pair whose key is equal to o_key or returns zero if no equal key was found. If an equal key was found the method returns the position of the next pair of the deleted pair or zero, if the last pair was deleted.

t_Position **DelLastKeyAndValueCond** (t_Key o_key);

Deletes the last key-value pair whose key is equal to o_key or returns zero if no equal key was found. If an equal key was found the method returns the position of the next pair of the deleted pair or zero, if the last pair was deleted.

2.5.6 Pointer Map Containers (tuning/ptrmap.h)

A map container can manage 'value' objects of many different types (e.g. ct_String, int, float). If the 'value' type is a pointer type, some map container methods are very unhandily. The method GetValue returns a pointer to a pointer, AddKeyAndValue requires a parameter of type pointer to pointer etc.

The class template gct_PtrMap provides a comfortable interface for pointer map containers. A pointer map manages key-pointer pairs. The pointers refer to 'value' objects. The 'key' type requirements are very simple. A class type must contain a default and a copy constructor. Numeric and pointer types can also be used. The 'key' type must additionally provide the comparison function 'operator =='. So it’s possible to search for a specific key.

Note that a pointer map container can be the owner of the referenced value objects or it can manage pointers to value objects which have a different owner. The method DelKeyAndValue deletes a key-pointer pair and the referenced value object. The method DelKey simply deletes the key-pointer pair, the referenced value object remains unchanged.

The first template parameter t_container must be a container type which manages key-pointer pairs, e.g. gct_Std32Array <gct_PtrMapEntry <ct_String> >. The second template parameter t_value is the type of the value objects. The basic container is used as the base class of the pointer map container.

Key-pointer type example: The type gct_PtrMapEntry <ct_String> is based on the 'key' type ct_String. The pointer part of the key-pointer pair is of type void *. With this technique, many pointer map container instances can share the same binary code. The 'key' type is used as the base class of the key-pointer type. Numeric data types, e.g. int or char, must be extended by the template gct_Key, e.g. gct_PtrMapEntry <gct_Key <int> >. If the base container of a pointer map container is a sorted array, the 'key' type must provide the comparison function 'operator <'. If the base container is a hash table, the 'key' type must provide the method GetHash.

Base Classes

gct_AnyContainer (see above 'Container Interface')
[gct_ExtContainer (optional, see above 'Extended Container')]

Template Declaration

```
template <class t_container, class t_value>
  class gct_PtrMap: public t_container
    {
  public:
    typedef t_Object::t_Key        t_Key;
    typedef t_value                t_Value;

    inline bool          ContainsKey (t_Key o_key) const;
    t_Length             CountKeys (t_Key o_key) const;
```

```
t_Position          SearchFirstKey (t_Key o_key) const;
t_Position          SearchLastKey (t_Key o_key) const;
t_Position          SearchNextKey (t_Position o_pos,
                      t_Key o_key) const;
t_Position          SearchPrevKey (t_Position o_pos,
                      t_Key o_key) const;

inline t_Key        GetKey (t_Position o_pos) const;
inline t_Value *    GetValPtr (t_Position o_pos) const;
inline t_Value *    GetFirstValPtr (t_Key o_key) const;
inline t_Value *    GetLastValPtr (t_Key o_key) const;

t_Position          AddKeyAndValPtr (t_Key o_key,
                      const t_Value * po_value);
t_Position          AddKeyAndValPtrCond (t_Key o_key,
                      const t_Value * po_value);

inline t_Position   DelKey (t_Position o_pos);
inline t_Position   DelFirstKey (t_Key o_key);
inline t_Position   DelLastKey (t_Key o_key);
inline t_Position   DelFirstKeyCond (t_Key o_key);
inline t_Position   DelLastKeyCond (t_Key o_key);
inline void         DelAllKey ();

inline t_Position   DelKeyAndValue (t_Position o_pos);
inline t_Position   DelFirstKeyAndValue (t_Key o_key);
inline t_Position   DelLastKeyAndValue (t_Key o_key);
inline t_Position   DelFirstKeyAndValueCond (t_Key o_key);
inline t_Position   DelLastKeyAndValueCond (t_Key o_key);
void                DelAllKeyAndValue ();
};
```

Data Types

typedef t_Object::t_Key **t_Key**;

The nested type t_Key describes the 'key' type of key-pointer pairs.

typedef t_value **t_Value**;

The nested type t_Value describes the type of referenced objects of key-pointer pairs.

Search for Pairs

bool **ContainsKey** (t_Key o_key) const;

Returns true, if a contained key is equal to o_key.

t_Length **CountKeys** (t_Key o_key) const;

Returns the number of keys which are equal to o_key.

t_Position **SearchFirstKey** (t_Key o_key) const;

Returns the position of the first key-pointer pair whose key is equal to o_key or zero if no key was found.

t_Position **SearchLastKey** (t_Key o_key) const;

Returns the position of the last key-pointer pair whose key is equal to o_key or zero if no key was found.

t_Position **SearchNextKey** (t_Position o_pos, t_Key o_key) const;

Returns the position of the next key-pointer pair whose key is equal to o_key or zero if no key was found. o_pos must be a valid position value.

t_Position **SearchPrevKey** (t_Position o_pos, t_Key o_key) const;

Returns the position of the previous key-pointer pair whose key is equal to o_key or zero if no key was found. o_pos must be a valid position value.

Access to Key and Value

t_Key **GetKey** (t_Position o_pos) const;

Returns the key of the key-pointer pair at position o_pos. o_pos must be a valid position value.

t_Value * **GetValPtr** (t_Position o_pos) const;

Returns a pointer to the referenced value object of the key-pointer pair at position o_pos. o_pos must be a valid position value.

Access to Found Values

t_Value * **GetFirstValPtr** (t_Key o_key) const;

Returns a pointer to the referenced value object of the first key-pointer pair whose key is equal to o_key. There must be at least one equal key.

t_Value * **GetLastValPtr** (t_Key o_key) const;

Returns a pointer to the referenced value object of the last key-pointer pair whose key is equal to o_key. There must be at least one equal key.

Add Key-Pointer Pairs (Conditionally)

t_Position **AddKeyAndValPtr** (t_Key o_key, const t_Value * po_value);

Adds a key-pointer pair and returns the position of the new pair. The logical position of the new pair depends on the container implementation.

t_Position **AddKeyAndValPtrCond** (t_Key o_key, const t_Value * po_value);

Returns the position of the first key-pointer pair whose key is equal to o_key or the position of a new pair if no equal key was found. The logical position of the new pair depends on the container implementation.

Return Value of Delete Methods

Delete methods always return the position of the successor of the deleted entry. With this technique, a container can be iterated and modified at the same time. If the last object was deleted, the return value equals zero.

Delete Pairs

t_Position **DelKey** (t_Position o_pos);

Deletes the key-pointer pair at position o_pos. Calls the destructor of the key-pointer pair and releases the corresponding memory. The referenced value object remains unchanged. o_pos must be a valid position value. The method returns Next (o_pos), i.e. the position of the next pair or zero, if the last pair was deleted.

void **DelAllKey** ();

Deletes all contained key-pointer pairs. Calls the destructor of the pairs and releases the corresponding memory. The referenced value objects remain unchanged.

Delete Found Pairs

`t_Position` **`DelFirstKey`** `(t_Key o_key);`

Deletes the first key-pointer pair whose key is equal to `o_key`. The referenced value object remains unchanged. There must be at least one equal key. The method returns the position of the next pair of the deleted pair or zero, if the last pair was deleted.

`t_Position` **`DelLastKey`** `(t_Key o_key);`

Deletes the last key-pointer pair whose key is equal to `o_key`. The referenced value object remains unchanged. There must be at least one equal key. The method returns the position of the next pair of the deleted pair or zero, if the last pair was deleted.

Delete Found Pairs Conditionally

`t_Position` **`DelFirstKeyCond`** `(t_Key o_key);`

Deletes the first key-pointer pair whose key is equal to `o_key` or returns zero if no equal key was found. If an equal key was found the method returns the position of the next pair of the deleted pair or zero, if the last pair was deleted. The referenced value object remains unchanged.

`t_Position` **`DelLastKeyCond`** `(t_Key o_key);`

Deletes the last key-pointer pair whose key is equal to `o_key` or returns zero if no equal key was found. If an equal key was found the method returns the position of the next pair of the deleted pair or zero, if the last pair was deleted. The referenced value object remains unchanged.

Delete Pairs and Referenced Objects

`t_Position` **`DelKeyAndValue`** `(t_Position o_pos);`

This method works like `DelKey` and deletes the referenced value object.

`void` **`DelAllKeyAndValue`** `();`

This method works like `DelAllKey` and deletes the referenced value objects.

Delete Found Pairs and Referenced Objects

`t_Position` **`DelFirstKeyAndValue`** `(t_Key o_key);`

This method works like `DelFirstKey` and deletes the referenced value object.

`t_Position` **`DelLastKeyAndValue`** `(t_Key o_key);`

This method works like `DelLastKey` and deletes the referenced value object.

Delete Found Pairs and Referenced Objects Conditionally

`t_Position` **`DelFirstKeyAndValueCond`** `(t_Key o_key);`

This method works like `DelFirstKeyCond` and deletes the referenced value object.

`t_Position` **`DelLastKeyAndValueCond`** `(t_Key o_key);`

This method works like `DelLastKeyCond` and deletes the referenced value object.

2.6 Pointer Container Instances

2.6.1 Pointer Array Instances (tuning/xxx/ptrarray.h)

Some template instances are predefined to easily use pointer array containers. The macro `PTR_ARRAY_DCLS(Obj)` generates for each wrapper class of a global store one pointer array template.

The macro

```
PTR_ARRAY_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_PtrArray:
  public gct_PtrContainer <t_obj, gct_Any_Array <void *> > { };
template <class t_obj> class gct_Any8PtrArray:
  public gct_PtrContainer <t_obj, gct_Any8Array <void *> > { };
template <class t_obj> class gct_Any16PtrArray:
  public gct_PtrContainer <t_obj, gct_Any16Array <void *> > { };
template <class t_obj> class gct_Any32PtrArray:
  public gct_PtrContainer <t_obj, gct_Any32Array <void *> > { };
```

Every directory of a global store contains a file **'ptrarray.h'**.

The file 'tuning/std/ptrarray.h' contains the following declarations:

```
template <class t_obj> class gct_Std_PtrArray;
template <class t_obj> class gct_Std8PtrArray;
template <class t_obj> class gct_Std16PtrArray;
template <class t_obj> class gct_Std32PtrArray;
```

The file 'tuning/rnd/ptrarray.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_PtrArray;
template <class t_obj> class gct_Rnd8PtrArray;
template <class t_obj> class gct_Rnd16PtrArray;
template <class t_obj> class gct_Rnd32PtrArray;
```

The file 'tuning/chn/ptrarray.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_PtrArray;
template <class t_obj> class gct_Chn8PtrArray;
template <class t_obj> class gct_Chn16PtrArray;
template <class t_obj> class gct_Chn32PtrArray;
```

2.6.2 Pointer List Instances (tuning/xxx/ptrdlist.h)

Some template instances are predefined to easily use pointer list containers. The macro `PTR_DLIST_DCLS(Obj)` generates for each wrapper class of a global store one pointer list template.

The macro

```
PTR_DLIST_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_PtrDList:
  public gct_PtrContainer <t_obj, gct_Any_DList <void *> > { };
```

```
template <class t_obj> class gct_Any8PtrDList:
  public gct_PtrContainer <t_obj, gct_Any8DList <void *> > { };
template <class t_obj> class gct_Any16PtrDList:
  public gct_PtrContainer <t_obj, gct_Any16DList <void *> > { };
template <class t_obj> class gct_Any32PtrDList:
  public gct_PtrContainer <t_obj, gct_Any32DList <void *> > { };
```

Every directory of a global store contains a file **'ptrdlist.h'.**

The file 'tuning/std/ptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Std_PtrDList;
template <class t_obj> class gct_Std8PtrDList;
template <class t_obj> class gct_Std16PtrDList;
template <class t_obj> class gct_Std32PtrDList;
```

The file 'tuning/rnd/ptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_PtrDList;
template <class t_obj> class gct_Rnd8PtrDList;
template <class t_obj> class gct_Rnd16PtrDList;
template <class t_obj> class gct_Rnd32PtrDList;
```

The file 'tuning/chn/ptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_PtrDList;
template <class t_obj> class gct_Chn8PtrDList;
template <class t_obj> class gct_Chn16PtrDList;
template <class t_obj> class gct_Chn32PtrDList;
```

2.6.3 Pointer Sorted Array Instances (tuning/xxx/ptrsortedarray.h)

Some template instances are predefined to easily use pointer sorted array containers. The macro `PTR_SORTEDARRAY_DCLS(Obj)` generates for each wrapper class of a global store one pointer sorted array template.

The macro

```
PTR_SORTEDARRAY_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_PtrSortedArray:
  public gct_PtrContainer <t_obj, gct_Any_SortedArray <gct_SortedArrayRef <t_obj> > > { };
template <class t_obj> class gct_Any8PtrSortedArray:
  public gct_PtrContainer <t_obj, gct_Any8SortedArray <gct_SortedArrayRef <t_obj> > > { };
template <class t_obj> class gct_Any16PtrSortedArray:
  public gct_PtrContainer <t_obj, gct_Any16SortedArray <gct_SortedArrayRef <t_obj> > > { };
template <class t_obj> class gct_Any32PtrSortedArray:
  public gct_PtrContainer <t_obj, gct_Any32SortedArray <gct_SortedArrayRef <t_obj> > > { };
```

Every directory of a global store contains a file **'ptrsortedarray.h'.**

The file 'tuning/std/ptrsortedarray.h' contains the following declarations:

```
template <class t_obj> class gct_Std_PtrSortedArray;
template <class t_obj> class gct_Std8PtrSortedArray;
template <class t_obj> class gct_Std16PtrSortedArray;
template <class t_obj> class gct_Std32PtrSortedArray;
```

The file 'tuning/rnd/ptrsortedarray.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_PtrSortedArray;
template <class t_obj> class gct_Rnd8PtrSortedArray;
template <class t_obj> class gct_Rnd16PtrSortedArray;
template <class t_obj> class gct_Rnd32PtrSortedArray;
```

The file 'tuning/chn/ptrsortedarray.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_PtrSortedArray;
template <class t_obj> class gct_Chn8PtrSortedArray;
template <class t_obj> class gct_Chn16PtrSortedArray;
template <class t_obj> class gct_Chn32PtrSortedArray;
```

2.6.4 Pointer Hash Table Instances (tuning/xxx/ptrhashtable.h)

Some template instances are predefined to easily use pointer hash table containers. The macro `PTR_HASHTABLE_DCLS(Obj)` generates for each wrapper class of a global store one pointer hash table template.

The macro

```
PTR_HASHTABLE_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_PtrHashTable:
  public gct_PtrContainer <t_obj, gct_Any_HashTable <gct_HashTableRef <t_obj> > > { };
template <class t_obj> class gct_Any8PtrHashTable:
  public gct_PtrContainer <t_obj, gct_Any8HashTable <gct_HashTableRef <t_obj> > > { };
template <class t_obj> class gct_Any16PtrHashTable:
  public gct_PtrContainer <t_obj, gct_Any16HashTable <gct_HashTableRef <t_obj> > > { };
template <class t_obj> class gct_Any32PtrHashTable:
  public gct_PtrContainer <t_obj, gct_Any32HashTable <gct_HashTableRef <t_obj> > > { };
```

Every directory of a global store contains a file **'ptrhashtable.h'**.

The file 'tuning/std/ptrhashtable.h' contains the following declarations:

```
template <class t_obj> class gct_Std_PtrHashTable;
template <class t_obj> class gct_Std8PtrHashTable;
template <class t_obj> class gct_Std16PtrHashTable;
template <class t_obj> class gct_Std32PtrHashTable;
```

The file 'tuning/rnd/ptrhashtable.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_PtrHashTable;
template <class t_obj> class gct_Rnd8PtrHashTable;
template <class t_obj> class gct_Rnd16PtrHashTable;
template <class t_obj> class gct_Rnd32PtrHashTable;
```

The file 'tuning/chn/ptrhashtable.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_PtrHashTable;
template <class t_obj> class gct_Chn8PtrHashTable;
template <class t_obj> class gct_Chn16PtrHashTable;
template <class t_obj> class gct_Chn32PtrHashTable;
```

2.6.5 Block Pointer List Instances (tuning/xxx/blockptrdlist.h)

Some template instances are predefined to easily use block pointer list containers. The macro BLOCKPTR_DLIST_DCLS(Obj) generates for each wrapper class of a global store one block pointer list template.

The macro

```
BLOCKPTR_DLIST_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_BlockPtrDList:
  public gct_PtrContainer <t_obj, gct_Any_BlockDList <void *> > { };
template <class t_obj> class gct_Any8BlockPtrDList:
  public gct_PtrContainer <t_obj, gct_Any8BlockDList <void *> > { };
template <class t_obj> class gct_Any16BlockPtrDList:
  public gct_PtrContainer <t_obj, gct_Any16BlockDList <void *> > { };
template <class t_obj> class gct_Any32BlockPtrDList:
  public gct_PtrContainer <t_obj, gct_Any32BlockDList <void *> > { };
```

Every directory of a global store contains a file **'blockptrdlist.h'**.

The file 'tuning/std/blockptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Std_BlockPtrDList;
template <class t_obj> class gct_Std8BlockPtrDList;
template <class t_obj> class gct_Std16BlockPtrDList;
template <class t_obj> class gct_Std32BlockPtrDList;
```

The file 'tuning/rnd/blockptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_BlockPtrDList;
template <class t_obj> class gct_Rnd8BlockPtrDList;
template <class t_obj> class gct_Rnd16BlockPtrDList;
template <class t_obj> class gct_Rnd32BlockPtrDList;
```

The file 'tuning/chn/blockptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_BlockPtrDList;
template <class t_obj> class gct_Chn8BlockPtrDList;
template <class t_obj> class gct_Chn16BlockPtrDList;
template <class t_obj> class gct_Chn32BlockPtrDList;
```

2.6.6 Ref Pointer List Instances (tuning/xxx/refptrdlist.h)

Some template instances are predefined to easily use ref pointer list containers. The macro REFPTR_DLIST_DCLS(Obj) generates for each wrapper class of a global store one ref pointer list template.

The macro

```
REFPTR_DLIST_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_RefPtrDList:
  public gct_PtrContainer <t_obj, gct_Any_RefDList <void *> > { };
template <class t_obj> class gct_Any8RefPtrDList:
  public gct_PtrContainer <t_obj, gct_Any8RefDList <void *> > { };
template <class t_obj> class gct_Any16RefPtrDList:
```

```
  public gct_PtrContainer <t_obj, gct_Any16RefDList <void *> > { };
template <class t_obj> class gct_Any32RefPtrDList:
  public gct_PtrContainer <t_obj, gct_Any32RefDList <void *> > { };
```

Every directory of a global store contains a file **'refptrdlist.h'.**

The file 'tuning/std/refptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Std_RefPtrDList;
template <class t_obj> class gct_Std8RefPtrDList;
template <class t_obj> class gct_Std16RefPtrDList;
template <class t_obj> class gct_Std32RefPtrDList;
```

The file 'tuning/rnd/refptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_RefPtrDList;
template <class t_obj> class gct_Rnd8RefPtrDList;
template <class t_obj> class gct_Rnd16RefPtrDList;
template <class t_obj> class gct_Rnd32RefPtrDList;
```

The file 'tuning/chn/refptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_RefPtrDList;
template <class t_obj> class gct_Chn8RefPtrDList;
template <class t_obj> class gct_Chn16RefPtrDList;
template <class t_obj> class gct_Chn32RefPtrDList;
```

2.6.7 Block-Ref Pointer List Instances (tuning/xxx/blockrefptrdlist.h)

Some template instances are predefined to easily use block-ref pointer list containers. The macro `BLOCKREFPTR_DLIST_DCLS(Obj)` generates for each wrapper class of a global store one block-ref pointer list template.

The macro

```
BLOCKREFPTR_DLIST_DCLS (Any)
```

expands to:

```
template <class t_obj> class gct_Any_BlockRefPtrDList: public
  gct_PtrContainer <t_obj, gct_Any_BlockRefDList <void *> > { };
template <class t_obj> class gct_Any8BlockRefPtrDList: public
  gct_PtrContainer <t_obj, gct_Any8BlockRefDList <void *> > { };
template <class t_obj> class gct_Any16BlockRefPtrDList: public
  gct_PtrContainer <t_obj, gct_Any16BlockRefDList <void *> > { };
template <class t_obj> class gct_Any32BlockRefPtrDList: public
  gct_PtrContainer <t_obj, gct_Any32BlockRefDList <void *> > { };
```

Every directory of a global store contains a file **'blockrefptrdlist.h'.**

The file 'tuning/std/blockrefptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Std_BlockRefPtrDList;
template <class t_obj> class gct_Std8BlockRefPtrDList;
template <class t_obj> class gct_Std16BlockRefPtrDList;
template <class t_obj> class gct_Std32BlockRefPtrDList;
```

The file 'tuning/rnd/blockrefptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Rnd_BlockRefPtrDList;
template <class t_obj> class gct_Rnd8BlockRefPtrDList;
template <class t_obj> class gct_Rnd16BlockRefPtrDList;
template <class t_obj> class gct_Rnd32BlockRefPtrDList;
```

The file 'tuning/chn/blockrefptrdlist.h' contains the following declarations:

```
template <class t_obj> class gct_Chn_BlockRefPtrDList;
template <class t_obj> class gct_Chn8BlockRefPtrDList;
template <class t_obj> class gct_Chn16BlockRefPtrDList;
template <class t_obj> class gct_Chn32BlockRefPtrDList;
```

2.7 Overview of Container Instances

2.7.1 Predefined Template Instances

This section describes the naming convention of predefined template instances. A predefined template name consists of 7 parts.

1. Prefix

A predefined container name begins with the prefix `gct_`.

2. Global Store

Predefined containers allocate memory from one of the global store objects: `Std`, `Rnd` or `Chn`.

3. Length Type

The nested type `t_Length` describes the number of contained objects, examples are `t_UInt`, `t_UInt8`, `t_UInt16` and `t_UInt32`. The corresponding abbreviations are `_`, `8`, `16` and `32`.

4. Optional Block

If a block store is used to implement a list container, the name will contain the abbreviation `Block`.

5. Optional Ref

If a ref-store is used to implement a list container, the name will contain the abbreviation `Ref`.

6. Optional Ptr

If the container is a pointer container, the name will contain the abbreviation `Ptr`.

7. Container Type

A predefined container name ends with the abbreviation for the container type: `Array`, `DList`, `SortedArray` or `HashTable`.

The following table summarizes the naming convention.

Prefix	Glob. Store	t_Length	Opt. Block	Opt. Ref	Opt. Ptr	Cont. Type
gct_	Std	_	Block	Ref	Ptr	Array
	Rnd	8	-	-	-	DList
	Chn	16				SortedArray
		32				HashTable

2.7.2 User Defined Container Templates

In addition to the predefined containers, various other container templates can be defined. Predefined containers are based on the block template `gct_Block`. The alternative block implementations `gct_FixBlock`, `gct_MiniBlock` and `gct_ResBlock` can also be used. It is recommended to use the same naming convention as the predefined containers. The following sample code demonstrates how to use the block template `gct_MiniBlock` to implement some container templates.

```
typedef gct_EmptyBaseMiniBlock <ct_Chn_Store>  ct_Chn_MiniBlock;
typedef gct_EmptyBaseMiniBlock <ct_Chn32Store> ct_Chn32MiniBlock;
typedef gct_BlockStore <ct_PageBlock, gct_CharBlock <ct_Chn_MiniBlock, char> > ct_Chn_PageBlockStore;

template <class t_obj>
  class gct_Chn_MiniArray: public gct_ExtContainer
    <gct_FixItemArray <t_obj, ct_Chn_MiniBlock> > { };

template <class t_obj>
  class gct_Chn_MiniSortedArray: public gct_ExtContainer
    <gct_FixItemSortedArray <t_obj, ct_Chn_MiniBlock> > { };

template <class t_obj>
  class gct_Chn_MiniPtrArray:
    public gct_PtrContainer <t_obj, gct_Chn_MiniArray <void *> > { };

template <class t_obj>
  class gct_Chn32MiniHashTable:
    public gct_ExtContainer <gct_HashTable <t_obj, ct_Chn32MiniBlock> > { };

template <class t_obj>
  class gct_Chn32MiniPtrHashTable:
    public gct_PtrContainer <t_obj, gct_Chn32MiniHashTable
      <gct_HashTableRef <t_obj> > > { };
```

2.8 Collections

2.8.1 Abstract Object (tuning/object.hpp)

Containers and collections are two different concepts to manage sets of C++ objects. A container manages a uniform set of objects. It also contains the objects itself, i.e. the underlying memory. A collection can manage a polymorphic set of objects which are derived from a common base class. All objects to be used by the **Spirick** collection classes must inherit from the abstract base class `ct_Object`.

Class Declaration

```
class ct_Object
```

```
  {
public:
  virtual             ~ct_Object ();
  virtual bool        operator < (const ct_Object & co_comp) const;
  virtual t_UInt      GetHash () const;
  };
```

Methods

~ct_Object ();

The virtual destructor ensures type-safe destruction of derived classes.

bool **operator <** (const ct_Object & co_comp) const;

The comparison function 'operator <' is used by the collection class ct_SortedArray.

t_UInt **GetHash** () const;

The method GetHash is used by hash table containers.

2.8.2 Abstract Collection (tuning/collection.hpp)

The collection interface is identical to the pointer container interface (see above 'Pointer Containers').
The following differences exist between pointer containers and collections:
- Pointer containers are templates, collections are classes.
- Pointer containers can manage pointers of arbitrary type, collections manage pointers to ct_Object.

All collections are derived from the abstract base class ct_Collection, all methods are virtual. A specific collection class is implemented by using the methods of a specific pointer container.

Base Class

ct_Object (see above 'Abstract Object')

Class Declaration

```
class ct_Collection: public ct_Object
  {
public:
  typedef t_UInt      t_Length;
  typedef t_UInt      t_Position;

  virtual bool        IsEmpty () const = 0;
  virtual t_Length    GetLen () const = 0;

  virtual t_Position  First () const = 0;
  virtual t_Position  Last () const = 0;
  virtual t_Position  Next (t_Position o_pos) const = 0;
  virtual t_Position  Prev (t_Position o_pos) const = 0;
  virtual t_Position  Nth (t_Length u_idx) const = 0;

  virtual ct_Object * GetPtr (t_Position o_pos) const = 0;
  virtual ct_Object * GetFirstPtr () const = 0;
  virtual ct_Object * GetLastPtr () const = 0;
  virtual ct_Object * GetNextPtr (t_Position o_pos) const = 0;
  virtual ct_Object * GetPrevPtr (t_Position o_pos) const = 0;
  virtual ct_Object * GetNthPtr (t_Length u_idx) const = 0;

  virtual t_Position  AddPtr (const ct_Object * po_obj) = 0;
  virtual t_Position  AddPtrBefore (t_Position o_pos, const ct_Object * po_obj) = 0;
  virtual t_Position  AddPtrAfter (t_Position o_pos, const ct_Object * po_obj) = 0;
```

```
virtual t_Position   AddPtrBeforeFirst (const ct_Object * po_obj) = 0;
virtual t_Position   AddPtrAfterLast (const ct_Object * po_obj) = 0;
virtual t_Position   AddPtrBeforeNth (t_Length u_idx, const ct_Object * po_obj) = 0;
virtual t_Position   AddPtrAfterNth (t_Length u_idx, const ct_Object * po_obj) = 0;

virtual t_Position   DelPtr (t_Position o_pos) = 0;
virtual t_Position   DelFirstPtr () = 0;
virtual t_Position   DelLastPtr () = 0;
virtual t_Position   DelNextPtr (t_Position o_pos) = 0;
virtual t_Position   DelPrevPtr (t_Position o_pos) = 0;
virtual t_Position   DelNthPtr (t_Length u_idx) = 0;
virtual void         DelAllPtr () = 0;

virtual t_Position   DelPtrAndObj (t_Position o_pos) = 0;
virtual t_Position   DelFirstPtrAndObj () = 0;
virtual t_Position   DelLastPtrAndObj () = 0;
virtual t_Position   DelNextPtrAndObj (t_Position o_pos) = 0;
virtual t_Position   DelPrevPtrAndObj (t_Position o_pos) = 0;
virtual t_Position   DelNthPtrAndObj (t_Length u_idx) = 0;
virtual void         DelAllPtrAndObj () = 0;

virtual bool         ContainsPtr (const ct_Object * po_obj) const = 0;
virtual t_Length     CountPtrs (const ct_Object * po_obj) const = 0;

virtual t_Position   SearchFirstPtr (const ct_Object * po_obj) const = 0;
virtual t_Position   SearchLastPtr (const ct_Object * po_obj) const = 0;
virtual t_Position   SearchNextPtr (t_Position o_pos, const ct_Object * po_obj) const = 0;
virtual t_Position   SearchPrevPtr (t_Position o_pos, const ct_Object * po_obj) const = 0;

virtual t_Position   AddPtrCond (const ct_Object * po_obj) = 0;
virtual t_Position   AddPtrBeforeFirstCond (const ct_Object * po_obj) = 0;
virtual t_Position   AddPtrAfterLastCond (const ct_Object * po_obj) = 0;

virtual t_Position   DelFirstEqualPtr (const ct_Object * po_obj) = 0;
virtual t_Position   DelLastEqualPtr (const ct_Object * po_obj) = 0;
virtual t_Position   DelFirstEqualPtrCond (const ct_Object * po_obj) = 0;
virtual t_Position   DelLastEqualPtrCond (const ct_Object * po_obj) = 0;

virtual t_Position   DelFirstEqualPtrAndObj (const ct_Object * po_obj) = 0;
virtual t_Position   DelLastEqualPtrAndObj (const ct_Object * po_obj) = 0;
virtual t_Position   DelFirstEqualPtrAndObjCond (const ct_Object * po_obj) = 0;
virtual t_Position   DelLastEqualPtrAndObjCond (const ct_Object * po_obj) = 0;
};
```

2.8.3 Collection Operations

Insert, Copy and Delete Objects

The following sample code demonstrates some simple collection operations. The class `ct_Int` is described in the section 'Sample Programs'.

```
ct_Int co_int = 1;
ct_Int * pco_int;
ct_AnyCollection co_collection;
ct_AnyCollection::t_Position o_pos;

// Add a new object by calling the default constructor
o_pos = co_collection. AddPtr (new ct_Int);

// Access the object and initialize it
pco_int = dynamic_cast <ct_Int *> (co_collection. GetPtr (o_pos));
(* pco_int) = 2;
```

```
// Copy an existing object into the collection
o_pos = co_collection. AddPtr (new ct_Int (co_int));

// Delete a single pointer and the referenced object
co_collection. DelPtrAndObj (o_pos);
```

Iterate Forward

The following sample code demonstrates a forward iteration over a collection.

```
ct_AnyCollection co_collection;
ct_AnyCollection::t_Position o_pos;

for (o_pos = co_collection. First ();
     o_pos != 0;
     o_pos = co_collection. Next (o_pos))
  {
  ct_Object * pco_object = co_collection. GetPtr (o_pos);
  // ...
  }
```

Iterate Backward

The following sample code demonstrates a backward iteration over a collection.

```
ct_AnyCollection co_collection;
ct_AnyCollection::t_Position o_pos;

for (o_pos = co_collection. Last ();
     o_pos != 0;
     o_pos = co_collection. Prev (o_pos))
  {
  ct_Object * pco_object = co_collection. GetPtr (o_pos);
  // ...
  }
```

Iterate and Modify

The following sample code demonstrates how to iterate and modify a collection.

```
ct_AnyCollection co_collection;
ct_AnyCollection::t_Position o_pos;

for (o_pos = co_collection. First ();
     o_pos != 0;
     o_pos = /* delete entry ? */ ?
             co_collection. DelPtrAndObj (o_pos) :
             co_collection. Next (o_pos))
  {
  ct_Object * pco_object = co_collection. GetPtr (o_pos);
  // ...
  }
```

Alternatively a `while` loop can be used.

```
ct_AnyCollection co_collection;
ct_AnyCollection::t_Position o_pos;

o_pos = co_collection. First ();

while (o_pos != 0)
  {
  ct_Object * pco_object = co_collection. GetPtr (o_pos);
```

```
  // ...
  if ( /* delete entry ? */ )
    o_pos = co_collection. DelPtrAndObj (o_pos);
  else
    o_pos = co_collection. Next (o_pos);
  }
```

2.8.4 Abstract Ref-Collection (tuning/refcollection.hpp)

The ref-collection interface is identical to the ref-list interface (see above 'Ref-Lists', template `gct_RefDList`). A specific ref-collection class is implemented by using the methods of a specific ref pointer list, e.g. `gct_Chn_RefPtrDList <ct_Object>`.

Base Classes

`ct_Object` (see above 'Abstract Object')
  `ct_Collection` (see above 'Abstract Collection')

Class Declaration

```
class ct_RefCollection: public ct_Collection
  {
public:
  virtual void        IncRef (t_Position o_pos) = 0;
  virtual void        DecRef (t_Position o_pos) = 0;
  virtual t_RefCount  GetRef (t_Position o_pos) const = 0;
  virtual bool        IsAlloc (t_Position o_pos) const = 0;
  virtual bool        IsFree (t_Position o_pos) const = 0;
  };
```

2.8.5 Predefined Collections

Some collection classes are predefined to easily use the collection and ref-collection interfaces. The macro `COLLMAP_DCL` declares a collection class. The macro `COLLMAP_DEF` generates the implementation of the class methods using a pointer container ('**tuning/collmap.hpp**'). The macros `REFCOLLMAP_DCL` and `REFCOLLMAP_DEF` are used to declare and implement ref-collections ('**tuning/refcollmap.hpp**'). The header file of a collection class does not include any container header file.

	Implementation	Compile	Runtime
Container	templates, inline methods	slower	faster
Collection	virtual methods	faster	slower

The macro

```
COLLMAP_DCL (Array)
```

is located in a header file and expands to:

```
class ct_Array: public ct_Collection
  {
  // ...
  };
```

The macro COLLMAP_DEF is located in a cpp file. Predefined collection and ref-collection classes are based on pointer containers of type gct_Chn_....

```
#include "tuning/chn/ptrarray.h"
COLLMAP_DEF (Array, gct_Chn_PtrArray)
```

The file 'tuning/array.hpp' contains the following declaration:

```
class ct_Array: public ct_Collection { /*...*/ };
```

The file 'tuning/dlist.hpp' contains the following declaration:

```
class ct_DList: public ct_Collection { /*...*/ };
```

The file 'tuning/sortedarray.hpp' contains the following declaration:

```
class ct_SortedArray: public ct_Collection { /*...*/ };
```

The file 'tuning/blockdlist.hpp' contains the following declaration:

```
class ct_BlockDList: public ct_Collection { /*...*/ };
```

The file 'tuning/refdlist.hpp' contains the following declaration:

```
class ct_RefDList: public ct_RefCollection { /*...*/ };
```

The file 'tuning/blockrefdlist.hpp' contains the following declaration:

```
class ct_BlockRefDList: public ct_RefCollection { /*...*/ };
```

3 STRINGS AND UTILITIES

3.1 System Interface

3.1.1 Resource Errors (tuning/sys/creserror.hpp)

This enum defines different resource errors.

Enumeration

```
enum et_ResError
  {
  ec_ResOK = 0,
  ec_ResUnknownError,
  ec_ResUninitialized,
  ec_ResAlreadyInitialized,
  ec_ResInvalidKey,
  ec_ResInvalidValue,
  ec_ResNoKey,
  ec_ResAlreadyExists,
  ec_ResAccessDenied,
  ec_ResNotFound,
  ec_ResLockCountMismatch,
  ec_ResLockFailed,
  ec_ResUnlockFailed,
  ec_ResMemMapFailed,
  ec_ResUnmapFailed,
  ec_ResQuerySizeFailed
  };
```

`ec_ResOK`

No errors occured.

`ec_ResUnknownError`

Unknown error.

`ec_ResUninitialized`

Attempt to use an uninitialized object.

`ec_ResAlreadyInitialized`

Attempt to reinitialize an initialized object.

`ec_ResInvalidKey`

Invalid key.

`ec_ResInvalidValue`

Invalid function parameter.

`ec_ResNoKey`

Attempt to use an object without a key.

ec_ResAlreadyExists

Object with a specific key already exists.

ec_ResAccessDenied

Access denied.

ec_ResNotFound

Object with a specific key not found.

ec_ResLockCountMismatch

Mutex lock/unlock mismatch.

ec_ResLockFailed

Mutex lock failed.

ec_ResUnlockFailed

Mutex unlock failed.

ec_ResMemMapFailed

Shared memory mapping failed.

ec_ResUnmapFailed

Shared memory unmapping failed.

ec_ResQuerySizeFailed

Query shared memory size failed.

3.1.2 Character and String Conversion (tuning/sys/cstring.hpp)

This section describes several character and string conversion functions. Each 8-bit character function has a matching wide character version. Length parameters refer to the number of characters, not to the size in bytes.

The character case conversion functions are implemented in two different ways. The first implementation (tl_ToUpper/tl_ToLower) is very fast. It uses the Windows-1252 character set (this is a superset of ISO 8859-1 (Latin-1)). These functions use a static conversion table independent of the current locale. This implementation is not compatible with UTF strings.

The second implementation (tl_ToUpper2/tl_ToLower2) uses fast, wide character based system calls (MS Windows: CharUpperW, Linux: towupper). The matching 8-bit character versions use a temporary wide character string. This implementation is partially compatible with UTF strings (see also next section).

Multibyte strings (char) are partially compatible with UTF-8. Wide character strings (wchar_t) are partially compatible with UTF-16 (MS Windows: 16 bit, Linux: 16 or 32 bit). See next section for full UTF compatible functions. The conversion between multibyte and wide character strings consists of two steps: calculate the size of the target buffer and perform the conversion. The conversion functions rely on corresponding system functions (e.g. MS Windows: MultiByteToWideChar, Linux: mbstowcs).

Functions

char **tl_ToUpper** (char c);
wchar_t **tl_ToUpper** (wchar_t c);

Converts a single character to upper case (Windows-1252).

```
char tl_ToLower (char c);
wchar_t tl_ToLower (wchar_t c);
```

Converts a single character to lower case (Windows-1252).

```
bool tl_ToUpper (char * pc_str);
bool tl_ToUpper (wchar_t * pc_str);
```

Converts a null-terminated string to upper case (Windows-1252).

```
bool tl_ToLower (char * pc_str);
bool tl_ToLower (wchar_t * pc_str);
```

Converts a null-terminated string to lower case (Windows-1252).

```
wchar_t tl_ToUpper2 (wchar_t c);
```

Converts a single character to upper case (partially UTF compatible).

```
wchar_t tl_ToLower2 (wchar_t c);
```

Converts a single character to lower case (partially UTF compatible).

```
bool tl_ToUpper2 (char * pc_str);
bool tl_ToUpper2 (wchar_t * pc_str);
```

Converts a null-terminated string to upper case (partially UTF compatible).

```
bool tl_ToLower2 (char * pc_str);
bool tl_ToLower2 (wchar_t * pc_str);
```

Converts a null-terminated string to lower case (partially UTF compatible).

```
t_UInt tl_StringLength (const char * pc);
t_UInt tl_StringLength (const wchar_t * pc);
```

Calculates the length of a string up to, but not including the terminating null character.

```
unsigned tl_StringHash (const char * pc, t_UInt u_length);
unsigned tl_StringHash (const wchar_t * pc, t_UInt u_length);
```

Calculates the string's hash value.

```
t_UInt tl_MbConvertCount (wchar_t *, const char * pc_src);
```

Counts the number of wide characters inclusive the terminating null character to convert a null-terminated multibyte string. The type of the first parameter is used to resolve overloaded functions, the parameter value is not used.

```
bool tl_MbConvert (wchar_t * pc_dst, const char * pc_src, t_UInt u_count);
```

Converts a null-terminated multibyte string to a null-terminated wide character string. `u_count` is the wide character size of the destination buffer.

```
t_UInt tl_MbConvertCount (char *, const wchar_t * pc_src);
```

Counts the number of 8-bit characters inclusive the terminating null character to convert a null-terminated wide character string. The type of the first parameter is used to resolve overloaded functions, the parameter value is not used.

```
bool tl_MbConvert (char * pc_dst, const wchar_t * pc_src, t_UInt u_count);
```

Converts a null-terminated wide character string to a null-terminated multibyte string. `u_count` is the 8-bit character size of the destination buffer.

Appropriate Classes

The classes ct_String and ct_WString rely on the global functions of this section.

3.1.3 Unicode (UTF) (tuning/sys/cutf.hpp)

The implementation of multibyte and wide character functions (previous section) relies on corresponding system functions (e.g. MS Windows: MultiByteToWideChar, Linux: mbstowcs). These functions are partially compatible with UTF strings, and the runtime behavior is OS and locale dependent.

The conversion functions of this section don't use any external resources. The algorithms are fully compatible with the UTF encodings, and the runtime behavior is OS and locale independent. They work on null-terminated and non-null-terminated strings. In case of an UTF format error, a precise error code and the precise error position are returned.

Enumeration

```
enum et_UtfError
  {
  ec_UtfOK = 0,
  ec_UtfMissingNull,      // Missing null character
  ec_UtfNullInside,       // Null character inside of string
  ec_UtfMbMissingStart,   // Multibyte (10xx xxxx) without startbyte (11xx xxxx)
  ec_UtfMbInvalidStart,   // Invalid startbyte (1111 1xxx)
  ec_UtfMbExpected,       // Multibyte (10xx xxxx) expected
  ec_UtfMbEnd,            // String end in multibyte sequence
  ec_UtfWideRange,        // Wide character out of range
  ec_UtfSurrogate,        // UTF-16 surrogate in wide character
  ec_UtfHighSurrExpected, // High surrogate expected
  ec_UtfLowSurrExpected,  // Low surrogate expected
  ec_UtfSurrEnd,          // String end in surrogate
  ec_UtfDestTooSmall,     // Destination buffer size too small
  ec_UtfDestTooLarge,     // Destination buffer size too large
  ec_UtfEOS,              // End of string
  ec_UtfLastError
  };
```

An UTF-8 character is of type t_UInt8, an UTF-16 character is of type t_UInt16, and an UTF-32 character is of type t_UInt32. Length parameters refer to the number of characters, not to the size in bytes.

The following UTF conversions are implemented: UTF-8 <-> UTF-32, UTF-16 <-> UTF-32 and UTF-8 <-> UTF-16. A string conversion consists of two steps: calculate the size of the target buffer and perform the conversion. If the parameter b_null equals true, the conversion includes the terminating null character.

The length functions count the number of UTF characters (inclusive the terminating null character, if the parameter b_null equals true). The upper/lower functions convert UTF strings to upper/lower case. The conversion is done for UTF characters of the Basic Multilingual Plane (< 0x10000) which don't change the size.

If the source pointer pu_src is of type 'reference to pointer', the parameter is used to store the error position in case of an UTF error. If the parameter b_null equals true, the string must be terminated by a null character, and inside of the string null characters are not allowed.

Functions

et_UtfError **tl_UtfConvertCount** (t_UInt**Y** *, t_UInt & u_dstLen, const t_UInt**X** * & pu_src, t_UInt u_srcLen, bool b_null = true);

Counts the number of UInt-Y characters to convert the UTF-X string (pu_src, u_srcLen) to UTF-Y, and stores the result in u_dstLen. The type of the first parameter is used to resolve overloaded functions, the parameter value is not used.

et_UtfError **tl_UtfConvert** (t_UInt**Y** * pu_dst, t_UInt u_dstLen, const t_UInt**X** * pu_src, t_UInt u_srcLen, bool b_null = true);

Converts the UTF-X string (pu_src, u_srcLen) to the destination buffer (pu_dst, u_dstLen) of type UTF-Y.

et_UtfError **tl_UtfLength** (t_UInt & u_len, const t_UInt**X** * & pu_src, t_UInt u_srcLen, bool b_null = true);

Counts the number of UTF characters of the UTF-X string (pu_src, u_srcLen) and store the result in u_len.

et_UtfError **tl_UtfToUpper** (t_UInt**X** * & pu_src, t_UInt u_srcLen);

Converts the UTF-X string (pu_src, u_srcLen) to upper case.

et_UtfError **tl_UtfToLower** (t_UInt**X** * & pu_src, t_UInt u_srcLen);

Converts the UTF-X string (pu_src, u_srcLen) to lower case.

3.1.4 Unicode Const Iterator (tuning/utfcit.h)

The UTF const iterator is a utility to iterate over constant UTF-8, UTF-16 and UTF-32 strings without converting the data into a temporary UTF-32 buffer. The iterator converts the current (possibly multiword) UTF character to UTF-32 and provides some position and length information. An UTF-8 character is of type t_UInt8, an UTF-16 character is of type t_UInt16, and an UTF-32 character is of type t_UInt32. Length parameters refer to the number of characters, not to the size in bytes. The UTF string may contain null characters. Modifying the string while iterating it is not allowed.

Template Declaration

```
template <class t_char>
  class gct_UtfCit
    {
  public:
    typedef t_char      t_Char;

    inline              gct_UtfCit ();
    inline              gct_UtfCit (const t_Char * pu_src, t_UInt u_srcLen);

    void                First (const t_Char * pu_src, t_UInt u_srcLen);
    bool                Ready () const;
    void                Next ();

    t_UInt32            GetChar () const;
    t_UInt              GetCharPos () const;
    t_UInt              GetRawPos () const;
    t_UInt              GetRawLen () const;
    et_UtfError         GetError () const;
    };
```

Methods

gct_UtfCit ();

Initializes an empty iterator.

gct_UtfCit (const t_UIntX * pu_src, t_UInt u_srcLen);

Initializes the iterator and reads the first UTF character from the UTF-X string (pu_src, u_srcLen).

void **First** (const t_UIntX * pu_src, t_UInt u_srcLen);

Reads the first UTF character from the UTF-X string (pu_src, u_srcLen).

bool **Ready** () const;

Returns true if an UTF character was read successfully.

void **Next** ();

Reads the next UTF character from the source string.

t_UInt32 **GetChar** () const;

Returns the current UTF character in UTF-32 format.

t_UInt **GetCharPos** () const;

Returns the sequential number of the current UTF character.

t_UInt **GetRawPos** () const;

Returns the position of the current UTF character in t_UIntX format.

t_Uint **GetRawLen** () const;

Returns the length of the current UTF character in t_UIntX format.

et_UtfError **GetError** () const;

Returns the error code of the current UTF character.
ec_UtfOK: UTF character was read successfully.
ec_UtfEOS: End of string.
Other error: UTF format error. Iteration aborted.

Sample Code

The following sample code demonstrates a forward iteration over an UTF-X string.

```
gct_UtfCit <t_UIntX> co_cit;

for (co_cit. First (pu_src, u_srcLen);
     co_cit. Ready ();
     co_cit. Next ())
  {
  t_UInt32 u_char = co_cit. GetChar ();
  // ...
  }

if (co_cit. GetError () != ec_UtfEOS)
  {
  // error handling
  }
```

3.1.5 Precision Time (tuning/sys/ctimedate.hpp)

The system time (next section) is inaccurate in the microsecond range. The following function provides a more precise measurement.

Data Types

`typedef t_Int64 t_MicroTime;`

Data type for precision time values.

Functions

`t_MicroTime tl_QueryPrecisionTime ();`

Returns the time in microseconds since the first call of the function.

3.1.6 Time and Date (tuning/sys/ctimedate.hpp)

The following functions can be used for calendar and time calculations. Time values are expressed in microseconds since 1/1/1970. The current time can be queried in UTC and local time.

Data Types, Constants

`typedef t_Int64 t_MicroTime;`

Time values are expressed in microseconds since 1/1/1970.

```
const t_MicroTime co_MicroSecondFactor =           1ll;
const t_MicroTime co_MilliSecondFactor =        1000ll;
const t_MicroTime co_SecondFactor      =     1000000ll;
const t_MicroTime co_MinuteFactor      =    60000000ll;
const t_MicroTime co_HourFactor        =  3600000000ll;
const t_MicroTime co_DayFactor         = 86400000000ll;
```

These constants are conversion factors from microseconds to milliseconds, seconds, minutes, hours and days.

Functions

`t_MicroTime tl_QueryUTCTime ();`

Returns the current time, as reported by the system clock, in UTC.

`t_MicroTime tl_QueryLocalTime ();`

Returns the current time, as reported by the system clock, in the local time zone.

`t_MicroTime tl_UTCToLocalTime (t_MicroTime i_time);`

Converts UTC to local time.

`t_MicroTime tl_LocalToUTCTime (t_MicroTime i_time);`

Converts local time to UTC.

Appropriate Class

The class `ct_TimeDate` relies on the global functions of this section.

3.1.7 CPU Time (tuning/sys/ctimedate.hpp)

The following functions retrieve timing information for a process or thread.

Structure Declaration

```
struct st_UserKernelTime
  {
  t_MicroTime          o_UserTime;
  t_MicroTime          o_KernelTime;
  };
```

This struct contains two microsecond values.
`o_UserTime`: Amount of time that the process/thread has executed in user mode.
`o_KernelTime`: Amount of time that the process/thread has executed in kernel mode.

Functions

`bool` **`tl_QueryProcessTimes`** `(st_UserKernelTime * pso_times);`

Retrieves timing information for the current process.

`bool` **`tl_QueryThreadTimes`** `(st_UserKernelTime * pso_times);`

Retrieves timing information for the current thread.

3.1.8 Thread Utilities (tuning/sys/cprocess.hpp)

The following functions can be used for multithreading.

Functions

`t_Int32` **`tl_InterlockedRead`** `(volatile t_Int32 * pi_value);`

Returns a 32-bit value, loaded as an atomic operation.

`t_Int32` **`tl_InterlockedWrite`** `(volatile t_Int32 * pi_value, t_Int32 i_new);`

Writes a 32-bit value as an atomic operation.

`t_Int32` **`tl_InterlockedAdd`** `(volatile t_Int32 * pi_value, t_Int32 i_add);`

Performs an atomic addition operation on a 32-bit value and returns the result.

`t_Int32` **`tl_InterlockedIncrement`** `(volatile t_Int32 * pi_value);`
`t_Int32` **`tl_InterlockedDecrement`** `(volatile t_Int32 * pi_value);`

Increments/decrements a 32-bit value as an atomic operation and returns the result.

`void` **`tl_Delay`** `(int i_milliSec);`

Suspends the current thread for the specified number of milliseconds.

`void` **`tl_RelinquishTimeSlice`** `();`

The current thread relinquishes the remainder of its time slice to any other thread.

`ct_String` **`tl_GetEnv`** `(const char * pc_name);`

Returns the value of the environment variable specified by the null-terminated string `pc_name`.

`ct_String` **`tl_GetTempPath`** `();`

Returns the path for temporary files.

3.1.9 Threads (tuning/sys/cthread.hpp)

The following functions can be used to create and terminate threads.

Data Types

`typedef void (* ft_ThreadFunc) (void *);`

Pointer to the thread function.

Functions

`bool tl_BeginThread (ft_ThreadFunc fo_func, void * pv_param, t_UInt u_stackSize = 8u * 1024u);`

Creates and starts a new thread and returns `true` on success. The parameter `fo_func` points to the thread function. The parameter `pv_param` is passed to this function. Optionally the stack size of the new thread can be specified. The thread is terminated by returning from the thread function or by calling `tl_EndThread`.

`void tl_EndThread ();`

Terminates the current thread. The MS Windows implementation does not call destructors of local objects.

`t_UInt64 tl_ThreadId ();`

Returns an OS dependent thread id.

3.1.10 Processes (tuning/sys/cprocess.hpp)

The following functions can be used to create and terminate processes.

Functions

`int tl_Exec (const char * pc_path, unsigned u_params, const char * * ppc_params, bool b_wait = false);`

Creates and starts a new process. The parameter `pc_path` specifies the path to the executable file. Optionally `u_params` string parameters can be passed to the new process. The parameter `ppc_params` must point to an array containing `u_params` pointers. A string parameter pointer must be equal to the null pointer or it must point to a null-terminated string. Null pointers are replaced by pointers to an empty string. A string parameter may contain whitespace, and it may begin and end with '"'.

On error the function returns -1. If the parameter `b_wait` equals `false`, the function returns an OS dependent id of the new process. Otherwise the function waits for termination of the new process and returns its exit code. See also the sample programs 'texec' and 'texechelper'.

`void tl_EndProcess (unsigned u_exitCode);`

Terminates the current process without calling destructors. The parameter `u_exitCode` is passed to the operating system.

`int tl_ProcessId ();`

Returns an OS dependent process id.

`bool tl_IsProcessRunning (int i_processId);`

Returns `true` if the process specified by `i_processId` was started successfully and is still running.

3.1.11 Thread Mutex (tuning/sys/cthmutex.hpp)

A thread mutex is an object to synchronize multiple threads of a process.

Class Declaration

```
class ct_ThMutex
  {
public:
  bool                GetInitSuccess () const;
  et_ResError         TryLock (bool & b_success);
  et_ResError         Lock ();
  et_ResError         Unlock ();
  };
```

The class `ct_ThMutex` can be used to protect access to a shared resource (mutual exclusion). If a thread locks a mutex, the same thread must unlock the mutex. The implementation is recursive, i.e. a thread may lock an already locked mutex. Mutex objects must not be copied by a copy constructor, an assignment operator, `memcpy` or `memmove`.

Methods

`bool` **`GetInitSuccess`** `();`

Returns `true` if the mutex object was initialized successfully.

`et_ResError` **`TryLock`** `(bool & b_success);`

Tries to lock the mutex and stores `true` or `false` in `b_success`. The method returns immediately without blocking the thread.

`et_ResError` **`Lock`** `();`

Locks the mutex and returns immediately on success. If another thread has locked the mutex the current thread will be blocked until the mutex is unlocked.

`et_ResError` **`Unlock`** `();`

Unlocks the mutex.

Functions

The following functions use a predefined global mutex object.

`bool` **`tl_CriticalSectionInitSuccess`** `();`

Returns `true` if the global mutex object was initialized successfully.

`void` **`tl_DeleteCriticalSection`** `();`

Deletes the global mutex object. This function may be called optionally at the end of the program.

`et_ResError` **`tl_TryEnterCriticalSection`** `(bool & b_success);`

Tries to lock the global mutex object and stores `true` or `false` in `b_success`. The method returns immediately without blocking the thread.

`et_ResError` **`tl_EnterCriticalSection`** `();`

Locks the global mutex object and returns immediately on success. If another thread has locked the mutex the current thread will be blocked until the mutex is unlocked.

et_ResError **tl_LeaveCriticalSection** ();

Unlocks the global mutex object.

3.1.12 Thread Semaphore (tuning/sys/cthsemaphore.hpp)

A thread semaphore is an object to synchronize multiple threads of a process.

Class Declaration

```
class ct_ThSemaphore
  {
public:
                      ct_ThSemaphore (t_Int32 i_initValue = 1);
                      ~ct_ThSemaphore ();

  bool                GetInitSuccess () const;
  et_ResError         TryAcquire (bool & b_success, t_UInt32 u_milliSec = 0);
  et_ResError         Acquire ();
  et_ResError         Release ();
  };
```

The class ct_ThSemaphore implements a counting semaphore. A semaphore can be acquired and released by multiple threads in arbitrary order. The method Acquire decrements the internal counter, Release increments the counter. If the counter becomes zero, the current thread will be blocked until another thread releases the semaphore.

If the counter initially equals 1, a counting semaphore can be used like a mutex. In this case the method Acquire works like Lock, and Release works like Unlock. If the counter initially equals zero, a counting semaphore can be used to implement a message queue (see sample program 'tsemaphore'). Semaphore objects must not be copied by a copy constructor, an assignment operator, memcpy or memmove.

Methods

ct_ThSemaphore (t_Int32 i_initValue = 1);

Initializes the object and sets the internal counter to i_initValue.

bool **GetInitSuccess** ();

Returns true if the semaphore object was initialized successfully.

et_ResError **TryAcquire** (bool & b_success, t_UInt32 u_milliSec = 0);

Tries to acquire the semaphore and stores true or false in b_success. The method will wait for at most u_milliSec milliseconds.

et_ResError **Acquire** ();

Acquires the semaphore (i.e. decrement the counter) and returns immediately on success. If the counter becomes zero, the current thread will be blocked until another thread releases the semaphore.

et_ResError **Release** ();

Releases the semaphore (i.e. increments the counter).

3.1.13 Shared Resource (tuning/sys/csharedres.hpp)

The class ct_SharedResource is the base class for objects which can be shared by multiple processes. A shared resource is identified by a key (an 8-bit character string).

Before using a shared resource, a key must be assigned and the object must be initialized by calling Open or Create of a derived class. Once a shared resource has been initialized, the key must not be changed.

Class Declaration

```
class ct_SharedResource
  {
public:
                      ct_SharedResource ();
                      ct_SharedResource (const char * pc_key);
                      ct_SharedResource (const char * pc_key, unsigned u_idx);
  virtual             ~ct_SharedResource ();

  bool                GetInitSuccess () const;
  const char *        GetKey () const;
  et_ResError         SetKey (const char * pc_key);
  et_ResError         SetKey (const char * pc_key, unsigned u_idx);
  };
```

Methods

ct_SharedResource ();

Constructs a shared resource without a key.

ct_SharedResource (const char * pc_key);

Constructs a shared resource identified by pc_key.

ct_SharedResource (const char * pc_key, unsigned u_idx);

Constructs a shared resource identified by pc_key and u_idx. The value of u_idx is converted to a string and appended to pc_key.

virtual ~ct_SharedResource ();

The virtual destructor ensures type-safe destruction of derived classes.

bool GetInitSuccess ();

Returns true if the shared resource was initialized successfully.

const char * GetKey () const;

Returns the key.

et_ResError SetKey (const char * pc_key);

Sets the key to pc_key.

et_ResError SetKey (const char * pc_key, unsigned u_idx);

Sets the key to pc_key. The value of u_idx is converted to a string and appended to pc_key.

3.1.14 Process Mutex (tuning/sys/cprmutex.hpp)

A process mutex is an object to synchronize multiple processes.

Base Class

ct_SharedResource (see above 'Shared Resource')

Class Declaration

```
class ct_PrMutex: public ct_SharedResource
  {
public:
                       ct_PrMutex ();
                       ct_PrMutex (const char * pc_key);
                       ct_PrMutex (const char * pc_key, unsigned u_idx);
                       ~ct_PrMutex ();

  et_ResError          Open ();
  et_ResError          Create (bool b_createNew = false);
  et_ResError          Close ();

  et_ResError          TryLock (bool & b_success, t_UInt32 u_milliSec = 0);
  et_ResError          Lock ();
  et_ResError          Unlock ();
  };
```

The class ct_PrMutex can be used to protect access to a shared resource (mutual exclusion). A process mutex is fully initialized if the key has been set and Open or Create has returned ec_ResOK. If a process locks a mutex, the same process must unlock the mutex. The MS Windows implementation is recursive, i.e. a process may lock an already locked mutex. The Linux implementation is not recursive. The methods TryLock, Lock and Unlock are thread-safe. Mutex objects must not be copied by a copy constructor or an assignment operator.

Methods

ct_PrMutex ();

Constructs a process mutex using a predefined key.

ct_PrMutex (const char * pc_key);

Constructs a process mutex identified by pc_key.

ct_PrMutex (const char * pc_key, unsigned u_idx);

Constructs a process mutex identified by pc_key and u_idx. The value of u_idx is converted to a string and appended to pc_key.

~ct_PrMutex ();

The destructor closes the mutex.

et_ResError **Open** ();

Opens an existing process mutex.

et_ResError **Create** (bool b_createNew = false);

Creates a new process mutex. Returns ec_ResAlreadyExists if b_createNew equals true and a process mutex with the same key already exists.

et_ResError **Close** ();

Closes an open process mutex.

et_ResError **TryLock** (bool & b_success, t_UInt32 u_milliSec = 0);

Tries to lock the process mutex and stores true or false in b_success. The method will wait for at most u_milliSec milliseconds.

et_ResError **Lock** ();

Locks the process mutex and returns immediately on success. If another process has locked the mutex the current thread will be blocked until the mutex is unlocked.

et_ResError **Unlock** ();

Unlocks the process mutex.

Functions

The following functions use a predefined global mutex object.

bool **tl_CriticalPrSectionInitSuccess** ();

Returns true if the global mutex object was initialized successfully.

void **tl_DeleteCriticalPrSection** ();

Deletes the global mutex object. This function may be called optionally at the end of the program.

et_ResError **tl_TryEnterCriticalPrSection** (bool & b_success, t_UInt32 u_milliSec = 0);

Tries to lock the global mutex object and stores true or false in b_success. The method will wait for at most u_milliSec milliseconds.

et_ResError **tl_EnterCriticalPrSection** ();

Locks the global mutex object and returns immediately on success. If another process has locked the mutex the current thread will be blocked until the mutex is unlocked.

et_ResError **tl_LeaveCriticalPrSection** ();

Unlocks the global mutex object.

3.1.15 Process Semaphore (tuning/sys/cprsemaphore.hpp)

A process semaphore is an object to synchronize multiple processes.

Base Class

ct_SharedResource (see above 'Shared Resource')

Class Declaration

```
class ct_PrSemaphore: public ct_SharedResource
  {
public:
                      ct_PrSemaphore ();
                      ct_PrSemaphore (const char * pc_key);
                      ct_PrSemaphore (const char * pc_key, unsigned u_idx);
                      ~ct_PrSemaphore ();

  et_ResError         Open ();
  et_ResError         Create (t_Int32 i_initValue = 1, bool b_createNew = false);
  et_ResError         Close ();

  et_ResError         TryAcquire (bool & b_success, t_UInt32 u_milliSec = 0);
  et_ResError         Acquire ();
  et_ResError         Release ();
  };
```

The class ct_PrSemaphore implements a counting semaphore. A process semaphore is fully initialized if the key has been set and Open or Create has returned ec_ResOK. A semaphore can be acquired and released by multiple processes in arbitrary order. The method Acquire decrements the internal counter, Release increments the counter. If the counter becomes zero, the current thread will be blocked until another process releases the semaphore.

If the counter initially equals 1, a counting semaphore can be used like a mutex. In this case the method Acquire works like Lock, and Release works like Unlock. If the counter initially equals zero, a counting semaphore can be used to implement a message queue (see sample program 'tsemaphore'). The methods TryAcquire, Acquire and Release are thread-safe. Semaphore objects must not be copied by a copy constructor or an assignment operator.

Methods

ct_PrSemaphore ();

Constructs a process semaphore using a predefined key.

ct_PrSemaphore (const char * pc_key);

Constructs a process semaphore identified by pc_key.

ct_PrSemaphore (const char * pc_key, unsigned u_idx);

Constructs a process semaphore identified by pc_key and u_idx. The value of u_idx is converted to a string and appended to pc_key.

~ct_PrSemaphore ();

The destructor closes the semaphore.

et_ResError **Open** ();

Opens an existing process semaphore.

et_ResError **Create** (t_Int32 i_initValue = 1, bool b_createNew = false);

Creates a new process semaphore and sets the internal counter to i_initValue. Returns ec_ResAlreadyExists if b_createNew equals true and a process semaphore with the same key already exists.

et_ResError **Close** ();

Closes an open process semaphore.

et_ResError **TryAcquire** (bool & b_success, t_UInt32 u_milliSec = 0);

Tries to acquire the semaphore and stores true or false in b_success. The method will wait for at most u_milliSec milliseconds.

et_ResError **Acquire** ();

Acquires the semaphore (i.e. decrement the counter) and returns immediately on success. If the counter becomes zero, the current thread will be blocked until another process releases the semaphore.

et_ResError **Release** ();

Releases the semaphore (i.e. increments the counter).

3.1.16 Shared Memory (tuning/sys/csharedmem.hpp)

The class ct_SharedMemory provides access to a shared memory block by multiple processes. A shared memory object is fully initialized if the key has been set and Open or Create has returned ec_ResOK.

Base Class

ct_SharedResource (see above 'Shared Resource')

Class Declaration

```
class ct_SharedMemory: public ct_SharedResource
  {
public:
                       ct_SharedMemory ();
                       ct_SharedMemory (const char * pc_key);
                       ct_SharedMemory (const char * pc_key, unsigned u_idx);
                       ~ct_SharedMemory ();

  et_ResError          Open (bool b_readOnly);
  et_ResError          Create (t_UInt u_size, bool b_createNew = false);
  et_ResError          Close ();

  t_UInt               GetSize () const;
  void *               GetData () const;
  };
```

Methods

ct_SharedMemory ();

Constructs a shared memory object using a predefined key.

ct_SharedMemory (const char * pc_key);

Constructs a shared memory object identified by pc_key.

ct_SharedMemory (const char * pc_key, unsigned u_idx);

Constructs a shared memory object identified by pc_key and u_idx. The value of u_idx is converted to a string and appended to pc_key.

~ct_SharedMemory ();

The destructor closes the shared memory object.

et_ResError **Open** (bool b_readOnly);

Opens an existing shared memory object. The parameter b_readOnly determines the access mode.

et_ResError **Create** (t_UInt u_size, bool b_createNew = false);

Creates a new shared memory block of u_size bytes. Returns ec_ResAlreadyExists if b_createNew equals true and a shared memory object with the same key already exists.

et_ResError **Close** ();

Closes an open shared memory object.

t_UInt **GetSize** () const;

Returns the size of the shared memory block.

void * **GetData** () const;

Returns a pointer to the contents of the shared memory block.

3.1.17 File I/O (tuning/sys/cfile.hpp)

Within the **Spirick Tuning** library all file and directory paths are interpreted as UTF-8 strings. The Linux implementation passes the path names unchanged to the corresponding system functions. The MS Windows implementation converts the path names temporarily to UTF-16.

The following functions are based on operating system related functions. In most cases, the OS API functions perform better than the compiler's runtime system (`fopen` etc.). The functions `tl_OpenFile` and `tl_CreateFile` are protected against race conditions. All functions return `true` on success and `false` on failure, no exceptions are thrown.

Data Types, Constants

```
typedef ... t_FileId;
const t_FileId co_InvalidFileId = ...;
typedef t_Int64 t_FileSize;
```

A file id is an OS dependent identification number that references an open file. The constant `co_InvalidFileId` is invalid by definition. `t_FileSize` is used for size and position values.

Functions

```
bool tl_OpenFile (const char * pc_name, t_FileId & o_file, bool b_readOnly = true, bool b_sequential = true);
```

Opens the existing file `pc_name`. The parameter `b_readOnly` determines the access mode. The parameter `b_sequential` is a hint to optimize file caching (sequential or random access). Set `o_file` to `co_InvalidFileId` before calling the function. Returns `true` on success and stores the file id in `o_file`.

```
bool tl_CreateFile (const char * pc_name, t_FileId & o_file, bool b_createNew = false);
```

Creates the new file `pc_name` and opens it for read/write access. Returns `false` if `b_createNew` equals `true` and the specified file already exists. Otherwise the function overwrites the existing file. Set `o_file` to `co_InvalidFileId` before calling the function. Returns `true` on success and stores the file id in `o_file`.

```
bool tl_CloseFile (t_FileId o_file);
```

Closes the file `o_file`.

```
bool tl_ExistsFile (const char * pc_name);
```

Returns `true` if the file `pc_name` exists.

```
bool tl_MoveFile (const char * pc_old, const char * pc_new);
```

Moves (renames) a file either in the same directory or across directories.

```
bool tl_CopyFile (const char * pc_old, const char * pc_new, bool b_overwrite = true);
```

Copies an existing file to a new file. Returns `false` if `b_overwrite` equals `false` and the specified file already exists.

```
bool tl_DeleteFile (const char * pc_name);
```

Deletes an existing file.

```
bool tl_QuerySize (t_FileId o_file, t_FileSize & o_size);
```

Retrieves the size of the specified file and stores the result in `o_size`.

```
bool tl_QueryPos (t_FileId o_file, t_FileSize & o_pos);
```

Retrieves the file pointer of the specified file and stores the result in `o_pos`.

bool **tl_SeekAbs** (t_FileId o_file, t_FileSize o_pos);

Moves the file pointer of the specified file to the absolute position o_pos (an offset from the beginning of the file).

bool **tl_SeekRel** (t_FileId o_file, t_FileSize o_pos);

Moves the file pointer of the specified file to the relative position o_pos (relative to the current position).

bool **tl_Truncate** (t_FileId o_file, t_FileSize o_size);

Sets the size for the specified file to o_size.

bool **tl_Read** (t_FileId o_file, void * pv_dst, t_FileSize o_len);

Reads o_len bytes from the specified file to the buffer pv_dst and moves the file pointer.

bool **tl_Write** (t_FileId o_file, const void * pv_src, t_FileSize o_len);

Writes o_len bytes from the buffer pv_src to the specified file and moves the file pointer.

Appropriate Class

The class ct_File relies on the global functions of this section.

3.1.18 Directory (tuning/sys/cdir.hpp)

Within the **Spirick Tuning** library all file and directory paths are interpreted as UTF-8 strings. The Linux implementation passes the path names unchanged to the corresponding system functions. The MS Windows implementation converts the path names temporarily to UTF-16.

The following functions can be used to create, move and delete directories. All functions return true on success and false on failure, no exceptions are thrown.

Functions

bool **tl_QueryCurrentDirectory** (const char * pc_drive, t_UInt u_driveLen, ct_String & co_currentDirectory);

Retrieves the current directory and stores the result in co_currentDirectory. MS Windows only: Retrieves the current directory of the drive (pc_drive, u_driveLen). If u_driveLen equals zero the current drive is used.

bool **tl_CreateDirectory** (const char * pc_name);

Creates the new directory pc_name.

bool **tl_MoveDirectory** (const char * pc_old, const char * pc_new);

Moves (renames) a directory either in the same directory or across directories.

bool **tl_DeleteDirectory** (const char * pc_name);

Deletes the existing directory pc_name.

Appropriate Class

The class ct_Directory relies on the global functions of this section.

3.1.19 System-Related Information (tuning/sys/cinfo.hpp)

The following functions retrieve several system-related information. Strings are static allocated.

Structure Declaration

```
struct st_FileSystemInfo
  {
  t_UInt64           u_TotalBytes;
  t_UInt64           u_FreeBytes;
  t_UInt64           u_AvailableBytes;
  };
```

The struct `st_FileSystemInfo` provides information about a mounted filesystem (a disk volume).
`u_TotalBytes`: The total number of bytes on a filesystem.
`u_FreeBytes`: The total number of free bytes on a filesystem.
`u_AvailableBytes`: The total number of free bytes on a filesystem that are available to the curr. user.

Structure Declaration

```
struct st_HardwareInfo
  {
  t_UInt64           u_TotalBytes;
  t_UInt64           u_AvailableBytes;
  unsigned           u_TotalProcessors;
  unsigned           u_AvailableProcessors;
  const char *       pc_CPUName;
  };
```

The struct `st_HardwareInfo` provides information about hardware components.
`u_TotalBytes`: The amount of physical memory.
`u_AvailableBytes`: The amount of physical memory currently available.
`u_TotalProcessors`: The number of logical processors (CPU cores).
`u_AvailableProcessors`: The number of logical processors (CPU cores) currently available.
`pc_CPUName`: The name of the CPU.

Note that if a 32-bit process is running in a 64-bit environment, the reported memory size may be greater than 4 GB.

Structure Declaration

```
struct st_ProcessMemoryInfo
  {
  t_UInt             u_VMBytes;
  t_UInt             u_RSSBytes;
  };
```

The struct `st_ProcessMemoryInfo` provides information about the memory usage of the current process.
`u_VMBytes`: The virtual memory size (memory that is committed for the process).
`u_RSSBytes`: The resident set size (memory that is currently resident in physical memory).

Note that the calculation of these values is OS dependent, e.g. the inclusion of shared memory.

Structure Declaration

```
enum et_Compiler
  {
  ec_CompilerMSVC,
  ec_CompilerGCC
  };

struct st_CompilerInfo
  {
  et_Compiler        eo_Compiler;
```

```
  const char *        pc_CompilerVersion;
  const char *        pc_RuntimeVersion;
  };
```

The struct st_CompilerInfo provides information about the compiler and the runtime system.
eo_Compiler: The compiler type.
pc_CompilerVersion: The compiler version.
pc_RuntimeVersion: The runtime version.

Structure Declaration

```
enum et_System
  {
  ec_SystemMSWindows,
  ec_SystemLinux
  };

struct st_SystemInfo
  {
  et_System           eo_System;
  const char *        pc_SystemVersion;
  const char *        pc_ComputerName;
  const char *        pc_UserName;
  };
```

The struct st_SystemInfo provides information about the operating system.
eo_System: The operating system type.
pc_SystemVersion: The operating system version.
pc_ComputerName: The name of the computer.
pc_UserName: The name of the current user.

Structure Declaration

```
struct st_BatteryInfo
  {
  bool                b_ACLine;
  bool                b_BatteryFound;
  int                 i_LifePercent;
  };
```

The struct st_BatteryInfo provides information about the power supply.
b_ACLine: Is the system running on line power?
b_BatteryFound: Does the system contain a battery?
i_LifePercent: The percentage of full battery charge remaining.

Functions

bool **tl_QueryFileSystemInfo** (const char * pc_path, st_FileSystemInfo * pso_info);

Retrieves information about the specified filesystem and stores the result in pso_info.

bool **tl_QueryHardwareInfo** (st_HardwareInfo * pso_info);

Retrieves information about hardware components and stores the result in pso_info.

bool **tl_QueryProcessMemoryInfo** (st_ProcessMemoryInfo * pso_info);

Retrieves information about the memory usage and stores the result in pso_info.

bool **tl_QueryCompilerInfo** (st_CompilerInfo * pso_info);

Retrieves information about the compiler and stores the result in pso_info.

bool **tl_QuerySystemInfo** (st_SystemInfo * pso_info);

Retrieves information about the operating system and stores the result in pso_info.

bool **tl_QueryBatteryInfo** (st_BatteryInfo * pso_info);

Retrieves information about the power supply and stores the result in pso_info.

3.2 Strings and Filenames

3.2.1 String Template (tuning/string.h)

The **Spirick** string classes manage null-terminated strings and contain additionally a length attribute. The terminating null character ensures compatibility with many other API's. The redundant length attribute speeds up string operations. Position values are zero-based. The string length does not count the terminating null character. Length values refer to the number of characters, not to the size in bytes.

The class template gct_String is the base class of all other string classes. The first template parameter t_block must at least contain the character block interface, e.g. gct_CharBlock <ct_Chn32Block, char>. To reduce the memory consumption of empty strings, it is recommended to use the template gct_NullDataBlock, e.g. gct_CharBlock <gct_NullDataBlock <ct_Chn32Block, char>, char>. The second template parameter t_staticStore must be a store class with static methods, e.g. ct_Chn32Store. It is used for temporary data inside of the method ReplaceAll.

Base Class

gct_CharBlock (see above 'Character Block')

Template Declaration

```
template <class t_block, class t_staticStore>
  class gct_String: public t_block
    {
  public:
    typedef t_block       t_Block;
    typedef t_staticStore t_StaticStore;
    typedef t_block::t_Char t_Char;
    typedef t_block::t_Size t_Size;

    inline              gct_String ();
    inline              gct_String (t_Char c_init);
    inline              gct_String (t_Char c_init, t_Size o_len);
    inline              gct_String (const t_Char * pc_init);
    inline              gct_String (const t_Char * pc_init, t_Size o_len);
    inline              gct_String (const gct_String & co_init);

    inline t_UInt       GetHash () const;
    inline bool         IsEmpty () const;
    inline t_Size       GetMaxLen () const;
    inline t_Size       GetLen () const;
    inline const t_Char * GetStr () const;
    inline const t_Char * operator () () const;
    inline const t_Char * GetStr (t_Size o_pos) const;
    inline const t_Char * operator () (t_Size o_pos) const;
    inline t_Char &     GetChar (t_Size o_pos) const;
    inline t_Char &     operator [] (t_Size o_pos) const;
    inline t_Char &     GetRevChar (t_Size o_pos) const;
    gct_String          SubStr (t_Size o_len) const;
    gct_String          RevSubStr (t_Size o_len) const;
```

```
gct_String          SubStr (t_Size o_pos, t_Size o_len) const;
gct_String          operator () (t_Size o_pos, t_Size o_len) const;

t_Int               First (t_Char c_search, t_Size o_pos = 0) const;
t_Int               First (const t_Char * pc_search, t_Size o_pos = 0) const;
t_Int               First (const gct_String & co_search, t_Size o_pos = 0) const;

t_Int               Last (t_Char c_search, t_Size o_pos = 0) const;
t_Int               Last (const t_Char * pc_search, t_Size o_pos = 0) const;
t_Int               Last (const gct_String & co_search, t_Size o_pos = 0) const;

inline int          CompSubStr (t_Size o_pos, t_Char c_comp) const;
inline int          CompSubStr (t_Size o_pos, const t_Char * pc_comp) const;
inline int          CompSubStr (t_Size o_pos, const t_Char * pc_comp, t_Size o_len) const;
inline int          CompSubStr (t_Size o_pos, const gct_String & co_comp) const;

inline int          CompTo (t_Char c_comp) const;
inline int          CompTo (const t_Char * pc_comp) const;
inline int          CompTo (const t_Char * pc_comp, t_Size o_len) const;
inline int          CompTo (const gct_String & co_comp) const;

inline void         Clear ();
inline void         Assign (t_Char c_asgn);
inline void         Assign (t_Char c_asgn, t_Size o_len);
void                Assign (const t_Char * pc_asgn);
inline void         Assign (const t_Char * pc_asgn, t_Size o_len);
void                Assign (const gct_String & co_asgn);
inline void         Append (t_Char c_app);
inline void         Append (t_Char c_app, t_Size o_len);
void                Append (const t_Char * pc_app);
inline void         Append (const t_Char * pc_app, t_Size o_len);
void                Append (const gct_String & co_app);

inline void         Insert (t_Size o_pos, t_Char c_ins);
inline void         Insert (t_Size o_pos, t_Char c_ins, t_Size o_len);
inline void         Insert (t_Size o_pos, const t_Char * pc_ins);
inline void         Insert (t_Size o_pos, const t_Char * pc_ins, t_Size o_len);
inline void         Insert (t_Size o_pos, const gct_String & co_ins);
inline void         Delete (t_Size o_pos);
inline void         Delete (t_Size o_pos, t_Size o_len);
inline void         DeleteRev (t_Size o_len);
void                Replace (t_Size o_pos, t_Size o_delLen, t_Char c_ins);
void                Replace (t_Size o_pos, t_Size o_delLen, t_Char c_ins, t_Size o_insLen);
void                Replace (t_Size o_pos, t_Size o_delLen, const t_Char * pc_ins);
void                Replace (t_Size o_pos, t_Size o_delLen, const t_Char * pc_ins, t_Size o_insLen);
void                Replace (t_Size o_pos, t_Size o_delLen, const gct_String & co_ins);
t_Size              ReplaceAll (const gct_String & co_search, const gct_String & co_replace);

int                 AssignF (const t_Char * pc_format, ...);
int                 AppendF (const t_Char * pc_format, ...);
int                 InsertF (t_Size o_pos, const t_Char * pc_format, ...);
int                 ReplaceF (t_Size o_pos, t_Size o_delLen, const t_Char * pc_format, ...);

inline bool         ToUpper ();
inline bool         ToLower ();
inline bool         ToUpper2 ();
inline bool         ToLower2 ();

inline bool         operator == (const t_Char * pc_comp) const;
inline bool         operator == (const gct_String & co_comp) const;
inline bool         operator != (const t_Char * pc_comp) const;
inline bool         operator != (const gct_String & co_comp) const;
inline bool         operator <  (const t_Char * pc_comp) const;
inline bool         operator <  (const gct_String & co_comp) const;
inline bool         operator <= (const t_Char * pc_comp) const;
inline bool         operator <= (const gct_String & co_comp) const;
```

```
    inline bool         operator >  (const t_Char * pc_comp) const;
    inline bool         operator >  (const gct_String & co_comp) const;
    inline bool         operator >= (const t_Char * pc_comp) const;
    inline bool         operator >= (const gct_String & co_comp) const;

    inline gct_String &  operator = (t_Char c_asgn);
    inline gct_String &  operator = (const t_Char * pc_asgn);
    inline gct_String &  operator = (const gct_String & co_asgn);
    inline gct_String &  operator += (t_Char c_app);
    inline gct_String &  operator += (const t_Char * pc_app);
    inline gct_String &  operator += (const gct_String & co_app);

    inline gct_String    operator + (t_Char c_app) const;
    inline gct_String    operator + (const t_Char * pc_app) const;
    inline gct_String    operator + (const gct_String & co_app) const;

    friend inline gct_String operator + (t_Char c_init, const gct_String & co_app);
    friend inline gct_String operator + (const t_Char * pc_init, const gct_String & co_app);
    template <class t_string>
      void Convert (const t_string & co_asgn);
    template <class t_string>
      bool MbConvert (const t_string & co_asgn);
    template <class t_asgnChar>
      bool MbConvert (const t_asgnChar * po_asgn);
    };
```

Kinds of String Parameters

1. Single character (`t_Char c`): The character is interpreted as a string of length 1.
2. Multiple characters (`t_Char c, t_Size o_len`): The parameter list is interpreted as a string of length `o_len` filled with the character `c`.
3. Null-terminated string (`const t_Char * pc`): The string is processed up to the null character.
4. String with length information (`const t_Char * pc, t_Size o_len`): The first `o_len` characters of the string are processed. The string must not contain null characters.
5. String object (`const gct_String & co`): The complete string `co` is processed.
6. Formatted string (`const t_Char * pc_format, ...`): The parameter list is interpreted like a `printf` parameter list. This kind of string parameters can't be used by overloaded methods.

Self-Assignment

Some frequently used string methods check for self-assignment. In some cases, a check for self-assignment is very expensive, e.g. while processing substrings. Please refer to the description of the respective methods.

Data Types

`typedef t_block::t_Size` **`t_Size`**`;`

The nested type `t_Size` is used for position and length values.

Constructors

`gct_String` `();`

Initializes an empty string object.

`gct_String` `(t_Char c_init);`

Initializes a string object of length 1 containing the character `c_init`.

`gct_String` `(t_Char c_init, t_Size o_len);`

Initializes a string object of length `o_len` containing `o_len` characters `c_init`.

gct_String (const t_Char * pc_init);

Initializes a string object containing a copy of the null-terminated string pc_init.

gct_String (const t_Char * pc_init, t_Size o_len);

Initializes a string object of length o_len containing a copy of the first o_len characters of pc_init.

gct_String (const gct_String & co_init);

Initializes a string object containing a copy of the string object co_init.

Access to Length and Contents

t_UInt **GetHash** () const;

Calculates the string's hash value.

bool **IsEmpty** () const;

Returns true if the string is empty.

t_Size **GetMaxLen** () const;

Returns the maximum length (without the terminating null character).

t_Size **GetLen** () const;

Returns the current length (without the terminating null character).

const t_Char * **GetStr** () const;
const t_Char * **operator ()** () const;

Returns a pointer to the first character. If the string is empty, the methods return a pointer to the terminating null character.

const t_Char * **GetStr** (t_Size o_pos) const;
const t_Char * **operator ()** (t_Size o_pos) const;

Returns a pointer to the character at position o_pos (o_pos <= GetLen ()). If o_pos equals GetLen (), the methods return a pointer to the terminating null character.

t_Char & **GetChar** (t_Size o_pos) const;
t_Char & **operator []** (t_Size o_pos) const;

Returns a reference to the character at position o_pos (o_pos < GetLen ()).

t_Char & **GetRevChar** (t_Size o_pos) const;

Returns a reference to the character at position GetLen () - 1 - o_pos (o_pos < GetLen ()). If o_pos equals zero, the method returns a reference to the last character.

gct_String **SubStr** (t_Size o_len) const;

Returns a string object containing a copy of the first o_len characters (o_len <= GetLen ()).

gct_String **RevSubStr** (t_Size o_len) const;

Returns a string object containing a copy of the last o_len characters (o_len <= GetLen ()).

gct_String **SubStr** (t_Size o_pos, t_Size o_len) const;
gct_String **operator ()** (t_Size o_pos, t_Size o_len) const;

Returns a string object containing a copy of the o_len characters beginning at position o_pos (o_pos + o_len <= GetLen ()).

Search for Characters and Strings

`t_Int First (t_Char c_search, t_Size o_pos = 0) const;`

If successful, it returns the position of the first occurrence of `c_search` starting at position `o_pos`. Otherwise it returns a negative value.

`t_Int First (const t_Char * pc_search, t_Size o_pos = 0) const;`

If successful, it returns the position of the first occurrence of `pc_search` starting at position `o_pos`. Otherwise it returns a negative value.

`t_Int First (const gct_String & co_search, t_Size o_pos = 0) const;`

If successful, it returns the position of the first occurrence of `co_search` starting at position `o_pos`. Otherwise it returns a negative value.

`t_Int Last (t_Char c_search, t_Size o_pos = 0) const;`

If successful, it returns the position of the last occurrence of `c_search` starting at position `o_pos`. Otherwise it returns a negative value.

`t_Int Last (const t_Char * pc_search, t_Size o_pos = 0) const;`

If successful, it returns the position of the last occurrence of `pc_search` starting at position `o_pos`. Otherwise it returns a negative value.

`t_Int Last (const gct_String & co_search, t_Size o_pos = 0) const;`

If successful, it returns the position of the last occurrence of `co_search` starting at position `o_pos`. Otherwise it returns a negative value.

Compare Substrings

The return value of the following methods is less than zero if `this < param`, equal to zero if `this == param`, and greater than zero if `this > param`. The characters are compared as unsigned values.

The following methods compare a substring beginning at position `o_pos` to the string specified by the arguments. In contrast to the full string comparison (see below), a substring comparison ends at the end of the shorter string.

`int CompSubStr (t_Size o_pos, t_Char c_comp) const;`

Compares the substring beginning at position `o_pos` to the character `c_comp`.

`int CompSubStr (t_Size o_pos, const t_Char * pc_comp) const;`

Compares the substring beginning at position `o_pos` to the null-terminated string `pc_comp`.

`int CompSubStr (t_Size o_pos, const t_Char * pc_comp, t_Size o_len) const;`

Compares the substring beginning at position `o_pos` to the first `o_len` characters of the string `pc_comp`.

`int CompSubStr (t_Size o_pos, const gct_String & co_comp) const;`

Compares the substring beginning at position `o_pos` to the string object `co_comp`.

Compare Strings

The return value of the following methods is less than zero if `this < param`, equal to zero if `this == param`, and greater than zero if `this > param`. The characters are compared as unsigned values.

The following methods compare this string to the string specified by the arguments. If the strings are equal when compared up to the shortest length, the longer string is considered greater than the shorter one.

`int CompTo (t_Char c_comp) const;`

Compares this string to the character `c_comp`.

`int CompTo (const t_Char * pc_comp) const;`

Compares this string to the null-terminated string `pc_comp`.

`int CompTo (const t_Char * pc_comp, t_Size o_len) const;`

Compares this string to the first `o_len` characters of the string `pc_comp`.

`int CompTo (const gct_String & co_comp) const;`

Compares this string to the string object `co_comp`.

Assignment

`void Clear ();`

Clears the string.

`void Assign (t_Char c_asgn);`

Replaces the contents with the character `c_asgn`.

`void Assign (t_Char c_asgn, t_Size o_len);`

Replaces the contents with `o_len` characters `c_asgn`.

`void Assign (const t_Char * pc_asgn);`

Replaces the contents with a copy of the null-terminated string `pc_asgn` (check for self-assignment).

`void Assign (const t_Char * pc_asgn, t_Size o_len);`

Replaces the contents with a copy of the first `o_len` characters of the string `pc_asgn` (no check for self-assignment).

`void Assign (const gct_String & co_asgn);`

Replaces the contents with a copy of the string object `co_asgn` (check for self-assignment).

Append

`void Append (t_Char c_app);`

Appends the character `c_app`.

`void Append (t_Char c_app, t_Size o_len);`

Appends `o_len` characters `c_app`.

`void Append (const t_Char * pc_app);`

Appends a copy of the null-terminated string `pc_app` (check for self-assignment).

`void Append (const t_Char * pc_app, t_Size o_len);`

Appends a copy of the first `o_len` characters of the string `pc_app` (no check for self-assignment).

`void Append (const gct_String & co_app);`

Appends a copy of the string object `co_app` (check for self-assignment).

Insert

`void **Insert** (t_Size o_pos, t_Char c_ins);`

Inserts the character `c_ins` at the position `o_pos` (`o_pos <= GetLen ()`).

`void Insert (t_Size o_pos, t_Char c_ins, t_Size o_len);`

Inserts `o_len` characters `c_ins` at the position `o_pos` (`o_pos <= GetLen ()`).

`void Insert (t_Size o_pos, const t_Char * pc_ins);`

Inserts a copy of the null-terminated string `pc_ins` at the position `o_pos` (`o_pos <= GetLen ()`).

`void Insert (t_Size o_pos, const t_Char * pc_ins, t_Size o_len);`

Inserts a copy of the first `o_len` characters of the string `pc_ins` at the position `o_pos` (`o_pos <= GetLen ()`).

`void Insert (t_Size o_pos, const gct_String & co_ins);`

Inserts a copy of the string object `co_ins` at the position `o_pos` (`o_pos <= GetLen ()`).

Delete

`void Delete (t_Size o_pos);`

Deletes the characters from the position `o_pos` to the end of the string (`o_pos <= GetLen ()`).

`void Delete (t_Size o_pos, t_Size o_len);`

Deletes `o_len` characters starting at the position `o_pos` (`o_pos + o_len <= GetLen ()`).

`void DeleteRev (t_Size o_len);`

Deletes the last `o_len` characters (`o_len <= GetLen ()`).

Replace

`void Replace (t_Size o_pos, t_Size o_delLen, t_Char c_ins);`

Replaces `o_delLen` characters starting at position `o_pos` with the character `c_ins` (`o_pos + o_delLen <= GetLen ()`).

`void Replace (t_Size o_pos, t_Size o_delLen, t_Char c_ins, t_Size o_insLen);`

Replaces `o_delLen` characters starting at position `o_pos` with `o_insLen` characters `c_ins` (`o_pos + o_delLen <= GetLen ()`).

`void Replace (t_Size o_pos, t_Size o_delLen, const t_Char * pc_ins);`

Replaces `o_delLen` characters starting at position `o_pos` with a copy of the null-terminated string `pc_ins` (`o_pos + o_delLen <= GetLen ()`).

`void Replace (t_Size o_pos, t_Size o_delLen, const t_Char * pc_ins, t_Size o_insLen);`

Replaces `o_delLen` characters starting at position `o_pos` with a copy of the first `o_insLen` characters of the string `pc_ins` (`o_pos + o_delLen <= GetLen ()`).

`void Replace (t_Size o_pos, t_Size o_delLen, const gct_String & co_ins);`

Replaces `o_delLen` characters starting at position `o_pos` with a copy of the string object `co_ins` (`o_pos + o_delLen <= GetLen ()`).

Replace All

`t_Size ReplaceAll (const gct_String & co_search, const gct_String & co_replace);`

Replaces all occurrences of `co_search` with a copy of `co_replace` and returns the number of replacements done. The implementation is optimized for minimal reallocations.

Formatted String Parameters

The following methods work like `Assign`, `Append`, `Insert` and `Replace`, but the parameter list is interpreted like a `printf` parameter list. All methods return the length of the resulting string parameter. On failure, a negative number is returned (see below 'Formatted Strings').

`int AssignF (const t_Char * pc_format, ...);`

Replaces the contents with the formatted string parameter.

`int AppendF (const t_Char * pc_format, ...);`

Appends the formatted string parameter.

`int InsertF (t_Size o_pos, const t_Char * pc_format, ...);`

Inserts the formatted string parameter at the position `o_pos` (`o_pos <= GetLen ()`).

`int ReplaceF (t_Size o_pos, t_Size o_delLen, const t_Char * pc_format, ...);`

Replaces `o_delLen` characters starting at position `o_pos` with the formatted string parameter (`o_pos + o_delLen <= GetLen ()`).

Upper/Lower Case

The following methods use global system interface functions (see above 'Character and String Conversion').

`bool ToUpper ();`

Converts the string to upper case (Windows-1252).

`bool ToLower ();`

Converts the string to lower case (Windows-1252).

`bool ToUpper2 ();`

Converts the string to upper case (partially UTF compatible).

`bool ToLower2 ();`

Converts the string to lower case (partially UTF compatible).

Comparison Operators

The following comparison functions are based on the method `CompTo` (see above).

```
bool operator == (const t_Char * pc_comp) const;
bool operator == (const gct_String & co_comp) const;
bool operator != (const t_Char * pc_comp) const;
bool operator != (const gct_String & co_comp) const;
bool operator <  (const t_Char * pc_comp) const;
bool operator <  (const gct_String & co_comp) const;
bool operator <= (const t_Char * pc_comp) const;
bool operator <= (const gct_String & co_comp) const;
bool operator >  (const t_Char * pc_comp) const;
bool operator >  (const gct_String & co_comp) const;
bool operator >= (const t_Char * pc_comp) const;
bool operator >= (const gct_String & co_comp) const;
```

Assignment Operators

`gct_String & operator = (t_Char c_asgn);`

Replaces the contents with the character `c_asgn`.

`gct_String & operator = (const t_Char * pc_asgn);`

Replaces the contents with a copy of the null-terminated string `pc_asgn` (check for self-assignment).

`gct_String & operator = (const gct_String & co_asgn);`

Replaces the contents with a copy of the string object `co_asgn` (check for self-assignment).

Append Operators

`gct_String & operator += (t_Char c_app);`

Appends the character `c_app`.

`gct_String & operator += (const t_Char * pc_app);`

Appends a copy of the null-terminated string `pc_app` (check for self-assignment).

`gct_String & operator += (const gct_String & co_app);`

Appends a copy of the string object `co_app` (check for self-assignment).

Concatenation Operators

The following concatenation operators return a temporary object containing the concatenation of the two operands. The two operands remain unchanged.

```
gct_String operator + (t_Char c_app) const;
gct_String operator + (const t_Char * pc_app) const;
gct_String operator + (const gct_String & co_app) const;
friend gct_String operator + (t_Char c_init, const gct_String & co_app);
friend gct_String operator + (const t_Char * pc_init, const gct_String & co_app);
```

Conversion

The following methods use global system interface functions to convert `char` and `wchar_t` strings (see above 'Character and String Conversion').

`template <class t_string> void Convert (const t_string & co_asgn);`

Replaces the contents with a copy of the string object `co_asgn` (no multibyte conversion).

`template <class t_string> bool MbConvert (const t_string & co_asgn);`

Replaces the contents with a copy of the string object `co_asgn` (multibyte conversion).

```
template <class t_asgnChar> bool MbConvert (const t_asgnChar * po_asgn);
```

Replaces the contents with a copy of the null-terminated string pc_asgn (multibyte conversion).

3.2.2 String Instances (tuning/xxx/[w]string.h)

Some template instances are predefined to easily use the string interface. The macros STRING_DCL(t_Block, StoreSpec) and WSTRING_DCL(t_Block, StoreSpec) generate for a wrapper class of a global store one string class.

The macro

```
STRING_DCL (gct_AnyBlock, ct_Any32)
```

expands to:

```
typedef gct_String <gct_CharBlock <gct_NullDataBlock
  <gct_AnyBlock <ct_Any32Store>, char>, char>, ct_Any32Store> ct_Any32String;
```

The macro

```
WSTRING_DCL (gct_AnyBlock, ct_Any32)
```

expands to:

```
typedef gct_String <gct_CharBlock <gct_NullDataBlock
  <gct_AnyBlock <ct_Any32Store>, wchar_t>, wchar_t>, ct_Any32Store> ct_Any32WString;
```

Every directory of a global store contains the files **'string.h'** and **'wstring.h'**.

The file 'tuning/std/[w]string.h' contains the following declaration:

```
typedef ... ct_Std_[W]String;
```

The file 'tuning/rnd/[w]string.h' contains the following declaration:

```
typedef ... ct_Rnd_[W]String;
```

The file 'tuning/chn/[w]string.h' contains the following declaration:

```
typedef ... ct_Chn_[W]String;
```

3.2.3 Polymorphic String Classes (tuning/[w]string.hpp)

Polymorphic string classes are derived from the abstract base class ct_Object. They can be managed by polymorphic collections and used by other polymorphic API's. The two string classes ct_String and ct_WString are predefined, other polymorphic string classes can be defined if necessary. The macro OBJ_STRING_DCL(StoreSpec) generates a string class using a predefined template instance.

The macro

```
OBJ_STRING_DCL(ct_Chn_Obj)
```

expands to:

```
class ct_Chn_ObjectString: public ct_Chn_ObjString
  {
public:
  inline ct_Chn_ObjectString ();
  inline ct_Chn_ObjectString (t_Char c_init);
```

```
  inline ct_Chn_ObjectString (t_Char c_init, t_Size o_len);
  inline ct_Chn_ObjectString (const t_Char * pc_init);
  inline ct_Chn_ObjectString (const t_Char * pc_init, t_Size o_len);
  inline ct_Chn_ObjectString (const ct_Chn_ObjString & co_init);
  inline ct_Chn_ObjectString (const ct_Chn_ObjectString & co_init);
  TL_CLASSID (ct_Chn_ObjectString)
  virtual bool operator < (const ct_Object & co_comp) const;
  virtual t_UInt GetHash () const;
  inline ct_Chn_ObjectString & operator = (t_Char c_asgn);
  inline ct_Chn_ObjectString & operator = (const t_Char * pc_asgn);
  inline ct_Chn_ObjectString & operator = (const ct_Chn_ObjectString & co_asgn);
  };
...
```

Additional Methods

`bool operator < (const ct_Object & co_comp) const;`

This comparison operator is used by sorted array collections.

The file 'tuning/string.hpp' contains the following declaration:

```
OBJ_STRING_DCL(ct_Chn_Obj)
typedef ct_Chn_ObjectString ct_String;
```

The file 'tuning/wstring.hpp' contains the following declaration:

```
OBJ_STRING_DCL(ct_Chn_WObj)
typedef ct_Chn_WObjectString ct_WString;
```

3.2.4 Filename (tuning/filename.hpp)

The class `ct_FileName` provides several methods to manipulate filenames. A filename is stored as a null-terminated string. Filename components are determined by offset values stored in the filename object.

A filename consists of four components: `Drive`, `Path`, `Name` and `Ext`. The combination of `Drive` and `Path` is called `DrivePath`, the combination of `Name` and `Ext` is called `NameExt`. The path component always includes a terminating [back]slash. A `Path` without the terminating [back]slash is called `PurePath`, a `DrivePath` without the terminating [back]slash is called `PureDrivePath`.

The class `ct_FileName` supports the Universal Naming Convention (UNC). The `Drive` component can contain a drive specification (e.g. "`C:`") or a network path (e.g. "`\\server\\share`"). The methods `HasDrive` and `HasUNC` can be used to distinguish between these two cases.

The MS Windows implementation automatically replaces slash characters with backslash characters (Linux impl. vice versa). The terminating [back]slash of a path component is appended if necessary. The extension component does not include a period.

There are two different assignment methods. The method 'Assign as Name' tries to locate the name and extension components at the end of the string. The method 'Assign as Path' interprets the whole string as a drive-path component.

Base Classes

`ct_Object` (see above 'Abstract Object')
  `ct_String` (see above 'Polymorphic String')

Class Declaration

```
class ct_FileName: public ct_String
  {
```

```
                      ct_FileName ();
                      ct_FileName (const char * pc_init);
ct_FileName &         operator = (const char * pc_asgn);
ct_FileName &         operator = (const ct_FileName & co_asgn);

inline void           AssignAsPath (const char * pc_path);
void                  AssignAsPath (const char * pc_path, t_Size u_len);
inline void           AssignAsPath (const ct_String & co_path);
inline void           AssignAsName (const char * pc_name);
void                  AssignAsName (const char * pc_name, t_Size u_len);
inline void           AssignAsName (const ct_String & co_name);

bool                  HasDriveOrUNC () const;
bool                  HasDrive () const;
bool                  HasUNC () const;
bool                  HasPath () const;
bool                  HasName () const;
bool                  HasExt () const;
bool                  HasDot () const;
bool                  HasWildCards () const;

inline t_Size         GetDriveLen () const;
inline t_Size         GetPathLen () const;
inline t_Size         GetPurePathLen () const;
inline t_Size         GetDrivePathLen () const;
inline t_Size         GetPureDrivePathLen () const;
inline t_Size         GetNameLen () const;
inline t_Size         GetExtLen () const;
inline t_Size         GetNameExtLen () const;
inline t_Size         GetDotLen () const;
inline t_Size         GetAllLen () const;

inline t_Size         GetDriveOffs () const;
inline t_Size         GetPathOffs () const;
inline t_Size         GetNameOffs () const;
inline t_Size         GetExtOffs () const;

inline const char *   GetDriveStr () const;
inline const char *   GetPathStr () const;
inline const char *   GetNameStr () const;
inline const char *   GetExtStr () const;
inline const char *   GetAllStr () const;

inline ct_String      GetDrive () const;
inline ct_String      GetPath () const;
inline ct_String      GetPurePath () const;
inline ct_String      GetDrivePath () const;
inline ct_String      GetPureDrivePath () const;
inline ct_String      GetName () const;
inline ct_String      GetExt () const;
inline ct_String      GetNameExt () const;

inline void           SetDrive (const char * pc);
void                  SetDrive (const char * pc, t_Size u_len);
inline void           SetDrive (const ct_String & co);
inline void           SetPath (const char * pc);
void                  SetPath (const char * pc, t_Size u_len);
inline void           SetPath (const ct_String & co);
inline void           SetDrivePath (const char * pc);
void                  SetDrivePath (const char * pc, t_Size u_len);
inline void           SetDrivePath (const ct_String & co);
inline void           SetName (const char * pc);
void                  SetName (const char * pc, t_Size u_len);
inline void           SetName (const ct_String & co);
inline void           SetExt (const char * pc);
void                  SetExt (const char * pc, t_Size u_len);
```

```
inline void          SetExt (const ct_String & co);
inline void          SetNameExt (const char * pc);
void                 SetNameExt (const char * pc, t_Size u_len);
inline void          SetNameExt (const ct_String & co);

inline void          CopyDriveFrom (const ct_FileName * pco_copy);
inline void          CopyPathFrom (const ct_FileName * pco_copy);
inline void          CopyDrivePathFrom (const ct_FileName * pco_copy);
inline void          CopyNameFrom (const ct_FileName * pco_copy);
inline void          CopyExtFrom (const ct_FileName * pco_copy);
inline void          CopyNameExtFrom (const ct_FileName * pco_copy);

inline void          InsertPath (const char * pc_path);
void                 InsertPath (const char * pc_path, t_Size u_len);
inline void          InsertPath (const ct_String & co_path);
inline void          InsertDrivePath (const char * pc_path);
void                 InsertDrivePath (const char * pc_path, t_Size u_len);
inline void          InsertDrivePath (const ct_String & co_path);
inline void          AppendPath (const char * pc_path);
void                 AppendPath (const char * pc_path, t_Size u_len);
inline void          AppendPath (const ct_String & co_path);
void                 CompressPath ();
bool                 IsAbs () const;
bool                 IsRel () const;
void                 ToAbs (const char * pc_currDrivePath, bool b_withDrive = true);
void                 ToRel (const char * pc_currDrivePath, bool b_withDrive = false);
};
```

Methods

ct_FileName ();

Initializes an empty filename object.

ct_FileName (const char * pc_init);

Initializes a filename object by calling the method AssignAsName.

ct_FileName & **operator =** (const char * pc_asgn);

Calls the method AssignAsName.

ct_FileName & **operator =** (const ct_FileName & co_asgn);

Replaces the contents with a copy of the filename object co_asgn.

void **AssignAsPath** (const char * pc_path);
void **AssignAsPath** (const char * pc_path, t_Size u_len);
void **AssignAsPath** (const ct_String & co_path);

Replace the contents with a copy of the arguments. These methods interpret the whole string as a drive-path component.

void **AssignAsName** (const char * pc_name);
void **AssignAsName** (const char * pc_name, t_Size u_len);
void **AssignAsName** (const ct_String & co_name);

Replace the contents with a copy of the arguments. These methods try to locate the name and extension components at the end of the string.

```
bool HasDriveOrUNC () const;
bool HasDrive () const;
bool HasUNC () const;
bool HasPath () const;
bool HasName () const;
bool HasExt () const;
```

These methods return `true` if a specific component exists.

```
bool HasDot () const;
```

Returns `true` if there is a period (dot) between name and extension.

```
bool HasWildCards () const;
```

Returns `true` if name or extension contain wildcard characters ('*' or '?').

```
t_Size GetDriveLen () const;
t_Size GetPathLen () const;
t_Size GetPurePathLen () const;
t_Size GetDrivePathLen () const;
t_Size GetPureDrivePathLen () const;
t_Size GetNameLen () const;
t_Size GetExtLen () const;
t_Size GetNameExtLen () const;
```

These methods return the length of a specific component.

```
t_Size GetDotLen () const;
```

Returns 1 if there is a period (dot) between name and extension, otherwise zero is returned.

```
t_Size GetAllLen () const;
```

Returns the length of the whole filename.

```
t_Size GetDriveOffs () const;
t_Size GetPathOffs () const;
t_Size GetNameOffs () const;
t_Size GetExtOffs () const;
```

These methods return the position (offset) of a specific component.

```
const char * GetDriveStr () const;
const char * GetPathStr () const;
const char * GetNameStr () const;
const char * GetExtStr () const;
const char * GetAllStr () const;
```

These methods return a pointer to the beginning of a specific component.

```
ct_String GetDrive () const;
ct_String GetPath () const;
ct_String GetPurePath () const;
ct_String GetDrivePath () const;
ct_String GetPureDrivePath () const;
ct_String GetName () const;
ct_String GetExt () const;
ct_String GetNameExt () const;
```

These methods return a specific component as a string object.

```
void SetDrive (const char * pc);
void SetDrive (const char * pc, t_Size u_len);
void SetDrive (const ct_String & co);
void SetPath (const char * pc);
void SetPath (const char * pc, t_Size u_len);
void SetPath (const ct_String & co);
void SetDrivePath (const char * pc);
void SetDrivePath (const char * pc, t_Size u_len);
void SetDrivePath (const ct_String & co);
void SetName (const char * pc);
void SetName (const char * pc, t_Size u_len);
void SetName (const ct_String & co);
void SetExt (const char * pc);
void SetExt (const char * pc, t_Size u_len);
void SetExt (const ct_String & co);
void SetNameExt (const char * pc);
void SetNameExt (const char * pc, t_Size u_len);
void SetNameExt (const ct_String & co);
```

Replace the contents of a specific component with a copy of the arguments.

```
void CopyDriveFrom (const ct_FileName * pco_copy);
void CopyPathFrom (const ct_FileName * pco_copy);
void CopyDrivePathFrom (const ct_FileName * pco_copy);
void CopyNameFrom (const ct_FileName * pco_copy);
void CopyExtFrom (const ct_FileName * pco_copy);
void CopyNameExtFrom (const ct_FileName * pco_copy);
```

Copy the contents of a specific component from another filename object.

```
void InsertPath (const char * pc_path);
void InsertPath (const char * pc_path, t_Size u_len);
void InsertPath (const ct_String & co_path);
```

Insert a copy of the arguments at the beginning of the path component.

```
void InsertDrivePath (const char * pc_path);
void InsertDrivePath (const char * pc_path, t_Size u_len);
void InsertDrivePath (const ct_String & co_path);
```

Insert a copy of the arguments at the beginning of the path component and replace the drive component.

```
void AppendPath (const char * pc_path);
void AppendPath (const char * pc_path, t_Size u_len);
void AppendPath (const ct_String & co_path);
```

Append a copy of the arguments at the end of the path component.

```
void CompressPath ();
```

Compresses the path component, i.e. deletes `".\"` and `"path\..\"` patterns.

```
bool IsAbs () const;
```

Returns `true` if the path component is an absolute path (beginning with a [back]slash).

```
bool IsRel () const;
```

Returns `true` if the path component is a relative path (not beginning with a [back]slash).

```
void ToAbs (const char * pc_currDrivePath, bool b_withDrive = true);
```

Converts the path component, which is relative to the directory `pc_currDrivePath`, to an absolute path. If `b_withDrive` equals `true`, the drive component is copied from `pc_currDrivePath`, otherwise the drive component is cleared.

void **ToRel** (const char * pc_currDrivePath, bool b_withDrive = false);

Converts the path component, which is an absolute path, to a path relative to the directory pc_currDrivePath. If b_withDrive equals true, the drive component is copied from pc_currDrivePath, otherwise the drive component is cleared.

3.2.5 Formatted Strings (tuning/printf.hpp)

The char and wchar_t versions of tl_VSprintf interpret the parameter list like a printf parameter list. The destination buffer is dynamically allocated. It is recommended to use the gct_String methods AssignF, AppendF, InsertF and ReplaceF instead of tl_VSprintf. See also the sample program 'tstring'.

Functions

int **tl_VSprintf** (char * * ppc_buffer, const char * pc_format, va_list o_argList);
int **tl_VSprintf** (wchar_t * * ppc_buffer, const wchar_t * pc_format, va_list o_argList);

Formats the string pc_format with the parameters o_argList and writes the resulting string to a destination buffer which is allocated by malloc. On success, the length of the resulting string is returned (without the terminating null character), and the buffer * ppc_buffer must be released by free. On failure, a negative number is returned, and the pointer * ppc_buffer can be ignored.

3.2.6 String Sort Algorithm (tuning/stringsort.hpp)

This section describes an optimized string sort algorithm. Strings consist of characters, and characters have a value range from 0 to 255. To sort values in this range, no special sort algorithm is required. The values can be entered into an array of size 256. Afterwards the array can be iterated, and the values will appear in sorted order. This method can be applied to the first, the second, the third etc. character of a set of strings.

The sort order can be changed by a 'sort page' of size 256. The first entry of a sort page must be equal to zero. The private method GetDefaultSortPage returns a sort page containing consecutive numbers.

To sort N null-terminated strings, an array of N pointers to strings (const char * * ppc_strings) must be prepared by the caller of the algorithm. The results are written to an array of N t_Int values (t_Int * pi_sortedIndex) allocated by the caller. At the end of the calculation, this array will contain indices into the string array in sorted order.

The computing time depends on the maximum number of leading equal characters. The sort algorithm requires the following memory:

1. The input array char * apc [n] and the output array t_Int ai [n].
2. The array t_Int ai_temp [n] to store temporary chains.
3. x * 256 * sizeof (t_Int) bytes to store temporary order data. x is the maximum number of leading equal characters.

Class Declaration

```
class ct_StringSort
  {
public:
  bool        Sort (const char * * ppc_strings, t_Int * pi_sortedIndex, t_Int i_numOfStrings,
                const char * pc_sortPage = GetDefaultSortPage ());
  };
```

Methods

bool **Sort** (const char * * ppc_strings, t_Int * pi_sortedIndex, t_Int i_numOfStrings, const char * pc_sortPage = GetDefaultSortPage ());

Sorts the input data ppc_strings and stores the result in pi_sortedIndex. Temporary data are allocated and released automatically.

3.2.7 Number Sort Algorithm (tuning/stringsort.hpp)

The string sort algorithm (see above) can be modified to sort unsigned integer values. A t_UInt32 value can be interpreted as a sequence of 4 unsigned characters. The implementation of the number sort algorithm supports little-endian hardware.

Class Declaration

```
class ct_UInt32Sort
  {
public:
  bool                 Sort (const t_UInt32 * pu_ints, t_Int * pi_sortedIndex,
                         t_Int i_numOfInts);
  };
```

Methods

bool **Sort** (const t_UInt32 * pu_ints, t_Int * pi_sortedIndex, t_Int i_numOfInts);

Sorts the input data pu_ints and stores the result in pi_sortedIndex. Temporary data are allocated and released automatically.

3.3 Files and Directories

3.3.1 Files (tuning/file.hpp)

Within the **Spirick Tuning** library all file and directory paths are interpreted as UTF-8 strings. The Linux implementation passes the path names unchanged to the corresponding system functions. The MS Windows implementation converts the path names temporarily to UTF-16.

The class ct_File provides an object oriented interface for the global file functions (see above 'File I/O'). The methods TryOpen, Open, Create, Load, Save, Exists, Move, Copy and Delete must not be called while the file is open.

Base Classes

```
ct_Object       (see above 'Abstract Object')
  ct_String     (see above 'Polymorphic String')
    ct_FileName (see above 'Filename')
```

Class Declaration

```
class ct_File: public ct_FileName
  {
public:
                       ct_File ();
                       ct_File (const char * pc_init);
```

```
                        ct_File (const ct_FileName & co_init);
                        ~ct_File ();
ct_File &               operator = (const char * pc_asgn);
ct_File &               operator = (const ct_FileName & co_asgn);

bool                    TryOpen (bool b_readOnly = true, bool b_sequential = true,
                          t_UInt32 u_milliSec = 0);
bool                    Open (bool b_readOnly = true, bool b_sequential = true);
bool                    Create (bool b_createNew = false);
bool                    Close ();

bool                    Load (ct_String * pco_str);
bool                    Save (const ct_String * pco_str);

bool                    Exists ();
bool                    Move (const char * pc_new);
bool                    Copy (const char * pc_new, bool b_overwrite = true);
bool                    Delete ();

bool                    QuerySize (t_FileSize & o_size) const;
bool                    QueryPos (t_FileSize & o_pos) const;
bool                    EndOfFile (bool & b_eof) const;
bool                    SeekAbs (t_FileSize o_pos) const;
bool                    SeekRel (t_FileSize o_pos) const;
bool                    Truncate (t_FileSize o_size) const;
bool                    Read (void * pv_dst, t_FileSize o_len) const;
bool                    Write (const void * pv_src, t_FileSize o_len) const;
};
```

Methods

ct_File ();

Initializes an empty file object.

ct_File (const char * pc_init);

Initializes a file object by calling the method ct_FileName::AssignAsName.

ct_File (const ct_FileName & co_init);

Initializes a file object by calling the copy constructor of ct_FileName.

~ct_File ();

The destructor closes the file object.

ct_File & **operator =** (const char * pc_asgn);

Calls ct_FileName::AssignAsName (pc_asgn).

ct_File & **operator =** (const ct_FileName & co_asgn);

Assigns a new filename.

bool **TryOpen** (bool b_readOnly = true, bool b_sequential = true, t_UInt32 u_milliSec = 0);

Tries to open an existing file. The method will wait for at most u_milliSec milliseconds. The parameter b_readOnly determines the access mode. The parameter b_sequential is a hint to optimize file caching (sequential or random access).

bool **Open** (bool b_readOnly = true, bool b_sequential = true);

Opens an existing file. The parameter b_readOnly determines the access mode. The parameter b_sequential is a hint to optimize file caching (sequential or random access).

bool **Create** (bool b_createNew = false);

Creates a new file and opens it for read/write access. Returns false if b_createNew equals true and the specified file already exists. Otherwise the function overwrites the existing file.

bool **Close** ();

Closes an open file.

bool **Load** (ct_String * pco_str);

Loads the entire contents of the file into the string object pco_str (open, read, close). The file must not contain null characters.

bool **Save** (const ct_String * pco_str);

Saves the entire contents of the string object pco_str into the file (open, write, close).

bool **Exists** ();

Returns true if the file exists.

bool **Move** (const char * pc_new);

Moves (renames) the file either in the same directory or across directories. On success the own filename (base class ct_FileName) is changed as well.

bool **Copy** (const char * pc_new, bool b_overwrite = true);

Copies the existing file to a new file. Returns false if b_overwrite equals false and the specified file already exists.

bool **Delete** ();

Deletes the existing file.

bool **QuerySize** (t_FileSize & o_size) const;

Retrieves the size of the open file and stores the result in o_size.

bool **QueryPos** (t_FileSize & o_pos) const;

Retrieves the file pointer of the open file and stores the result in o_pos.

bool **EndOfFile** (bool & b_eof) const;

Sets b_eof to true if the file pointer is located at the end of the file.

bool **SeekAbs** (t_FileSize o_pos) const;

Moves the file pointer of the open file to the absolute position o_pos (an offset from the beginning of the file).

bool **SeekRel** (t_FileSize o_pos) const;

Moves the file pointer of the open file to the relative position o_pos (relative to the current position).

bool **Truncate** (t_FileSize o_size);

Sets the size for the open file to o_size.

bool **Read** (void * pv_dst, t_FileSize o_len) const;

Reads o_len bytes from the open file to the buffer pv_dst and moves the file pointer.

bool **Write** (const void * pv_src, t_FileSize o_len) const;

Writes o_len bytes from the buffer pv_src to the open file and moves the file pointer.

3.3.2 Directories (tuning/dir.hpp)

Within the **Spirick Tuning** library all file and directory paths are interpreted as UTF-8 strings. The Linux implementation passes the path names unchanged to the corresponding system functions. The MS Windows implementation converts the path names temporarily to UTF-16.

The class `ct_Directory` provides an object oriented interface for the global directory functions (see above 'sys/cdir.hpp'). This class uses the drive and path components of the base class `ct_FileName` (`PureDrivePath`), the name and extension components of the filename are ignored.

Base Classes

```
ct_Object       (see above 'Abstract Object')
  ct_String     (see above 'Polymorphic String')
    ct_FileName (see above 'Filename')
```

Class Declaration

```
class ct_Directory: public ct_FileName
  {
public:
                      ct_Directory ();
                      ct_Directory (const char * pc_init);
                      ct_Directory (const ct_FileName & co_init);
  ct_Directory &      operator = (const char * pc_asgn);
  ct_Directory &      operator = (const ct_FileName & co_asgn);

  bool                QueryCurrentDrive ();
  bool                QueryCurrentDirectory ();
  bool                QueryCurrentDriveDirectory ();

  bool                Create ();
  bool                Exists ();
  bool                Move (const char * pc_new);
  bool                Delete ();
  };
```

Methods

`ct_Directory ();`

Initializes an empty directory object.

`ct_Directory (const char * pc_init);`

Initializes a directory object by calling the method `ct_FileName::AssignAsPath`.

`ct_Directory (const ct_FileName & co_init);`

Initializes a directory object by calling the copy constructor of `ct_FileName`.

`ct_Directory & operator = (const char * pc_asgn);`

Calls `ct_FileName::AssignAsPath (pc_asgn)`.

`ct_Directory & operator = (const ct_FileName & co_asgn);`

Assigns a new filename.

`bool QueryCurrentDrive ();`

Retrieves the current drive and stores the result in the drive component of the filename.

`bool QueryCurrentDirectory ();`

Retrieves the current directory of the drive specified by the drive component and stores the result in the path component of the filename.

`bool QueryCurrentDriveDirectory ();`

Retrieves the current directory and stores the result in the drive-path component of the filename.

`bool Create ();`

Creates a directory.

`bool Exists ();`

Returns `true` if the directory exists.

`bool Move (const char * pc_new);`

Moves (renames) the directory either in the same directory or across directories. On success the own filename (base class `ct_FileName`) is changed as well.

`bool Delete ();`

Deletes an empty directory.

3.3.3 Directory Scan (tuning/dirscan.hpp)

Within the **Spirick Tuning** library all file and directory paths are interpreted as UTF-8 strings. The Linux implementation passes the path names unchanged to the corresponding system functions. The MS Windows implementation converts the path names temporarily to UTF-16.

The class `ct_DirScan` is derived from `ct_Directory`. The drive and path components of the filename determine the directory to scan. The name and extension components are used for input and output data. *Before* scanning a directory, these components contain the search pattern. *While* scanning a directory, these components contain the name and extension of the current directory entry.

Note that changing the contents of a directory while scanning it can lead to unpredictable results. It is recommended to cache the results of a directory scan in a data stucture before changing the contents of the directory.

The class `ct_DirScan` can also be used to retrieve information about a single file or directory. If the search pattern does not contain any wildcard characters (`'*'` or `'?'`), multiple information about a directory entry are retrieved by a single function call. The `FindOnce` methods consist of three steps: abort an active scan, assign a new search pattern and start a new scan.

Base Classes

`ct_Object`	(see above 'Abstract Object')
`ct_String`	(see above 'Polymorphic String')
`ct_FileName`	(see above 'Filename')
`ct_Directory`	(see above 'Directory')

Data Types, Constants

```
typedef unsigned t_FileAttributes;

const t_FileAttributes co_AttrArchive   = 0x01;
const t_FileAttributes co_AttrDirectory = 0x02;
const t_FileAttributes co_AttrHidden    = 0x04;
const t_FileAttributes co_AttrReadOnly  = 0x08;
```

```
const t_FileAttributes co_AttrSystem    = 0x10;
```

Values of the integer type t_FileAttributes can combine multiple attribute flags via an OR operation.

Class Declaration

```
class ct_DirScan: public ct_Directory
  {
public:
                       ct_DirScan ();
                       ct_DirScan (const char * pc_init);
                       ct_DirScan (const ct_FileName & co_init);
                       ~ct_DirScan ();
  ct_DirScan &         operator = (const char * pc_asgn);
  ct_DirScan &         operator = (const ct_FileName & co_asgn);

  bool                 FindOnce ();
  bool                 FindOnce (const char * pc_find);
  bool                 FindOnce (const ct_FileName & co_find);
  bool                 FindOncePath ();
  bool                 FindOncePath (const ct_FileName & co_find);

  bool                 FindFirst ();
  bool                 FindFirstFile ();
  bool                 FindFirstDirectory ();
  bool                 FindNext ();
  bool                 FindNextFile ();
  bool                 FindNextDirectory ();
  void                 AbortFind ();
  bool                 Found ();

  t_MicroTime          GetCreationTime () const;
  t_MicroTime          GetLastAccessTime () const;
  t_MicroTime          GetLastWriteTime () const;
  t_FileSize           GetSize () const;
  t_FileAttributes     GetAttributes () const;
  bool                 IsArchive () const;
  bool                 IsDirectory () const;
  bool                 IsHidden () const;
  bool                 IsReadOnly () const;
  bool                 IsSystem () const;
  };
```

Methods

ct_DirScan ();

Initializes an empty dirscan object.

ct_DirScan (const char * pc_init);

Initializes a dirscan object by calling the method ct_FileName::AssignAsName.

ct_DirScan (const ct_FileName & co_init);

Initializes a dirscan object by calling the copy constructor of ct_FileName.

~ct_DirScan ();

Releases all temporary data.

ct_DirScan & **operator =** (const char * pc_asgn);

Calls ct_FileName::AssignAsName (pc_asgn).

`ct_DirScan & operator = (const ct_FileName & co_asgn);`

Assigns a new filename.

`bool FindOnce ();`

Aborts an active scan and starts a new scan using the current filename.

`bool FindOnce (const char * pc_find);`

Aborts an active scan, calls `ct_FileName::AssignAsName (pc_find)` and starts a new scan.

`bool FindOnce (const ct_FileName & co_find);`

Aborts an active scan, calls `ct_FileName::AssignAsName (co_find)` and starts a new scan.

`bool FindOncePath ();`

Aborts an active scan, calls `ct_FileName::AssignAsName (GetPureDrivePath ())` and starts a new scan, i.e. the method retrieves information about the drive-path component.

`bool FindOncePath (const ct_FileName & co_find);`

Aborts an active scan, calls `ct_FileName::AssignAsName (co_find. GetPureDrivePath ())` and starts a new scan, i.e. the method retrieves information about the drive-path component of `co_find`.

`bool FindFirst ();`

Starts a new scan (files and directories) using the current filename and retrieves information about the first directory entry.

`bool FindFirstFile ();`

Starts a new scan (files only) using the current filename and retrieves information about the first directory entry.

`bool FindFirstDirectory ();`

Starts a new scan (directories only) using the current filename and retrieves information about the first directory entry.

`bool FindNext ();`

Iterates to the next directory entry (files and directories) and retrieves information about it.

`bool FindNextFile ();`

Iterates to the next directory entry (files only) and retrieves information about it.

`bool FindNextDirectory ();`

Iterates to the next directory entry (directories only) and retrieves information about it.

`void AbortFind ();`

Aborts an active scan and releases all temporary data.

`bool Found ();`

Returns `true` if the previous call of `FindFirst` or `FindNext` has returned `true`.

`t_MicroTime GetCreationTime () const;`

Returns the creation time of the current directory entry in UTC (see above 'Time and Date').

`t_MicroTime GetLastAccessTime () const;`

Returns the last access time of the current directory entry in UTC (see above 'Time and Date').

`t_MicroTime` **`GetLastWriteTime`** `() const;`

Returns the last write time of the current directory entry in UTC (see above 'Time and Date').

`t_FileSize` **`GetSize`** `() const;`

Returns the size of the current directory entry.

`t_FileAttributes` **`GetAttributes`** `() const;`

Returns the attributes of the current directory entry.

```
bool IsArchive () const;
bool IsDirectory () const;
bool IsHidden () const;
bool IsReadOnly () const;
bool IsSystem () const;
```

Returns `true` if a specific flag is set.

Search Patterns

The class `ct_DirScan` is derived from `ct_Directory`. The drive and path components of the filename determine the directory to scan. The method `ct_Directory::Exists` can be used to check if the directory exists.

```
ct_DirScan co_dirScan;
co_dirScan. SetDrivePath ("c:\\spirick\\tuning");

if (co_dirScan. Exists ())
  // ...
```

The name and extension components are used for input and output data. *Before* scanning a directory, these components contain the search pattern.

```
co_dirScan. SetNameExt ("*");
```

The search pattern "*" starts an unfiltered directory scan.

```
co_dirScan. SetNameExt ("*.?pp");
```

MS Windows only: The search pattern can be a combination of literal and wildcard characters (`'*'` or `'?'`).

```
co_dirScan. SetNameExt ("dirscan.hpp");
```

If the search pattern is a unique name of a file or directory, multiple information about the directory entry are retrieved by a single function call.

Sample Code

The following sample code demonstrates an unfiltered directory scan.

```
ct_DirScan co_dirScan ("c:\\spirick\\tuning\\*");

for (co_dirScan. FindFirst ();
     co_dirScan. Found ();
     co_dirScan. FindNext ())
  {
  // ...
  }
```

Scan files only:

```
ct_DirScan co_dirScan ("c:\\spirick\\tuning\\*");

for (co_dirScan. FindFirstFile ();
     co_dirScan. Found ();
     co_dirScan. FindNextFile ())
  {
  // ...
  }
```

Scan directories only:

```
ct_DirScan co_dirScan ("c:\\spirick\\tuning\\*");

for (co_dirScan. FindFirstDirectory ();
     co_dirScan. Found ();
     co_dirScan. FindNextDirectory ())
  {
  // ...
  }
```

3.4 Additional Utilities

3.4.1 Time and Date (tuning/timedate.hpp)

The class `ct_TimeDate` provides an object oriented interface for the global time and date functions (see above 'sys/ctimedate.hpp'). Time values are expressed in microseconds since 1/1/1970. The current time can be queried in UTC and local time.

Class Declaration

```
class ct_TimeDate
  {
public:
                      ct_TimeDate ();
                      ct_TimeDate (t_MicroTime i_time);

  void                Clear ();
  t_MicroTime         GetTime () const;
  void                SetTime (t_MicroTime i_time);

  void                QueryUTCTime ();
  void                QueryLocalTime ();

  inline unsigned     GetYear () const;
  inline unsigned     GetMonth () const;
  inline unsigned     GetDay () const;
  inline unsigned     GetDayOfWeek () const;
  inline unsigned     GetHour () const;
  inline unsigned     GetMinute () const;
  inline unsigned     GetSecond () const;
  inline unsigned     GetMicroSecond () const;

  inline void         SetYear (unsigned u);
  inline void         SetMonth (unsigned u);
  inline void         SetDay (unsigned u);
  inline void         SetDayOfWeek (unsigned u);
  inline void         SetHour (unsigned u);
  inline void         SetMinute (unsigned u);
  inline void         SetSecond (unsigned u);
```

```
    inline void          SetMicroSecond (unsigned u);

    inline bool          operator == (const ct_TimeDate & co_td) const;
    inline bool          operator != (const ct_TimeDate & co_td) const;
    inline bool          operator <  (const ct_TimeDate & co_td) const;
    inline bool          operator <= (const ct_TimeDate & co_td) const;
    inline bool          operator >  (const ct_TimeDate & co_td) const;
    inline bool          operator >= (const ct_TimeDate & co_td) const;
    };
```

Methods

`ct_TimeDate ();`

Initializes an empty time-date object.

`ct_TimeDate (t_MicroTime i_time);`

Converts a microsecond value to time-date components.

`void Clear ();`

Clears the time-date object.

`t_MicroTime GetTime () const;`

Converts time-date components to a microsecond value.

`void SetTime (t_MicroTime i_time);`

Converts a microsecond value to time-date components.

`void QueryUTCTime ();`

Retrieves the current time, as reported by the system clock, in UTC.

`void QueryLocalTime ();`

Retrieves the current time, as reported by the system clock, in the local time zone.

```
unsigned GetYear () const;
unsigned GetMonth () const;
unsigned GetDay () const;
unsigned GetDayOfWeek () const;
unsigned GetHour () const;
unsigned GetMinute () const;
unsigned GetSecond () const;
unsigned GetMicroSecond () const;
```

These methods return a specific component as an unsigned integer value.

```
void SetYear (unsigned u);
void SetMonth (unsigned u);
void SetDay (unsigned u);
void SetDayOfWeek (unsigned u);
void SetHour (unsigned u);
void SetMinute (unsigned u);
void SetSecond (unsigned u);
void SetMicroSecond (unsigned u);
```

These methods set a specific component to an unsigned integer value.

```
bool operator == (const ct_TimeDate & co_td) const;
bool operator != (const ct_TimeDate & co_td) const;
bool operator <  (const ct_TimeDate & co_td) const;
bool operator <= (const ct_TimeDate & co_td) const;
bool operator >  (const ct_TimeDate & co_td) const;
bool operator >= (const ct_TimeDate & co_td) const;
```

These methods compare two time-date objects.

3.4.2 MD5 Sum (tuning/md5.hpp)

The class `ct_MD5` can be used for a single MD5 sum calculation. The source data can be located in a single memory block, or they can consist of several parts. The results of the calculation can be retrieved in a textual and a binary format.

Class Declaration

```
typedef t_UInt8 t_MD5Result [16];

class ct_MD5
  {
public:
                      ct_MD5 ();
                      ct_MD5 (const t_MD5Result & ac_init);
                      ct_MD5 (const void * pv_data, t_UInt u_len);

  void                Update (const void * pv_data, t_UInt u_len);
  void                Finalize ();
  const t_MD5Result & GetResult () const;
  const char *        GetResultStr ();
  bool                operator == (const ct_MD5 & co_comp) const;
  };
```

Methods

`ct_MD5 ();`

Initializes an empty MD5 object.

`ct_MD5 (const t_MD5Result & ac_init);`

Copies the MD5 results from another MD5 object.

`ct_MD5 (const void * pv_data, t_UInt u_len);`

Initializes a MD5 object and calls the methods `Update` and `Finalize`.

`void Update (const void * pv_data, t_UInt u_len);`

Processes a single part of the source data. Location and length of the data block are determined by `pv_data` and `u_len`.

`void Finalize ();`

Stops the MD5 sum calculation. Afterwards the results can be retrieved.

`const t_MD5Result & GetResult () const;`

Returns the result in a binary format.

`const char * GetResultStr ();`

Returns the result in a textual format. The string consists of 32 lower case hexadecimal characters and a terminating null character.

bool **operator ==** (const ct_MD5 & co_comp) const;

Compares the results of two MD5 objects.

3.4.3 Universally Unique Identifier (tuning/uuid.hpp)

The class ct_UUID provides an interface to create and process Universally Unique Identifiers.

Class Declaration

```
typedef t_UInt8 t_UUID [16];

class ct_UUID
  {
public:
                       ct_UUID ();
                       ct_UUID (const ct_UUID & co_init);
                       ct_UUID (const t_UUID & ao_init);
  ct_UUID &            operator = (const ct_UUID & co_asgn);

  bool                 IsEmpty () const;
  t_UInt               GetHash () const;
  const t_UUID &       GetUUID () const;
  void                 Clear ();
  bool                 Create ();
  bool                 ToStr (char * pc_dst, t_UInt u_len, bool b_upperCase) const;
  bool                 FromStr (const char * pc_src, t_UInt u_len);

  bool                 operator == (const ct_UUID & co_comp) const;
  bool                 operator != (const ct_UUID & co_comp) const;
  };
```

Methods

ct_UUID ();

Initializes an empty UUID object.

ct_UUID (const ct_UUID & co_init);

Copies the data from another UUID object.

ct_UUID (const t_UUID & ao_init);

Copies the binary UUID data.

ct_UUID & **operator =** (const ct_UUID & co_asgn);

Copies the data from another UUID object.

bool **IsEmpty** () const;

Returns true if the UUID object is empty.

t_UInt **GetHash** () const;

Returns a hash value.

const t_UUID & **GetUUID** () const;

Returns a reference to the binary UUID data.

```
void Clear ();
```

Clears the UUID object.

```
bool Create ();
```

Creates a new Universally Unique Identifier.

```
bool ToStr (char * pc_dst, t_UInt u_len, bool b_upperCase) const;
```

Converts the binary UUID to a string and writes the result to the buffer (`pc_dst`, `u_len`) (`u_len >= 36`). The formatted string consists of 36 characters <u>without</u> a terminating null character. If `b_upperCase` equals `true` upper case characters are used.

```
bool FromStr (const char * pc_src, t_UInt u_len);
```

Converts a formatted string to a binary UUID. The first 36 characters of the buffer (`pc_src`, `u_len`) (`u_len >= 36`) are interpreted as a textual UUID.

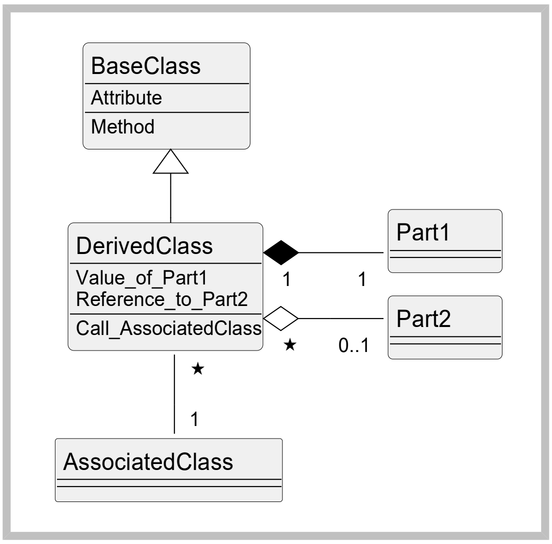
```
bool operator == (const ct_UUID & co_comp) const;
bool operator != (const ct_UUID & co_comp) const;
```

Compare two UUID objects.

4 DESIGN DIAGRAMS

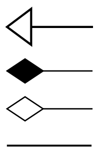
4.1 Notation

The following sections contain some design diagrams describing the interaction of several components of the **Spirick Tuning** library. The diagrams are based on the '**U**nified **M**odeling **L**anguage' (UML). The following diagram shows some important parts of UML class diagrams.



Classes are represented by rectangles which show the name of the class and optionally the attributes and methods. The following relationships can be used:

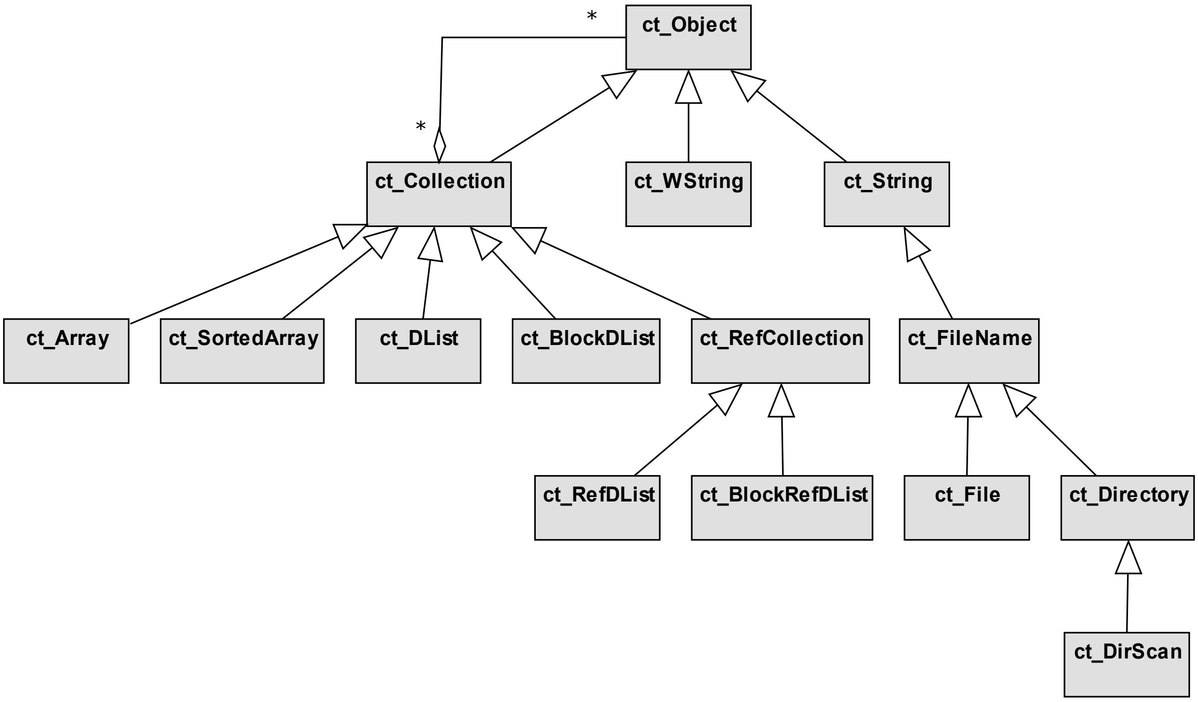
- Inheritance
- Composition
- Aggregation
- Association



Some connectors may include cardinality at each end.

4.2 Polymorphic Class Hierarchy

The following diagram shows all classes which inherit from the abstract base class `ct_Object`.



4.3 An Array Container

The following (partially simplified) diagram shows all classes which are used to implement an array container. The container instance was defined by the following sample code:

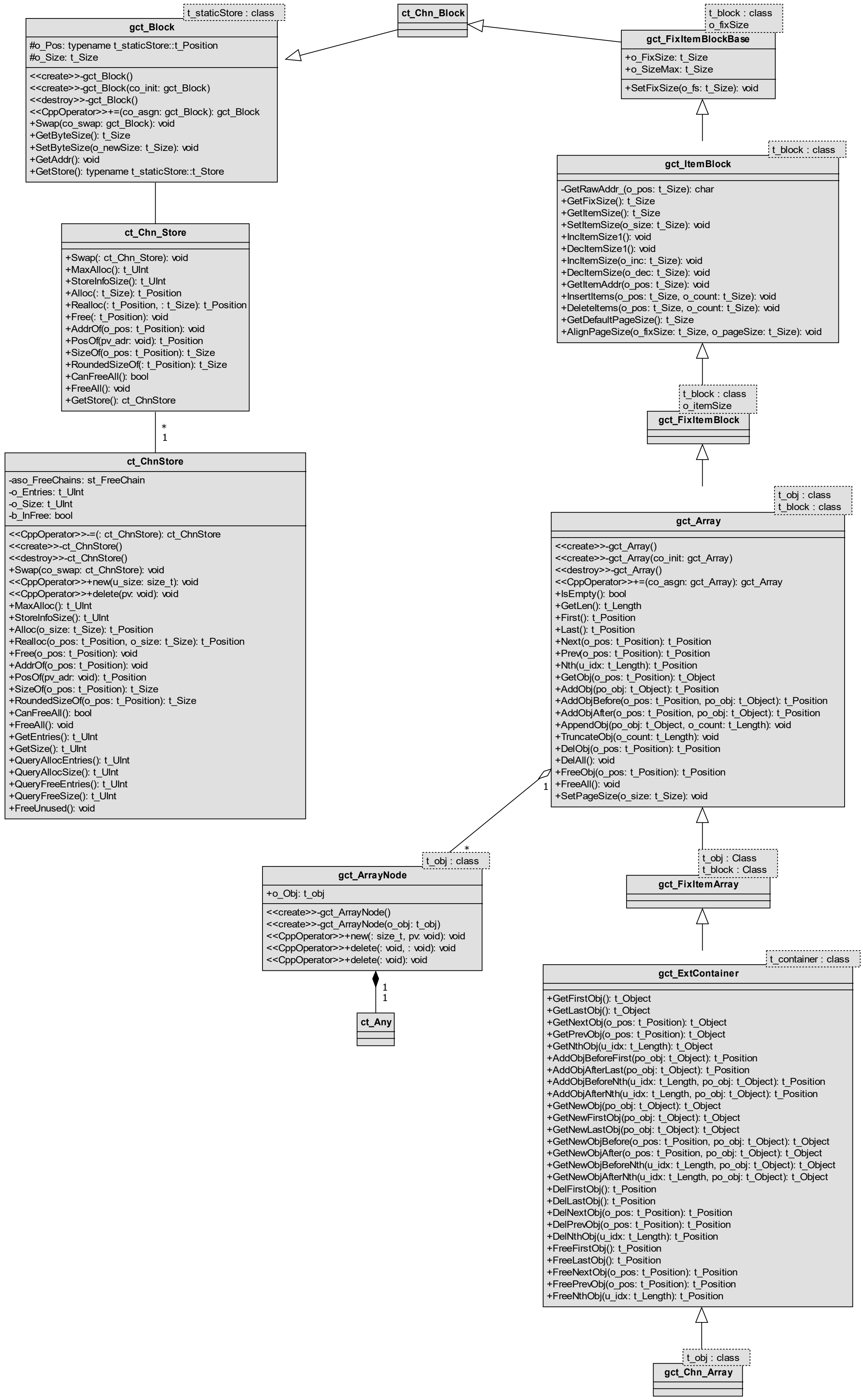
```
#include "tuning/chn/array.h"
class ct_Any { /* ... */ };
gct_Chn_Array <ct_Any> co_AnyArray;
```

The array container allocates memory using the store class `ct_ChnStore`. The wrapper class `ct_Chn_Store` maps methods of the global store object to static class methods. The abbreviation _ determines the nested size type `t_UInt`.

The class `ct_Chn_Block` is a predefined instance of the block template `gct_Block` using the wrapper class `ct_Chn_Store`. The class template `gct_ItemBlock` is an extension of the common block interface. The helper templates `gct_FixItemBlockBase` and `gct_FixItemBlock` are used for compile time configuration of the item size.

The container template `gct_Array` is instantiated using the parameters `ct_Any` and `gct_FixItemBlock <t_block, sizeof (gct_ArrayNode <ct_Any>)>`. The helper template `gct_ArrayNode` is used to construct and destruct the contained objects. The helper template `gct_FixItemArray` passes the size of an object to the template `gct_FixItemBlock`.

The class template `gct_ExtContainer` enhances the usability of the basic container interface. The template `gct_Chn_Array` is a predefined shortcut for `gct_ExtContainer <gct_FixItemArray <t_obj, ct_Chn_Block> >`.



t_staticStore : class
gct_Block
#o_Pos: typename t_staticStore::t_Position
#o_Size: t_Size
<<create>>-gct_Block()
<<create>>-gct_Block(co_init: gct_Block)
<<destroy>>-gct_Block()
<<CppOperator>>+=(co_asgn: gct_Block): gct_Block
+Swap(co_swap: gct_Block): void
+GetByteSize(): t_Size
+SetByteSize(o_newSize: t_Size): void
+GetAddr(): void
+GetStore(): typename t_staticStore::t_Store
ct_Chn_Block
t_block : class
o_fixSize
gct_FixItemBlockBase
+o_FixSize: t_Size
+o_SizeMax: t_Size
+SetFixSize(o_fs: t_Size): void
t_block : class
gct_ItemBlock
-GetRawAddr_(o_pos: t_Size): char
+GetFixSize(): t_Size
+GetItemSize(): t_Size
+SetItemSize(o_size: t_Size): void
+IncItemSize1(): void
+DecItemSize1(): void
+IncItemSize(o_inc: t_Size): void
+DecItemSize(o_dec: t_Size): void
+GetItemAddr(o_pos: t_Size): void
+InsertItems(o_pos: t_Size, o_count: t_Size): void
+DeleteItems(o_pos: t_Size, o_count: t_Size): void
+GetDefaultPageSize(): t_Size
+AlignPageSize(o_fixSize: t_Size, o_pageSize: t_Size): void
ct_Chn_Store
+Swap(: ct_Chn_Store): void
+MaxAlloc(): t_UInt
+StoreInfoSize(): t_UInt
+Alloc(: t_Size): t_Position
+Realloc(: t_Position, : t_Size): t_Position
+Free(: t_Position): void
+AddrOf(o_pos: t_Position): void
+PosOf(pv_adr: void): t_Position
+SizeOf(o_pos: t_Position): t_Size
+RoundedSizeOf(: t_Position): t_Size
+CanFreeAll(): bool
+FreeAll(): void
+GetStore(): ct_ChnStore
*
1
t_block : class
o_itemSize
gct_FixItemBlock
ct_ChnStore
-aso_FreeChains: st_FreeChain
-o_Entries: t_UInt
-o_Size: t_UInt
-b_InFree: bool
<<CppOperator>>-=(: ct_ChnStore): ct_ChnStore
<<create>>-ct_ChnStore()
<<destroy>>-ct_ChnStore()
+Swap(co_swap: ct_ChnStore): void
<<CppOperator>>+new(u_size: size_t): void
<<CppOperator>>+delete(pv: void): void
+MaxAlloc(): t_UInt
+StoreInfoSize(): t_UInt
+Alloc(o_size: t_Size): t_Position
+Realloc(o_pos: t_Position, o_size: t_Size): t_Position
+Free(o_pos: t_Position): void
+AddrOf(o_pos: t_Position): void
+PosOf(pv_adr: void): t_Position
+SizeOf(o_pos: t_Position): t_Size
+RoundedSizeOf(o_pos: t_Position): t_Size
+CanFreeAll(): bool
+FreeAll(): void
+GetEntries(): t_UInt
+GetSize(): t_UInt
+QueryAllocEntries(): t_UInt
+QueryAllocSize(): t_UInt
+QueryFreeEntries(): t_UInt
+QueryFreeSize(): t_UInt
+FreeUnused(): void
t_obj : class
t_block : class
gct_Array
<<create>>-gct_Array()
<<create>>-gct_Array(co_init: gct_Array)
<<destroy>>-gct_Array()
<<CppOperator>>+=(co_asgn: gct_Array): gct_Array
+IsEmpty(): bool
+GetLen(): t_Length
+First(): t_Position
+Last(): t_Position
+Next(o_pos: t_Position): t_Position
+Prev(o_pos: t_Position): t_Position
+Nth(u_idx: t_Length): t_Position
+GetObj(o_pos: t_Position): t_Object
+AddObj(po_obj: t_Object): t_Position
+AddObjBefore(o_pos: t_Position, po_obj: t_Object): t_Position
+AddObjAfter(o_pos: t_Position, po_obj: t_Object): t_Position
+AppendObj(po_obj: t_Object, o_count: t_Length): void
+TruncateObj(o_count: t_Length): void
+DelObj(o_pos: t_Position): t_Position
+DelAll(): void
+FreeObj(o_pos: t_Position): t_Position
+FreeAll(): void
+SetPageSize(o_size: t_Size): void
1
*
t_obj : class
gct_ArrayNode
+o_Obj: t_obj
<<create>>-gct_ArrayNode()
<<create>>-gct_ArrayNode(o_obj: t_obj)
<<CppOperator>>+new(: size_t, pv: void): void
<<CppOperator>>+delete(: void, : void): void
<<CppOperator>>+delete(: void): void
1
1
ct_Any
t_obj : Class
t_block : Class
gct_FixItemArray
t_container : class
gct_ExtContainer
+GetFirstObj(): t_Object
+GetLastObj(): t_Object
+GetNextObj(o_pos: t_Position): t_Object
+GetPrevObj(o_pos: t_Position): t_Object
+GetNthObj(u_idx: t_Length): t_Object
+AddObjBeforeFirst(po_obj: t_Object): t_Position
+AddObjAfterLast(po_obj: t_Object): t_Position
+AddObjBeforeNth(u_idx: t_Length, po_obj: t_Object): t_Position
+AddObjAfterNth(u_idx: t_Length, po_obj: t_Object): t_Position
+GetNewObj(po_obj: t_Object): t_Object
+GetNewFirstObj(po_obj: t_Object): t_Object
+GetNewLastObj(po_obj: t_Object): t_Object
+GetNewObjBefore(o_pos: t_Position, po_obj: t_Object): t_Object
+GetNewObjAfter(o_pos: t_Position, po_obj: t_Object): t_Object
+GetNewObjBeforeNth(u_idx: t_Length, po_obj: t_Object): t_Object
+GetNewObjAfterNth(u_idx: t_Length, po_obj: t_Object): t_Object
+DelFirstObj(): t_Position
+DelLastObj(): t_Position
+DelNextObj(o_pos: t_Position): t_Position
+DelPrevObj(o_pos: t_Position): t_Position
+DelNthObj(u_idx: t_Length): t_Position
+FreeFirstObj(): t_Position
+FreeLastObj(): t_Position
+FreeNextObj(o_pos: t_Position): t_Position
+FreePrevObj(o_pos: t_Position): t_Position
+FreeNthObj(u_idx: t_Length): t_Position
t_obj : class
gct_Chn_Array

4.4 A Pointer Array Container

The following (partially simplified) diagram shows all classes which are used to implement a pointer array container. The container instance was defined by the following sample code:

```
#include "tuning/chn/ptrarray.h"
class ct_Any { /* ... */ };
gct_Chn_PtrArray <ct_Any> co_AnyPtrArray;
```

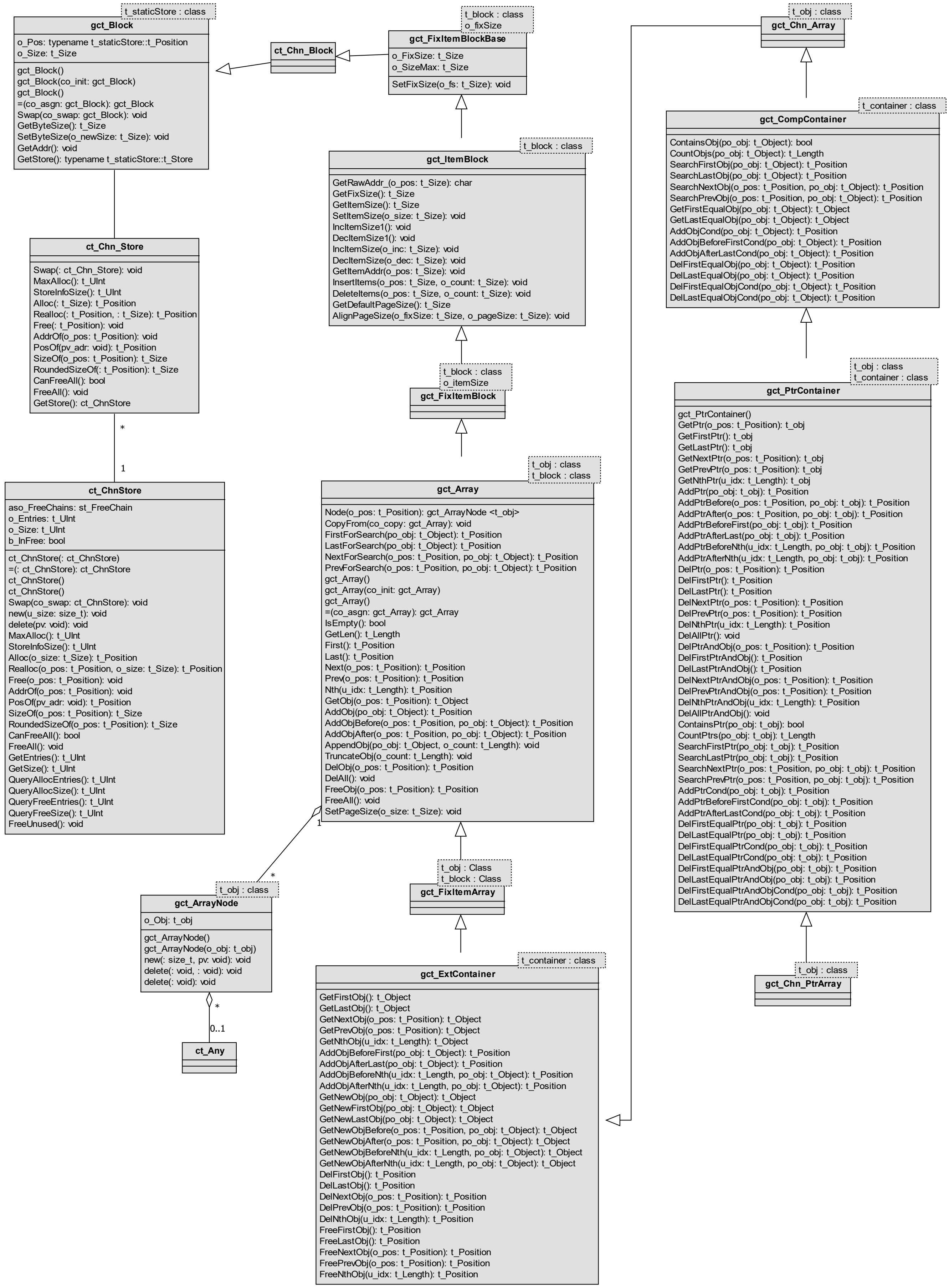
The pointer array container allocates memory using the store class ct_ChnStore. The wrapper class ct_Chn_Store maps methods of the global store object to static class methods. The abbreviation _ determines the nested size type t_UInt.

The class ct_Chn_Block is a predefined instance of the block template gct_Block using the wrapper class ct_Chn_Store. The class template gct_ItemBlock is an extension of the common block interface. The helper templates gct_FixItemBlockBase and gct_FixItemBlock are used for compile time configuration of the item size.

The container template gct_Array is instantiated using the parameters void * and gct_FixItemBlock <t_block, sizeof (gct_ArrayNode <void *>)>. The helper template gct_ArrayNode is used to construct and destruct the contained pointers. The helper template gct_FixItemArray passes the size of a pointer to the template gct_FixItemBlock.

The class template gct_ExtContainer enhances the usability of the basic container interface. The template gct_Chn_Array is a predefined shortcut for gct_ExtContainer <gct_FixItemArray <t_obj, ct_Chn_Block> >.

The class template gct_CompContainer implements some count, search and conditional methods. The class template gct_PtrContainer provides a comfortable interface for pointer containers. It maps many methods of the basic, extended and comp-container interface and provides some additional methods. The template gct_Chn_PtrArray is a predefined shortcut for gct_PtrContainer <ct_Any, gct_Chn_Array <void *> >.


t_staticStore : class
gct_Block
o_Pos: typename t_staticStore::t_Position
o_Size: t_Size
gct_Block()
gct_Block(co_init: gct_Block)
gct_Block()
=(co_asgn: gct_Block): gct_Block
Swap(co_swap: gct_Block): void
GetByteSize(): t_Size
SetByteSize(o_newSize: t_Size): void
GetAddr(): void
GetStore(): typename t_staticStore::t_Store
ct_Chn_Block
t_block : class
o_fixSize
gct_FixItemBlockBase
o_FixSize: t_Size
o_SizeMax: t_Size
SetFixSize(o_fs: t_Size): void
t_block : class
gct_ItemBlock
GetRawAddr_(o_pos: t_Size): char
GetFixSize(): t_Size
GetItemSize(): t_Size
SetItemSize(o_size: t_Size): void
IncItemSize1(): void
DecItemSize1(): void
IncItemSize(o_inc: t_Size): void
DecItemSize(o_dec: t_Size): void
GetItemAddr(o_pos: t_Size): void
InsertItems(o_pos: t_Size, o_count: t_Size): void
DeleteItems(o_pos: t_Size, o_count: t_Size): void
GetDefaultPageSize(): t_Size
AlignPageSize(o_fixSize: t_Size, o_pageSize: t_Size): void
t_block : class
o_itemSize
gct_FixItemBlock
ct_Chn_Store
Swap(: ct_Chn_Store): void
MaxAlloc(): t_UInt
StoreInfoSize(): t_UInt
Alloc(: t_Size): t_Position
Realloc(: t_Position, : t_Size): t_Position
Free(: t_Position): void
AddrOf(o_pos: t_Position): void
PosOf(pv_adr: void): t_Position
SizeOf(o_pos: t_Position): t_Size
RoundedSizeOf(: t_Position): t_Size
CanFreeAll(): bool
FreeAll(): void
GetStore(): ct_ChnStore
*
1
ct_ChnStore
aso_FreeChains: st_FreeChain
o_Entries: t_UInt
o_Size: t_UInt
b_InFree: bool
ct_ChnStore(: ct_ChnStore)
=(: ct_ChnStore): ct_ChnStore
ct_ChnStore()
ct_ChnStore()
Swap(co_swap: ct_ChnStore): void
new(u_size: size_t): void
delete(pv: void): void
MaxAlloc(): t_UInt
StoreInfoSize(): t_UInt
Alloc(o_size: t_Size): t_Position
Realloc(o_pos: t_Position, o_size: t_Size): t_Position
Free(o_pos: t_Position): void
AddrOf(o_pos: t_Position): void
PosOf(pv_adr: void): t_Position
SizeOf(o_pos: t_Position): t_Size
RoundedSizeOf(o_pos: t_Position): t_Size
CanFreeAll(): bool
FreeAll(): void
GetEntries(): t_UInt
GetSize(): t_UInt
QueryAllocEntries(): t_UInt
QueryAllocSize(): t_UInt
QueryFreeEntries(): t_UInt
QueryFreeSize(): t_UInt
FreeUnused(): void
t_obj : class
t_block : class
gct_Array
Node(o_pos: t_Position): gct_ArrayNode <t_obj>
CopyFrom(co_copy: gct_Array): void
FirstForSearch(po_obj: t_Object): t_Position
LastForSearch(po_obj: t_Object): t_Position
NextForSearch(o_pos: t_Position, po_obj: t_Object): t_Position
PrevForSearch(o_pos: t_Position, po_obj: t_Object): t_Position
gct_Array()
gct_Array(co_init: gct_Array)
gct_Array()
=(co_asgn: gct_Array): gct_Array
IsEmpty(): bool
GetLen(): t_Length
First(): t_Position
Last(): t_Position
Next(o_pos: t_Position): t_Position
Prev(o_pos: t_Position): t_Position
Nth(u_idx: t_Length): t_Position
GetObj(o_pos: t_Position): t_Object
AddObj(po_obj: t_Object): t_Position
AddObjBefore(o_pos: t_Position, po_obj: t_Object): t_Position
AddObjAfter(o_pos: t_Position, po_obj: t_Object): t_Position
AppendObj(po_obj: t_Object, o_count: t_Length): void
TruncateObj(o_count: t_Length): void
DelObj(o_pos: t_Position): t_Position
DelAll(): void
FreeObj(o_pos: t_Position): t_Position
FreeAll(): void
SetPageSize(o_size: t_Size): void
1
*
t_obj : class
gct_ArrayNode
o_Obj: t_obj
gct_ArrayNode()
gct_ArrayNode(o_obj: t_obj)
new(: size_t, pv: void): void
delete(: void, : void): void
delete(: void): void
*
0..1
ct_Any
t_obj : Class
t_block : Class
gct_FixItemArray
t_container : class
gct_ExtContainer
GetFirstObj(): t_Object
GetLastObj(): t_Object
GetNextObj(o_pos: t_Position): t_Object
GetPrevObj(o_pos: t_Position): t_Object
GetNthObj(u_idx: t_Length): t_Object
AddObjBeforeFirst(po_obj: t_Object): t_Position
AddObjAfterLast(po_obj: t_Object): t_Position
AddObjBeforeNth(u_idx: t_Length, po_obj: t_Object): t_Position
AddObjAfterNth(u_idx: t_Length, po_obj: t_Object): t_Position
GetNewObj(po_obj: t_Object): t_Object
GetNewFirstObj(po_obj: t_Object): t_Object
GetNewLastObj(po_obj: t_Object): t_Object
GetNewObjBefore(o_pos: t_Position, po_obj: t_Object): t_Object
GetNewObjAfter(o_pos: t_Position, po_obj: t_Object): t_Object
GetNewObjBeforeNth(u_idx: t_Length, po_obj: t_Object): t_Object
GetNewObjAfterNth(u_idx: t_Length, po_obj: t_Object): t_Object
DelFirstObj(): t_Position
DelLastObj(): t_Position
DelNextObj(o_pos: t_Position): t_Position
DelPrevObj(o_pos: t_Position): t_Position
DelNthObj(u_idx: t_Length): t_Position
FreeFirstObj(): t_Position
FreeLastObj(): t_Position
FreeNextObj(o_pos: t_Position): t_Position
FreePrevObj(o_pos: t_Position): t_Position
FreeNthObj(u_idx: t_Length): t_Position
t_obj : class
gct_Chn_Array
t_container : class
gct_CompContainer
ContainsObj(po_obj: t_Object): bool
CountObjs(po_obj: t_Object): t_Length
SearchFirstObj(po_obj: t_Object): t_Position
SearchLastObj(po_obj: t_Object): t_Position
SearchNextObj(o_pos: t_Position, po_obj: t_Object): t_Position
SearchPrevObj(o_pos: t_Position, po_obj: t_Object): t_Position
GetFirstEqualObj(po_obj: t_Object): t_Object
GetLastEqualObj(po_obj: t_Object): t_Object
AddObjCond(po_obj: t_Object): t_Position
AddObjBeforeFirstCond(po_obj: t_Object): t_Position
AddObjAfterLastCond(po_obj: t_Object): t_Position
DelFirstEqualObj(po_obj: t_Object): t_Position
DelLastEqualObj(po_obj: t_Object): t_Position
DelFirstEqualObjCond(po_obj: t_Object): t_Position
DelLastEqualObjCond(po_obj: t_Object): t_Position
t_obj : class
t_container : class
gct_PtrContainer
gct_PtrContainer()
GetPtr(o_pos: t_Position): t_obj
GetFirstPtr(): t_obj
GetLastPtr(): t_obj
GetNextPtr(o_pos: t_Position): t_obj
GetPrevPtr(o_pos: t_Position): t_obj
GetNthPtr(u_idx: t_Length): t_obj
AddPtr(po_obj: t_obj): t_Position
AddPtrBefore(o_pos: t_Position, po_obj: t_obj): t_Position
AddPtrAfter(o_pos: t_Position, po_obj: t_obj): t_Position
AddPtrBeforeFirst(po_obj: t_obj): t_Position
AddPtrAfterLast(po_obj: t_obj): t_Position
AddPtrBeforeNth(u_idx: t_Length, po_obj: t_obj): t_Position
AddPtrAfterNth(u_idx: t_Length, po_obj: t_obj): t_Position
DelPtr(o_pos: t_Position): t_Position
DelFirstPtr(): t_Position
DelLastPtr(): t_Position
DelNextPtr(o_pos: t_Position): t_Position
DelPrevPtr(o_pos: t_Position): t_Position
DelNthPtr(u_idx: t_Length): t_Position
DelAllPtr(): void
DelPtrAndObj(o_pos: t_Position): t_Position
DelFirstPtrAndObj(): t_Position
DelLastPtrAndObj(): t_Position
DelNextPtrAndObj(o_pos: t_Position): t_Position
DelPrevPtrAndObj(o_pos: t_Position): t_Position
DelNthPtrAndObj(u_idx: t_Length): t_Position
DelAllPtrAndObj(): void
ContainsPtr(po_obj: t_obj): bool
CountPtrs(po_obj: t_obj): t_Length
SearchFirstPtr(po_obj: t_obj): t_Position
SearchLastPtr(po_obj: t_obj): t_Position
SearchNextPtr(o_pos: t_Position, po_obj: t_obj): t_Position
SearchPrevPtr(o_pos: t_Position, po_obj: t_obj): t_Position
AddPtrCond(po_obj: t_obj): t_Position
AddPtrBeforeFirstCond(po_obj: t_obj): t_Position
AddPtrAfterLastCond(po_obj: t_obj): t_Position
DelFirstEqualPtr(po_obj: t_obj): t_Position
DelLastEqualPtr(po_obj: t_obj): t_Position
DelFirstEqualPtrCond(po_obj: t_obj): t_Position
DelLastEqualPtrCond(po_obj: t_obj): t_Position
DelFirstEqualPtrAndObj(po_obj: t_obj): t_Position
DelLastEqualPtrAndObj(po_obj: t_obj): t_Position
DelFirstEqualPtrAndObjCond(po_obj: t_obj): t_Position
DelLastEqualPtrAndObjCond(po_obj: t_obj): t_Position
t_obj : class
gct_Chn_PtrArray

4.5 A List Container

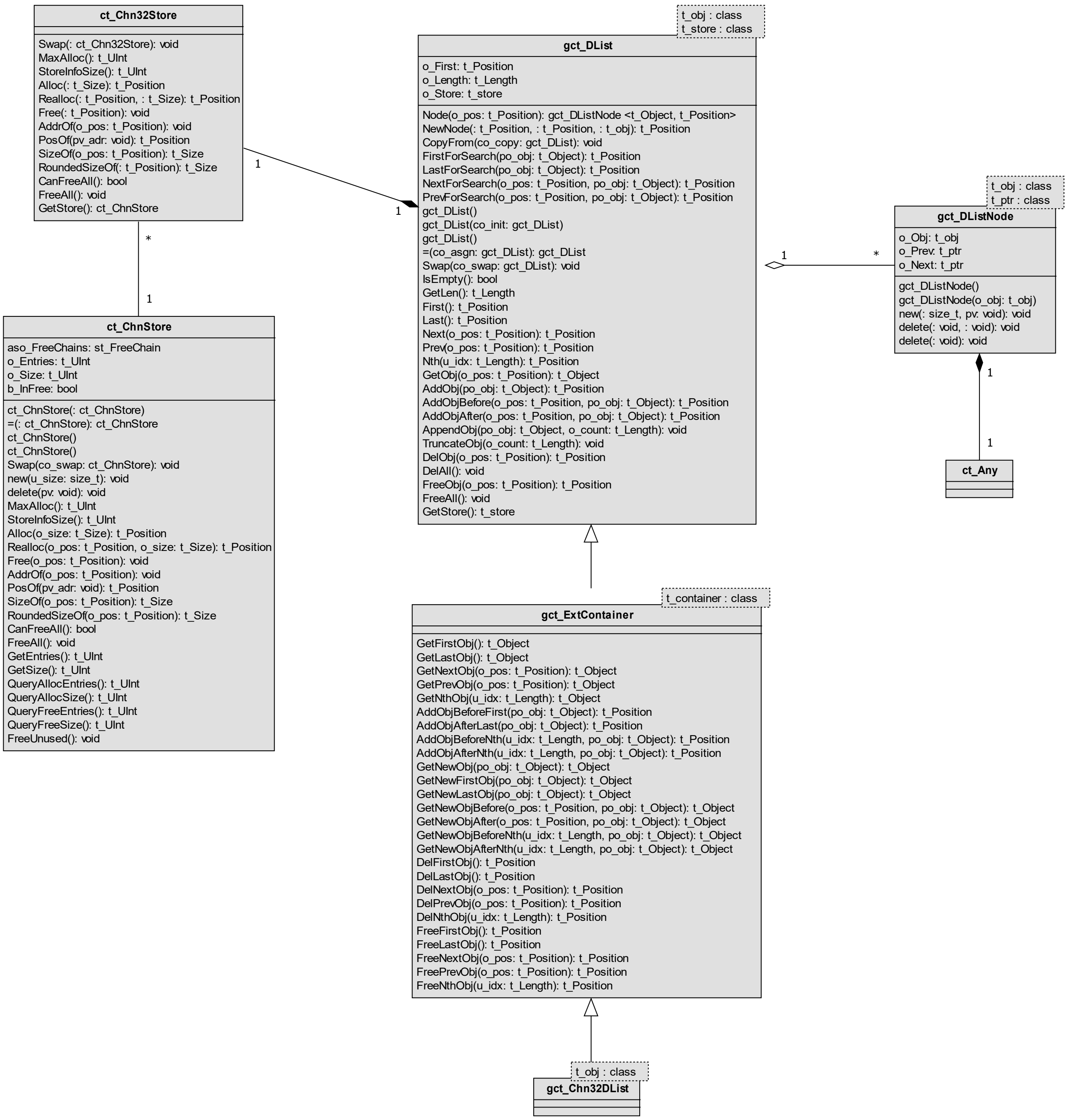
The following (partially simplified) diagram shows all classes which are used to implement a list container. The container instance was defined by the following sample code:

```
#include "tuning/chn/dlist.h"
class ct_Any { /* ... */ };
gct_Chn32DList <ct_Any> co_AnyDList;
```

The list container allocates memory using the store class ct_ChnStore. The wrapper class ct_Chn32Store maps methods of the global store object to static class methods. The abbreviation 32 determines the nested size type t_UInt32.

The container template gct_DList is instantiated using the parameters ct_Any and ct_Chn32Store. The list class contains a data member of type t_store. The helper template gct_DListNode is used to construct and destruct the contained objects. Every list node contains references (position values) to the direct neighbors.

The class template gct_ExtContainer enhances the usability of the basic container interface. The template gct_Chn32DList is a predefined shortcut for gct_ExtContainer <gct_DList <ct_Any, ct_Chn32Store> >.


ct_Chn32Store
Swap(: ct_Chn32Store): void
MaxAlloc(): t_UInt
StoreInfoSize(): t_UInt
Alloc(: t_Size): t_Position
Realloc(: t_Position, : t_Size): t_Position
Free(: t_Position): void
AddrOf(o_pos: t_Position): void
PosOf(pv_adr: void): t_Position
SizeOf(o_pos: t_Position): t_Size
RoundedSizeOf(: t_Position): t_Size
CanFreeAll(): bool
FreeAll(): void
GetStore(): ct_ChnStore
1
1
*
1
ct_ChnStore
aso_FreeChains: st_FreeChain
o_Entries: t_UInt
o_Size: t_UInt
b_InFree: bool
ct_ChnStore(: ct_ChnStore)
=(: ct_ChnStore): ct_ChnStore
ct_ChnStore()
ct_ChnStore()
Swap(co_swap: ct_ChnStore): void
new(u_size: size_t): void
delete(pv: void): void
MaxAlloc(): t_UInt
StoreInfoSize(): t_UInt
Alloc(o_size: t_Size): t_Position
Realloc(o_pos: t_Position, o_size: t_Size): t_Position
Free(o_pos: t_Position): void
AddrOf(o_pos: t_Position): void
PosOf(pv_adr: void): t_Position
SizeOf(o_pos: t_Position): t_Size
RoundedSizeOf(o_pos: t_Position): t_Size
CanFreeAll(): bool
FreeAll(): void
GetEntries(): t_UInt
GetSize(): t_UInt
QueryAllocEntries(): t_UInt
QueryAllocSize(): t_UInt
QueryFreeEntries(): t_UInt
QueryFreeSize(): t_UInt
FreeUnused(): void
t_obj : class
t_store : class
gct_DList
o_First: t_Position
o_Length: t_Length
o_Store: t_store
Node(o_pos: t_Position): gct_DListNode <t_Object, t_Position>
NewNode(: t_Position, : t_Position, : t_obj): t_Position
CopyFrom(co_copy: gct_DList): void
FirstForSearch(po_obj: t_Object): t_Position
LastForSearch(po_obj: t_Object): t_Position
NextForSearch(o_pos: t_Position, po_obj: t_Object): t_Position
PrevForSearch(o_pos: t_Position, po_obj: t_Object): t_Position
gct_DList()
gct_DList(co_init: gct_DList)
gct_DList()
=(co_asgn: gct_DList): gct_DList
Swap(co_swap: gct_DList): void
IsEmpty(): bool
GetLen(): t_Length
First(): t_Position
Last(): t_Position
Next(o_pos: t_Position): t_Position
Prev(o_pos: t_Position): t_Position
Nth(u_idx: t_Length): t_Position
GetObj(o_pos: t_Position): t_Object
AddObj(po_obj: t_Object): t_Position
AddObjBefore(o_pos: t_Position, po_obj: t_Object): t_Position
AddObjAfter(o_pos: t_Position, po_obj: t_Object): t_Position
AppendObj(po_obj: t_Object, o_count: t_Length): void
TruncateObj(o_count: t_Length): void
DelObj(o_pos: t_Position): t_Position
DelAll(): void
FreeObj(o_pos: t_Position): t_Position
FreeAll(): void
GetStore(): t_store
1
*
t_obj : class
t_ptr : class
gct_DListNode
o_Obj: t_obj
o_Prev: t_ptr
o_Next: t_ptr
gct_DListNode()
gct_DListNode(o_obj: t_obj)
new(: size_t, pv: void): void
delete(: void, : void): void
delete(: void): void
1
1
ct_Any
t_container : class
gct_ExtContainer
GetFirstObj(): t_Object
GetLastObj(): t_Object
GetNextObj(o_pos: t_Position): t_Object
GetPrevObj(o_pos: t_Position): t_Object
GetNthObj(u_idx: t_Length): t_Object
AddObjBeforeFirst(po_obj: t_Object): t_Position
AddObjAfterLast(po_obj: t_Object): t_Position
AddObjBeforeNth(u_idx: t_Length, po_obj: t_Object): t_Position
AddObjAfterNth(u_idx: t_Length, po_obj: t_Object): t_Position
GetNewObj(po_obj: t_Object): t_Object
GetNewFirstObj(po_obj: t_Object): t_Object
GetNewLastObj(po_obj: t_Object): t_Object
GetNewObjBefore(o_pos: t_Position, po_obj: t_Object): t_Object
GetNewObjAfter(o_pos: t_Position, po_obj: t_Object): t_Object
GetNewObjBeforeNth(u_idx: t_Length, po_obj: t_Object): t_Object
GetNewObjAfterNth(u_idx: t_Length, po_obj: t_Object): t_Object
DelFirstObj(): t_Position
DelLastObj(): t_Position
DelNextObj(o_pos: t_Position): t_Position
DelPrevObj(o_pos: t_Position): t_Position
DelNthObj(u_idx: t_Length): t_Position
FreeFirstObj(): t_Position
FreeLastObj(): t_Position
FreeNextObj(o_pos: t_Position): t_Position
FreePrevObj(o_pos: t_Position): t_Position
FreeNthObj(u_idx: t_Length): t_Position
t_obj : class
gct_Chn32DList

4.6 A Block List Container

The following (partially simplified) diagram shows all classes which are used to implement a block list container. The container instance was defined by the following sample code:

```
#include "tuning/chn/blockdlist.h"
class ct_Any { /* ... */ };
gct_Chn32BlockDList <ct_Any> co_AnyBlockDList;
```

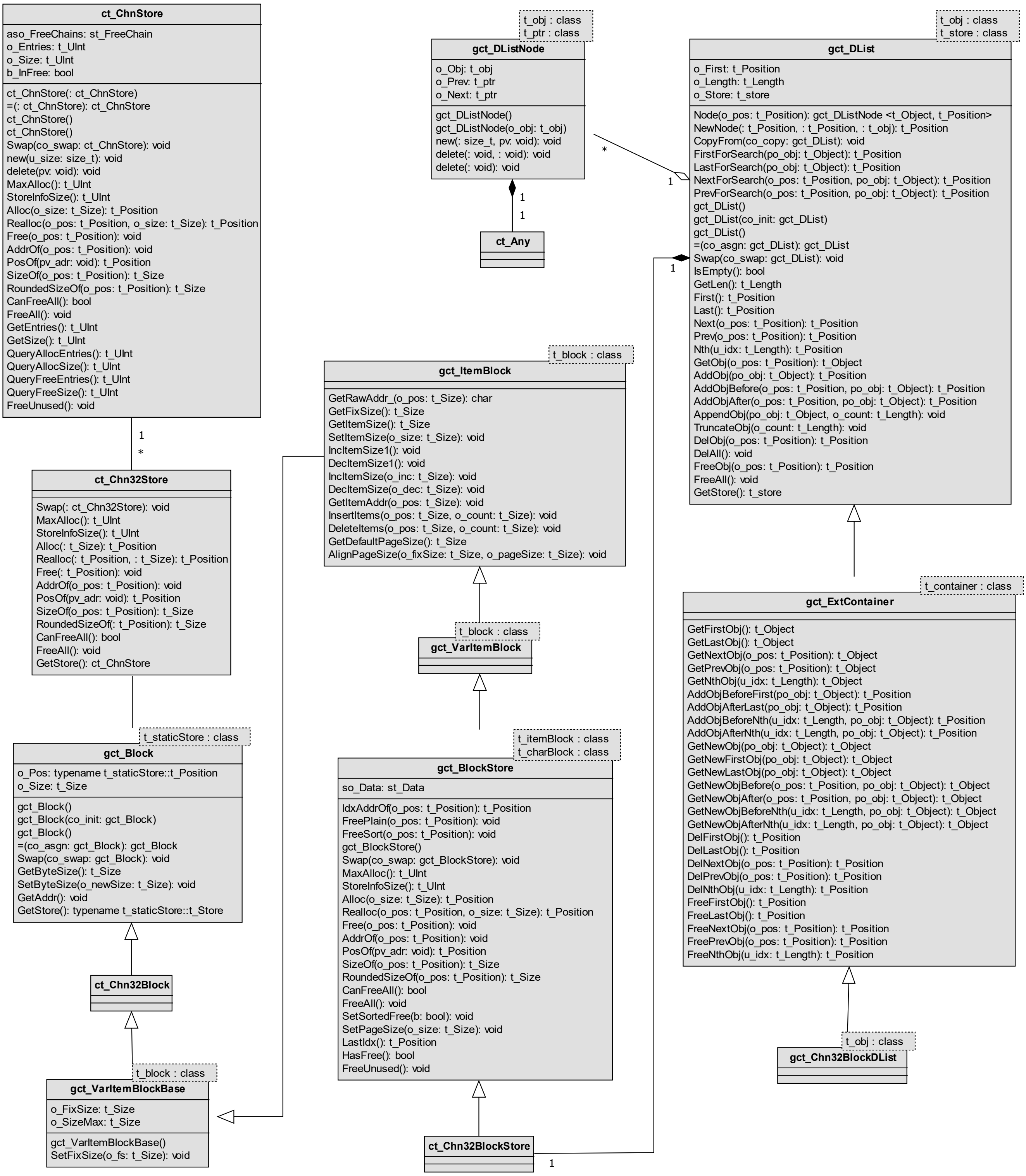
The block list container allocates memory using the store class ct_ChnStore. The wrapper class ct_Chn32Store maps methods of the global store object to static class methods. The abbreviation 32 determines the nested size type t_UInt32.

The class ct_Chn32Block is a predefined instance of the block template gct_Block using the wrapper class ct_Chn32Store. The class template gct_ItemBlock is an extension of the common block interface. The helper templates gct_VarItemBlockBase and gct_VarItemBlock are used for runtime configuration of the item size.

A block store uses an item block for compact storage of smaller, equal-sized memory blocks. The store template gct_BlockStore is instantiated using the parameters gct_VarItemBlock <ct_Chn32Block> and gct_CharBlock <ct_Chn32Block, char>. The template ct_Chn32BlockStore is a predefined shortcut for gct_BlockStore <gct_Var..., gct_Char...>.

The container template gct_DList is instantiated using the parameters ct_Any and ct_Chn32BlockStore. The list class contains a data member of type t_store. The helper template gct_DListNode is used to construct and destruct the contained objects. Every list node contains references (position values) to the direct neighbors.

The class template gct_ExtContainer enhances the usability of the basic container interface. The template gct_Chn32BlockDList is a predefined shortcut for gct_ExtContainer <gct_DList <ct_Any, ct_Chn32BlockStore> >.


ct_ChnStore
aso_FreeChains: st_FreeChain
o_Entries: t_UInt
o_Size: t_UInt
b_InFree: bool
ct_ChnStore(: ct_ChnStore)
=(: ct_ChnStore): ct_ChnStore
ct_ChnStore()
ct_ChnStore()
Swap(co_swap: ct_ChnStore): void
new(u_size: size_t): void
delete(pv: void): void
MaxAlloc(): t_UInt
StoreInfoSize(): t_UInt
Alloc(o_size: t_Size): t_Position
Realloc(o_pos: t_Position, o_size: t_Size): t_Position
Free(o_pos: t_Position): void
AddrOf(o_pos: t_Position): void
PosOf(pv_adr: void): t_Position
SizeOf(o_pos: t_Position): t_Size
RoundedSizeOf(o_pos: t_Position): t_Size
CanFreeAll(): bool
FreeAll(): void
GetEntries(): t_UInt
GetSize(): t_UInt
QueryAllocEntries(): t_UInt
QueryAllocSize(): t_UInt
QueryFreeEntries(): t_UInt
QueryFreeSize(): t_UInt
FreeUnused(): void
1
*
ct_Chn32Store
Swap(: ct_Chn32Store): void
MaxAlloc(): t_UInt
StoreInfoSize(): t_UInt
Alloc(: t_Size): t_Position
Realloc(: t_Position, : t_Size): t_Position
Free(: t_Position): void
AddrOf(o_pos: t_Position): void
PosOf(pv_adr: void): t_Position
SizeOf(o_pos: t_Position): t_Size
RoundedSizeOf(: t_Position): t_Size
CanFreeAll(): bool
FreeAll(): void
GetStore(): ct_ChnStore
t_staticStore : class
gct_Block
o_Pos: typename t_staticStore::t_Position
o_Size: t_Size
gct_Block()
gct_Block(co_init: gct_Block)
gct_Block()
=(co_asgn: gct_Block): gct_Block
Swap(co_swap: gct_Block): void
GetByteSize(): t_Size
SetByteSize(o_newSize: t_Size): void
GetAddr(): void
GetStore(): typename t_staticStore::t_Store
ct_Chn32Block
t_block : class
gct_VarItemBlockBase
o_FixSize: t_Size
o_SizeMax: t_Size
gct_VarItemBlockBase()
SetFixSize(o_fs: t_Size): void
t_obj : class
t_ptr : class
gct_DListNode
o_Obj: t_obj
o_Prev: t_ptr
o_Next: t_ptr
gct_DListNode()
gct_DListNode(o_obj: t_obj)
new(: size_t, pv: void): void
delete(: void, : void): void
delete(: void): void
1
1
ct_Any
*
1
t_block : class
gct_ItemBlock
GetRawAddr_(o_pos: t_Size): char
GetFixSize(): t_Size
GetItemSize(): t_Size
SetItemSize(o_size: t_Size): void
IncItemSize1(): void
DecItemSize1(): void
IncItemSize(o_inc: t_Size): void
DecItemSize(o_dec: t_Size): void
GetItemAddr(o_pos: t_Size): void
InsertItems(o_pos: t_Size, o_count: t_Size): void
DeleteItems(o_pos: t_Size, o_count: t_Size): void
GetDefaultPageSize(): t_Size
AlignPageSize(o_fixSize: t_Size, o_pageSize: t_Size): void
t_block : class
gct_VarItemBlock
t_itemBlock : class
t_charBlock : class
gct_BlockStore
so_Data: st_Data
IdxAddrOf(o_pos: t_Position): t_Position
FreePlain(o_pos: t_Position): void
FreeSort(o_pos: t_Position): void
gct_BlockStore()
Swap(co_swap: gct_BlockStore): void
MaxAlloc(): t_UInt
StoreInfoSize(): t_UInt
Alloc(o_size: t_Size): t_Position
Realloc(o_pos: t_Position, o_size: t_Size): t_Position
Free(o_pos: t_Position): void
AddrOf(o_pos: t_Position): void
PosOf(pv_adr: void): t_Position
SizeOf(o_pos: t_Position): t_Size
RoundedSizeOf(o_pos: t_Position): t_Size
CanFreeAll(): bool
FreeAll(): void
SetSortedFree(b: bool): void
SetPageSize(o_size: t_Size): void
LastIdx(): t_Position
HasFree(): bool
FreeUnused(): void
ct_Chn32BlockStore
1
t_obj : class
t_store : class
gct_DList
o_First: t_Position
o_Length: t_Length
o_Store: t_store
Node(o_pos: t_Position): gct_DListNode <t_Object, t_Position>
NewNode(: t_Position, : t_Position, : t_obj): t_Position
CopyFrom(co_copy: gct_DList): void
FirstForSearch(po_obj: t_Object): t_Position
LastForSearch(po_obj: t_Object): t_Position
NextForSearch(o_pos: t_Position, po_obj: t_Object): t_Position
PrevForSearch(o_pos: t_Position, po_obj: t_Object): t_Position
gct_DList()
gct_DList(co_init: gct_DList)
gct_DList()
=(co_asgn: gct_DList): gct_DList
Swap(co_swap: gct_DList): void
IsEmpty(): bool
GetLen(): t_Length
First(): t_Position
Last(): t_Position
Next(o_pos: t_Position): t_Position
Prev(o_pos: t_Position): t_Position
Nth(u_idx: t_Length): t_Position
GetObj(o_pos: t_Position): t_Object
AddObj(po_obj: t_Object): t_Position
AddObjBefore(o_pos: t_Position, po_obj: t_Object): t_Position
AddObjAfter(o_pos: t_Position, po_obj: t_Object): t_Position
AppendObj(po_obj: t_Object, o_count: t_Length): void
TruncateObj(o_count: t_Length): void
DelObj(o_pos: t_Position): t_Position
DelAll(): void
FreeObj(o_pos: t_Position): t_Position
FreeAll(): void
GetStore(): t_store
t_container : class
gct_ExtContainer
GetFirstObj(): t_Object
GetLastObj(): t_Object
GetNextObj(o_pos: t_Position): t_Object
GetPrevObj(o_pos: t_Position): t_Object
GetNthObj(u_idx: t_Length): t_Object
AddObjBeforeFirst(po_obj: t_Object): t_Position
AddObjAfterLast(po_obj: t_Object): t_Position
AddObjBeforeNth(u_idx: t_Length, po_obj: t_Object): t_Position
AddObjAfterNth(u_idx: t_Length, po_obj: t_Object): t_Position
GetNewObj(po_obj: t_Object): t_Object
GetNewFirstObj(po_obj: t_Object): t_Object
GetNewLastObj(po_obj: t_Object): t_Object
GetNewObjBefore(o_pos: t_Position, po_obj: t_Object): t_Object
GetNewObjAfter(o_pos: t_Position, po_obj: t_Object): t_Object
GetNewObjBeforeNth(u_idx: t_Length, po_obj: t_Object): t_Object
GetNewObjAfterNth(u_idx: t_Length, po_obj: t_Object): t_Object
DelFirstObj(): t_Position
DelLastObj(): t_Position
DelNextObj(o_pos: t_Position): t_Position
DelPrevObj(o_pos: t_Position): t_Position
DelNthObj(u_idx: t_Length): t_Position
FreeFirstObj(): t_Position
FreeLastObj(): t_Position
FreeNextObj(o_pos: t_Position): t_Position
FreePrevObj(o_pos: t_Position): t_Position
FreeNthObj(u_idx: t_Length): t_Position
t_obj : class
gct_Chn32BlockDList

5 INSTALLATION

5.1 Installation

5.1.1 Available Platforms

The **Spirick Tuning** library is currently available for the following operating systems: MS Windows XP, MS Windows 7, MS Windows 10 and Linux (x86/x86-64, kernel 2.6.32 to 5.13.0). The library can be used in 32-bit and 64-bit environments, in single-threaded or multi-threaded mode. The source code is developed and tested for the following compilers: MS Visual C++ 8.0 (2005) to 17.0 (2022) and g++ 4.4.5 to 11.2.0.

5.1.2 Dependencies

The **Spirick Tuning** library uses the compiler runtime system and OS dependent low-level functions. On Linux platforms the library Pthreads is used for multithreading. There are no dependencies or interactions to other libraries.

5.1.3 Makefiles

The source code of the **Spirick Tuning** library can be integrated in any existing build system. Alternatively the **Spirick** makefiles can be used. These makefiles automatically detect the make utility (MS Windows: nmake, Linux: gmake). The **Spirick** makefiles use the following environment variables:

`TL_PROJECT_TARGETDIR`: The compiler and linker target directory.
`TL_COMPILER`: A shortcut for the compiler version, e.g. "msc192164".
`TL_RELEASE`: Switch between debug and release build.
`MSDEVDIR`: MS Windows only: Detect the MSVC compiler.
`TL_BUILD_DLL`: MSVC only: Switch between `_declspec (dllexport)` and `_declspec (dllimport)`.

The **Spirick** makefiles use the sd utility (Spirick Source Dependencies). The source code of the sd utility is included in the **Spirick Tuning** library. Bootstrap method: If the sd executable is not available, use an empty sd script file (MS Windows: sd.bat, Linux: sd.sh).

5.1.4 Global Objects

Each global store object (see above 'Global Stores') has its own global access function. The global object is created in the first call of the access function. This technique ensures safe access to store objects from constructors of global C++ objects. A global store object may be created directly by a global `Create` function.

Global store objects are not destroyed automatically during program termination. This technique ensures safe access to store objects from destructors of global C++ objects. The destruction of global store objects is not necessary. They manage raw memory blocks, and this memory is released by the OS automatically. A global store object may be destroyed directly by a global `Delete` function.

Note that a heap walker may report the global store objects as memory leaks at the end of the program. This problem can be avoided by explicitly deleting these objects. Please ensure that a global store object is not used after deleting it.

The file **'tuning/sys/cprocess.cpp'** contains access functions for two global mutex objects (see above 'Thread Mutex' and 'Process Mutex'). These objects are created in the first call of the access functions or before starting the first thread. At the end of the program the global mutex objects can be destroyed by calling a global `Delete` function.

5.1.5 Exception Handling

Exception handling can be enabled or disabled by compiler options. In some C++ projects exception handling is disabled to improve performance. The **Spirick Tuning** library can be used with or without exception handling. All functions return `true` on success and `false` or an error code on failure, no exceptions are thrown.

While working with containers, exceptions may occur inside of constructors and destructors of contained objects. **Spirick** container classes contain minimal exception handlers. These handlers ensure the consistency of the container object and pass the exception unchanged to a higher-level handler (see above 'Container Interface').

Index