

Spirick Tuning

Eine C++ Klassen- und Template-Bibliothek
für performancekritische Anwendungen

Referenzhandbuch



Version 1.48

Stand November 2021

Copyright © Dietmar Deimling 1996 - 2021. All rights reserved.

Kein Teil dieses Werkes darf ohne schriftliche Genehmigung des Autors in irgendeiner Form (Fotokopie, Mikrofilm oder andere Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden. Bei der Zusammenstellung wurde mit größter Sorgfalt vorgegangen. Fehler können trotzdem nicht völlig ausgeschlossen werden, so daß der Autor für fehlerhafte Angaben und deren Folgen keine juristische Verantwortung oder irgendeine Haftung übernimmt. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, daß solche Namen im Sinne der Warenzeichen- und Markenschutzgesetzgebung als frei betrachtet wären. Für Verbesserungsvorschläge und Hinweise auf Fehler ist der Autor stets dankbar.

Inhaltsverzeichnis

1 SPEICHERVERWALTUNG

7

1.1 Systemschnittstelle	7
1.1.1 Globale Definitionen (tuning/defs.hpp)	7
1.1.2 Reservespeicher (tuning/sys/calloc.hpp)	7
1.1.3 Dynamischer Speicher (tuning/sys/calloc.hpp)	8
1.1.4 Heapoperationen (tuning/sys/calloc.hpp)	9
1.1.5 Speicheroperationen (tuning/sys/cmemory.hpp)	10
1.2 Store	11
1.2.1 Storeschnittstelle	11
1.2.2 Globale Stores (tuning/defs.hpp)	13
1.2.3 Beispiel für eine Wrapperklasse	14
1.3 Dynamische Stores	15
1.3.1 Standardstore (tuning/std/store.hpp)	15
1.3.2 Roundstore (tuning/rnd/store.hpp)	16
1.3.3 Chainstore (tuning/chn/store.hpp)	17
1.3.4 Operatoren new und delete (tuning/newdel.cpp)	20
1.4 Block	20
1.4.1 Blockschnittstelle	20
1.4.2 Allgemeiner Block (tuning/block.h)	22
1.4.3 Miniblock (tuning/miniblock.h)	24
1.4.4 Reserveblock (tuning/resblock.h)	25
1.4.5 Fixblock (tuning/fixblock.h)	27
1.4.6 Nulldatablock (tuning/nulldatablock.h)	28
1.4.7 Zeichenblock (tuning/charblock.h)	28
1.4.8 Elementblock (tuning/itemblock.h)	30
1.4.9 Pageblock (tuning/pageblock.hpp)	32
1.4.10 Block-Instanzen (tuning/xxx/block.h)	35
1.5 Spezielle Stores	36
1.5.1 Blockstore (tuning/blockstore.h)	36
1.5.2 Blockstore-Instanzen (tuning/xxx/blockstore.h)	38
1.5.3 Referenzzähler (tuning/refcount.hpp)	38
1.5.4 Refstore (tuning/refstore.h)	40
1.5.5 Refstore-Instanzen (tuning/xxx/refstore.h)	41
1.5.6 Blockrefstore-Instanzen (tuning/xxx/blockrefstore.h)	42
1.5.7 Packstore (tuning/packstore.hpp)	43

2 OBJEKTVERWALTUNG

45

2.1 Container	45
2.1.1 Containerschnittstelle	45
2.1.2 Operationen mit Containern	49
2.1.3 Erweiterter Container (tuning/extcont.h)	51
2.2 Arrays und Listen	54
2.2.1 Array (tuning/array.h)	54
2.2.2 Array-Instanzen (tuning/xxx/array.h)	56
2.2.3 Liste (tuning/dlist.h)	56
2.2.4 Listen-Instanzen (tuning/xxx/dlist.h)	57
2.3 Sortierte Container	58
2.3.1 Sortiertes Array (tuning/sortarr.h)	58
2.3.2 Sortierte Array-Instanzen (tuning/xxx/sortedarray.h)	60

2.3.3 Hashtabelle (tuning/hashtable.h).....	61
2.3.4 Hashtabellen-Instanzen (tuning/xxx/hashtable.h).....	62
2.4 Block- und Reflisten.....	63
2.4.1 Blockliste.....	63
2.4.2 Blocklisten-Instanzen (tuning/xxx/blockdlist.h).....	63
2.4.3 Refliste (tuning/refdlist.h).....	64
2.4.4 Reflisten-Instanzen (tuning/xxx/refdlist.h).....	65
2.4.5 Blockreflisten-Instanzen (tuning/xxx/blockrefdlist.h).....	66
2.5 Vergleichs-, Zeiger- und Mapcontainer.....	67
2.5.1 Vergleichscontainer (tuning/compcontainer.h).....	67
2.5.2 Zeigercontainer (tuning/ptrcontainer.h).....	69
2.5.3 Operationen mit Zeigercontainern.....	75
2.5.4 Zeigervergleichscontainer (tuning/ptrcompcontainer.h).....	77
2.5.5 Mapcontainer (tuning/map.h).....	79
2.5.6 Zeigermappedcontainer (tuning/ptrmap.h).....	82
2.6 Zeigercontainer-Instanzen.....	86
2.6.1 Zeigerarray-Instanzen (tuning/xxx/ptrarray.h).....	86
2.6.2 Zeigerlisten-Instanzen (tuning/xxx/ptrdlist.h).....	87
2.6.3 Sortierte Zeigerarray-Instanzen (tuning/xxx/ptrsortedarray.h).....	88
2.6.4 Zeigerhashtabellen-Instanzen (tuning/xxx/ptrhashtable.h).....	88
2.6.5 Blockzeigerlisten-Instanzen (tuning/xxx/blockptrdlist.h).....	89
2.6.6 Refzeigerlisten-Instanzen (tuning/xxx/refptrdlist.h).....	90
2.6.7 Blockrefzeigerlisten-Instanzen (tuning/xxx/blockrefptrdlist.h).....	91
2.7 Übersicht Container-Instanzen.....	91
2.7.1 Vordefinierte Templateinstanzen.....	91
2.7.2 Selbstdefinierte Templateinstanzen.....	92
2.8 Collections.....	93
2.8.1 Abstraktes Objekt (tuning/object.hpp).....	93
2.8.2 Abstrakte Collection (tuning/collection.hpp).....	94
2.8.3 Operationen mit Collections.....	95
2.8.4 Abstrakte Refcollection (tuning/refcollection.hpp).....	97
2.8.5 Konkrete Collections.....	98

3 ZEICHENKETTEN UND SYSTEMDIENSTE

100

3.1 Systemschnittstelle.....	100
3.1.1 Ressourcenfehler (tuning/sys/creserror.hpp).....	100
3.1.2 Zeichen und Zeichenketten (tuning/sys/cstring.hpp).....	101
3.1.3 Unicode (UTF) (tuning/sys/utf.hpp).....	103
3.1.4 Unicode-Const-Iterator (tuning/utfcit.h).....	104
3.1.5 Präzisionszeit (tuning/sys/ctimedate.hpp).....	106
3.1.6 Uhrzeit und Datum (tuning/sys/ctimedate.hpp).....	106
3.1.7 Prozessorzeit (tuning/sys/ctimedate.hpp).....	107
3.1.8 Taskumgebung (tuning/sys/cprocess.hpp).....	107
3.1.9 Threads (tuning/sys/cthread.hpp).....	108
3.1.10 Prozesse (tuning/sys/cprocess.hpp).....	108
3.1.11 Thread-Mutex (tuning/sys/cthmutex.hpp).....	109
3.1.12 Thread-Semaphor (tuning/sys/cthsemaphore.hpp).....	110
3.1.13 Gemeinsame Ressource (tuning/sys/csharedres.hpp).....	111
3.1.14 Prozeß-Mutex (tuning/sys/cprmutex.hpp).....	112
3.1.15 Prozeß-Semaphor (tuning/sys/cprsemaphore.hpp).....	114
3.1.16 Gemeinsamer Speicher (tuning/sys/csharedmem.hpp).....	115
3.1.17 Datei (tuning/sys/cfile.hpp).....	116
3.1.18 Verzeichnis (tuning/sys/cdir.hpp).....	118
3.1.19 Systemnahe Informationen (tuning/sys/cinfo.hpp).....	118
3.2 Zeichenketten und Dateinamen.....	121
3.2.1 Stringtemplate (tuning/string.h).....	121

3.2.2 String-Instanzen (tuning/xxx/[w]string.h).....	131
3.2.3 Polymorphe Stringklassen (tuning/[w]string.hpp).....	132
3.2.4 Dateiname (tuning/filename.hpp).....	132
3.2.5 Zeichenketten formatieren (tuning/printf.hpp).....	137
3.2.6 Zeichenketten sortieren (tuning/stringsort.hpp).....	137
3.2.7 Zahlen sortieren (tuning/stringsort.hpp).....	138
3.3 Dateien und Verzeichnisse.....	139
3.3.1 Datei (tuning/file.hpp).....	139
3.3.2 Verzeichnis (tuning/dir.hpp).....	141
3.3.3 Verzeichnis durchlaufen (tuning/dirscan.hpp).....	143
3.4 Weitere Werkzeuge.....	147
3.4.1 Uhrzeit und Datum (tuning/timedata.hpp).....	147
3.4.2 MD5 Summe (tuning/md5.hpp).....	149
3.4.3 Universally Unique Identifier (tuning/uuid.hpp).....	150

4 DESIGNDIAGRAMME

152

4.1 Zur Notation.....	152
4.2 Polymorphe Klassenhierarchie.....	153
4.3 Ein Array.....	154
4.4 Ein Zeigerarray.....	156
4.5 Eine Liste.....	158
4.6 Eine Blockliste.....	160

5 INSTALLATION UND BEISPIELE

162

5.1 Hinweise zur Installation.....	162
5.1.1 Verfügbare Plattformen.....	162
5.1.2 Abhängigkeiten.....	162
5.1.3 Installation.....	162
5.1.4 Performance-Tests.....	163
5.1.5 Inline-Methoden.....	163
5.1.6 DLL's.....	163
5.1.7 Globale Objekte.....	163
5.1.8 Multithreading.....	164
5.1.9 Exception Handling.....	164
5.2 Beispielprogramme.....	165
5.2.1 Protokollklasse (samples/int.cpp).....	165
5.2.2 Speicherüberlauf (samples/talloc.cpp).....	165
5.2.3 Alignment (samples/talign.cpp).....	166
5.2.4 Globale Stores (samples/tstore.cpp).....	166
5.2.5 Block (samples/tblock.cpp).....	167
5.2.6 Block- und Packstore (samples/tblockstore.cpp).....	167
5.2.7 Container (samples/tcontainer.cpp).....	168
5.2.8 Collections (samples/tcollection.cpp).....	168
5.2.9 [Zeiger]Mapcontainer (samples/t[ptr]map.cpp).....	169
5.2.10 Zugriffsbeschleunigung (samples/taccess.cpp).....	169
5.2.11 Exceptions in Containern (samples/texception.cpp).....	169
5.2.12 Interlocked (samples/tinterlocked.cpp).....	171
5.2.13 Threads (samples/tthread.cpp).....	171
5.2.14 Semaphoren (samples/tsemaphore.cpp).....	172
5.2.15 Prozesse (samples/texec.cpp).....	172
5.2.16 Starthilfe (samples/texechelper.cpp).....	172
5.2.17 Gemeinsame Ressourcen (samples/tshared.cpp).....	172
5.2.18 Zeichenketten (samples/tstring.cpp).....	172
5.2.19 Zeichenketten sortieren (samples/tsort.cpp).....	173
5.2.20 Dateiname (samples/tfilename.cpp).....	173

5.2.21 Datei (samples/tfile.cpp).....	174
5.2.22 Verzeichnis (samples/tdir.cpp).....	174
5.2.23 Verzeichnis durchlaufen (samples/tdirscan.cpp).....	174
5.2.24 Verzeichnisbaum (samples/ttree.cpp).....	174
5.2.25 Uhrzeit und Datum (samples/ttimedate.cpp).....	175
5.2.26 Systemnahe Informationen (samples/tinfo.cpp).....	175
5.2.27 MD5 und UUID (samples/tmd5.cpp und tuuid.cpp).....	175

1 SPEICHERVERWALTUNG

1.1 Systemschnittstelle

1.1.1 Globale Definitionen (tuning/defs.hpp)

In der Datei **'tuning/defs.hpp'** werden compilerspezifische Makros abgefragt und eigene globale Datentypen und Makros definiert. Diese Datei wird von allen anderen Headerdateien der Bibliothek **Spirick Tuning** zuerst inkludiert. Am Ende wird optional die Datei **'tl_user.hpp'** inkludiert. Damit ist es möglich, das Verhalten der Klassenbibliothek an eigene Anforderungen anzupassen ohne den Quelltext zu verändern. Z. B. kann auf diese Weise das Makro `TL_ASSERT` undefiniert werden.

Datentypen

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

Diskrete numerische Datentypen mit bestimmter Anzahl von Bits, jeweils mit oder ohne Vorzeichen. `t_Int` und `t_UInt` umfassen in einer 32-Bit-Umgebung 32 Bit und in einer 64-Bit-Umgebung 64 Bit.

1.1.2 Reservespeicher (tuning/sys/calloc.hpp)

Mit Hilfe des Reservespeichers können bei Speichermangel elementare Operationen zu Ende geführt werden, ohne daß jede einzelne Speicheranforderung geprüft werden muß. Der Reservespeicher sollte einmalig zu Programmbeginn angefordert werden. Er wird von `tl_Alloc` und `tl_Realloc` automatisch freigegeben, wenn die C-Standardbibliothek keinen Speicher mehr bereitstellen kann. Danach liefert die Funktion `tl_HasReserve` den Wert `false`. Die Verwaltung des Reservespeichers ist gegen den konkurrierenden Zugriff mehrerer Threads geschützt.

Speicherüberlauf

Innerhalb der Bibliothek **Spirick Tuning** wird an sehr vielen Stellen neuer Speicher angefordert oder vorhandener vergrößert. An jeder einzelnen Stelle im Programmcode kann ein Speicherüberlauf eintreten. Eine Behandlung des Speicherüberlaufs in jedem Einzelfall würde den Programmcode stark vergrößern und den Rechenzeitbedarf erhöhen. Ein Speicherüberlauf ist jedoch ein Ausnahmefall, der in der Praxis selten auftritt. Die Bibliothek **Spirick Tuning** ist darauf optimiert, im Normalfall eine bestmögliche Performance zu erzielen. Deshalb wird der Speicherüberlauf an einer zentralen Stelle, in den Funktionen `tl_Alloc` und `tl_Realloc`, behandelt. **Alle anderen Programmteile gehen davon aus, daß eine angeforderte Speicheroperation korrekt ausgeführt wurde.**

Bei einer Speicheranforderung oder dem Vergrößern eines vorhandenen Speicherblocks laufen nacheinander die folgenden Schritte ab. Zunächst wird versucht, die Speicheroperation mit Hilfe der C-Standardbibliothek (`malloc`, `realloc`) auszuführen. Gelingt es nicht, wird der (eventuell vorhandene) Reservespeicher freigegeben, und die C-Standardbibliothek wird erneut aufgerufen. Liegt kein positives Resultat vor, wird der Overflowhandler aufgerufen. Sollte anschließend die Speicheranforderung von der

C-Standardbibliothek immer noch nicht erfüllt werden können, kann das Programm nicht weiterarbeiten. Jede weitere Operation, z. B. das Schreiben in eine Log-Datei oder das Anzeigen einer Dialogbox, würde wahrscheinlich fehlschlagen, da kein Speicher mehr zur Verfügung steht. Deshalb wird das Programm ohne den Aufruf der Destruktoren globaler Objekte mit der Funktion `tl_EndProcess` beendet.

Datentypen

```
typedef void (* tpf_AllocHandler) ();
```

Zeiger auf eine globale Funktion, die keinen Rückgabewert und keine Parameter besitzt.

Funktionen

```
tpf_AllocHandler tl_SetReserveHandler (tpf_AllocHandler pf_allocHandler);
```

Setzt die globale Funktion für den Reservehandler und liefert die Adresse des vorher eingestellten Reservehandlers. Der Reservehandler wird stets aufgerufen, wenn sich der Reservespeicher verändert hat, also wenn Reservespeicher angefordert oder freigegeben wurde oder wenn sich seine Größe verändert hat.

```
tpf_AllocHandler tl_SetOverflowHandler (tpf_AllocHandler pf_allocHandler);
```

Setzt die globale Funktion für den Overflowhandler und liefert die Adresse des vorher eingestellten Overflowhandlers. Der Overflowhandler wird aufgerufen, wenn kein Reservespeicher mehr zur Verfügung steht und eine Speicheranforderung von der C-Standardbibliothek nicht erfüllt werden konnte. Innerhalb der Bibliothek **Spirick Tuning** wird der Speicherüberlauf an einer zentralen Stelle, in den Funktionen `tl_Alloc` und `tl_Realloc`, behandelt. **Alle anderen Programmteile gehen davon aus, daß eine angeforderte Speicheroperation korrekt ausgeführt wurde.** Deshalb darf im Overflowhandler keine Exception ausgelöst werden. Diese Exception würde in der Bibliothek **Spirick Tuning** nicht behandelt werden und dazu führen, daß das Objekt, das gerade Speicher angefordert hat, in einem inkonsistenten Zustand verbleibt.

```
void tl_SetReserveSize (t_UInt u_resSize);
```

Setzt die neue Größe des Reservespeichers auf `u_resSize`. Anschließend kann mit `tl_HasReserve` gefragt werden, ob Reservespeicher mit der neuen Größe bereitgestellt werden konnte.

```
t_UInt tl_GetReserveSize ();
```

Liefert die Größe des Reservespeichers. Der Rückgabewert ist unabhängig davon, ob gerade Reservespeicher bereitsteht oder nicht.

```
bool tl_HasReserve ();
```

Liefert `true`, wenn Reservespeicher bereitsteht.

```
void tl_FreeReserve ();
```

Gibt den Reservespeicher frei. Anschließend liefert `tl_HasReserve` den Wert `false`.

```
void tl_AllocReserve ();
```

Versucht, den Reservespeicher anzufordern. Anschließend kann mit `tl_HasReserve` gefragt werden, ob Reservespeicher bereitsteht.

1.1.3 Dynamischer Speicher (tuning/sys/calloc.hpp)

Die Systemschnittstelle baut direkt auf der C-Standardbibliothek auf. Sie verwendet die globalen Funktionen `malloc`, `realloc` und `free`. Debughilfen und Heapwalker der C-Standardbibliothek können uneingeschränkt weitergenutzt werden. Die Funktionen `tl_Alloc` und `tl_Realloc` nutzen darüberhinaus den Reservespeicher. Kann z. B. `malloc` keinen Speicher mehr liefern, gibt `tl_Alloc` den Reservespeicher frei und ruft `malloc` erneut auf.

Funktionen

`t_UInt t1_StoreInfoSize ();`

Liefert die Anzahl Bytes Verwaltungsspeicher pro Speicherblock. Dieser Wert wird für die Berechnung gerundeter Blockgrößen benötigt.

`t_UInt t1_MaxAlloc ();`

Liefert die maximale Anzahl Bytes, die zusammenhängend bereitgestellt werden können, d. h. die maximale Größe eines einzelnen Speicherblocks.

`void * t1_Alloc (t_UInt u_size);`

Stellt einen zusammenhängenden Speicherblock der Größe `u_size` bereit. Ist `u_size` gleich Null, wird Null zurückgegeben. Bei Speicherüberlauf werden Reservehandler und Overflowhandler aufgerufen (siehe Abschnitt 'Reservespeicher').

`void * t1_Realloc (void * pv_ptr, t_UInt u_size);`

Verändert die Größe des Speicherblocks `pv_ptr` auf `u_size`. Bei `pv_ptr` gleich Null ist `t1_Realloc` identisch mit `t1_Alloc`. Bei `u_size` gleich Null ist `t1_Realloc` identisch mit `t1_Free`. Bei Speicherüberlauf werden Reservehandler und Overflowhandler aufgerufen (siehe Abschnitt 'Reservespeicher').

`void t1_Free (void * pv_ptr);`

Gibt den Speicherblock `pv_ptr` frei. Der Wert `pv_ptr` gleich Null ist erlaubt.

Zugehörige Klassen

Die globalen Funktionen dieser Schnittstelle dienen als Grundlage der Klassen `ct_StdStore`, `ct_RndStore` und `ct_ChnStore`.

1.1.4 Heapoperationen (tuning/sys/calloc.hpp)

Debughilfen und Heapwalker sind leider nicht standardisiert. Deshalb enthält die Systemschnittstelle nur einige ausgewählte Heapinformationen. Die Struktur `st_HeapInfo` enthält die Anzahl und die Gesamtgröße genutzter und ungenutzter Speicherblöcke sowie die Gesamtgröße des Heaps. Die ungenutzten Speicherblöcke sind ein Maß für die Speicherfragmentierung. Die Heapgröße gibt Auskunft über den Gesamtspeicherbedarf des Programms. Zu beachten ist, daß MS Visual C++ keine Informationen über die Freiliste liefert.

Strukturdeklaration

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

Funktionen

`bool t1_QueryHeapInfo (st_HeapInfo * pso_info);`

Speichert in `pso_info` Daten über den aktuellen Zustand des Heaps. Wurden im Heap keine Fehler gefunden, liefert die Funktion den Wert `true`. Der Rückgabewert `false` deutet auf Inkonsistenzen im Heap hin.

```
bool t1_FreeUnused ();
```

Versucht, ungenutzten Freispeicher an das Betriebssystem zurückzugeben. Der Rückgabewert `false` deutet auf Inkonsistenzen im Heap hin.

1.1.5 Speicheroperationen (tuning/sys/cmemory.hpp)

Die Systemschnittstelle für Speicheroperationen baut direkt auf der C-Standardbibliothek auf. Sie verwendet globale Funktionen wie `memcpy` und `memcmp`. Zusätzlich werden einige Sonderfälle berücksichtigt, die von der C-Standardbibliothek nicht immer korrekt behandelt werden. Ist z. B. die Länge einer Operation gleich Null, können die Zeigerparameter ungültige Werte enthalten. Alle Parameter werden mit `ASSERT`-Makros überprüft. Von allen Funktionen existiert jeweils eine Version für die Datentypen `char` und `wchar_t`. Alle Längenangaben beziehen sich auf die Anzahl der Zeichen und nicht auf die Anzahl der Bytes.

Funktionen

```
void t1_CopyMemory (char * pc_dst, const char * pc_src, t_UInt u_len);  
void t1_CopyMemory (wchar_t * pc_dst, const wchar_t * pc_src, t_UInt u_len);
```

Kopiert `u_len` Zeichen von `pc_src` nach `pc_dst`. Diese Funktion ist *nicht* für überlappende Speicherbereiche geeignet.

```
void t1_MoveMemory (char * pc_dst, const char * pc_src, t_UInt u_len);  
void t1_MoveMemory (wchar_t * pc_dst, const wchar_t * pc_src, t_UInt u_len);
```

Kopiert `u_len` Zeichen von `pc_src` nach `pc_dst`. Diese Funktion ist auch für überlappende Speicherbereiche geeignet.

```
char * t1_FillMemory (char * pc_dst, t_UInt u_len, char c_fill);  
wchar_t * t1_FillMemory (wchar_t * pc_dst, t_UInt u_len, wchar_t c_fill);
```

Füllt `u_len` Zeichen beginnend bei `pc_dst` mit dem Zeichen `c_fill`.

```
int t1_CompareChar (char c1, char c2);  
int t1_CompareChar (wchar_t c1, wchar_t c2);
```

Vergleicht die Zeichen `c1` und `c2` miteinander. Das Resultat ist bei `c1 < c2` kleiner Null, bei `c1 == c2` gleich Null und bei `c1 > c2` größer Null. Die beiden Zeichen werden ohne Vorzeichen miteinander verglichen. Z. B. gilt `'\x40' < '\xC0'`.

```
int t1_CompareMemory (const char * pc1, const char * pc2, t_UInt u_len);  
int t1_CompareMemory (const wchar_t * pc1, const wchar_t * pc2, t_UInt u_len);
```

Vergleicht die ersten `u_len` Zeichen der Speicherbereiche `pc1` und `pc2`. Das Resultat ist bei `pc1 < pc2` kleiner Null, bei `pc1 == pc2` gleich Null und bei `pc1 > pc2` größer Null. Die einzelnen Zeichen werden ohne Vorzeichen miteinander verglichen. Z. B. gilt `"\x40" < "\xC0"`.

```
const char * t1_FirstChar (const char * pc_mem, t_UInt u_len, char c_search);  
const wchar_t * t1_FirstChar (const wchar_t * pc_mem, t_UInt u_len, wchar_t c_search);
```

Sucht in den ersten `u_len` Zeichen des Speicherbereiches `pc_mem` nach dem ersten Auftreten des Zeichens `c_search`. Wurde das Zeichen nicht gefunden, ist der Rückgabewert gleich Null.

```
const char * t1_FirstMemory (const char * pc_mem, t_UInt u_len, const char * pc_search, t_UInt u_searchLen);  
const wchar_t * t1_FirstMemory (const wchar_t * pc_mem, t_UInt u_len, const wchar_t * pc_search, t_UInt  
u_searchLen);
```

Sucht in den ersten `u_len` Zeichen des Speicherbereiches `pc_mem` nach dem ersten Auftreten der Zeichenfolge `pc_search`, die `u_searchLen` Zeichen lang ist. Wurde die Zeichenfolge nicht gefunden, ist der Rückgabewert gleich Null.

```
const char * t1_LastChar (const char * pc_mem, t_UInt u_len, char c_search);
const wchar_t * t1_LastChar (const wchar_t * pc_mem, t_UInt u_len, wchar_t c_search);
```

Sucht in den ersten `u_len` Zeichen des Speicherbereiches `pc_mem` nach dem letzten Auftreten des Zeichens `c_search`. Wurde das Zeichen nicht gefunden, ist der Rückgabewert gleich Null.

```
const char * t1_LastMemory (const char * pc_mem, t_UInt u_len, const char * pc_search, t_UInt u_searchLen);
const wchar_t * t1_LastMemory (const wchar_t * pc_mem, t_UInt u_len, const wchar_t * pc_search, t_UInt u_searchLen);
```

Sucht in den ersten `u_len` Zeichen des Speicherbereiches `pc_mem` nach dem letzten Auftreten der Zeichenfolge `pc_search`, die `u_searchLen` Zeichen lang ist. Wurde die Zeichenfolge nicht gefunden, ist der Rückgabewert gleich Null.

```
template <t_UInt u_len>
void t1_SwapMemory (void * pv1, void * pv2);
```

Tauscht den Inhalt der Speicherbereiche `pv1` und `pv2` mit der Länge `u_len` Bytes aus.

```
template <class t_obj>
void t1_SwapObj (t_obj & o1, t_obj & o2);
```

Tauscht den Wert der Objekte `o1` und `o2` durch dreimaliges Aufrufen von `operator =` aus. Dabei wird ein drittes lokales Objekt verwendet.

Zugehörige Klassen

Die globalen Funktionen dieser Schnittstelle dienen als Grundlage der Templates `gct_CharBlock` und `gct_String`.

1.2 Store

1.2.1 Storeschnittstelle

Stores sind Speicherverwaltungsobjekte. Zur Erhöhung der Flexibilität und Performance besitzen sie keine gemeinsame Basisklasse mit virtuellen Methoden. Sie verfügen jedoch über eine einheitliche Schnittstelle. Diese vereinfacht die Handhabung und ermöglicht das leichte Austauschen eines Stores gegen einen anderen. Es werden nicht alle Methoden von allen Stores unterstützt. Damit beim Verwenden einer Storeklasse als Templateparameter keine Syntaxfehler auftreten, enthält die Deklaration der Klasse auch nicht unterstützte Methoden. Diese enthalten jedoch in ihrer Definition die Anweisung `ASSERT (false)`.

Klassendeklaration

```
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 ();
};

```

Datentypen

```
typedef t_UInt t_Size;
```

Der geschachtelte Typ `t_Size` beschreibt die Größe der Speicherblöcke, die der Store verwalten kann. Neben `t_UInt` werden auch `t_UInt8`, `t_UInt16` und `t_UInt32` verwendet. Ist z. B. `t_Size` auf `t_UInt8` definiert, kann ein Speicherblock maximal 255 Bytes umfassen. Ein angepaßter Größentyp verringert den Speicherbedarf von Objekten, die Größenangaben enthalten.

```
typedef void * t_Position;
```

Stores verwalten ihre Speicherblöcke mit Hilfe von Positionszeigern. Neben `void *` werden auch `t_UInt`, `t_UInt8`, `t_UInt16` und `t_UInt32` verwendet. Bei allen Positionstypen ist der Wert Null per Definition ungültig. Der Zugriff auf den Speicher erfolgt i. a. mit der Methode `AddrOf`. Bei einigen Stores, die `void *` als Positionstyp verwenden, ist ein Positionszeiger gleich dem physischen Zeiger. Auch in diesen Fällen sollte die Methode `AddrOf` verwendet werden, denn sie ist inline definiert und benötigt keine zusätzliche Rechenzeit.

Methoden

```
void Swap (ct_AnyStore & co_swap);
```

Tauscht den Inhalt der beiden Objekte aus.

```
t_UInt StoreInfoSize ();
```

Liefert die Anzahl Bytes Verwaltungsspeicher pro Speicherblock. Diese Methode wird nicht von allen Stores unterstützt.

```
t_UInt MaxAlloc ();
```

Liefert die maximale Anzahl Bytes, die zusammenhängend bereitgestellt werden können, d. h. die maximale Größe eines einzelnen Speicherblocks.

```
t_Position Alloc (t_Size o_size);
```

Stellt einen zusammenhängenden Speicherblock der Größe `o_size` bereit. Ist `o_size` gleich Null, wird Null zurückgegeben. Bei Speicherüberlauf werden Reservehandler und Overflowhandler aufgerufen (siehe Abschnitt 'Reservespeicher').

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

Verändert die Größe des Speicherblocks `o_pos` auf `o_size`. Bei `o_pos` gleich Null ist `Realloc` identisch mit `Alloc`. Bei `o_size` gleich Null ist `Realloc` identisch mit `Free`. Bei Speicherüberlauf werden Reservehandler und Overflowhandler aufgerufen (siehe Abschnitt 'Reservespeicher').

```
void Free (t_Position o_pos);
```

Gibt den Speicherblock `o_pos` frei. Der Wert `o_pos` gleich Null ist erlaubt.

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

Berechnet die zum Positionszeiger `o_pos` gehörende Speicheradresse. Bei `o_pos` gleich Null liefert `AddrOf` den Nullzeiger.

`t_Position PosOf (void * pv_adr);`

Berechnet den zur Speicheradresse `pv_adr` gehörenden Positionszeiger. Diese Methode wird nicht von allen Stores unterstützt.

`t_Size SizeOf (t_Position o_pos);`

Berechnet die exakte Größe des Speicherbereichs, auf den `o_pos` zeigt, d. h. die Größe, die bei `Alloc` oder `Realloc` angegeben wurde. Diese Methode wird nicht von allen Stores unterstützt.

`t_Size RoundedSizeOf (t_Position o_pos);`

Berechnet die aufgerundete interne Größe des Speicherbereichs, auf den `o_pos` zeigt. Diese Methode wird nicht von allen Stores unterstützt.

`bool CanFreeAll ();`

Liefert `true`, wenn der Store sämtlichen Speicher, der von ihm angefordert wurde, zusammenhängend freigeben kann.

`void FreeAll ();`

Gibt sämtlichen Speicher, der vom Store angefordert wurde, frei. Diese Methode wird nicht von allen Stores unterstützt.

1.2.2 Globale Stores (tuning/defs.hpp)

Stores werden innerhalb der Bibliothek **Spirick Tuning** sehr unterschiedlich eingesetzt. Von den drei dynamischen Stores (siehe folgende Abschnitte) wird je eine globale Instanz erzeugt, auf die mit generierten Wrapperklassen zugegriffen wird. Z. B. reicht es in den meisten Fällen aus, vom Roundstore nur eine einzelne Instanz zu bilden. Die eingestellten Parameter zum Runden der Blockgröße gelten dann für das gesamte Programm.

Zahlreiche Templates erwarten als Parameter eine Storeklasse und bilden eine Instanz davon. Z. B. enthält jeder Listencontainer ein Storeobjekt, von dem er den Speicher für seine Nodes anfordert. Eine Blockliste enthält einen 'echten' Store (einen Blockstore). Eine 'normale' Liste nutzt jedoch ein globales Storeobjekt und greift mit Hilfe eines Wrapperobjektes darauf zu.

Zu jedem globalen Storeobjekt werden vier Wrapperklassen generiert. Diese unterscheiden sich nur durch den Größentyp `t_Size`. Alle Methoden einer Wrapperklasse sind `static` deklariert. Sie können entweder direkt (z. B. bei `get_Block`) oder über ein Wrapperobjekt (z. B. bei `get_DList`) aufgerufen werden.

Eine Wrapperklasse mappt ihre eigenen Methoden auf Methodenaufrufe des globalen Objekts. Sind für eine Storeklasse die Positionszeiger gleich den physischen Zeigern, kann der Zugriff auf den Speicher über die Wrapperklasse beschleunigt werden. Anstatt die Methode `AddrOf` des globalen Objekts aufzurufen, kann die Methode `AddrOf` der Wrapperklasse inline definiert werden.

Auf das generierte globale Storeobjekt kann mit einer `Get`-Funktion direkt zugegriffen werden. Das Objekt wird nicht als ein globales C++-Objekt erzeugt, sondern beim ersten Zugriff über die `Get`-Funktion oder beim Starten des ersten Threads. Dadurch ist der globale Store unabhängig von der Reihenfolge der Initialisierung globaler Objekte. Andere globale Objekte besitzen in ihrem Konstruktor einen sicheren Zugriff auf globale Stores. Bei Bedarf können globale Stores mit einer `Create`-Funktion explizit erzeugt werden.

Globale Stores werden nicht automatisch zerstört. Dadurch können die Destruktoren anderer globaler Objekte noch sicher auf angeforderten Speicher zugreifen. Das Zerstören globaler Stores ist nicht notwendig, denn sie verwalten nur rohe Speicherblöcke, die am Programmende vom Betriebssystem automatisch freigegeben werden. Bei Bedarf können globale Stores mit einer `Delete`-Funktion explizit zerstört werden.

GLOBAL_STORE_DCLS(t_store, Obj, inl_or_stat)

Dieses Makro wird am Ende der Headerdatei der Storeklasse platziert. Der Parameter `t_store` enthält die ursprüngliche Storeklasse. `Obj` ist ein Namens Kürzel für die generierten Namen. Es erlaubt das Generieren mehrerer globaler Instanzen einer Storeklasse. Der Parameter `inl_or_stat` legt fest, ob die Methoden `AddrOf` und `PosOf` der Wrapperklasse `inline` oder `static` deklariert werden sollen. Die Makroverwendung

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

enthält die folgenden Deklarationen (Makroparameter sind fett hervorgehoben):

```
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)

Dieses Makro wird in der Implementierungsdatei der Storeklasse platziert. Es erwartet dieselben Parameter wie `GLOBAL_STORE_DCLS` und generiert die Definition der Methoden.

1.2.3 Beispiel für eine Wrapperklasse

Die vollständige Deklaration der Wrapperklasse `ct_My16Store` aus dem vorigen Beispiel lautet:

```
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 ();
};
```

Im Makro `GLOBAL_STORE_DEFS` werden drei Funktionen für das globale Objekt definiert:

```
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;
}

```

Die generierte Definition der Methode `ct_My16Store::Alloc` lautet:

```

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

```

1.3 Dynamische Stores

1.3.1 Standardstore (tuning/std/store.hpp)

Die Klasse `ct_StdStore` enthält keine eigene Funktionalität. Sie mappt die globalen Funktionen der Systemschnittstelle auf Methoden der Storeschnittstelle. Als Beispiel folgt der Klassendeklaration die Definition der Methode `Alloc`:

Klassendeklaration

```

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); }

```

Besonderheiten, Wrapperklassen

Die folgenden Methoden werden vom Standardstore nicht unterstützt: `SizeOf`, `RoundedSizeOf` und `FreeAll`. Da die Klasse `ct_StdStore` auf der Systemschnittstelle aufbaut, nutzt sie indirekt auch die Funktionalität des Reservespeichers. Z. B. kann mit der globalen Funktion `tl_HasReserve` gefragt werden, ob noch Reservespeicher bereit steht. Jede Speicheranforderung führt über die Systemschnittstelle zu einem

Aufruf von `malloc`. Debughilfen und Heapwalker der C-Standardbibliothek können uneingeschränkt weitergenutzt werden.

In der Headerdatei des Standardstores werden Funktionen für das globale Objekt und vier Wrapperklassen deklariert:

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

1.3.2 Roundstore (tuning/rnd/store.hpp)

Die Klasse `ct_RndStore` nutzt ähnlich wie `ct_StdStore` die Systemschnittstelle, rundet jedoch alle Größenangaben, bevor sie an die globalen Funktionen weitergegeben werden. Die Berechnung der gerundeten Werte erfolgt in der privaten Methode `Round`.

Klassendeklaration

```
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)); }
```

Die Rundung der Blockgrößen wirkt der Speicherfragmentierung entgegen. Dadurch verkleinert sich der ungenutzte Freispeicher, und die Speicherverwaltung wird spürbar schneller. Zur Berechnung der gerundeten Werte dienen zwei Parameter, die Mindestgröße und der Schritt-Teiler. Der Schritt-Teiler steuert die Granularität der Rundung. Je kleiner der Schritt-Teiler ist, desto grober wird die Rundung. Beim Schritt-Teiler Eins werden alle Größen auf Zweierpotenzen gerundet. Der Schritt-Teiler `n` lässt

zwischen zwei aufeinanderfolgenden Zweierpotenzen n Werte zu (einschließlich der Grenze). Ein größerer Schritt-Teiler führt also zu einer feineren Rundung.

Je stärker die Belastung der Speicherverwaltung ist, desto grober sollte die Rundung sein. Bei geringer Belastung tritt kaum eine Fragmentierung ein. Die Speicherverwaltung kann wenig beschleunigt werden, und es genügt der Schritt-Teiler Vier oder Acht. Bei stärkerer Belastung sollte ein kleinerer Schritt-Teiler (Zwei oder Eins) gewählt werden. Der Speicherplatz, der durch die grobe Rundung verlorengelassen wird, wird durch die geringere Fragmentierung ausgeglichen, und die Speicherverwaltung wird spürbar schneller. Bei einer sehr starken Belastung des dynamischen Speichers sollte der Chainstore verwendet werden. Er besitzt einen deutlich höheren Wirkungsgrad als der Roundstore.

Der Wirkungsgrad des Roundstores hängt stark von der Implementierung der C-Standardbibliothek ab. Enthält diese bereits eigene Rundungsmechanismen, fällt die Nachbereitung durch den Roundstore weniger ins Gewicht. Der Roundstore führt bei vielen älteren Compilern zu einem meßbaren Geschwindigkeitsgewinn gegenüber dem Standardstore.

Neben den allgemeinen Storemethoden enthält die Klasse `ct_RndStore` noch Zugriffsmethoden für die Rundungsparameter. Da vom Roundstore ein globales Objekt gebildet wird, sind die Rundungsparameter gegen den konkurrierenden Zugriff mehrerer Threads geschützt.

Zusätzliche Methoden

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

Liest bzw. setzt die Minimalgröße für die Rundung. Zur Beschleunigung der Berechnung muß der Parameter `o_minSize` eine Zweierpotenz sein.

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

Liest bzw. setzt den Schritt-Teiler für die Rundung. Zur Beschleunigung der Berechnung muß der Parameter `u_stepDiv` eine Zweierpotenz sein.

Besonderheiten, Wrapperklassen

Die folgenden Methoden werden vom Roundstore nicht unterstützt: `SizeOf`, `RoundedSizeOf` und `FreeAll`. Da die Klasse `ct_RndStore` auf der Systemschnittstelle aufbaut, nutzt sie indirekt auch die Funktionalität des Reservespeichers. Z. B. kann mit der globalen Funktion `tl_HasReserve` gefragt werden, ob noch Reservespeicher bereit steht. Jede Speicheranforderung führt über die Systemschnittstelle zu einem Aufruf von `malloc`. Debughilfen und Heapwalker der C-Standardbibliothek können uneingeschränkt weitergenutzt werden.

In der Headerdatei des Roundstores werden Funktionen für das globale Objekt und vier Wrapperklassen deklariert:

```
void CreateRndStore ();
void DeleteRndStore ();
ct_RndStore * GetRndStore ();
class ct_Rnd_Store;
class ct_Rnd8Store;
class ct_Rnd16Store;
class ct_Rnd32Store;
```

1.3.3 Chainstore (tuning/chn/store.hpp)

Die Klasse `ct_ChnStore` ist eine Weiterentwicklung des Roundstores. Der Chainstore ist auf Programme mit starker Belastung der Speicherverwaltung ausgerichtet. Er enthält eine Optimierungstechnologie für maximale Geschwindigkeit. Sie wirkt auch der Speicherfragmentierung effektiv entgegen und benötigt in

einigen Fällen etwas weniger, in anderen Fällen bis zu 25% mehr Gesamtspeicher als der Standardstore. Der Chainstore bringt Programmen mit geringer Belastung der Speicherverwaltung keine Nachteile und ist universell einsetzbar.

Klassendeklaration

```
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 ();
};
```

Der Chainstore rundet ähnlich wie der Roundstore beim Schritt-Teiler Eins alle Anforderungen auf die nächst höhere Zweierpotenz. Normalerweise bildet eine dynamische Speicherverwaltung eine lineare Liste der Freielemente. Bei jeder neuen Speicheranforderung wird diese Liste durchlaufen, bis ein passendes Element gefunden wird. Die Suche benötigt bei zunehmender Speicherfragmentierung (längere Liste) immer mehr Rechenzeit. Die Rundung führt jedoch zu wesentlich weniger möglichen Blockgrößen. Damit wächst die Wahrscheinlichkeit, sehr schnell einen passenden Block zu finden.

Der Chainstore besitzt zusätzlich eine eigene Verwaltung des Freispeichers. Er legt für jede einzelne Blockgröße eine eigene Liste (Chain) der Freielemente an. Bei einer neuen Speicheranforderung greift er direkt auf die passende Freiliste zu. Existiert dort ein Element, wird es aus der Liste entfernt. Andernfalls wird über die Systemschnittstelle mit der Funktion `tl_Alloc` neuer Speicher angefordert.

Wird ein Speicherblock an den Chainstore zurückgegeben, reicht dieser ihn nicht sofort an die C-Standardbibliothek weiter, sondern ordnet ihn der passenden eigenen Freiliste zu. Dort steht der Speicherblock für neue Anforderungen direkt zur Verfügung. Zum Ermitteln der Blockgröße benötigt der Chainstore acht zusätzliche Bytes pro Block. Am Anfang jedes Speicherblocks wird seine exakte und gerundete Größe untergebracht. Mit Hilfe dieser Zusatzinformationen kann der Chainstore die Methoden `SizeOf` und `RoundedSizeOf` der allgemeinen Storeschnittstelle unterstützen, und sie ermöglichen eine Buchführung über die genutzten und freien Blöcke.

Die eigene Verwaltung des Freispeichers ist besonders effektiv bei speicherintensiven Rechenvorgängen, die etwa gleichviel Speicher freigeben und wieder anfordern. Wurde jedoch wesentlich mehr Speicher freigegeben als neu angefordert (z. B. beim Schließen eines Dokuments in einer interaktiven Anwendung), besitzt der Chainstore unnötig große Freilisten. Am Ende des Vorgangs sollte die Methode `FreeUnused` aufgerufen werden. Sie leert sämtliche Freilisten und gibt deren Elemente mit der Funktion `tl_Free` an die C-Standardbibliothek zurück.

Mit zunehmender Blockgröße wird die Wahrscheinlichkeit der Speicherfragmentierung immer geringer. Gleichzeitig erhöht sich die Wahrscheinlichkeit, daß sich in den Freilisten unnötig viel ungenutzter Speicher befindet. Z. B. ist es bei einem Gesamtspeicher von 4 GB unwahrscheinlich, daß Blöcke mit einer Größe von 100 MB oder mehr fragmentieren. Wenn mehrere Blöcke der Größe 100 MB ungenutzt in einer Freiliste auf eine neue Verwendung warten, erhöht sich unnötig der Gesamtspeicherbedarf. Deshalb kann man im Chainstore die maximale Größe für die Verwendung von Freilisten einstellen. Blöcke mit einer Größe oberhalb dieser Grenze werden bei der Freigabe nicht in die zugehörige Freiliste einsortiert, sondern mit der Funktion `tl_Free` direkt an die C-Standardbibliothek zurückgegeben. Der Chainstore funktioniert oberhalb dieser Grenze also ähnlich wie der Roundstore beim Schritt-Teiler Eins.

Neben den allgemeinen Storemethoden enthält die Klasse `ct_ChnStore` noch Buchführungsmethoden. Da vom Chainstore ein globales Objekt gebildet wird, sind seine privaten Attribute gegen den konkurrierenden Zugriff mehrerer Threads geschützt.

Zusätzliche Methoden

`unsigned GetMaxChainExp ();`

Liefert den maximalen Exponenten für Freilisten.

`void SetMaxChainExp (unsigned u_exp);`

Setzt den maximalen Exponenten für Freilisten. Die maximale Größe für Blöcke in Freilisten wird nicht als Bytegröße, sondern als Exponent angegeben. Z. B. bedeutet der Exponent 10, daß alle Blöcke, die größer als 2^{10} (1 KB) sind, nicht in Freilisten einsortiert werden. Der Defaultwert ist 22 (4 MB).

`t_UInt GetEntries ();`

Liefert die Gesamtzahl der genutzten und ungenutzten Speicherblöcke, die von dieser Instanz des Chainstores verwaltet werden.

`t_UInt GetSize ();`

Liefert die Gesamtgröße der genutzten und ungenutzten Speicherblöcke.

`t_UInt QueryAllocEntries ();`

Berechnet die Anzahl der genutzten Speicherblöcke.

`t_UInt QueryAllocSize ();`

Berechnet die Gesamtgröße der genutzten Speicherblöcke.

`t_UInt QueryFreeEntries ();`

Berechnet die Anzahl der ungenutzten Speicherblöcke.

`t_UInt QueryFreeSize ();`

Berechnet die Gesamtgröße der ungenutzten Speicherblöcke.

`void FreeUnused ();`

Leert alle Freilisten und gibt deren Speicher an die C-Standardbibliothek zurück.

Besonderheiten, Wrapperklassen

Die Methode `FreeAll` wird vom Chainstore nicht unterstützt. Da die Klasse `ct_ChnStore` auf der Systemschnittstelle aufbaut, nutzt sie indirekt auch die Funktionalität des Reservespeichers. Z. B. kann mit der globalen Funktion `tl_HasReserve` gefragt werden, ob noch Reservespeicher bereit steht. Jede Speicheranforderung führt über die Systemschnittstelle zu einem Aufruf von `malloc`. Debughilfen und Heapwalker der C-Standardbibliothek können uneingeschränkt weitergenutzt werden. Es ist jedoch zu beachten, daß Elemente der eigenen Freilisten des Chainstores beim Durchlaufen des Heaps nicht als frei, sondern als genutzt erscheinen, und daß sich am Anfang jedes Blocks acht Bytes Zusatzinformationen befinden.

In der Headerdatei des Chainstores werden Funktionen für das globale Objekt und vier Wrapperklassen deklariert:

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

1.3.4 Operatoren `new` und `delete` (`tuning/newdel.cpp`)

Um die Vorteile des Chainstores der globalen C++-Speicherverwaltung zur Verfügung zu stellen, werden in der Datei '`tuning/newdel.cpp`' die globalen Operatoren `new` und `delete` überschrieben. Sie greifen auf das globale Chainstore-Objekt zu. Unterstützt der verwendete Compiler die Operatoren `new []` und `delete []`, werden auch diese überschrieben.

```
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 Blockschnittstelle

Zahlreiche Klassen der Bibliothek **Spirick Tuning** verwenden dynamische Speicherblöcke zur Unterbringung ihrer Daten. Ihre gemeinsame Grundlage ist das Blockkonzept. Ein Block ist ein Objekt, das einen Speicherbereich dynamischer Größe verwaltet. Ähnlich wie Storeklassen besitzen auch Blockklassen keine gemeinsame Basisklasse mit virtuellen Methoden, aber eine einheitliche Schnittstelle.

Diese vereinfacht die Handhabung und ermöglicht das leichte Austauschen eines Blocks gegen einen anderen. Blockklassen dienen als Templateparameter für Strings, Arrays und Blockstores.

Klassendeklaration

```
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;
};
```

Datentypen

```
typedef t_UInt t_Size;
```

Der geschachtelte Größentyp einer Blockklasse bestimmt den Wertebereich der Größen- und Positionsangaben. Außer `t_UInt` werden auch `t_UInt8`, `t_UInt16` und `t_UInt32` verwendet. Ist z. B. der Größentyp auf `t_UInt8` definiert, kann der dynamische Speicherbereich maximal 255 Bytes umfassen. Der Größentyp beeinflusst auch die Größe des Blockobjekts, denn die meisten Blockklassen enthalten ein Attribut des Typs `t_Size`.

Konstruktoren, Destruktor, Gleichoperator, Swap

Blockobjekte werden häufig kopiert. Deshalb enthält jede Blockklasse einen Konstruktor, Kopierkonstruktor, Destruktor und Gleichoperator. Viele Anwender von Blockklassen verlassen sich auf das einwandfreie Funktionieren dieser Methoden.

```
ct_AnyBlock ();
```

Initialisiert ein leeres Blockobjekt (Größe Null).

```
ct_AnyBlock (const ct_AnyBlock & co_init);
```

Initialisiert ein Blockobjekt durch Kopieren des Inhalts von `co_init`. Es wird eine echte Kopie (deep copy) angefertigt. Der Inhalt von `co_init` wird in einen eigenen Speicherbereich kopiert.

```
~ct_AnyBlock ();
```

Gibt den belegten Speicher frei.

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

Weist dem Blockobjekt einen neuen Inhalt zu. Es wird eine echte Kopie (deep copy) angefertigt. Nach der Anpassung der Größe wird der Inhalt von `co_asgn` in den eigenen Speicherbereich kopiert.

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

Tauscht den Inhalt der beiden Objekte aus.

Weitere Methoden

```
static t_UInt GetMaxByteSize ();
```

Liefert die maximale Größe des dynamischen Speicherbereichs.

```
t_Size GetByteSize () const;
```

Liefert die Größe des dynamischen Speicherbereichs.

```
void SetByteSize (t_Size o_newSize);
```

Setzt die Größe des dynamischen Speicherbereichs auf `o_newSize`. Der Wert Null ist erlaubt.

```
void * GetAddr () const;
```

Liefert die Anfangsadresse des dynamischen Speicherbereichs. Ist die Größe gleich Null, wird ein Nullzeiger zurückgegeben.

In den folgenden Abschnitten werden verschiedene Implementierungen der Blockschnittstelle vorgestellt.

1.4.2 Allgemeiner Block (tuning/block.h)

Das Klassentemplate `gct_Block` enthält die Standardimplementierung der Blockschnittstelle. Es basiert auf einer Storeklasse, in der alle Methoden `static` deklariert sind. Diese Bedingung erfüllen die Wrapperklassen für globale Storeobjekte, z. B. `ct_Rnd16Store`. Die Implementierung besteht aus der Basisklasse `gct_BlockBase`, der eigentlichen Blockklasse `gct_Block` und den Erweiterungen `gct_EmptyBaseBlock` und `gct_ObjectBaseBlock`.

Basisklasse

Die Basisklasse `gct_BlockBase` enthält je ein Attribut der Typen `t_Position` und `t_Size` der Storeklasse. Durch einen angepassten Größentyp `t_Size` (z. B. `t_UInt16` statt `t_UInt32`) kann die Größe des Blockobjekts optimiert werden. Für die korrekte Ausrichtung der Attribute im Speicher kann der Compiler Paddingbytes einfügen. Diese Paddingbytes können nur innerhalb einer Klasse durch zusätzliche Attribute nutzbar gemacht werden. Es ist z. B. nicht möglich, in einer abgeleiteten Klasse die Paddingbytes am Ende der Basisklasse zu nutzen. Im Beispielpogramm `TBlock` werden Paddingbytes in einer modifizierten Block-Basisklasse verwendet.

Das Klassentemplate `gct_BlockBase` erwartet als Parameter `t_staticStore` eine statische Storeklasse und als Parameter `t_base` eine frei definierbare Basisklasse. Wegen der Möglichkeit zusätzlicher Attribute und der variablen Basisklasse wird die Blockmethode `Swap` nicht in `gct_Block`, sondern in `gct_BlockBase` definiert.

Templatedeklaration

```
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;
```

```
};
```

Blockklasse

Das Klassentemplate `gct_Block` erwartet als Parameter `t_blockBase` eine Klasse, die mindestens die Datentypen, Attribute und Methoden wie `gct_BlockBase` enthält.

Templatedeklaration

```
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;
};
```

Die Methoden des Klassentemplates `gct_Block` enthalten nur wenige Anweisungen und sind durchgängig inline definiert. Da die Methoden der Storeklasse static deklariert sind, werden sie direkt aufgerufen. Der Rechenzeitbedarf der Blockmethoden ist sehr gering.

```
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);
}
```

Erweiterungen

Als oberste Basisklasse können z. B. die leere Klasse `ct_Empty` oder `ct_Object` verwendet werden. Dafür existieren die beiden Erweiterungen `gct_EmptyBaseBlock` und `gct_ObjectBaseBlock`.

Templatedeklaration

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

Templatedeklaration

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

1.4.3 Miniblock (tuning/miniblock.h)

Eine aus dem Template `gct_Block` abgeleitete Klasse enthält ein Größen- und ein Positionsattribut. Unterstützt der zugrunde gelegte Store die Methode `SizeOf`, ist das Größenattribut redundant. Diese Eigenschaft wurde im Template `gct_Miniblock` berücksichtigt. Die Implementierung besteht ähnlich wie bei `gct_Block` aus der Basisklasse `gct_MiniblockBase`, der eigentlichen Blockklasse `gct_Miniblock` und den Erweiterungen `gct_EmptyBaseMiniblock` und `gct_ObjectBaseMiniblock`.

Basisklasse

Das Klassentemplate `gct_MiniblockBase` enthält ein Attribut des Typs `t_Position` der Storeklasse. Es erwartet als Parameter `t_staticStore` eine statische Storeklasse und als Parameter `t_base` eine frei definierbare Basisklasse. Wegen der Möglichkeit zusätzlicher Attribute und der variablen Basisklasse wird die Blockmethode `Swap` nicht in `gct_Miniblock`, sondern in `gct_MiniblockBase` definiert.

Templatedeklaration

```
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;
};
```

Blockklasse

Das Klassentemplate `gct_Miniblock` erwartet als Parameter `t_blockBase` eine Klasse, die mindestens die Datentypen, Attribute und Methoden wie `gct_MiniblockBase` enthält.

Templatedeklaration

```
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;
};
```

Ein Miniblockobjekt ist kleiner als ein vergleichbares Blockobjekt. Die Methode `GetByteSize`, die von Blockanwendern (z. B. Strings) häufig aufgerufen wird, ist jedoch etwas langsamer. Das Template `gct_Miniblock` ist insbesondere für Objekte geeignet, die in großen Stückzahlen auftreten.


```
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);
    }
```

Erweiterungen

Als oberste Basisklasse können z. B. die leere Klasse `ct_Empty` oder `ct_Object` verwendet werden. Dafür existieren die beiden Erweiterungen `gct_EmptyBaseMiniBlock` und `gct_ObjectBaseMiniBlock`.

Templatedeklaration

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

Templatedeklaration

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

1.4.4 Reserveblock (tuning/resblock.h)

Bei den Templates `gct_Block` und `gct_MiniBlock` werden Reallokationen vom verwendeten Store optimiert. In einigen Fällen möchte der Anwender jedoch diese Optimierung selber durchführen, indem er zeitweise mehr Speicher allokiert, als tatsächlich verwendet wird. Diese Eigenschaft wurde im Template `gct_ResBlock` berücksichtigt. Es hat als zusätzliches Attribut eine Minimalgröße. Die Größe des allokierten Speichers ist gleich dem Maximum aus Größe und Minimalgröße. D. h. bei Bedarf kann zusätzlicher Speicher reserviert werden, um die Anzahl der Reallokationen zu verringern.

Ein typischer Anwendungsfall ist eine Zeichenkettenverarbeitung, bei der in einem bestimmten Arbeitsschritt sehr viele Änderungen an einem Objekt vorgenommen werden, die i. a. auch mit einer Größenänderung verbunden sind. Wenn zusätzlich bekannt ist, daß die Zeichenkette nicht größer als z. B. 4096 Bytes wird, so setzt man die Minimalgröße vor dem Arbeitsschritt auf 4096 und am Ende wieder auf Null.

Die Implementierung besteht ähnlich wie bei `gct_Block` aus der Basisklasse `gct_ResBlockBase`, der eigentlichen Blockklasse `gct_ResBlock` und den Erweiterungen `gct_EmptyBaseResBlock` und `gct_ObjectBaseResBlock`.

Basisklasse

Das Klassentemplate `gct_ResBlockBase` enthält ein Attribut des Typs `t_Position` und zwei Attribute des Typs `t_Size` der Storeklasse. Es erwartet als Parameter `t_staticStore` eine statische Storeklasse und als Parameter `t_base` eine frei definierbare Basisklasse. Wegen der Möglichkeit zusätzlicher Attribute und der variablen Basisklasse wird die Blockmethode `Swap` nicht in `gct_ResBlock`, sondern in `gct_ResBlockBase` definiert.

Templatedeklaration

```
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;
};

```

Blockklasse

Das Klassentemplate `gct_ResBlock` erwartet als Parameter `t_blockBase` eine Klasse, die mindestens die Datentypen, Attribute und Methoden wie `gct_ResBlockBase` enthält.

Templatedeklaration

```

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);
};

```

Zusätzliche Methoden

`t_Size GetMinByteSize () const;`

Liefert die Minimalgröße in Bytes.

`t_Size GetAllocByteSize () const;`

Liefert die allokierten Bytes, d. h. das Maximum von Größe und Minimalgröße.

`void SetMinByteSize (t_Size o_newSize);`

Setzt die neue Minimalgröße in Bytes.

Erweiterungen

Als oberste Basisklasse können z. B. die leere Klasse `ct_Empty` oder `ct_Object` verwendet werden. Dafür existieren die beiden Erweiterungen `gct_EmptyBaseResBlock` und `gct_ObjectBaseResBlock`.

Templatedeclaration

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

Templatedeclaration

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

1.4.5 Fixblock (tuning/fixblock.h)

Jede dynamische Speicherverwaltung besitzt eine Minimalgröße für Speicherblöcke und beansprucht pro Block einige Bytes Verwaltungsspeicher. Dieser doppelte Overhead wirkt sich besonders bei kleinen Anforderungen von 10 oder 16 Bytes aus. Ist von Instanzen eines Blocktyps bekannt, daß ihre Größe einen bestimmten Wert nicht überschreitet, kann mit Hilfe des Templates `gct_FixBlock` der Verwaltungsaufwand gesenkt werden.

Templatedeclaration

```
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;
};
```

Ein Fixblockobjekt fordert den benötigten Speicher nicht von einem Store an, sondern enthält ihn als Attribut `ac_Block`. Die Parameter `t_size` und `u_fixSize` sollten aufeinander abgestimmt sein. Z. B. ist `gct_FixBlock <t_UInt8, 15>` eine sinnvolle Kombination. Das Blockobjekt umfaßt insgesamt 16 Bytes.

Der Parameter `t_size` beeinflusst die Ausrichtung des Arrays `ac_Block` im Speicher. Ist z. B. `t_size` auf `t_UInt16` gesetzt, dann belegt `o_Size` 2 Bytes, `ac_Block` liegt auf einer 2-Byte-Grenze und das Blockobjekt endet auf einer 2-Byte-Grenze. In diesem Block können nur Objekte gespeichert werden, die eine 1- oder 2-Byte-Ausrichtung erfordern.

1.4.6 Nulldatablock (tuning/nulldatablock.h)

Stringklassen, die nullterminierte Zeichenketten verwalten, belegen auch im leeren Zustand den Speicher für das Nullzeichen. Durch die Rundung der Blockgrößen werden effektiv mindestens 8 oder 16 Bytes belegt. Treten in einer Anwendung sehr häufig leere Stringobjekte auf, kann sich dieser Overhead zu einem großen Betrag summieren. Der Nulldatablock behandelt dieses Problem, indem er statischen Speicher für ein einzelnes Zeichen bereitstellt und bei Blockgröße 1 keinen dynamischen Speicher verwendet.

Templatedeklaration

```
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;
};
```

Der Anwender des Nulldatablock muß darauf achten, daß bei Blockgröße 1 nur das Nullzeichen in den Speicher geschrieben wird und keine anderen Daten. Auf diese Weise ist der Nulldatablock auch ohne Synchronisierung sicher beim Zugriff durch mehrere Threads. Das Klassentemplate `gct_NullDataBlock` erwartet als Parameter eine Blockklasse, z. B. `gct_EmptyBaseBlock <ct_Chn_Store>`, und einen Zeichendatentyp, also `char` oder `wchar_t`.

1.4.7 Zeichenblock (tuning/charblock.h)

Das Klassentemplate `gct_CharBlock` erweitert die Blockschnittstelle um Zusatzfunktionen für Zugriff, Einfügen und Löschen von Zeichen. Es erwartet als Parameter eine Blockklasse, z. B. `gct_EmptyBaseBlock <ct_Chn_Store>`, und einen Zeichendatentyp, also `char` oder `wchar_t`. Die byte-orientierten Aufrufe werden privat deklariert, damit byte- und zeichen-orientierte Aufrufe nicht gemischt verwendet werden können, z. B. `SetByteSize` und `GetCharSize`.

Basisklasse

`ct_AnyBlock` (siehe Abschnitt 'Blockschnittstelle')

Templatedeklaration

```
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);
};

```

Methoden

`t_Size GetMaxCharSize ();`

Liefert die maximale Anzahl der Zeichen im Block.

`t_Size GetCharSize () const;`

Liefert die Anzahl der Zeichen im Block.

`void SetCharSize (t_Size o_size);`

Setzt die Anzahl der Zeichen im Block auf `o_size`.

`void IncCharSize (t_Size o_inc);`

Vergrößert den Block um `o_inc` Zeichen.

`void DecCharSize (t_Size o_dec);`

Verkleinert den Block um `o_dec` Zeichen. Es muß `o_dec <= GetCharSize ()` gelten.

`t_char * GetRawAddr () const;`

Liefert die Anfangsadresse des Blocks mit dem Typ `t_char *`.

`t_char * GetRawAddr (t_Size o_pos) const;`

Liefert die Adresse des Zeichens an der Position `o_pos`. Es muß `o_pos <= GetCharSize ()` gelten. Die angegebene Position kann also auch hinter dem letzten Zeichen sein.

`t_char * GetCharAddr () const;`

Liefert die Anfangsadresse des Blocks mit dem Typ `t_char *`. Der Block muß mindestens ein Zeichen enthalten, d. h. der Rückgabewert zeigt garantiert auf ein Zeichen innerhalb des Blocks.

`t_char * GetCharAddr (t_Size o_pos) const;`

Liefert die Adresse des Zeichens an der Position `o_pos`. Es muß `o_pos < GetCharSize ()` gelten, d. h. der Rückgabewert zeigt garantiert auf ein Zeichen innerhalb des Blocks.

`t_char * AppendChars (t_Size o_len);`

Vergrößert den Block um `o_len` Zeichen und gibt die Adresse des freigewordenen Speicherbereichs zurück.

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

Vergrößert den Block um `o_len` Zeichen, verschiebt den Speicher an der Position `o_pos` um `o_len` Zeichen nach hinten (zu höheren Positionen) und gibt die Adresse des freigewordenen Speicherbereichs zurück.

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

Verschiebt den Speicher an der Position `o_pos` um `o_len` Zeichen nach vorn (zu niedrigeren Positionen), verkleinert den Block um `o_len` Zeichen und gibt die Adresse des verschobenen Speicherbereichs zurück.

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

Füllt `o_len` Zeichen ab der Position `o_pos` mit dem Zeichen `c_fill` und gibt die Adresse des veränderten Speicherbereichs zurück.

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

Setzt die Größe auf `o_len` Zeichen und kopiert die ersten `o_len` Zeichen der Zeichenkette `pc_asgn` in den eigenen Speicherbereich (mit Prüfung auf Selbstzuweisung).

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

Vergrößert den Block um `o_len` Zeichen und kopiert die ersten `o_len` Zeichen der Zeichenkette `pc_app` in den freigewordenen Speicherbereich (ohne Prüfung auf Selbstzuweisung).

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

Vergrößert den Block um `o_len` Zeichen, verschiebt den Speicher an der Position `o_pos` um `o_len` Zeichen nach hinten (zu höheren Positionen) und kopiert die ersten `o_len` Zeichen der Zeichenkette `pc_ins` in den freigewordenen Speicherbereich.

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

Ersetzt `o_delLen` Zeichen an der Position `o_pos` durch die ersten `o_insLen` Zeichen der Zeichenkette `pc_ins`. Dabei kann der Block vergrößert oder verkleinert werden.

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

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

Mit diesen beiden Methoden wird `gct_CharBlock` kompatibel zur `PageBlock`-Schnittstelle..

1.4.8 Elementblock (tuning/itemblock.h)

Das Klassentemplate `gct_ItemBlock` erweitert die Blockschnittstelle um Zusatzfunktionen für Zugriff, Einfügen und Löschen von Elementen gleicher Größe. Die byte-orientierten Aufrufe werden privat deklariert, damit byte- und element-orientierte Aufrufe nicht gemischt verwendet werden können, z. B. `SetByteSize` und `GetItemSize`.

Basisklasse

`ct_AnyBlock` (siehe Abschnitt 'Blockschnittstelle')

Templatedeklaration

```
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);
};

```

Methoden

`t_Size GetFixSize () const;`

Liefert die Größe eines Elements in Bytes.

`t_Size GetMaxItemSize () const;`

Liefert die maximale Anzahl der Elemente im Block.

`t_Size GetItemSize () const;`

Liefert die Anzahl der Elemente im Block.

`void SetItemSize (t_Size o_size) const;`

Setzt die Anzahl der Elemente im Block auf `o_size`.

`void IncItemSize1 ();`

Vergrößert den Block um 1 Element.

`void DecItemSize1 ();`

Verkleinert den Block um 1 Element. Es muß `1 <= GetItemSize ()` gelten.

`void IncItemSize (t_Size o_inc);`

Vergrößert den Block um `o_inc` Elemente.

`void DecItemSize (t_Size o_dec);`

Verkleinert den Block um `o_dec` Elemente. Es muß `o_dec <= GetItemSize ()` gelten.

`void * GetItemAddr (t_Size o_pos) const;`

Liefert die Adresse des Elements an der Position `o_pos`. Es muß `o_pos < GetItemSize ()` gelten, d. h. der Rückgabewert zeigt garantiert auf ein Element innerhalb des Blocks.

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

Vergrößert den Block um `o_count` Elemente und verschiebt den Speicher an der Position `o_pos` um `o_count` Elemente nach hinten (zu höheren Positionen). Die Adresse des freigewordenen Speicherbereichs wird zurückgegeben.

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

Verschiebt den Speicher an der Position `o_pos` um `o_count` Elemente nach vorn (zu niedrigeren Positionen) und verkleinert den Block um `o_count` Elemente. Die Adresse des verschobenen Speicherbereichs wird zurückgegeben.

`t_Size GetDefaultPageSize () const;`

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

Mit diesen beiden Methoden wird `gct_ItemBlock` kompatibel zur `PageBlock`-Schnittstelle..

Erweiterungen

Die Größe eines Elements im Klassentemplate `gct_ItemBlock` kann zur Laufzeit oder zur Übersetzungszeit festgelegt werden. Dafür existieren die beiden Erweiterungen `gct_VarItemBlock` und `gct_FixItemBlock`.

Das Klassentemplate `gct_VarItemBlock` erweitert `gct_ItemBlock`. Die Größe eines Elements wird zur Laufzeit mit der Methode `AlignPageSize` festgelegt, während der Block noch die Größe Null hat. Eine typische Anwendung ist der Blockstore.

Templatedeklaration

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

Das Klassentemplate `gct_FixItemBlock` erweitert `gct_ItemBlock`. Die Größe eines Elements wird mit dem Templateparameter `o_itemSize` festgelegt. Eine typische Anwendung ist der Arraycontainer.

Templatedeklaration

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

1.4.9 Pageblock (tuning/pageblock.hpp)

Der Pageblock unterteilt den angeforderten Speicher in mehrere, gleichgroße Pages. Dadurch ergeben sich gegenüber einem zusammenhängenden Block folgende Vorteile:

1. Geringere Anzahl an Speichieranforderungen und -freigaben bei Größenänderung.
2. Geringere Speicherfragmentierung durch wenige, immer gleichgroße Teilblöcke.
3. Kein Umkopieren beim Ändern der Größe.
4. Speicheradressen innerhalb des Blocks bleiben auch beim Ändern der Größe gültig.

Der Pageblock ist nur für größere Speichermengen sinnvoll, denn auch bei einer geringen Blockgröße wird immer mindestens eine Page belegt. Die Größe des Blockobjekts spielt im Verhältnis zum verwalteten Datenspeicher keine wesentliche Rolle. Deshalb wurde der Pageblock als eine Klasse implementiert und nicht als ein Template mit variablen Größen- und Positionstypen.

Neben den gleichgroßen Pages mit Nutzdaten enthält der Pageblock noch einen Speicherblock, der Zeiger auf die Pages verwaltet. Für beide Speichertypen können unterschiedliche Storeobjekte verwendet werden. Der Verwaltungsspeicher kann eine feste oder eine variable Größe haben. Ein fest dimensionierter Verwaltungsspeicher hat in einer multithreaded Umgebung den Vorteil, daß für die Berechnung der Speicheradresse aus einem Index (`GetCharAddr` oder `GetItemAddr`) kein Mutex benötigt wird. Die maximale Anzahl der Pages und damit auch die Maximalgröße des Blocks sind in diesem Fall jedoch begrenzt, und es muß darauf geachtet werden, daß es nicht zu einem Überlauf kommt.

Da ganze Pages relativ selten angefordert und freigegeben werden, erfolgt der Zugriff auf die Storeobjekte nicht als Templateparameter, sondern über virtuelle Methoden. Die Implementierung besteht aus der Basisklasse `ct_PageBlockBase` mit rein virtuellen Methoden und der abgeleiteten Klasse `ct_PageBlock` mit dem Zugriff auf zwei Default-Storeobjekte.

Neben der allgemeinen Blockschnittstelle enthält der Pageblock auch die Methoden von `gct_CharBlock` und `gct_ItemBlock`. Bei der Verwendung als Elementblock muß darauf geachtet werden, daß der Speicher eines Elements nicht über eine Pagegrenze gehen darf. Deshalb muß der Pageblock, solange er noch die Größe Null hat, mit der Methode `AlignPageSize` justiert werden.

Klassendeklaration

```
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;
};
```

Zusätzliche Methoden

`void LastPageWarning ();`

Diese virtuelle Methode wird aufgerufen, wenn der Verwaltungsspeicher für Pages fest dimensioniert ist und die letzte Page allokiert werden soll. Das bedeutet, daß für weitere Vergrößerungen des Blocks nur noch Speicher im Umfang von einer Page zur Verfügung steht.

`void LastPageError ();`

Diese virtuelle Methode wird aufgerufen, wenn der Verwaltungsspeicher für Pages fest dimensioniert ist und die letzte Page aufgebraucht ist, d. h. der Pageblock kann nicht weiter vergrößert werden. Der Aufrufer muß vor dem Vergrößern des Pageblocks sicherstellen, daß dieser Fall nicht eintritt. Andernfalls kann das Programm nicht sinnvoll weitergeführt werden.

Bei Klassen und Templates, die die Blockschnittstelle verwenden, wird dieser Fall nicht behandelt. Deshalb darf im Pageerrorhandler keine Exception ausgelöst werden. Diese Exception würde in der Bibliothek **Spirick Tuning** nicht behandelt werden und dazu führen, daß das Objekt, das gerade Speicher angefordert hat, in einem inkonsistenten Zustand verbleibt (siehe Funktion `tl_SetOverflowHandler`).

`t_Size GetDefaultPageSize () const;`

Die `DefaultPageSize` kann von abgeleiteten Klassen verwendet werden, wenn sie keine eigene Größe für Pages konfigurieren.

`t_Size GetFixPagePtrs () const;`

Liefert die Anzahl der Pages für den fest dimensionierten Verwaltungsspeicher. Das ist gleichzeitig die maximale Anzahl der Pages. Der Wert Null bedeutet, daß die Größe des Verwaltungsspeichers nicht fest, sondern dynamisch ist.

`void SetFixPagePtrs (t_Size o_ptr);`

Setzt die Anzahl der Pages für den fest dimensionierten Verwaltungsspeicher auf den Wert `o_ptr`. Der Aufruf ist nur möglich, wenn die Blockgröße gleich Null ist.

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

Die interne Größe für Pages wird so justiert, daß sie ein Vielfaches der Elementgröße `o_fixSize` und größer oder gleich `o_pageSize` ist. Der Aufruf ist nur möglich, wenn die Blockgröße gleich Null ist.

`t_Size GetPageSize () const;`

Liefert die tatsächliche Größe einer Page nach dem Aufruf von `AlignPageSize`.

`t_Size GetRoundedSize () const;`

Liefert das Produkt aus Größe einer Page und Anzahl der Pages.

Klassendeklaration

```
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 ();
};
```

Methoden

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

Allokiert Speicher für den Zeigerblock (Verwaltungsspeicher).

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

Reallokiert Speicher für den Zeigerblock (Verwaltungsspeicher).

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

Allokiert eine Page für Nutzdaten.

```
void FreeData (void * pv_mem);
```

Gibt eine Page für Nutzdaten frei.

```
~ct_PageBlock ();
```

Im Destruktor der abgeleiteten Klasse muß der angeforderte Speicher frei gegeben werden, denn im Destruktor der Basisklasse besteht kein Zugriff auf die virtuellen Methoden mehr.

1.4.10 Block-Instanzen (tuning/xxx/block.h)

Zur Erleichterung des Umgangs mit der Blockschnittstelle werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen des Klassentemplates `gct_Block` vordefiniert. Das Makro `BLOCK_DCLS(Obj)` generiert für jede der vier Wrapperklassen eines globalen Storeobjekts je eine Blockklasse. Die Makroverwendung

```
BLOCK_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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> { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei **'block.h'**. Darin werden mit Hilfe des Makros `BLOCK_DCLS` vier Blockklassen deklariert. Z. B. enthält die Klasse `ct_Std8Block` den Größentyp `t_UInt8` und fordert den Speicher für den dynamischen Block vom globalen Standardstoreobjekt an.

In der Datei **'tuning/std/block.h'** werden deklariert:

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

In der Datei **'tuning/rnd/block.h'** werden deklariert:

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

In der Datei **'tuning/chn/block.h'** werden deklariert:

```
class ct_Chn_Block;
```

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

1.5 Spezielle Stores

1.5.1 Blockstore (tuning/blockstore.h)

Blockstores verwalten in einem umfassenden Block mehrere gleichgroße Speicherblöcke. Ihr Hauptanwendungsgebiet sind Listencontainer, denn deren Nodes besitzen stets dieselbe Größe. Eine dynamische Speicherverwaltung rundet (auch ohne Round- und Chainstore) die Blockgrößen und beansprucht pro Block einige Bytes Verwaltungsspeicher. Werden viele gleichgroße Speicherblöcke angefordert, ist dieser Verwaltungsaufwand unnötig. Eine dynamische Speicherverwaltung ist darauf nicht eingerichtet. Ein Blockstore bringt jedoch die gleichgroßen Elemente fortlaufend ohne Zwischenraum im umfassenden Block unter und erzielt damit eine bessere Speicherauslastung.

Das Klassentemplate `gct_BlockStore` erwartet als Parameter `t_itemBlock` eine Blockklasse mit Elementblock-Schnittstelle, z. B. `gct_VarItemBlock <ct_Chn16Block>` oder `ct_PageBlock`. Sie dient dem Blockstore als Basisklasse. Der zweite Templateparameter `t_charBlock` ist eine Blockklasse mit Zeichenblock-Schnittstelle, z. B. `gct_CharBlock <ct_Chn16Block, char>`. Sie wird in der Methode `FreeUnused` als temporärer Zwischenspeicher verwendet.

Basisklassen

`t_itemBlock` (siehe Abschnitt 'Elementblock')

Templatedeklaration

```
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 ();
};
```

Größen- und Positionstyp eines Blockstores sind gleich dem Größentyp der übergebenen Blockklasse. Als Positionszeiger dienen Indizes. Es wird nicht das Byte-Offset eines inneren Blocks verwendet, sondern seine fortlaufende Nummer. Der Positionswert Null ist wie bei allen anderen Stores per Definition ungültig. Die Positionswerte eines Blockstores beginnen also mit 1, 2, 3 usw.

Ein Blockstore stellt sicher, daß Positionszeiger stets auf dieselben Elemente verweisen. Die Speicheradressen sind jedoch nur eingeschränkt gültig. Beim Vergrößern oder Verkleinern des umfassenden Blocks wird dieser u. U. an eine andere Stelle im Speicher verschoben. Dabei ändern sich die physischen Adressen der Elemente im Blockstore. Die Adressen (die mit `AddrOf` ermittelt werden) sind nur solange gültig, wie der Store nicht mit `Alloc`, `Realloc` oder `Free` verändert wurde. Wurde als Parameter `t_itemBlock` jedoch `ct_PageBlock` angegeben, bleiben die Speicheradressen der Elemente im Blockstore erhalten.

Wird ein Element freigegeben, kann der Blockstore nicht die dahinterliegenden Elemente verschieben, denn damit würden sich ihre Positionszeiger ändern. Ein Blockstore muß also ähnlich wie ein dynamischer Store eine Liste der Freielemente verwalten. Dafür existieren zwei Strategien.

Die erste ist auf eine maximale Geschwindigkeit ausgerichtet. Beim Freigeben eines Elements wird nur geprüft, ob es sich am physischen Ende befindet. In diesem Fall wird der umfassende Block verkleinert. Befindet sich das Element 'mittendrin', wird es der Freiliste ohne weitere Prüfung zugeordnet. Diese Strategie ist sehr schnell. Die Wahrscheinlichkeit, daß am Ende des umfassenden Blocks etwas freigegeben werden kann und sich dieser verkleinert, ist jedoch gering.

Die zweite Strategie ist auf eine gute Speichernutzung ausgerichtet und arbeitet mit einer sortierten Freiliste. Wird am Ende des umfassenden Blocks ein Element freigegeben, kann mit Hilfe der Sortierung leicht festgestellt werden, ob sich unmittelbar davor weitere freie Elemente befinden und der umfassende Block um mehrere Einheiten verkürzt werden kann. Befindet sich das Element 'mittendrin', wird es in aufsteigender Reihenfolge in die Freiliste einsortiert. Die Sortierung der Freiliste ermöglicht es, beim Anfordern eines neuen Elements das freie mit dem kleinsten Index zu verwenden. Damit verdichtet sich die Auslastung am physischen Anfang des umfassenden Blocks, und es steigt die Wahrscheinlichkeit, daß am Ende etwas freigegeben werden kann.

Die beiden Strategien führen zu einer unterschiedlichen Implementierung der Blockstoremethode `Free`. Wegen des größeren Rechenzeitaufwandes der zweiten Strategie wird standardmäßig die erste verwendet. Die Methode `SetSortedFree` steuert ein Umschalten auf die zweite Strategie. Beim Verwenden der ersten Strategie besteht die Möglichkeit, von Zeit zu Zeit mit der Methode `FreeUnused` die Freiliste zu sortieren und Freielemente am physischen Ende des umfassenden Blocks zu entfernen.

Die Methode `SizeOf` wird vom Template `gct_BlockStore` nicht unterstützt. Neben der allgemeinen Storeschnittstelle enthält es noch die folgenden Methoden:

Zusätzliche Methoden

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

Steuert das Umschalten der Strategie zum Sortieren der Freiliste.

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

Setzt die Größe der Pages, wenn als Parameter `t_itemBlock` die Klasse `ct_PageBlock` angegeben wurde.

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

Liefert den größten gültigen Positionswert, unabhängig davon, ob das zugehörige Element frei oder belegt ist. Bei einem leeren Blockstore ist der Rückgabewert gleich Null.

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

Liefert `true`, wenn sich mindestens ein Element in der Freiliste befindet.

```
void FreeUnused ();
```

Sortiert die Freiliste aufsteigend und löscht Freielemente am physischen Ende des umfassenden Blocks.

1.5.2 Blockstore-Instanzen (tuning/xxx/blockstore.h)

Zur Erleichterung des Umgangs mit Blockstores werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen des Klassentemplates `gct_BlockStore` vordefiniert. Das Makro `BLOCK_STORE_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je eine Blockstoreklasse. Die Makroverwendung

```
BLOCK_STORE_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**blockstore.h**'. Darin werden mit Hilfe des Makros `BLOCK_STORE_DCLS` nach obigem Muster vier Blockstoreklassen deklariert.

In der Datei '**tuning/std/blockstore.h**' werden deklariert:

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

In der Datei '**tuning/rnd/blockstore.h**' werden deklariert:

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

In der Datei '**tuning/chn/blockstore.h**' werden deklariert:

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

1.5.3 Referenzzähler (tuning/refcount.hpp)

Die Bibliothek **Spirick Tuning** enthält die spezialisierte Referenzzählerklasse `ct_RefCount`. Sie wird in Refstores und allen darauf aufbauenden Klassen eingesetzt und ist an deren Bedürfnisse angepaßt. Ein Refstore ordnet jedem Speicherblock ein `ct_RefCount`-Objekt zu. Neben einem 'flachen' Referenzzähler enthält `ct_RefCount` ein boolesches Attribut (das Alloc-Bit). Es besitzt den Wert `true`, wenn der zugehörige Block belegt ist.

Klassendeklaration

```
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;
};
```

Datentypen

```
typedef t_UInt32 t_RefCount;
```

Der 'flache' Referenzzählertyp besitzt eine Breite von 32 Bits.

Methoden

```
ct_RefCount ();
```

Setzt den Referenzzähler auf Null und das Alloc-Bit auf true.

```
void Initialize ();
```

Setzt den Referenzzähler auf Null und das Alloc-Bit auf true.

```
t_RefCount GetRef () const;
```

Liefert den Wert des 'flachen' Referenzzählers.

```
void IncRef ();
```

Erhöht den Referenzzähler um Eins.

```
void DecRef ();
```

Verkleinert den Referenzzähler um Eins.

```
bool IsAlloc () const;
```

Liefert true, wenn das Alloc-Bit gesetzt ist.

```
void SetAlloc ();
```

Setzt das Alloc-Bit.

```
bool IsFree () const;
```

Liefert true, wenn das Alloc-Bit nicht gesetzt ist.

```
void SetFree ();
```

Löscht das Alloc-Bit.

```
bool IsNull () const;
```

Liefert true, wenn das Alloc-Bit nicht gesetzt und der Referenzzähler gleich Null ist.

1.5.4 Refstore (tuning/refstore.h)

Ein Refstore besitzt keine eigene Speicherverwaltung, sondern baut auf einer vorhandenen auf und ordnet jedem Speicherblock einen Referenzzähler zu. Mit deren Hilfe können sichere Zeiger implementiert werden. Das Klassentemplate `gct_RefStore` erwartet als Parameter eine Storeklasse, übernimmt von ihr den Größen- und Positionstyp und enthält ein Objekt der Storeklasse.

Templatedeklaration

```
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 ();
};
```

Wird von einem Refstore Speicher angefordert, gibt er die Anforderung an den darunterliegenden Store weiter, platziert am Anfang des bereitgestellten Speicherblocks ein `ct_RefCount`-Objekt und initialisiert es. Der 'flache' Referenzzähler erhält den Wert Null, und das Alloc-Bit wird auf `true` gesetzt. Auf den Referenzzähler und den dahinterliegenden Nutzerbereich wird mit Hilfe desselben Positionszeigers zugegriffen. Die Methode `AddrOf` der allgemeinen Storeschnittstelle liefert die Adresse des Nutzerbereichs. Zur Manipulation des Referenzzählers dienen die zusätzlichen Methoden `IncRef` und `DecRef`.

Wird mit der Methode `Free` ein Speicherblock an den Refstore zurückgegeben, löscht er das Alloc-Bit im zugehörigen `ct_RefCount`-Objekt. Ist zusätzlich der Wert des Referenzzählers gleich Null, wird der Speicherblock im darunter liegenden Store freigegeben. Andernfalls bleibt der Speicher weiter genutzt, und der Positionszeiger behält seine Gültigkeit. Der Versuch, mit `AddrOf` auf den Nutzerbereich zuzugreifen, führt zu einer `ASSERT`-Meldung. Mit den Methoden `IncRef` und `DecRef` kann jedoch der Referenzzähler weiterhin geändert werden. Erreicht er den Wert Null, gibt der Refstore den Speicherblock im darunter liegenden Store frei, und der Positionszeiger verliert seine Gültigkeit.

Die Methode `FreeAll` wird vom Template `gct_RefStore` nicht unterstützt. Neben der allgemeinen Storeschnittstelle enthält es noch die folgenden Methoden:

Zusätzliche Methoden

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

Erhöht den zum Positionszeiger `o_pos` gehörenden Referenzzähler. `o_pos` muß eine gültige Position sein.

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

Verkleinert den zum Positionszeiger `o_pos` gehörenden Referenzzähler. `o_pos` muß eine gültige Position sein.

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

Liefert den Wert des zum Positionszeiger `o_pos` gehörenden Referenzzählers. `o_pos` muß eine gültige Position sein.

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

Liefert `true`, wenn der zum Positionszeiger `o_pos` gehörende Speicherbereich im Refstore genutzt ist und mit `AddrOf` auf den Nutzerbereich zugegriffen werden kann. `o_pos` muß eine gültige Position sein.

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

Diese Methode ist die logische Negation von `IsAlloc`. `o_pos` muß eine gültige Position sein.

```
t_store * GetStore ();
```

Liefert einen Zeiger auf das enthaltene Storeobjekt.

1.5.5 Refstore-Instanzen (tuning/xxx/refstore.h)

Zur Erleichterung des Umgangs mit Refstores werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen des Klassentemplates `gct_RefStore` vordefiniert. Das Makro `REF_STORE_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je eine Refstoreklasse. Die Makroverwendung

```
REF_STORE_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

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

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**refstore.h**'. Darin werden mit Hilfe des Makros `REF_STORE_DCLS` nach obigem Muster vier Refstoreklassen deklariert.

In der Datei '**tuning/std/refstore.h**' werden deklariert:

```
class ct_StdRefStore;
class ct_Std8RefStore;
class ct_Std16RefStore;
class ct_Std32RefStore;
```

In der Datei '**tuning/rnd/refstore.h**' werden deklariert:

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

In der Datei 'tuning/chn/refstore.h' werden deklariert:

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

1.5.6 Blockrefstore-Instanzen (tuning/xxx/blockrefstore.h)

Ein Blockrefstore entsteht, wenn dem Klassentemplate `gct_RefStore` als Parameter `t_store` eine Blockstoreklasse übergeben wird. Er nutzt die Speicherverwaltung des Blockstores, der gleichgroße Speicherblöcke in einem umfassenden Block unterbringt. Zusätzlich ordnet der Blockrefstore jedem Speicherblock einen Referenzzähler zu.

Zur Erleichterung des Umgangs mit Blockrefstores werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `BLOCKREF_STORE_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je eine Blockrefstoreklasse. Die Makroverwendung

```
BLOCKREF_STORE_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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> { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**blockrefstore.h**'. Darin werden mit Hilfe des Makros `BLOCKREF_STORE_DCLS` nach obigem Muster vier Blockrefstoreklassen deklariert.

In der Datei 'tuning/std/blockrefstore.h' werden deklariert:

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

In der Datei 'tuning/rnd/blockrefstore.h' werden deklariert:

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

In der Datei 'tuning/chn/blockrefstore.h' werden deklariert:

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

1.5.7 Packstore (tuning/packstore.hpp)

Der Packstore ist darauf optimiert, mehrere zusammengehörige Speicheranforderungen ohne unnötigen Zwischenraum und Rechenzeitaufwand hintereinander im Speicher abzulegen. Er arbeitet nach einem sehr einfachen Verfahren. Die angeforderten Speicherblöcke werden nacheinander in gleichgroßen Pages untergebracht. Ist der Restspeicher in der aktuellen Page zu klein für eine neue Anforderung, wird eine neue Page allokiert. Ab einer konfigurierbaren Minimalgröße erhält eine Speicheranforderung eine eigene Page.

Die Freigabe einzelner Speicherblöcke ist nicht vorgesehen. Der Packstore kann jedoch mit der Methode `FreeAll` den gesamten Speicher freigeben. Wird ein Packstore-Objekt mehrmals mit `FreeAll` geleert und wiederverwendet, kann man mit dem Parameter `b_keepPage` verhindern, daß die erste Page freigegeben und anschließend neu allokiert wird.

Neben den gleichgroßen Pages mit Nutzdaten verwaltet der Packstore noch einen Speicherblock variabler Größe, der Zeiger auf die Pages enthält. Für beide Speichertypen können unterschiedliche Storeobjekte verwendet werden. Da ganze Pages relativ selten angefordert und freigegeben werden, erfolgt der Zugriff auf die Storeobjekte nicht als Templateparameter, sondern über virtuelle Methoden. Die Implementierung besteht aus der Basisklasse `ct_PackStoreBase` mit rein virtuellen Methoden und der abgeleiteten Klasse `ct_PackStore` mit dem Zugriff auf zwei Default-Storeobjekte.

Klassendeklaration

```
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);
```

Zusätzliche Methoden

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

Initialisiert den Packstore, solange noch kein Speicher angefordert wurde. Mit dem Parameter `o_align` wird das Alignment gesteuert. Zulässige Werte sind 1, 2, 4, 8 und 16. Der Parameter `o_pageSize` gibt die Größe einer Page an. Mit dem optionalen Parameter `o_ownPageSize` wird festgelegt, ab welcher Größe eine Speicheranforderung eine eigene Page erhält. Ist der Parameter nicht angegeben, wird ein Viertel der Pagesize verwendet. Der Rückgabewert ist `false`, wenn im Packstore bereits Speicher angefordert wurde oder ein Parameter einen ungültigen Wert enthält.

Klassendeklaration

```
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 ();
};
```

Methoden

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

Reallokiert Speicher für den Zeigerblock.

```
t_UInt MaxDataAlloc () const;
```

Liefert die maximale Anzahl Bytes für Nutzdaten.

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

Allokiert eine Page für Nutzdaten.

```
void FreeData (void * pv_mem);
```

Gibt eine Page für Nutzdaten frei.

```
~ct_PackStore ();
```

Im Destruktor der abgeleiteten Klasse muß der angeforderte Speicher frei gegeben werden, denn im Destruktor der Basisklasse besteht kein Zugriff auf die virtuellen Methoden mehr.

2 OBJEKTVERWALTUNG

2.1 Container

2.1.1 Containerschnittstelle

Die Bibliothek **Spirick Tuning** enthält zwei verschiedene Konzepte für die Objektverwaltung: Container und Collections. Collections sind auf leichte Bedienbarkeit und schnelles Übersetzen ausgerichtet. Sie sind polymorph, d. h. sie können Objekte unterschiedlicher Typen enthalten. Bei Containern steht die Laufzeiteffizienz im Vordergrund. Sie sind homogen, d. h. sie enthalten nur Objekte eines bestimmten Typs. Die Anpassung an den konkreten Objekttyp ermöglicht zahlreiche Optimierungen.

Ähnlich wie Storeklassen besitzen auch Containerklassen keine gemeinsame Basisklasse mit virtuellen Methoden, aber eine einheitliche Schnittstelle. Diese vereinfacht die Handhabung und ermöglicht das leichte Austauschen eines Containers gegen einen anderen.

Templatedeklaration

```
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 ();
};
```

Container werden als Klassentemplates mit mindestens einem Parameter, dem Typ der enthaltenen Objekte `t_obj`, deklariert. Dieser muß weder von einer abstrakten Basisklasse erben noch einen Gleich- oder Vergleichsoperator besitzen. Es muß nur sichergestellt sein, daß der normale und der Kopierkonstruktor verfügbar sind und korrekt arbeiten. Wegen der Implementierung als Template und der geringen Anforderungen an die enthaltenen Objekte sind Container universell einsetzbar und sehr effizient. Es können beliebige Klassen und auch primitive Datentypen wie `int` oder `float` in Containern untergebracht werden.

Collections verwalten Zeiger auf außerhalb erzeugte Objekte. Container enthalten dagegen ihre Objekte physisch und können den Speicher wesentlich besser auslasten. Container stellen den Speicherplatz der Objekte zur Verfügung und rufen deren Konstruktoren und Destruktoren auf. Ein neues Objekt wird mit seinem normalen Konstruktor erzeugt. Ein vorhandenes Objekt kann nicht übernommen, sondern nur mit seinem Kopierkonstruktor in den Container kopiert werden. Beim Löschen eines Objektes wird dessen Destruktor aufgerufen.

Bei vielen Operationen, die Veränderungen am Container bewirken, müssen die enthaltenen Objekte umorganisiert werden. Dafür existieren im wesentlichen zwei Strategien: Bei der ersten Strategie werden die Objekte mit Kopierkonstruktoren und Gleichoperatoren kopiert. Dabei kann ein erheblicher Overhead entstehen. Die Verwendung der C++11 Move-Semantik bringt nur wenig Abhilfe. Bei der zweiten Strategie werden die Objekte mit `memcpy` und `memmove` in einen anderen Speicherbereich kopiert. Dieses Verfahren kommt bei allen Containern in der Bibliothek **Spirick Tuning** zur Anwendung. Es muß darauf geachtet werden, daß die enthaltenen Objekte mit `memcpy` kopierbar sind. In realen C++-Programmen existieren nur sehr wenige Klassen, die diese Eigenschaft nicht besitzen. Z. B. dürfen die Klassen `ct_ThMutex` und `ct_ThSemaphore` nicht mit `memcpy` kopiert werden.

Die Implementierung der Container ist der Speicherverwaltung sehr nahe. Container besitzen zahlreiche Ähnlichkeiten mit Stores. Auch Container verwalten ihre Einträge mit Hilfe von Positionszeigern. Während ein Store nur 'rohe' Speicherblöcke verwaltet, verarbeitet ein Container auch noch den Inhalt, d. h. die darin enthaltenen Objekte. Der Storemethode `Alloc` entspricht etwa die Containermethode `AddObj`. Sie erzeugt im Container ein neues Objekt und liefert seine Position. Mit `AddrOf` erhält man eine untypisierte Adresse eines Speicherblocks. Die Containermethode `GetObj` liefert dagegen einen typisierten Zeiger auf den konkreten Objekttyp. Mit `Free` wird ein Speicherblock freigegeben. Die Containermethode `DelObj` ruft vorher noch den Destruktor des enthaltenen Objekts auf.

Im Gegensatz zu Stores sichern nicht alle Container die Gültigkeit der Positionszeiger. Die Positionszeiger eines Containers müssen nur dann ihre Gültigkeit behalten, wenn die Objekte von außen referenziert werden. Werden die Elemente jedoch ausschließlich mit `First` und `Next` durchlaufen, können die Positionszeiger zugunsten einer besseren Speicherauslastung nach Veränderungen des Containers ihre Gültigkeit verlieren. Das ist in der Bibliothek **Spirick Tuning** bei allen Arraytypen der Fall. Bei Listencontainern behalten jedoch die Positionszeiger auch nach einer Änderung ihre Gültigkeit und verweisen auf dasselbe Objekt.

Datentypen

```
typedef t_UInt t_Length;
```

Der geschachtelte Typ `t_Length` beschreibt die maximale Anzahl der Objekte. Neben `t_UInt` werden auch `t_UInt8`, `t_UInt16` und `t_UInt32` verwendet. Ist z. B. `t_Length` auf `t_UInt8` definiert, kann der Container nur maximal 255 Einträge verwalten. Jeder Container enthält ein Attribut des Typs `t_Length`. Ein angepaßter Längentyp verringert somit den Speicherbedarf des Containerobjekts.

```
typedef void * t_Position;
```

Container verwalten ähnlich wie Stores ihre Objekte mit Hilfe von Positionszeigern. Neben `void *` werden auch `t_UInt`, `t_UInt8`, `t_UInt16` und `t_UInt32` verwendet. Bei allen Positionstypen ist der Wert `Null` per Definition ungültig.

```
typedef t_obj t_Object;
```

Der geschachtelte Datentyp `t_Object` entspricht dem Parameter `t_obj` des Containertemplates. Die Typdefinition ermöglicht Anwendern des Containers und abgeleiteten Klassen den Zugriff auf den Objekttyp.

Konstruktor, Destruktor, Gleichoperator, Swap

```
gct_AnyContainer ();
```

Der normale Konstruktor erzeugt einen leeren Container.

```
gct_AnyContainer (const gct_AnyContainer & co_init);
```

Der Kopierkonstruktor übernimmt alle Elemente eines vorhandenen Containers mit Hilfe des Kopierkonstruktors der enthaltenen Objekte.

```
~gct_AnyContainer ();
```

Im Destruktor eines Containers wird die Methode `DeleteAll` aufgerufen. Vor der Speicherfreigabe werden alle enthaltenen Objekte mit ihrem Destruktor zerstört.

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

Der Gleichoperator übernimmt ähnlich wie der Kopierkonstruktor alle Elemente eines vorhandenen Containers mit Hilfe des Kopierkonstruktors der enthaltenen Objekte.

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

Tauscht den Inhalt der beiden Objekte aus.

Anzahl der Objekte

```
bool IsEmpty () const;
```

Liefert `true`, wenn der Container keine Objekte enthält.

```
t_Length GetLen () const;
```

Liefert die Anzahl der enthaltenen Objekte.

Iterieren des Containers

```
t_Position First () const;
```

Liefert die Position des ersten Objekts oder Null bei einem leeren Container.

```
t_Position Last () const;
```

Liefert die Position des letzten Objekts oder Null bei einem leeren Container.

```
t_Position Next (t_Position o_pos) const;
```

Liefert die Position des nächsten Objekts oder Null, wenn `o_pos` die Position des letzten Elements war. `o_pos` muß eine gültige Position sein.

```
t_Position Prev (t_Position o_pos) const;
```

Liefert die Position des vorigen Objekts oder Null, wenn `o_pos` die Position des ersten Elements war. `o_pos` muß eine gültige Position sein.

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

Liefert die Position des Objekts mit der fortlaufenden Nummer `u_idx`. Der Index muß zwischen Eins und `GetLen` liegen.

Zugriff auf Objekte

`t_Object * GetObj (t_Position o_pos) const;`

Liefert einen typisierten Zeiger auf das durch `o_pos` identifizierte Objekt. `o_pos` muß eine gültige Position sein.

Einfügen von Objekten

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

Fügt ein neues Objekt in den Container ein und liefert dessen Position. Die Stelle des Einfügens ist abhängig von der Implementierung. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt vor einem anderen ein und liefert dessen Position. Ist `o_pos` gleich Null, wird das neue Objekt nach dem letzten platziert, d. h. es ist das neue letzte Element. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt nach einem anderen ein und liefert dessen Position. Ist `o_pos` gleich Null, wird das neue Objekt vor dem ersten platziert, d. h. es ist das neue erste Element. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

Anfügen und Löschen mehrerer Objekte

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

Fügt am Ende des Containers `o_count` Objekte an. Ist der Zeiger `po_obj` gleich Null, werden die Objekte mit ihrem normalen Konstruktor erzeugt. Andernfalls werden ihre Kopierkonstruktoren mit dem Parameter `* po_obj` aufgerufen.

`void TruncateObj (t_Length o_count = 1);`

Löscht am Ende des Containers `o_count` Objekte. Es werden die Destruktoren der Objekte aufgerufen und der zugehörige Verwaltungsspeicher freigegeben.

Rückgabewert von Löschmethoden

Löschmethoden liefern stets die Position des Nachfolgers des gelöschten Eintrags. Diese Technik ermöglicht das gleichzeitige Iterieren und Verändern eines Containers. Der Rückgabewert wird mit der Methode `Next` vor dem Löschen berechnet. Wurde der der Reihenfolge nach letzte Eintrag gelöscht (Methode `Last`), ist der Rückgabewert gleich Null.

Löschen von Objekten

`t_Position De1Obj (t_Position o_pos);`

Ruft den Destruktor eines Objekts auf und gibt den zugehörigen Speicher frei. `o_pos` muß eine gültige Position sein. Die Methode liefert `Next (o_pos)`, also die Position des nächsten Objekts oder Null, wenn das letzte Objekt gelöscht wurde.

`void De1All ();`

Ruft die Destruktoren aller Objekte auf und gibt deren Speicher frei. `De1All` ist i. a. schneller als das mehrfache Löschen mit `De1Obj`.


```
t_Position FreeObj (t_Position o_pos);
```

Gibt den Speicher eines Objekts frei, ohne dessen Destruktor aufzurufen. FreeObj ist für primitive Datentypen wie int oder float geeignet und schneller als DelObj. o_pos muß eine gültige Position sein. Die Methode liefert Next (o_pos), also die Position des nächsten Objekts oder Null, wenn das letzte Objekt gelöscht wurde.

```
void FreeAll ();
```

Gibt den gesamten von Objekten belegten Speicher frei, ohne deren Destrukturen aufzurufen.

Exception Handling

Bei der Arbeit mit Containern können in Konstruktoren und Destrukturen enthaltener Objekte Exceptions auftreten. Container enthalten minimale eigene Exceptionhandler. Diese versetzen nach dem Erkennen einer Exception das Containerobjekt in einen konsistenten Zustand und reichen die Exception unverändert an den übergeordneten Exceptionhandler, der sich im Programmcode des Containeranwenders befindet, weiter. Im einzelnen gelten folgende Regeln:

Tritt beim Einfügen eines einzelnen Objektes (AddObj) in dessen Konstruktor eine Exception auf, verbleibt der Container in seinem vorigen Zustand (Objekt wird nicht eingefügt).

Tritt beim Löschen eines einzelnen Objektes (DelObj) in dessen Destruktor eine Exception auf, wird das Objekt trotzdem aus dem Container entfernt.

Tritt beim Einfügen mehrerer Objekte mit der Methode AppendObj im Konstruktor eines Objektes eine Exception auf, wird das Einfügen abgebrochen. Die korrekt eingefügten Objekte verbleiben im Container.

Tritt beim Löschen mehrerer Objekte mit der Methode TruncateObj in einem Destruktor eine Exception auf, wird das Löschen abgebrochen. Die korrekt gelöschten Objekte bleiben gelöscht. Die noch nicht gelöschten Objekte verbleiben im Container. Das Objekt, das die Exception ausgelöst hat, gilt als gelöscht.

Tritt beim Löschen mehrerer Objekte mit der Methode DelAll in einem Destruktor eine Exception auf, wird das Löschen fortgesetzt. Anschließend befindet sich der Container im leeren Zustand. Dieses Verhalten ist für die folgenden vier Methoden relevant:

Tritt in einer der Methoden Konstruktor, Kopierkonstruktor, Destruktor oder Gleichoperator im Konstruktor oder Destruktor eines Objektes eine Exception auf, wird der Container in den leeren Zustand versetzt. Dabei werden die Destrukturen aller enthaltenen Objekte aufgerufen und sämtlicher Verwaltungsspeicher freigegeben.

2.1.2 Operationen mit Containern

Objekte einfügen, kopieren und löschen

Das folgende Programmbeispiel demonstriert das Einfügen, Kopieren und Löschen von Objekten in einem Container. Die Klasse ct_Int wird im Abschnitt 'Beispielprogramme' beschrieben.

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

// Neues Objekt im Container mit Defaultkonstruktor erzeugen
o_pos = co_container. AddObj ();

// Auf das Objekt zugreifen und es und initialisieren
pco_int = co_container. GetObj (o_pos);
(* pco_int) = 2;

// Vorhandenes Objekt in den Container kopieren
o_pos = co_container. AddObj (& co_int);

// Objekt aus dem Container nehmen und löschen
```

```
co_container. DelObj (o_pos);
```

Vorwärts iterieren

Zum Iterieren eines Containers in aufsteigender Reihenfolge der Einträge wird eine `for`-Schleife nach folgendem Muster empfohlen:

```
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);
    // ...
}
```

Rückwärts iterieren

Zum Iterieren eines Containers in absteigender Reihenfolge der Einträge wird eine `for`-Schleife nach folgendem Muster empfohlen:

```
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);
    // ...
}
```

Iterieren und verändern

Zum Iterieren und Verändern eines Containers wird eine `for`-Schleife nach folgendem Muster empfohlen:

```
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);
    // ...
}
```

Statt der `for`-Schleife kann auch eine `while`-Schleife nach folgendem Muster verwendet werden:

```
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 Erweiterter Container (tuning/extcont.h)

Das Klassentemplate `gct_ExtContainer` vereinfacht den Umgang mit der Containerschnittstelle. Z. B. müssen zum Ermitteln des fünften Objekts normalerweise zwei Methoden aufgerufen werden.

```

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

```

Das Klassentemplate `gct_ExtContainer` besitzt für diesen Fall die Methode `GetNthObj`. Die Containerklasse, die als Templateparameter übergeben wird, dient dem erweiterten Container als Basisklasse. Zur Illustration der Implementierung des erweiterten Containers wird die Definition einer Methode angefügt.

Basisklasse

`gct_AnyContainer` (siehe Abschnitt 'Containerschnittstelle')

Templatedeklaration

```

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);
};

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));
}

```

Zugriff auf Objekte

```
t_Object * GetFirstObj () const;
```

Liefert einen typisierten Zeiger auf das erste Objekt. Der Container muß mindestens ein Objekt enthalten.

```
t_Object * GetLastObj () const;
```

Liefert einen typisierten Zeiger auf das letzte Objekt. Der Container muß mindestens ein Objekt enthalten.

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

Liefert einen typisierten Zeiger auf das folgende Objekt. `o_pos` und `Next (o_pos)` müssen gültige Positionen sein.

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

Liefert einen typisierten Zeiger auf das vorhergehende Objekt. `o_pos` und `Prev (o_pos)` müssen gültige Positionen sein.

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

Liefert einen typisierten Zeiger auf das n-te Objekt. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen.

Einfügen von Objekten

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

Fügt ein neues Objekt in den Container ein und liefert dessen Position. Das Objekt wird vor dem ersten plazierte, d. h. es ist das neue erste Element. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt in den Container ein und liefert dessen Position. Das Objekt wird nach dem letzten plazierte, d. h. es ist das neue letzte Element. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt vor einem anderen ein und liefert dessen Position. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt nach einem anderen ein und liefert dessen Position. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

Zugriff auf neue Objekte

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

Fügt ein neues Objekt in den Container ein und liefert einen Zeiger darauf. Die Stelle des Einfügens ist abhängig von der Implementierung. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt in den Container ein und liefert einen Zeiger darauf. Das Objekt wird vor dem ersten platziert, d. h. es ist das neue erste Element. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt in den Container ein und liefert einen Zeiger darauf. Das Objekt wird nach dem letzten platziert, d. h. es ist das neue letzte Element. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt vor einem anderen ein und liefert einen Zeiger darauf. Ist `o_pos` gleich Null, wird das neue Objekt nach dem letzten platziert, d. h. es ist das neue letzte Element. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt nach einem anderen ein und liefert einen Zeiger darauf. Ist `o_pos` gleich Null, wird das neue Objekt vor dem ersten platziert, d. h. es ist das neue erste Element. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt vor einem anderen ein und liefert einen Zeiger darauf. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

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

Fügt ein neues Objekt nach einem anderen ein und liefert einen Zeiger darauf. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen. Ist der Zeiger `po_obj` gleich Null, wird das Objekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_obj` aufgerufen.

Rückgabewert von Löschmethoden

Löschmethoden liefern stets die Position des Nachfolgers des gelöschten Eintrags. Diese Technik ermöglicht das gleichzeitige Iterieren und Verändern eines Containers. Der Rückgabewert wird mit der Methode `Next` vor dem Löschen berechnet. Wurde der der Reihenfolge nach letzte Eintrag gelöscht (Methode `Last`), ist der Rückgabewert gleich Null.

Löschen von Objekten

`t_Position DelFirstObj ();`

Entfernt das erste Objekt aus dem Container und ruft dessen Destruktor auf. Der Container muß mindestens ein Objekt enthalten. Die Methode liefert die Position des neuen ersten Eintrags oder Null, wenn kein Eintrag mehr vorhanden ist.

`t_Position DelLastObj ();`

Entfernt das letzte Objekt aus dem Container und ruft dessen Destruktor auf. Der Container muß mindestens ein Objekt enthalten. Die Methode liefert Null, da der letzte Eintrag gelöscht wurde.

`t_Position DelNextObj (t_Position o_pos);`

Entfernt das Objekt `Next (o_pos)` aus dem Container und ruft dessen Destruktor auf. `o_pos` und `Next (o_pos)` müssen gültige Positionen sein. Die Methode liefert die Position des Nachfolgers des gelöschten Eintrags oder Null, wenn der letzte Eintrag gelöscht wurde.

`t_Position DelPrevObj (t_Position o_pos);`

Entfernt das Objekt `Prev (o_pos)` aus dem Container und ruft dessen Destruktor auf. `o_pos` und `Prev (o_pos)` müssen gültige Positionen sein. Die Methode liefert `o_pos` zurück, denn `o_pos` ist der Nachfolger des gelöschten Eintrags.

`t_Position DelNthObj (t_Length u_idx);`

Entfernt das Objekt `Nth (u_idx)` aus dem Container und ruft dessen Destruktor auf. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen. Die Methode liefert die Position des Nachfolgers des gelöschten Eintrags oder Null, wenn der letzte Eintrag gelöscht wurde.

`t_Position FreeFirstObj ();`

Entfernt das erste Objekt aus dem Container, ohne dessen Destruktor aufzurufen. Der Container muß mindestens ein Objekt enthalten. Die Methode liefert die Position des neuen ersten Eintrags oder Null, wenn kein Eintrag mehr vorhanden ist.

`t_Position FreeLastObj ();`

Entfernt das letzte Objekt aus dem Container, ohne dessen Destruktor aufzurufen. Der Container muß mindestens ein Objekt enthalten. Die Methode liefert Null, da der letzte Eintrag gelöscht wurde.

`t_Position FreeNextObj (t_Position o_pos);`

Entfernt das Objekt `Next (o_pos)` aus dem Container, ohne dessen Destruktor aufzurufen. `o_pos` und `Next (o_pos)` müssen gültige Positionen sein. Die Methode liefert die Position des Nachfolgers des gelöschten Eintrags oder Null, wenn der letzte Eintrag gelöscht wurde.

`t_Position FreePrevObj (t_Position o_pos);`

Entfernt das Objekt `Prev (o_pos)` aus dem Container, ohne dessen Destruktor aufzurufen. `o_pos` und `Prev (o_pos)` müssen gültige Positionen sein. Die Methode liefert `o_pos` zurück, denn `o_pos` ist der Nachfolger des gelöschten Eintrags.

`t_Position FreeNthObj (t_Length u_idx);`

Entfernt das Objekt `Nth (u_idx)` aus dem Container, ohne dessen Destruktor aufzurufen. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen. Die Methode liefert die Position des Nachfolgers des gelöschten Eintrags oder Null, wenn der letzte Eintrag gelöscht wurde.

2.2 Arrays und Listen

2.2.1 Array (tuning/array.h)

Der Arraycontainer ist auf eine bestmögliche Speicherauslastung optimiert. Er bringt seine Objekte ohne Zwischenraum in einem Block unter. Beim Einfügen oder Löschen von Objekten werden alle dahinterliegenden im Speicher verschoben, und es ändern sich deren Positionszeiger. In einem Arraycontainer ist der direkte Zugriff auf das n-te Element möglich.

Das Klassentemplate `gct_Array` besitzt zwei Parameter. `t_obj` ist der Objekttyp. `t_block` ist eine Blockklasse mit Elementblock-Schnittstelle, und dient dem Arraycontainer als Basisklasse. Bei einem Arraycontainer läßt sich die Elementgröße relativ einfach zur Übersetzungszeit ermitteln. Das Klassentemplate `gct_FixItemArray` vereinfacht die Handhabung, indem es die passenden Parameter für das Template `gct_FixItemBlock` bereitstellt.

Basisklassen

`gct_...ItemBlock` (siehe Abschnitt 'Elementblock')

Templatedeklaration

```
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);
    inline 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);
};
```

Zusätzliche Methoden

`t_Length GetMaxLen () const;`

Liefert die maximale Anzahl der Objekte im Container.

`void SetPageSize (t_Size o_size);`

Setzt die Größe der Pages, wenn als Parameter `t_block` die Klasse `ct_PageBlock` angegeben wurde.

Templatedeclaration

```
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-Instanzen (tuning/xxx/array.h)

Zur Erleichterung des Umgangs mit Arraycontainern werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen des Klassentemplates `gct_Array` vordefiniert. Das Makro `ARRAY_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Arraytemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
ARRAY_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**array.h**'. Darin werden mit Hilfe des Makros `ARRAY_DCLS` nach obigem Muster vier Arraytemplates deklariert.

In der Datei '**tuning/std/array.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/rnd/array.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/chn/array.h**' werden deklariert:

```
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 Liste (tuning/dlist.h)

Der Listencontainer verwaltet seine Objekte in Knoten (Nodes). Der Speicher für jedes einzelne Node wird von einem Store angefordert. Nodes und die darin enthaltenen Objekte werden mit Hilfe der Positionszeiger des Stores identifiziert. Deshalb behalten die Positionszeiger nach Änderungen des

Containers ihre Gültigkeit. Ob zusätzlich auch die Speicheradressen ihre Gültigkeit behalten, hängt vom verwendeten Store ab.

Der Listencontainer ist als eine doppelt verkettete Liste (double linked list) implementiert. Jedes Node enthält neben dem Objekt je einen Positionszeiger auf Vorgänger und Nachfolger. Die doppelte Verkettung ermöglicht das Durchlaufen des Containers in beiden Richtungen und beschleunigt das Einfügen und Löschen von Elementen. Zum Ermitteln des n-ten Elements müssen jedoch die Nodes einzeln abgezählt werden.

Das Klassentemplate `gct_DList` besitzt zwei Parameter. `t_obj` ist der Objekttyp. `t_store` ist eine Storeklasse. Der Listencontainer enthält ein Attribut dieses Typs und fordert von ihm den Speicher für seine Nodes an. Die zusätzliche Methode `GetStore` ermöglicht den Zugriff auf das Storeobjekt.

Templatedeklaration

```
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;
    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);
    inline 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 Listen-Instanzen (tuning/xxx/dlist.h)

Zur Erleichterung des Umgangs mit Listencontainern werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen des Klassentemplates `gct_DList` vordefiniert. Das Makro `DLIST_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Listentemplate, das nur noch den Parameter `t_obj` besitzt. Diese Listen fordern den Speicher jedes Nodes

einzelnen von einem globalen Storeobjekt an. Die Speicheradressen der enthaltenen Objekte behalten nach Änderungen des Containers ihre Gültigkeit. Die Makroverwendung

`DLIST_DCLS (Any)`

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**dlist.h**'. Darin werden mit Hilfe des Makros `DLIST_DCLS` nach obigem Muster vier Listentemplates deklariert.

In der Datei '**tuning/std/dlist.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/rnd/dlist.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/chn/dlist.h**' werden deklariert:

```
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 Sortierte Container

2.3.1 Sortiertes Array (tuning/sortarr.h)

Der sortierte Arraycontainer ist analog zum einfachen Arraycontainer implementiert. Auch er bringt seine Objekte ohne Zwischenraum in einem Block unter. Beim Einfügen oder Löschen von Objekten werden alle dahinterliegenden im Speicher verschoben, und es ändern sich deren Positionszeiger. Der direkte Zugriff auf das n-te Element ist möglich.

Das Klassentemplate `gct_SortedArray` besitzt wie `gct_Array` zwei Parameter. `t_obj` ist der Objekttyp. `t_block` ist eine Blockklasse mit Elementblock-Schnittstelle, und dient dem Arraycontainer als Basisklasse. Bei einem Arraycontainer läßt sich die Elementgröße relativ einfach zur Übersetzungszeit ermitteln. Das Klassentemplate `gct_FixItemSortedArray` vereinfacht die Handhabung, indem es die passenden Parameter für das Template `gct_FixItemBlock` bereitstellt.

Im Gegensatz zum einfachen Arraycontainer ordnet `gct_SortedArray` die Elemente in aufsteigender Reihenfolge an. Dazu muß der Objekttyp `t_obj` den Vergleichsoperator '`operator <`' bereitstellen. Positioniertes Einfügen mit den Methoden `AddObjBefore` und `AddObjAfter` ist nur möglich, wenn das Objekt

an dieser Stelle einzuordnen ist. Beim sortierten Arraycontainer werden neue Objekte normalerweise mit der Methode `AddObj` eingefügt. Diese sortiert das Objekt automatisch an der richtigen Stelle ein. Mehrere gleiche Objekte werden hintereinander angeordnet. Ihre Reihenfolge entspricht der des Einfügens. Das zuletzt eingefügte Objekt steht in der Folge der gleichen Objekte an letzter Stelle.

Besitzt der Objekttyp `t_obj` zusätzlich den Gleichheitsoperator `'operator =='`, kann der Container um die Vergleichscontainerschnittstelle erweitert werden. Im sortierten Arraycontainer wird zum Suchen der Objekte eine binäre Suche verwendet, die wesentlich effizienter als die lineare Suche im einfachen Arraycontainer ist.

Basisklassen

`gct_...ItemBlock` (siehe Abschnitt 'Elementblock')

Templatedeklaration

```
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;
};
```

Zusätzliche Methoden

`t_Length GetMaxLen () const;`

Liefert die maximale Anzahl der Objekte im Container.

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

Setzt die Größe der Pages, wenn als Parameter `t_block` die Klasse `ct_PageBlock` angegeben wurde.

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

Liefert die Position des letzten Objektes, das kleiner oder gleich `* po_obj` ist. Liefert Null, wenn das Objekt kleiner als das erste Objekt im Container ist. Liefert `Last ()`, wenn das Objekt nicht kleiner als das letzte Objekt im Container ist.

Templatedeklaration

```
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 Sortierte Array-Instanzen (tuning/xxx/sortedarray.h)

Zur Erleichterung des Umgangs mit sortierten Arraycontainern werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen des Klassentemplates `gct_SortedArray` vordefiniert. Das Makro `SORTEDARRAY_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Arraytemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
SORTEDARRAY_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
template <class t_obj> class gct_Any_SortedArray:
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> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**sortedarray.h**'. Darin werden mit Hilfe des Makros `SORTEDARRAY_DCLS` nach obigem Muster vier Arraytemplates deklariert.

In der Datei '**tuning/std/sortedarray.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/rnd/sortedarray.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/chn/sortedarray.h**' werden deklariert:

```
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 Hashtabelle (tuning/hashtable.h)

Hashtabellen sind spezialisierte Container mit einer Zugriffsbeschleunigung. Sie können sehr unterschiedlich implementiert werden. Das Grundprinzip besteht darin, Objekte gleichen Hashwertes (modulo der Hashtabellengröße) in einer lokalen Liste anzuordnen. Gelingt es, die Hashwerte breit zu streuen, und verwendet man eine Primzahl für die Hashtabellengröße, werden die lokalen Listen im Durchschnitt sehr klein. Bei der Suche nach einem Objekt kann mit seinem Hashwert direkt auf die lokale Liste zugegriffen werden, wo es sehr schnell zu finden ist.

Hashtabellen sind sortierte Container, da die enthaltenen Objekte in der Reihenfolge ihrer Hashwerte angeordnet werden. Ein positioniertes Einfügen mit `AddObjBefore` oder `AddObjAfter` ist nicht möglich. Neue Objekte können nur mit der Methode `AddObj` eingefügt werden.

Das Klassentemplate `gct_HashTable` ist als geschachtelter Arraycontainer implementiert. Es besitzt zwei Parameter. `t_obj` ist der Objekttyp. Er muß eine Methode `GetHash` bereitstellen, die einen ganzzahligen numerischen Wert liefert. `t_block` ist eine Blockklasse mit einfacher Schnittstelle, z. B. `ct_Chn16Block`. Sie dient der Implementierung des umfassenden und der lokalen Arraycontainer.

Solange der Container noch keine Elemente enthält, kann mit der Methode `SetHashSize` die Größe der Hashtabelle eingestellt werden. Die zusammen mit dem Klassentemplate definierten Konstanten `u_HashPrime1` bis `u_HashPrime16` sind Primzahlen, die den umfassenden Arraycontainer auf eine Größe knapp unterhalb einer Zweierpotenz bringen. Als Standardeinstellung dient die Konstante `u_HashPrime4`.

Besitzt der Objekttyp `t_obj` zusätzlich den Gleichheitsoperator `'operator =='`, kann der Container um die Vergleichscontainerschnittstelle erweitert werden. In der Hashtabelle werden Objekte mit Hilfe ihres Hashwertes gesucht. Die Suche ist wesentlich effizienter als im einfachen Arraycontainer oder im Listencontainer.

Templatedeklaration

```
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;
};

```

Konstanten

```

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;

```

Diese Konstanten sind empfohlene Vorgabewerte für die Größe der Hashtabelle. Es sind Primzahlen, die den umfassenden Arraycontainer auf eine Größe knapp unterhalb einer Zweierpotenz bringen.

Zusätzliche Methoden

```
void SetHashSize (t_Length o_size);
```

Setzt bei einem leeren Container die Größe der Hashtabelle. Alle Objekte werden mit ihrem Hashwert modulo der Größe der Hashtabelle einsortiert.

```
t_Length GetHashSize () const;
```

Liefert die Größe der Hashtabelle.

2.3.4 Hashtabellen-Instanzen (tuning/xxx/hashtable.h)

Zur Erleichterung des Umgangs mit Hashtabellencontainern werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen des Klassentemplates `gct_HashTable` vordefiniert. Das Makro `HASHTABLE_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Hashtabellentemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
HASHTABLE_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```

template <class t_obj> class gct_AnyHashTable:
public gct_ExtContainer <gct_HashTable <t_obj, ct_AnyBlock> > { };
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> > { };

```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**hashtable.h**'. Darin werden mit Hilfe des Makros `HASHTABLE_DCLS` nach obigem Muster vier Hashtabellentemplates deklariert.

In der Datei 'tuning/std/hashtable.h' werden deklariert:

```
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;
```

In der Datei 'tuning/rnd/hashtable.h' werden deklariert:

```
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;
```

In der Datei 'tuning/chn/hashtable.h' werden deklariert:

```
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- und Reflisten

2.4.1 Blockliste

Übergibt man dem Containertemplate `gct_DList` als Parameter `t_store` eine Blockstoreklasse, erhält man eine Blockliste. Sie verbindet die Bedieneigenschaften einer doppelt verketteten Liste mit der Speichereffizienz eines Blockstores. Dieser bringt die Nodes kompakt in einem zusammenhängenden Speicherblock unter. Bei der Verwendung von Blocklisten ist zu beachten, daß sich die Speicheradressen der Objekte ändern können.

In Blocklisten ist der Längentyp gleich dem Positionstyp. Durch Auswahl eines geeigneten Positionstyps kann der Speicherbedarf reduziert werden. Ist z. B. von Instanzen eines bestimmten Listentyps bekannt, daß die enthaltenen Objekte zusammengekommen nicht mehr als 64 KB Speicher benötigen, kann ein 16-Bit-Blockstore genutzt werden. Gegenüber einem 32-Bit-Blockstore verringert sich der Speicherbedarf jedes Nodes um vier Bytes, denn Nodes enthalten je zwei Positionszeiger.

2.4.2 Blocklisten-Instanzen (tuning/xxx/blockdlist.h)

Zur Erleichterung des Umgangs mit Blocklisten werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `BLOCK_DLIST_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Blocklistentemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
BLOCK_DLIST_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei **'blockdlist.h'**. Darin werden mit Hilfe des Makros **BLOCK_DLIST_DCLS** nach obigem Muster vier Listentemplates deklariert.

In der Datei 'tuning/std/blockdlist.h' werden deklariert:

```
template <class t_obj> class gct_Std_BlockDList;
template <class t_obj> class gct_Std8BBlockDList;
template <class t_obj> class gct_Std16BlockDList;
template <class t_obj> class gct_Std32BlockDList;
```

In der Datei 'tuning/rnd/blockdlist.h' werden deklariert:

```
template <class t_obj> class gct_Rnd_BlockDList;
template <class t_obj> class gct_Rnd8BBlockDList;
template <class t_obj> class gct_Rnd16BlockDList;
template <class t_obj> class gct_Rnd32BlockDList;
```

In der Datei 'tuning/chn/blockdlist.h' werden deklariert:

```
template <class t_obj> class gct_Chn_BlockDList;
template <class t_obj> class gct_Chn8BBlockDList;
template <class t_obj> class gct_Chn16BlockDList;
template <class t_obj> class gct_Chn32BlockDList;
```

2.4.3 Refliste (tuning/refdlist.h)

Das Containertemplate **gct_DList** nutzt nur die normale Storeschnittstelle des Parameters **t_store**. Verwendet man einen Refstore, müßte der Anwender der Liste mit Hilfe der Methode **GetStore** auf die erweiterten Storemethoden (z. B. **IncRef**) zugreifen. Das Template **gct_RefDList** vereinfacht diesen Zugriff. Es übernimmt die Ref-Methoden des Storeobjekts in die Listenschnittstelle und besitzt dieselben Templateparameter wie **gct_DList**. Die Definition der Methode **IncRef** demonstriert die Implementierung des Reflistentemplates.

Basisklassen

gct_DList (siehe Abschnitt 'Liste')

gct_ExtContainer (siehe Abschnitt 'Erweiterter Container')

Templatedeklaration

```
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;
};

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);
}
```


In einer Refliste wird jedem einzelnen Node ein Referenzzähler zugeordnet. Dieser ermöglicht die Implementierung von sicheren Zeigern auf Listeneinträge. Ein sicherer Zeiger erhöht den Referenzzähler des Eintrags, auf den er verweist.

Ein Positionszeiger einer Refliste behält seine Gültigkeit, solange der Eintrag nicht (z. B. mit `De1Obj`) gelöscht wurde oder der Referenzzähler ungleich Null ist. Wurde das Element mit `De1Obj` aus der Liste entfernt, liefert `IsAlloc` den Wert `false`, und es kann nicht mehr mit `GetObj` auf das Objekt zugegriffen werden. Erreicht der Referenzzähler mit `DecRef` den Wert Null, wird auch der zugehörige Speicher freigegeben, und der Positionszeiger verliert seine Gültigkeit.

Methoden

`void IncRef (t_Position o_pos);`

Erhöht den zum Listeneintrag `o_pos` gehörenden Referenzzähler. `o_pos` muß eine gültige Position sein.

`void DecRef (t_Position o_pos);`

Verkleinert den zum Listeneintrag `o_pos` gehörenden Referenzzähler. `o_pos` muß eine gültige Position sein.

`t_RefCount GetRef (t_Position o_pos) const;`

Liefert den Wert des zum Listeneintrag `o_pos` gehörenden Referenzzählers. `o_pos` muß eine gültige Position sein.

`bool IsAlloc (t_Position o_pos) const;`

Liefert `true`, wenn der zum Positionszeiger `o_pos` gehörende Listeneintrag nicht (z. B. mit `De1Obj`) gelöscht wurde und mit `GetObj` darauf zugegriffen werden kann. `o_pos` muß eine gültige Position sein.

`bool IsFree (t_Position o_pos) const;`

Diese Methode ist die logische Negation von `IsAlloc`. `o_pos` muß eine gültige Position sein.

2.4.4 Reflisten-Instanzen (tuning/xxx/refdlist.h)

Zur Erleichterung des Umgangs mit Reflisten werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen des Klassentemplates `gct_RefDList` vordefiniert. Das Makro `REF_DLIST_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Listentemplate, das nur noch den Parameter `t_obj` besitzt. Diese Reflisten fordern den Speicher jedes Nodes einzeln von einem globalen Storeobjekt an. Die Speicheradressen der enthaltenen Objekte behalten nach Änderungen des Containers ihre Gültigkeit. Die Makroverwendung

`REF_DLIST_DCLS (Any)`

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
template <class t_obj> class gct_Any_RefDList:
    public gct_RefDList <t_obj, ct_Any_RefStore> { };
template <class t_obj> class gct_Any8_RefDList:
    public gct_RefDList <t_obj, ct_Any8_RefStore> { };
template <class t_obj> class gct_Any16_RefDList:
    public gct_RefDList <t_obj, ct_Any16_RefStore> { };
template <class t_obj> class gct_Any32_RefDList:
    public gct_RefDList <t_obj, ct_Any32_RefStore> { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '`refdlist.h`'. Darin werden mit Hilfe des Makros `REF_DLIST_DCLS` nach obigem Muster vier Reflistentemplates deklariert.

In der Datei '`tuning/std/refdlist.h`' werden deklariert:

```
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;
```

In der Datei 'tuning/rnd/refdlist.h' werden deklariert:

```
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;
```

In der Datei 'tuning/chn/refdlist.h' werden deklariert:

```
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 Blockreflisten-Instanzen (tuning/xxx/blockrefdlist.h)

Übergibt man dem Containertemplate `gct_RefDList` als Parameter `t_store` eine Blockrefstoreklasse, erhält man eine Blockrefliste. Zur Erleichterung des Umgangs mit Blockreflisten werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `BLOCKREF_DLIST_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Blockreflistentemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
BLOCKREF_DLIST_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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> { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**blockrefdlist.h**'. Darin werden mit Hilfe des Makros `BLOCKREF_DLIST_DCLS` nach obigem Muster vier Listentemplates deklariert.

In der Datei 'tuning/std/blockrefdlist.h' werden deklariert:

```
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;
```

In der Datei 'tuning/rnd/blockrefdlist.h' werden deklariert:

```
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;
```

In der Datei 'tuning/chn/blockrefdlist.h' werden deklariert:

```
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 Vergleichs-, Zeiger- und Mapcontainer

2.5.1 Vergleichscontainer (tuning/compcontainer.h)

Die normale Containerschnittstelle ist auf universelle Anwendbarkeit ausgelegt und stellt nur geringe Anforderungen an die enthaltenen Objekte. Diese müssen nur einen normalen und einen Kopierkonstruktor zur Verfügung stellen. Einige Objektklassen besitzen jedoch einen Gleichheitsoperator (operator ==). Dieser ist implizit auch für alle primitiven Datentypen, z. B. `int` oder `void *`, definiert. Der Gleichheitsoperator ermöglicht zahlreiche weitere Containermethoden, z. B. das bedingte Einfügen und das Suchen eines Elements.

Das Klassentemplate `gct_CompContainer` erwartet als Parameter eine Containerklasse, deren Objekttyp einen Gleichheitsoperator enthält, z. B. `gct_Std32Array <float>`. Sie dient dem Vergleichscontainer als Basisklasse.

Basisklassen

`gct_AnyContainer` (siehe Abschnitt 'Containerschnittstelle')
 [`gct_ExtContainer` (optional, siehe Abschnitt 'Erweiterter Container')]

Templatedeklaration

```
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);
};
```

Suche nach Objekten

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

Liefert `true`, wenn der Container ein Objekt enthält, das gleich `* po_obj` ist.

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

Liefert die Anzahl der Objekte, die gleich `* po_obj` sind.

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

Liefert die Position des ersten Objekts, das gleich * po_obj ist, oder Null, wenn kein Objekt gefunden wurde.

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

Liefert die Position des letzten Objekts, das gleich * po_obj ist, oder Null, wenn kein Objekt gefunden wurde.

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

Liefert die Position des nächsten Objekts, das gleich * po_obj ist, oder Null, wenn kein Objekt gefunden wurde. o_pos muß eine gültige Position sein. Die Suche beginnt bei Next (o_pos).

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

Liefert die Position des vorhergehenden Objekts, das gleich * po_obj ist, oder Null, wenn kein Objekt gefunden wurde. o_pos muß eine gültige Position sein. Die Suche beginnt bei Prev (o_pos).

Zugriff auf gefundene Objekte

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

Liefert einen typisierten Zeiger des ersten Objekts, das gleich * po_obj ist. Es muß mindestens ein gleiches Objekt enthalten sein.

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

Liefert einen typisierten Zeiger des letzten Objekts, das gleich * po_obj ist. Es muß mindestens ein gleiches Objekt enthalten sein.

Bedingtes Einfügen

`t_Position AddObjCond (const t_Object * po_obj);`

Liefert die Position des ersten Objekts, das gleich * po_obj ist, oder die Position eines neu eingefügten Objekts, wenn kein Objekt gefunden wurde. Zum Einfügen des neuen Objekts wird die Methode AddObj aufgerufen.

`t_Position AddObjBeforeFirstCond (const t_Object * po_obj);`

Liefert die Position des ersten Objekts, das gleich * po_obj ist, oder die Position eines neu eingefügten Objekts, wenn kein Objekt gefunden wurde. Zum Einfügen des neuen Objekts wird die Methode AddObjBeforeFirst aufgerufen.

`t_Position AddObjAfterLastCond (const t_Object * po_obj);`

Liefert die Position des ersten Objekts, das gleich * po_obj ist, oder die Position eines neu eingefügten Objekts, wenn kein Objekt gefunden wurde. Zum Einfügen des neuen Objekts wird die Methode AddObjAfterLast aufgerufen.

Rückgabewert von Löschmethoden

Löschmethoden liefern stets die Position des Nachfolgers des gelöschten Eintrags. Diese Technik ermöglicht das gleichzeitige Iterieren und Verändern eines Containers. Der Rückgabewert wird mit der Methode Next vor dem Löschen berechnet. Wurde der der Reihenfolge nach letzte Eintrag gelöscht (Methode Last), ist der Rückgabewert gleich Null.

Löschen gefundener Objekte

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

Löscht das erste Objekt, das gleich * po_obj ist. Es muß mindestens ein gleiches Objekt enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Objekts oder Null, wenn das letzte Objekt gelöscht wurde.

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

Löscht das letzte Objekt, das gleich * po_obj ist. Es muß mindestens ein gleiches Objekt enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Objekts oder Null, wenn das letzte Objekt gelöscht wurde.

Bedingtes Löschen gefundener Objekte

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

Löscht das erste Objekt, das gleich * po_obj ist, sofern ein gleiches Objekt gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Objekts oder Null, wenn kein Objekt gefunden wurde oder das gelöschte Objekt das letzte war.

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

Löscht das letzte Objekt, das gleich * po_obj ist, sofern ein gleiches Objekt gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Objekts oder Null, wenn kein Objekt gefunden wurde oder das gelöschte Objekt das letzte war.

2.5.2 Zeigercontainer (tuning/ptrcontainer.h)

Container können nicht nur Objekte (z. B. ct_String) und primitive Datentypen (z. B. int, float) enthalten, sondern auch C++-Zeiger (z. B. ct_String *). Viele Methoden der normalen und erweiterten Containerschnittstelle sind in diesem Fall unhandlich. Z. B. liefert die Methode GetObj einen Zeiger auf ein Objekt. Ist das Objekt selbst ein Zeiger, liefert sie einen Zeiger auf einen Zeiger. Analog erwartet die Methode AddObj einen Zeiger auf einen Zeiger.

```
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);
```

Das Klassentemplate gct_PtrContainer stellt eine komfortable Schnittstelle für Zeigercontainer bereit. Es mappt viele Methoden der normalen, erweiterten und Vergleichscontainer-Schnittstelle und enthält einige zusätzliche Methoden. Zur besseren Unterscheidung von den Objektmethode(n) (z. B. GetObj) enthalten die zugehörigen Zeigermethoden die Abkürzung Ptr (z. B. GetPtr). Die Methode DelPtr wird auf FreeObj zurückgeführt und löscht einen Zeiger aus dem Container. Die Methode DelPtrAndObj löscht zusätzlich das referenzierte Objekt.

Beim Instantiieren von gleichartigen Containern, die Zeiger enthalten (z. B. gct_Chnl6DList <int *> und gct_Chnl6DList <ct_String *>), entsteht stets derselbe binäre Programmcode. Diese Templateinstanzen unterscheiden sich nur im Typ der Parameter und Rückgabewerte. Um das unnötige Duplizieren von Programmcode zu vermeiden, erwartet das Template gct_PtrContainer die Objektklasse und eine Containerklasse mit dem Objekttyp void *. Z. B. ist gct_PtrContainer <int, gct_Chnl6DList <void *> > ein Container, der Objekte des Typs int * verwaltet. Der Zeigercontainer selbst enthält nur Inline-Methoden, die im binären Programmcode nicht separat instantiiert werden.

Für C++-Zeiger ist der Gleichheitsoperator definiert. Deshalb wird die übergebene Containerklasse zunächst um die Vergleichscontainer-Schnittstelle erweitert. Der Vergleichscontainer dient dem Zeigercontainer als Basisklasse. Alle dort deklarierten Methoden stehen dem Anwender des

Zeigercontainers auch zur Verfügung. Zur Illustration der Implementierung des Zeigercontainers werden die Definitionen der Methoden `GetPtr` und `DelPtrAndObj` angefügt.

Basisklassen

`gct_AnyContainer` (siehe Abschnitt 'Containerschnittstelle')
`gct_ExtContainer` (siehe Abschnitt 'Erweiterter Container')
`gct_CompContainer` (siehe Abschnitt 'Vergleichscontainer')

Templatedeklaration

```
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);
};

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);
}

```

Datentypen

```
typedef t_obj t_RefObject;
```

Der geschachtelte Typ `t_RefObject` beschreibt den Typ der referenzierten Objekte und ist für die Verwendung in abgeleiteten Klassen bestimmt.

Destruktor

```
~gct_PtrContainer ();
```

Da C++-Zeiger keinen Destruktor besitzen, ruft der Destruktor des Zeigercontainers die Methode `FreeAll` auf. Der vom Container belegte Speicher wird effizient freigegeben. Es werden jedoch keine Destrukturen referenzierter Objekte aufgerufen.

Zugriff auf referenzierte Objekte

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

Liefert einen typisierten Zeiger auf das durch `o_pos` identifizierte Objekt. `o_pos` muß eine gültige Position sein.

```
t_obj * GetFirstPtr () const;
```

Liefert einen typisierten Zeiger auf das erste Objekt. Der Container muß mindestens ein Objekt enthalten.

```
t_obj * GetLastPtr () const;
```

Liefert einen typisierten Zeiger auf das letzte Objekt. Der Container muß mindestens ein Objekt enthalten.

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

Liefert einen typisierten Zeiger auf das folgende Objekt. `o_pos` und `Next (o_pos)` müssen gültige Positionen sein.

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

Liefert einen typisierten Zeiger auf das vorhergehende Objekt. `o_pos` und `Prev (o_pos)` müssen gültige Positionen sein.

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

Liefert einen typisierten Zeiger auf das `n`-te Objekt. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen.

Einfügen von Zeigern

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

Fügt den Zeiger `po_obj` in den Container ein und liefert dessen Position. Die Stelle des Einfügens ist abhängig von der Implementierung.

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

Fügt den Zeiger `po_obj` vor einem anderen ein und liefert dessen Position. Ist `o_pos` gleich Null, wird der Zeiger nach dem letzten platziert, d. h. er ist das neue letzte Element.

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

Fügt den Zeiger `po_obj` nach einem anderen ein und liefert dessen Position. Ist `o_pos` gleich Null, wird der Zeiger vor dem ersten platziert, d. h. er ist das neue erste Element.

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

Fügt den Zeiger `po_obj` vor dem ersten ein und liefert dessen Position. `po_obj` ist das neue erste Element.

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

Fügt den Zeiger `po_obj` nach dem letzten ein und liefert dessen Position. `po_obj` ist das neue letzte Element.

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

Fügt den Zeiger `po_obj` vor einem anderen ein und liefert dessen Position. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen.

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

Fügt den Zeiger `po_obj` nach einem anderen ein und liefert dessen Position. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen.

Rückgabewert von Löschmethoden

Löschmethoden liefern stets die Position des Nachfolgers des gelöschten Eintrags. Diese Technik ermöglicht das gleichzeitige Iterieren und Verändern eines Containers. Der Rückgabewert wird mit der Methode `Next` vor dem Löschen berechnet. Wurde der der Reihenfolge nach letzte Eintrag gelöscht (Methode `Last`), ist der Rückgabewert gleich Null.

Löschen von Zeigern

```
t_Position DelPtr (t_Position o_pos);
```

Die Methode ist identisch mit `FreeObj`. Sie entfernt einen Zeiger aus dem Container und beeinflusst nicht das referenzierte Objekt. `o_pos` muß eine gültige Position sein. Die Methode liefert `Next (o_pos)`, also die Position des nächsten Eintrags oder Null, wenn der letzte Eintrag gelöscht wurde.

```
t_Position DelFirstPtr ();
```

Die Methode ist identisch mit `FreeFirstObj`. Sie entfernt den ersten Zeiger aus dem Container und beeinflusst nicht das referenzierte Objekt. Der Container muß mindestens ein Objekt enthalten. Die Methode liefert die Position des neuen ersten Eintrags oder Null, wenn kein Eintrag mehr vorhanden ist.

`t_Position DelLastPtr ();`

Die Methode ist identisch mit `FreeLastObj`. Sie entfernt den letzten Zeiger aus dem Container und beeinflusst nicht das referenzierte Objekt. Der Container muß mindestens ein Objekt enthalten. Die Methode liefert Null, da der letzte Eintrag gelöscht wurde.

`t_Position DelNextPtr (t_Position o_pos);`

Die Methode ist identisch mit `FreeNextObj`. Sie entfernt den Zeiger `Next (o_pos)` aus dem Container und beeinflusst nicht das referenzierte Objekt. `o_pos` und `Next (o_pos)` müssen gültige Positionen sein. Die Methode liefert die Position des Nachfolgers des gelöschten Eintrags oder Null, wenn der letzte Eintrag gelöscht wurde.

`t_Position DelPrevPtr (t_Position o_pos);`

Die Methode ist identisch mit `FreePrevObj`. Sie entfernt den Zeiger `Prev (o_pos)` aus dem Container und beeinflusst nicht das referenzierte Objekt. `o_pos` und `Prev (o_pos)` müssen gültige Positionen sein. Die Methode liefert `o_pos` zurück, denn `o_pos` ist der Nachfolger des gelöschten Eintrags.

`t_Position DelNthPtr (t_Length u_idx);`

Die Methode ist identisch mit `FreeNthObj`. Sie entfernt den Zeiger `Nth (u_idx)` aus dem Container und beeinflusst nicht das referenzierte Objekt. Der Index `u_idx` muß zwischen Eins und `GetLen` liegen. Die Methode liefert die Position des Nachfolgers des gelöschten Eintrags oder Null, wenn der letzte Eintrag gelöscht wurde.

`void DelAllPtr ();`

Die Methode ist identisch mit `FreeAll`. Der vom Container belegte Speicher wird effizient freigegeben. Es werden jedoch keine Destruktoren referenzierter Objekte aufgerufen.

Löschen von Zeigern und referenzierten Objekten

`t_Position DelPtrAndObj (t_Position o_pos);`

Wirkt wie `DelPtr` und löscht zusätzlich das referenzierte Objekt.

`t_Position DelFirstPtrAndObj ();`

Wirkt wie `DelFirstPtr` und löscht zusätzlich das referenzierte Objekt.

`t_Position DelLastPtrAndObj ();`

Wirkt wie `DelLastPtr` und löscht zusätzlich das referenzierte Objekt.

`t_Position DelNextPtrAndObj (t_Position o_pos);`

Wirkt wie `DelNextPtr` und löscht zusätzlich das referenzierte Objekt.

`t_Position DelPrevPtrAndObj (t_Position o_pos);`

Wirkt wie `DelPrevPtr` und löscht zusätzlich das referenzierte Objekt.

`t_Position DelNthPtrAndObj (t_Length u_idx);`

Wirkt wie `DelNthPtr` und löscht zusätzlich das referenzierte Objekt.

`void DelAllPtrAndObj ();`

Wirkt wie `DelAllPtr` und löscht zusätzlich die referenzierten Objekte.

Vergleich im Zeigercontainer

Ein Zeigercontainer basiert auf einem Container des Typs `gct_AnyContainer <void *>`. Die enthaltenen Objekte sind untypisierte C++-Zeiger. Methoden des Vergleichscontainers, z. B. `SearchFirstObj`,

vergleichen Objekte des Typs `void *` miteinander. Die folgenden Methoden des Zeigercontainers, z. B. `SearchFirstPtr`, werden auf die Methoden des Vergleichscontainers zurückgeführt und vergleichen die im Container enthaltenen C++-Zeiger miteinander, nicht die referenzierten Objekte.

Suche nach Zeigern

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

Liefert `true`, wenn der Container einen Zeiger enthält, der gleich `po_obj` ist.

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

Liefert die Anzahl der Zeiger, die gleich `po_obj` sind.

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

Liefert die Position des ersten Zeigers, der gleich `po_obj` ist, oder `Null`, wenn kein Zeiger gefunden wurde.

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

Liefert die Position des letzten Zeigers, der gleich `po_obj` ist, oder `Null`, wenn kein Zeiger gefunden wurde.

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

Liefert die Position des nächsten Zeigers, der gleich `po_obj` ist, oder `Null`, wenn kein Zeiger gefunden wurde. `o_pos` muß eine gültige Position sein. Die Suche beginnt bei `Next (o_pos)`.

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

Liefert die Position des vorhergehenden Zeigers, der gleich `po_obj` ist, oder `Null`, wenn kein Zeiger gefunden wurde. `o_pos` muß eine gültige Position sein. Die Suche beginnt bei `Prev (o_pos)`.

Bedingtes Einfügen von Zeigern

`t_Position AddPtrCond (const t_obj * po_obj);`

Liefert die Position des ersten Zeigers, der gleich `po_obj` ist, oder die Position eines neu eingefügten Zeigers, wenn kein Zeiger gefunden wurde. Zum Einfügen des neuen Zeigers wird die Methode `AddPtr` aufgerufen.

`t_Position AddPtrBeforeFirstCond (const t_obj * po_obj);`

Liefert die Position des ersten Zeigers, der gleich `po_obj` ist, oder die Position eines neu eingefügten Zeigers, wenn kein Zeiger gefunden wurde. Zum Einfügen des neuen Zeigers wird die Methode `AddPtrBeforeFirst` aufgerufen.

`t_Position AddPtrAfterLastCond (const t_obj * po_obj);`

Liefert die Position des ersten Zeigers, der gleich `po_obj` ist, oder die Position eines neu eingefügten Zeigers, wenn kein Zeiger gefunden wurde. Zum Einfügen des neuen Zeigers wird die Methode `AddPtrAfterLast` aufgerufen.

Löschen gefundener Zeiger

`t_Position DelFirstEqualPtr (const t_obj * po_obj);`

Löscht den ersten Zeiger, der gleich `po_obj` ist. Es muß mindestens ein gleicher Zeiger enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Zeigers oder `Null`, wenn der letzte Zeiger gelöscht wurde.

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

Löscht den letzten Zeiger, der gleich `po_obj` ist. Es muß mindestens ein gleicher Zeiger enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Zeigers oder Null, wenn der letzte Zeiger gelöscht wurde.

Bedingtes Löschen gefundener Zeiger

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

Löscht den ersten Zeiger, der gleich `po_obj` ist, sofern ein gleicher Zeiger gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Zeigers oder Null, wenn kein Zeiger gefunden wurde oder der letzte Zeiger gelöscht wurde.

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

Löscht den letzten Zeiger, der gleich `po_obj` ist, sofern ein gleicher Zeiger gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Zeigers oder Null, wenn kein Zeiger gefunden wurde oder der letzte Zeiger gelöscht wurde.

Löschen gefundener Zeiger und referenzierter Objekte

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

Wirkt wie `DelFirstEqualPtr` und löscht zusätzlich das referenzierte Objekt.

```
t_Position DelLastEqualPtrAndObj (const t_obj * po_obj);
```

Wirkt wie `DelLastEqualPtr` und löscht zusätzlich das referenzierte Objekt.

Bedingtes Löschen gefundener Zeiger und referenzierter Objekte

```
t_Position DelFirstEqualPtrAndObjCond (const t_obj * po_obj);
```

Wirkt wie `DelFirstEqualPtrCond` und löscht zusätzlich das referenzierte Objekt.

```
t_Position DelLastEqualPtrAndObjCond (const t_obj * po_obj);
```

Wirkt wie `DelLastEqualPtrCond` und löscht zusätzlich das referenzierte Objekt.

2.5.3 Operationen mit Zeigercontainern

Objekte einfügen, kopieren und löschen

Das folgende Programmbeispiel demonstriert das Einfügen, Kopieren und Löschen von Objekten in einem Zeigercontainer. Die Klasse `ct_Int` wird im Abschnitt 'Beispielprogramme' beschrieben.

```
ct_Int co_int = 1;
ct_Int * pco_int;
gct_AnyPtrContainer <ct_Int> co_ptrContainer;
gct_AnyPtrContainer <ct_Int>::t_Position o_pos;

// Neues Objekt im Zeigercontainer mit Defaultkonstruktor erzeugen
o_pos = co_ptrContainer. AddPtr (new ct_Int);

// Auf das Objekt zugreifen und es und initialisieren
pco_int = co_ptrContainer. GetPtr (o_pos);
(* pco_int) = 2;

// Vorhandenes Objekt in den Zeigercontainer kopieren
o_pos = co_ptrContainer. AddPtr (new ct_Int (co_int));
```

```
// Objekt aus dem Zeigercontainer nehmen und löschen
co_ptrContainer. DelPtrAndObj (o_pos);
```

Vorwärts iterieren

Zum Iterieren eines Zeigercontainers in aufsteigender Reihenfolge der Einträge wird eine for-Schleife nach folgendem Muster empfohlen:

```
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);
    // ...
}
```

Rückwärts iterieren

Zum Iterieren eines Zeigercontainers in absteigender Reihenfolge der Einträge wird eine for-Schleife nach folgendem Muster empfohlen:

```
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);
    // ...
}
```

Iterieren und verändern

Zum Iterieren und Verändern eines Zeigercontainers wird eine for-Schleife nach folgendem Muster empfohlen:

```
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);
    // ...
}
```

Statt der for-Schleife kann auch eine while-Schleife nach folgendem Muster verwendet werden:

```
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 Zeigervergleichscontainer (tuning/ptrcompcontainer.h)

Die normale Zeigercontainerschnittstelle ist auf universelle Anwendbarkeit ausgelegt und stellt nur geringe Anforderungen an die referenzierten Objekte. Diese müssen nur einen Destruktor (für die Methode `DelPtrAndObj`) zur Verfügung stellen. Einige Objektklassen besitzen jedoch einen Gleichheitsoperator (`operator ==`). Dieser ist implizit auch für alle primitiven Datentypen, z. B. `int` oder `void *`, definiert. Der Gleichheitsoperator ermöglicht zahlreiche weitere Containermethoden, z. B. das bedingte Einfügen und das Suchen eines Elements.

Das Klassentemplate `gct_PtrCompContainer` erwartet als Parameter eine Zeigercontainerklasse, deren Objekttyp einen Gleichheitsoperator enthält, z. B. `gct_Std32PtrArray <float>`. Sie dient dem Zeigervergleichscontainer als Basisklasse.

Basisklassen

<code>gct_AnyContainer</code>	(siehe Abschnitt 'Containerschnittstelle')
<code>gct_ExtContainer</code>	(siehe Abschnitt 'Erweiterter Container')
<code>gct_CompContainer</code>	(siehe Abschnitt 'Vergleichscontainer')
<code>gct_PtrContainer</code>	(siehe Abschnitt 'Zeigercontainer')

Templatedeklaration

```
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);
};
```

Suche nach referenzierten Objekten

`bool ContainsRef (const t_RefObject * po_obj) const;`

Liefert `true`, wenn der Container ein Objekt enthält, das gleich `* po_obj` ist.

`t_Length CountRefs (const t_RefObject * po_obj) const;`

Liefert die Anzahl der Objekte, die gleich `* po_obj` sind.

`t_Position SearchFirstRef (const t_RefObject * po_obj) const;`

Liefert die Position des ersten Objekts, das gleich `* po_obj` ist, oder `Null`, wenn kein Objekt gefunden wurde.

`t_Position SearchLastRef (const t_RefObject * po_obj) const;`

Liefert die Position des letzten Objekts, das gleich `* po_obj` ist, oder `Null`, wenn kein Objekt gefunden wurde.

`t_Position SearchNextRef (t_Position o_pos, const t_RefObject * po_obj) const;`

Liefert die Position des nächsten Objekts, das gleich `* po_obj` ist, oder `Null`, wenn kein Objekt gefunden wurde. `o_pos` muß eine gültige Position sein. Die Suche beginnt bei `Next (o_pos)`.

`t_Position SearchPrevRef (t_Position o_pos, const t_RefObject * po_obj) const;`

Liefert die Position des vorhergehenden Objekts, das gleich `* po_obj` ist, oder `Null`, wenn kein Objekt gefunden wurde. `o_pos` muß eine gültige Position sein. Die Suche beginnt bei `Prev (o_pos)`.

Zugriff auf gefundene Objekte

`t_RefObject * GetFirstEqualRef (const t_RefObject * po_obj) const;`

Liefert einen typisierten Zeiger auf das erste Objekt, das gleich `* po_obj` ist. Es muß mindestens ein gleiches Objekt enthalten sein.

`t_RefObject * GetLastEqualRef (const t_RefObject * po_obj) const;`

Liefert einen typisierten Zeiger auf das letzte Objekt, das gleich `* po_obj` ist. Es muß mindestens ein gleiches Objekt enthalten sein.

Bedingtes Einfügen von Zeigern

`t_Position AddRefCond (const t_RefObject * po_obj);`

Liefert die Position des ersten Objekts, das gleich `* po_obj` ist, oder die Position eines neu eingefügten Zeigers, wenn kein Objekt gefunden wurde. Zum Einfügen des neuen Zeigers wird die Methode `AddPtr` aufgerufen.

`t_Position AddRefBeforeFirstCond (const t_RefObject * po_obj);`

Liefert die Position des ersten Objekts, das gleich `* po_obj` ist, oder die Position eines neu eingefügten Zeigers, wenn kein Objekt gefunden wurde. Zum Einfügen des neuen Zeigers wird die Methode `AddPtrBeforeFirst` aufgerufen.

`t_Position AddRefAfterLastCond (const t_RefObject * po_obj);`

Liefert die Position des ersten Objekts, das gleich `* po_obj` ist, oder die Position eines neu eingefügten Zeigers, wenn kein Objekt gefunden wurde. Zum Einfügen des neuen Zeigers wird die Methode `AddPtrAfterLast` aufgerufen.

Rückgabewert von Löschmethoden

Löschmethoden liefern stets die Position des Nachfolgers des gelöschten Eintrags. Diese Technik ermöglicht das gleichzeitige Iterieren und Verändern eines Containers. Der Rückgabewert wird mit der

Methode `Next` vor dem Löschen berechnet. Wurde der der Reihenfolge nach letzte Eintrag gelöscht (Methode `Last`), ist der Rückgabewert gleich Null.

Löschen von Zeigern gefundener Objekte

```
t_Position DelFirstEqualRef (const t_RefObject * po_obj);
```

Löscht den Zeiger auf das erste Objekt, das gleich `* po_obj` ist. Es muß mindestens ein gleiches Objekt enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Zeigers oder Null, wenn der letzte Zeiger gelöscht wurde.

```
t_Position DelLastEqualRef (const t_RefObject * po_obj);
```

Löscht den Zeiger auf das letzte Objekt, das gleich `* po_obj` ist. Es muß mindestens ein gleiches Objekt enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Zeigers oder Null, wenn der letzte Zeiger gelöscht wurde.

Bedingtes Löschen von Zeigern gefundener Objekte

```
t_Position DelFirstEqualRefCond (const t_RefObject * po_obj);
```

Löscht den Zeiger auf das erste Objekt, das gleich `* po_obj` ist, sofern ein gleiches Objekt gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Zeigers oder Null, wenn kein Objekt gefunden wurde oder der gelöschte Zeiger der letzte war.

```
t_Position DelLastEqualRefCond (const t_RefObject * po_obj);
```

Löscht den Zeiger auf das letzte Objekt, das gleich `* po_obj` ist, sofern ein gleiches Objekt gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Zeigers oder Null, wenn kein Objekt gefunden wurde oder der gelöschte Zeiger der letzte war.

Löschen gefundener Zeiger und referenzierter Objekte

```
t_Position DelFirstEqualRefAndObj (const t_RefObject * po_obj);
```

Wirkt wie `DelFirstEqualRef` und löscht zusätzlich das referenzierte Objekt.

```
t_Position DelLastEqualRefAndObj (const t_RefObject * po_obj);
```

Wirkt wie `DelLastEqualRef` und löscht zusätzlich das referenzierte Objekt.

Bedingtes Löschen gefundener Zeiger und referenzierter Objekte

```
t_Position DelFirstEqualRefAndObjCond (const t_RefObject * po_obj);
```

Wirkt wie `DelFirstEqualRefCond` und löscht zusätzlich das referenzierte Objekt.

```
t_Position DelLastEqualRefAndObjCond (const t_RefObject * po_obj);
```

Wirkt wie `DelLastEqualRefCond` und löscht zusätzlich das referenzierte Objekt.

2.5.5 Mapcontainer (tuning/map.h)

Der Mapcontainer ist ähnlich wie der Vergleichscontainer eine Erweiterung der normalen Containerschnittstelle. Er verwaltet jedoch keine einzelnen Objekte, sondern Schlüssel-Wert-Paare. An die Wertobjekte werden nur geringe Anforderungen gestellt. Sie müssen nur einen normalen und einen Kopierkonstruktor zur Verfügung stellen. Die Schlüsselobjekte müssen zusätzlich einen Gleichheitsoperator (`operator ==`) besitzen. Er ermöglicht das Suchen nach einem Wert mit einem gegebenen Schlüssel.

Das Klassentemplate `gct_Map` erwartet als Parameter eine Containerklasse, deren Objekttyp ein Schlüssel-Wert-Paar ist, z. B. `gct_Std32Array <gct_MapEntry <ct_String, ct_Int> >`. Sie dient dem Mapcontainer als Basisklasse. Mit dem Hilfstemplate `gct_MapEntry` können Schlüssel-Wert-Paare gebildet werden. Wird als Basiscontainer ein sortiertes Array verwendet, müssen die Schlüsselobjekte zusätzlich den Vergleichsoperator '`operator <`' besitzen. Wird als Basiscontainer eine Hashtabelle verwendet, müssen die Schlüsselobjekte zusätzlich die Methode `GetHash` besitzen.

Das Schlüsselobjekt dient dem Hilfstemplate `gct_MapEntry` als Basisklasse. Primitive Datentypen, z. B. `int` oder `char`, können nicht direkt als Schlüssel verwendet werden. Das Hilfstemplate `gct_Key` wandelt einen ganzzahligen Zahlentyp in einen Schlüsseltyp um, z. B. `gct_MapEntry <gct_Key <int>, ct_String>`.

Basisklassen

`gct_AnyContainer` (siehe Abschnitt 'Containerschnittstelle')
 [`gct_ExtContainer` (optional, siehe Abschnitt 'Erweiterter Container')]

Templatedeklaration

```
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 ();
};
```

Datentypen

`typedef t_Object::t_Key t_Key;`

Der geschachtelte Typ `t_Key` ist der Datentyp der Schlüsselobjekte und wird vom Hilfstemplate `gct_MapEntry` übernommen.

`typedef t_Object::t_Value t_Value;`

Der geschachtelte Typ `t_Value` ist der Datentyp der Wertobjekte und wird vom Hilfstemplate `gct_MapEntry` übernommen.

Suche nach Paaren

`bool ContainsKey (t_Key o_key) const;`

Liefert `true`, wenn der Container einen Schlüssel enthält, der gleich `o_key` ist.

`t_Length CountKeys (t_Key o_key) const;`

Liefert die Anzahl der Schlüssel, die gleich `o_key` sind.

`t_Position SearchFirstKey (t_Key o_key) const;`

Liefert die Position des ersten Schlüssel-Wert-Paares, dessen Schlüssel gleich `o_key` ist, oder `Null`, wenn kein Schlüssel gefunden wurde.

`t_Position SearchLastKey (t_Key o_key) const;`

Liefert die Position des letzten Schlüssel-Wert-Paares, dessen Schlüssel gleich `o_key` ist, oder `Null`, wenn kein Schlüssel gefunden wurde.

`t_Position SearchNextKey (t_Position o_pos, t_Key o_key) const;`

Liefert die Position des nächsten Schlüssel-Wert-Paares, dessen Schlüssel gleich `o_key` ist, oder `Null`, wenn kein Schlüssel gefunden wurde. `o_pos` muß eine gültige Position sein. Die Suche beginnt bei `Next (o_pos)`.

`t_Position SearchPrevKey (t_Position o_pos, t_Key o_key) const;`

Liefert die Position des vorhergehenden Schlüssel-Wert-Paares, dessen Schlüssel gleich `o_key` ist, oder `Null`, wenn kein Schlüssel gefunden wurde. `o_pos` muß eine gültige Position sein. Die Suche beginnt bei `Prev (o_pos)`.

Zugriff auf Schlüssel und Wert

`t_Key GetKey (t_Position o_pos) const;`

Liefert den Schlüssel des durch `o_pos` identifizierten Schlüssel-Wert-Paares. `o_pos` muß eine gültige Position sein.

`t_Value * GetValue (t_Position o_pos) const;`

Liefert einen typisierten Zeiger auf den Wert des durch `o_pos` identifizierten Schlüssel-Wert-Paares. `o_pos` muß eine gültige Position sein.

Zugriff auf gefundene Objekte

`t_Value * GetFirstValue (t_Key o_key) const;`

Liefert einen typisierten Zeiger auf den Wert des ersten Schlüssel-Wert-Paares, dessen Schlüssel gleich `o_key` ist. Es muß mindestens ein gleicher Schlüssel enthalten sein.

`t_Value * GetLastValue (t_Key o_key) const;`

Liefert einen typisierten Zeiger auf den Wert des letzten Schlüssel-Wert-Paares, dessen Schlüssel gleich `o_key` ist. Es muß mindestens ein gleicher Schlüssel enthalten sein.

Einfügen von Paaren

`t_Position AddKeyAndValue (t_Key o_key, const t_Value * po_value = 0);`

Fügt ein neues Schlüssel-Wert-Paar in den Container ein und liefert dessen Position. Ist der Zeiger `po_value` gleich `Null`, wird das Wertobjekt mit seinem normalen Konstruktor erzeugt. Andernfalls wird sein Kopierkonstruktor mit dem Parameter `* po_value` aufgerufen.

```
t_Position AddKeyAndValueCond (t_Key o_key, const t_Value * po_value = 0);
```

Liefert die Position des ersten Schlüssel-Wert-Paares, dessen Schlüssel gleich `o_key` ist, oder die Position eines neu eingefügten Paares, wenn kein Paar gefunden wurde. Zum Einfügen des neuen Paares wird die Methode `AddKeyAndValue` aufgerufen.

Rückgabewert von Löschmethoden

Löschmethoden liefern stets die Position des Nachfolgers des gelöschten Eintrags. Diese Technik ermöglicht das gleichzeitige Iterieren und Verändern eines Containers. Der Rückgabewert wird mit der Methode `Next` vor dem Löschen berechnet. Wurde der der Reihenfolge nach letzte Eintrag gelöscht (Methode `Last`), ist der Rückgabewert gleich Null.

Löschen von Paaren

```
t_Position DelKeyAndValue (t_Position o_pos);
```

Ruft den Destruktor des Schlüssel-Wert-Paares auf und gibt den zugehörigen Speicher frei. `o_pos` muß eine gültige Position sein. Die Methode liefert `Next (o_pos)`, also die Position des nächsten Paares oder Null, wenn das letzte Paar gelöscht wurde.

```
void DelAllKeyAndValue ();
```

Ruft die Destruktoren aller Schlüssel-Wert-Paare auf und gibt deren Speicher frei. `DelAllKeyAndValue` ist i. a. schneller als das mehrfache Löschen mit `DelKeyAndValue`.

Löschen gefundener Paare

```
t_Position DelFirstKeyAndValue (t_Key o_key);
```

Löscht das erste Schlüssel-Wert-Paar, dessen Schlüssel gleich `o_key` ist. Es muß mindestens ein gleicher Schlüssel enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Paares oder Null, wenn das letzte Paar gelöscht wurde.

```
t_Position DelLastKeyAndValue (t_Key o_key);
```

Löscht das letzte Schlüssel-Wert-Paar, dessen Schlüssel gleich `o_key` ist. Es muß mindestens ein gleicher Schlüssel enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Paares oder Null, wenn das letzte Paar gelöscht wurde.

Bedingtes Löschen gefundener Paare

```
t_Position DelFirstKeyAndValueCond (t_Key o_key);
```

Löscht das erste Schlüssel-Wert-Paar, dessen Schlüssel gleich `o_key` ist, sofern ein gleicher Schlüssel gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Paares oder Null, wenn kein Schlüssel gefunden wurde oder das gelöschte Paar das letzte war.

```
t_Position DelLastKeyAndValueCond (t_Key o_key);
```

Löscht das letzte Schlüssel-Wert-Paar, dessen Schlüssel gleich `o_key` ist, sofern ein gleicher Schlüssel gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Paares oder Null, wenn kein Schlüssel gefunden wurde oder das gelöschte Paar das letzte war.

2.5.6 Zeigermapcontainer (tuning/ptrmap.h)

Der Zeigermapcontainer ist ähnlich wie der Mapcontainer eine Erweiterung der normalen Containerschnittstelle. Er verwaltet jedoch keine Schlüssel-Wert-Paare, sondern Schlüssel-Zeiger-Paare. Die Schlüsselobjekte müssen einen Gleichheitsoperator (`operator ==`) besitzen. Er ermöglicht das Suchen nach einem Zeiger mit einem gegebenen Schlüssel.

Das Klassentemplate `gct_PtrMap` erwartet als Parameter eine Containerklasse, deren Objekttyp ein Schlüssel-Zeiger-Paar ist, z. B. `gct_Std32Array <gct_PtrMapEntry <ct_String> >`, und den Datentyp des Zeigers. Die Containerklasse dient dem Zeigermapcontainer als Basisklasse. Mit dem Hilfstemplate `gct_PtrMapEntry` wird ein Schlüssel-Zeiger-Paar gebildet. Der Zeiger ist untypisiert (`void *`) und wird erst in den Zugriffsmethoden in einen typisierten Zeiger umgewandelt. Wird als Basiscontainer ein sortiertes Array verwendet, müssen die Schlüsselobjekte zusätzlich den Vergleichsoperator `'operator <'` besitzen. Wird als Basiscontainer eine Hashtabelle verwendet, müssen die Schlüsselobjekte zusätzlich die Methode `GetHash` besitzen.

Das Schlüsselobjekt dient dem Hilfstemplate `gct_PtrMapEntry` als Basisklasse. Primitive Datentypen, z. B. `int` oder `char`, können nicht direkt als Schlüssel verwendet werden. Das Hilfstemplate `gct_Key` wandelt einen ganzzahligen Zahlentyp in einen Schlüsseltyp um, z. B. `gct_PtrMapEntry <gct_Key <int> >`.

Basisklassen

`gct_AnyContainer` (siehe Abschnitt 'Containerschnittstelle')
`[gct_ExtContainer (optional, siehe Abschnitt 'Erweiterter Container')]`

Templatedeklaration

```
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 ();
};
```

Datentypen

`typedef t_Object::t_Key t_Key;`

Der geschachtelte Typ `t_Key` ist der Datentyp der Schlüsselobjekte und wird vom Hilfstemplate `gct_PtrMapEntry` übernommen.

`typedef t_value t_Value;`

Der geschachtelte Typ `t_Value` ist der Datentyp der Wertobjekte und wird als Templateparameter übergeben.

Suche nach Paaren

`bool ContainsKey (t_Key o_key) const;`

Liefert `true`, wenn der Container einen Schlüssel enthält, der gleich `o_key` ist.

`t_Length CountKeys (t_Key o_key) const;`

Liefert die Anzahl der Schlüssel, die gleich `o_key` sind.

`t_Position SearchFirstKey (t_Key o_key) const;`

Liefert die Position des ersten Schlüssel-Zeiger-Paares, dessen Schlüssel gleich `o_key` ist, oder `Null`, wenn kein Schlüssel gefunden wurde.

`t_Position SearchLastKey (t_Key o_key) const;`

Liefert die Position des letzten Schlüssel-Zeiger-Paares, dessen Schlüssel gleich `o_key` ist, oder `Null`, wenn kein Schlüssel gefunden wurde.

`t_Position SearchNextKey (t_Position o_pos, t_Key o_key) const;`

Liefert die Position des nächsten Schlüssel-Zeiger-Paares, dessen Schlüssel gleich `o_key` ist, oder `Null`, wenn kein Schlüssel gefunden wurde. `o_pos` muß eine gültige Position sein. Die Suche beginnt bei `Next (o_pos)`.

`t_Position SearchPrevKey (t_Position o_pos, t_Key o_key) const;`

Liefert die Position des vorhergehenden Schlüssel-Zeiger-Paares, dessen Schlüssel gleich `o_key` ist, oder `Null`, wenn kein Schlüssel gefunden wurde. `o_pos` muß eine gültige Position sein. Die Suche beginnt bei `Prev (o_pos)`.

Zugriff auf Schlüssel und Wert

`t_Key GetKey (t_Position o_pos) const;`

Liefert den Schlüssel des durch `o_pos` identifizierten Schlüssel-Zeiger-Paares. `o_pos` muß eine gültige Position sein.

`t_Value * GetValPtr (t_Position o_pos) const;`

Liefert einen typisierten Zeiger auf den Wert des durch `o_pos` identifizierten Schlüssel-Zeiger-Paares. `o_pos` muß eine gültige Position sein.

Zugriff auf gefundene Objekte

`t_Value * GetFirstValPtr (t_Key o_key) const;`

Liefert einen typisierten Zeiger auf den Wert des ersten Schlüssel-Zeiger-Paares, dessen Schlüssel gleich `o_key` ist. Es muß mindestens ein gleicher Schlüssel enthalten sein.

```
t_Value * GetLastValPtr (t_Key o_key) const;
```

Liefert einen typisierten Zeiger auf den Wert des letzten Schlüssel-Zeiger-Paares, dessen Schlüssel gleich `o_key` ist. Es muß mindestens ein gleicher Schlüssel enthalten sein.

Einfügen von Paaren

```
t_Position AddKeyAndValPtr (t_Key o_key, const t_Value * po_value);
```

Fügt ein neues Schlüssel-Zeiger-Paar in den Container ein und liefert dessen Position.

```
t_Position AddKeyAndValPtrCond (t_Key o_key, const t_Value * po_value);
```

Liefert die Position des ersten Schlüssel-Zeiger-Paares, dessen Schlüssel gleich `o_key` ist, oder die Position eines neu eingefügten Paares, wenn kein Paar gefunden wurde. Zum Einfügen des neuen Paares wird die Methode `AddKeyAndValPtr` aufgerufen.

Rückgabewert von Löschmethoden

Löschmethoden liefern stets die Position des Nachfolgers des gelöschten Eintrags. Diese Technik ermöglicht das gleichzeitige Iterieren und Verändern eines Containers. Der Rückgabewert wird mit der Methode `Next` vor dem Löschen berechnet. Wurde der der Reihenfolge nach letzte Eintrag gelöscht (Methode `Last`), ist der Rückgabewert gleich Null.

Löschen von Paaren

```
t_Position DelKey (t_Position o_pos);
```

Ruft den Destruktor des Schlüssel-Zeiger-Paares auf und gibt den zugehörigen Speicher frei. `o_pos` muß eine gültige Position sein. Die Methode liefert `Next (o_pos)`, also die Position des nächsten Paares oder Null, wenn das letzte Paar gelöscht wurde.

```
void DelAllKey ();
```

Ruft die Destruktoren aller Schlüssel-Zeiger-Paare auf und gibt deren Speicher frei. `DelAllKey` ist i. a. schneller als das mehrfache Löschen mit `DelKey`.

Löschen gefundener Paare

```
t_Position DelFirstKey (t_Key o_key);
```

Löscht das erste Schlüssel-Zeiger-Paar, dessen Schlüssel gleich `o_key` ist. Es muß mindestens ein gleicher Schlüssel enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Paares oder Null, wenn das letzte Paar gelöscht wurde.

```
t_Position DelLastKey (t_Key o_key);
```

Löscht das letzte Schlüssel-Zeiger-Paar, dessen Schlüssel gleich `o_key` ist. Es muß mindestens ein gleicher Schlüssel enthalten sein. Die Methode liefert die Position des Nachfolgers des gelöschten Paares oder Null, wenn das letzte Paar gelöscht wurde.

Bedingtes Löschen gefundener Paare

```
t_Position DelFirstKeyCond (t_Key o_key);
```

Löscht das erste Schlüssel-Zeiger-Paar, dessen Schlüssel gleich `o_key` ist, sofern ein gleicher Schlüssel gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Paares oder Null, wenn kein Schlüssel gefunden wurde oder das gelöschte Paar das letzte war.

```
t_Position DelLastKeyCond (t_Key o_key);
```

Löscht das letzte Schlüssel-Zeiger-Paar, dessen Schlüssel gleich `o_key` ist, sofern ein gleicher Schlüssel gefunden wurde. Die Methode liefert die Position des Nachfolgers des gelöschten Paares oder Null, wenn kein Schlüssel gefunden wurde oder das gelöschte Paar das letzte war.

Löschen von Paaren und referenzierten Objekten

```
t_Position DelKeyAndValue (t_Position o_pos);
```

Wirkt wie `DelKey` und löscht zusätzlich das referenzierte Objekt.

```
void DelAllKeyAndValue ();
```

Wirkt wie `DelAllKey` und löscht zusätzlich die referenzierten Objekte.

Löschen gefundener Paare und referenzierter Objekte

```
t_Position DelFirstKeyAndValue (t_Key o_key);
```

Wirkt wie `DelFirstKey` und löscht zusätzlich das referenzierte Objekt.

```
t_Position DelLastKeyAndValue (t_Key o_key);
```

Wirkt wie `DelLastKey` und löscht zusätzlich das referenzierte Objekt.

Bedingtes Löschen gefundener Paare und referenzierter Objekte

```
t_Position DelFirstKeyAndValueCond (t_Key o_key);
```

Wirkt wie `DelFirstKeyCond` und löscht zusätzlich das referenzierte Objekt.

```
t_Position DelLastKeyAndValueCond (t_Key o_key);
```

Wirkt wie `DelLastKeyCond` und löscht zusätzlich das referenzierte Objekt.

2.6 Zeigercontainer-Instanzen

2.6.1 Zeigerarray-Instanzen (tuning/xxx/ptrarray.h)

Zur Erleichterung des Umgangs mit Zeigerarraycontainern werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `PTR_ARRAY_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Zeigerarraytemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
PTR_ARRAY_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
template <class t_obj> class gct_AnyPtrArray:
    public gct_PtrContainer <t_obj, gct_AnyArray <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 *> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '`ptrarray.h`'. Darin werden mit Hilfe des Makros `PTR_ARRAY_DCLS` nach obigem Muster vier Zeigerarraytemplates deklariert.

In der Datei 'tuning/std/ptrarray.h' werden deklariert:

```
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;
```

In der Datei 'tuning/rnd/ptrarray.h' werden deklariert:

```
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;
```

In der Datei 'tuning/chn/ptrarray.h' werden deklariert:

```
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 Zeigerlisten-Instanzen (tuning/xxx/ptrdlist.h)

Zur Erleichterung des Umgangs mit Zeigerlistencontainern werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `PTR_DLIST_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Zeigerlistentemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

`PTR_DLIST_DCLS (Any)`

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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 *> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**ptrdlist.h**'. Darin werden mit Hilfe des Makros `PTR_DLIST_DCLS` nach obigem Muster vier Zeigerlistentemplates deklariert.

In der Datei 'tuning/std/ptrdlist.h' werden deklariert:

```
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;
```

In der Datei 'tuning/rnd/ptrdlist.h' werden deklariert:

```
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;
```

In der Datei 'tuning/chn/ptrdlist.h' werden deklariert:

```
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 Sortierte Zeigerarray-Instanzen (tuning/xxx/ptrsortedarray.h)

Zur Erleichterung des Umgangs mit sortierten Zeigerarraycontainern werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `PTR_SORTEDARRAY_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Zeigerarraytemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
PTR_SORTEDARRAY_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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_Any8_PtrSortedArray:
    public gct_PtrContainer <t_obj, gct_Any8_SortedArray <gct_SortedArrayRef <t_obj> > > { };
template <class t_obj> class gct_Any16_PtrSortedArray:
    public gct_PtrContainer <t_obj, gct_Any16_SortedArray <gct_SortedArrayRef <t_obj> > > { };
template <class t_obj> class gct_Any32_PtrSortedArray:
    public gct_PtrContainer <t_obj, gct_Any32_SortedArray <gct_SortedArrayRef <t_obj> > > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**ptrsortedarray.h**'. Darin werden mit Hilfe des Makros `PTR_SORTEDARRAY_DCLS` nach obigem Muster vier Zeigerarraytemplates deklariert.

In der Datei '**tuning/std/ptrsortedarray.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/rnd/ptrsortedarray.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/chn/ptrsortedarray.h**' werden deklariert:

```
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 Zeigerhashtabellen-Instanzen (tuning/xxx/ptrhashtable.h)

Zur Erleichterung des Umgangs mit Zeigerhashtabellencontainern werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `PTR_HASHTABLE_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Zeigerhashtabellentemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
PTR_HASHTABLE_DCLS (Any)
```


expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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> > > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**ptrhashtable.h**'. Darin werden mit Hilfe des Makros `PTR_HASHTABLE_DCLS` nach obigem Muster vier Zeigerhashtabellentemplates deklariert.

In der Datei '**tuning/std/ptrhashtable.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/rnd/ptrhashtable.h**' werden deklariert:

```
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;
```

In der Datei '**tuning/chn/ptrhashtable.h**' werden deklariert:

```
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 Blockzeigerlisten-Instanzen (tuning/xxx/blockptrdlist.h)

Zur Erleichterung des Umgangs mit Blockzeigerlisten werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `BLOCKPTR_DLIST_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Blockzeigerlistentemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
BLOCKPTR_DLIST_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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 *> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**blockptrdlist.h**'. Darin werden mit Hilfe des Makros `BLOCKPTR_DLIST_DCLS` nach obigem Muster vier Blockzeigerlistentemplates deklariert.

In der Datei '**tuning/std/blockptrdlist.h**' werden deklariert:

```
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;
```

In der Datei 'tuning/rnd/blockptrdlist.h' werden deklariert:

```
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;
```

In der Datei 'tuning/chn/blockptrdlist.h' werden deklariert:

```
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 Refzeigerlisten-Instanzen (tuning/xxx/refptrdlist.h)

Zur Erleichterung des Umgangs mit Refzeigerlisten werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `REFPTR_DLST_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Refzeigerlistentemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
REFPTR_DLST_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
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 *> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**refptrdlist.h**'. Darin werden mit Hilfe des Makros `REFPTR_DLST_DCLS` nach obigem Muster vier Refzeigerlistentemplates deklariert.

In der Datei 'tuning/std/refptrdlist.h' werden deklariert:

```
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;
```

In der Datei 'tuning/rnd/refptrdlist.h' werden deklariert:

```
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;
```

In der Datei 'tuning/chn/refptrdlist.h' werden deklariert:

```
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_Chn32RefPtrDLList;
```

2.6.7 Blockrefzeigerlisten-Instanzen (tuning/xxx/blockrefptrdlist.h)

Zur Erleichterung des Umgangs mit Blockrefzeigerlisten werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen vordefiniert. Das Makro `BLOCKREFPTR_DLST_DCLS(Obj)` generiert ähnlich wie `BLOCK_DCLS(Obj)` für jede der vier Wrapperklassen eines globalen Storeobjekts je ein Blockrefzeigerlistentemplate, das nur noch den Parameter `t_obj` besitzt. Die Makroverwendung

```
BLOCKREFPTR_DLST_DCLS (Any)
```

expandiert zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
template <class t_obj> class gct_AnyBlockRefPtrDLList: public  
    gct_PtrContainer <t_obj, gct_AnyBlockRefDLList <void *> > { };  
template <class t_obj> class gct_Any8BlockRefPtrDLList: public  
    gct_PtrContainer <t_obj, gct_Any8BlockRefDLList <void *> > { };  
template <class t_obj> class gct_Any16BlockRefPtrDLList: public  
    gct_PtrContainer <t_obj, gct_Any16BlockRefDLList <void *> > { };  
template <class t_obj> class gct_Any32BlockRefPtrDLList: public  
    gct_PtrContainer <t_obj, gct_Any32BlockRefDLList <void *> > { };
```

Jedes Verzeichnis eines dynamischen Stores enthält eine Datei '**blockrefptrdlist.h**'. Darin werden mit Hilfe des Makros `BLOCKREFPTR_DLST_DCLS` nach obigem Muster vier Blockrefzeigerlistentemplates deklariert.

In der Datei '**tuning/std/blockrefptrdlist.h**' werden deklariert:

```
template <class t_obj> class gct_StdBlockRefPtrDLList;  
template <class t_obj> class gct_Std8BlockRefPtrDLList;  
template <class t_obj> class gct_Std16BlockRefPtrDLList;  
template <class t_obj> class gct_Std32BlockRefPtrDLList;
```

In der Datei '**tuning/rnd/blockrefptrdlist.h**' werden deklariert:

```
template <class t_obj> class gct_RndBlockRefPtrDLList;  
template <class t_obj> class gct_Rnd8BlockRefPtrDLList;  
template <class t_obj> class gct_Rnd16BlockRefPtrDLList;  
template <class t_obj> class gct_Rnd32BlockRefPtrDLList;
```

In der Datei '**tuning/chn/blockrefptrdlist.h**' werden deklariert:

```
template <class t_obj> class gct_ChnBlockRefPtrDLList;  
template <class t_obj> class gct_Chn8BlockRefPtrDLList;  
template <class t_obj> class gct_Chn16BlockRefPtrDLList;  
template <class t_obj> class gct_Chn32BlockRefPtrDLList;
```

2.7 Übersicht Container-Instanzen

2.7.1 Vordefinierte Templateinstanzen

Zur besseren Orientierung in der großen Menge vordefinierter Standardinstanzen wurde eine einheitliche Namensgebung verwendet. Die mit einem `DCLS`-Makro generierten Namen bestehen aus sieben Teilen.

1. Präfix

Jeder vordefinierte Container ist ein Template, besitzt das Präfix `gct_` und genau einen Templateparameter, den Typ der verwalteten Objekte.

2. Globaler Store

Es folgt das Kürzel für das globale Storeobjekt, von dem der Container seinen Speicher anfordert. Vordefiniert sind `Std`, `Rnd` und `Chn`.

3. Längentyp

Es folgt das Kürzel für den geschachtelten Längentyp. Dieser beeinflusst nicht nur die Anzahl verarbeitbarer Objekte, sondern auch den Speicherbedarf. Bei Arrays und Blocklisten ist der Längentyp gleich dem Positionstyp. Vordefiniert sind `_`, `8`, `16` und `32`.

4. Optional Block

Bei Listencontainern kann an dieser Stelle `optional Block` angegeben werden. Blocklisten bringen ihre Nodes kompakt in einem Blockstore unter.

5. Optional Ref

Bei Listencontainern kann an dieser Stelle `optional Ref` angegeben werden. Reflisten ordnen jedem Node einen Referenzzähler zu. Damit können sichere Zeiger auf Listeneinträge implementiert werden.

6. Optional Ptr

An dieser Stelle kann `optional Ptr` angegeben werden. Zeigercontainer enthalten ihre Objekte nicht selbst, sondern verwalten nur Zeiger darauf.

7. Containertyp

Am Ende wird der Containertyp `Array`, `DList`, `SortedArray` oder `HashTable` angegeben.

Die folgende Tabelle faßt die Namensbildung der vordefinierten Container zusammen.

Präfix	Glob. Store	t_Length	Opt. Block	Opt. Ref	Opt. Ptr	Cont.typ
gct_	Std	_	Block	Ref	Ptr	Array
	Rnd	8	-	-	-	DList
	Chn	16				SortedArray
		32				HashTable

2.7.2 Selbstdefinierte Templateinstanzen

Die vordefinierten Templateinstanzen basieren auf den drei dynamischen Stores Standardstore, Roundstore und Chainstore sowie dem allgemeinen Blocktemplate `gct_Block`. Neben den vordefinierten Instanzen können natürlich auch beliebige andere Instanzen gebildet werden, indem z. B. statt `gct_Block` `gct_FixBlock`, `gct_MiniBlock` oder `gct_ResBlock` verwendet wird. Weiterhin können auch selbstdefinierte Store- und Blockimplementierungen zum Einsatz kommen. Wegen der großen Zahl möglicher

Kombinationen können weitere Templateinstanzen nicht vordefiniert werden. Man sollte sich bei selbstdefinierten Templateinstanzen aber an die Struktur und Namensgebung der vordefinierten Instanzen halten, damit aus dem Namen die Eigenschaften der Instanzen erkennbar sind. Auf diese Weise können z. B. die folgenden Instanzen definiert werden:

```
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 Abstraktes Objekt (tuning/object.hpp)

Container sind homogen und enthalten stets gleichartige Objekte. Collections sind hingegen polymorph. Sie können Objekte unterschiedlichen Typs verwalten. Zum typisierten Zugriff auf diese Objekte wird die abstrakte Basisklasse `ct_Object` definiert. Sie enthält einen virtuellen Destruktor. Dieser stellt sicher, daß beim Zerstören abgeleiteter Objekte der richtige Destruktor aufgerufen wird. Alle von `ct_Object` abgeleiteten Klassen können in Collections verwaltet werden.

Klassendeklaration

```
class ct_Object
{
public:
    virtual ~ct_Object ();
    virtual bool operator < (const ct_Object & co_comp) const;
    virtual t_UInt GetHashCode () const;
};
```

Methoden

`~ct_Object ();`

Die Klasse `ct_Object` dient als eine abstrakte Basisklasse. Der virtuelle Destruktor sichert das korrekte Zerstören abgeleiteter Objekte in einem polymorphen Kontext.

```
bool operator < (const ct_Object & co_comp) const;
```

Der Vergleichsoperator 'operator <' wird in der Collection `ct_SortedArray` zum sortierten Einfügen eines neuen Elements benötigt.

```
t_UInt GetHashCode () const;
```

Die Methode `GetHash` wird in einem Hashtabellencontainer zum Einfügen eines neuen Elements benötigt.

2.8.2 Abstrakte Collection (tuning/collection.hpp)

Collections können nicht nur polymorphe Objekte verwalten, sondern bilden auch selber einen polymorphen Klassenbaum. Sie erben von der abstrakten Basisklasse `ct_Collection`. Die Collectionschnittstelle gleicht syntaktisch und semantisch der eines Zeigercontainers, z. B. `gct_Chn_PtrArray <ct_Object>`.

Die Verwendung einer einheitlichen Schnittstelle erleichtert dem Anwender das Austauschen von Containern und Collections. Sie ermöglicht zudem eine einfache Implementierung von Collections durch Mappen der Funktionalität eines Zeigercontainers. Im Gegensatz zu einem Zeigercontainer sind jedoch sämtliche Methoden der Klasse `ct_Collection` rein virtuell deklariert. Sie müssen in abgeleiteten Klassen (konkreten Collections) definiert werden.

Basisklasse

`ct_Object` (siehe Abschnitt 'Abstraktes Objekt')

Klassendeklaration

```
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;
};

```

Methoden

Die Beschreibung der Methoden ist identisch mit der Schnittstelle des Zeigercontainers und wird nicht wiederholt (siehe Abschnitt 'Zeigercontainer', Template `gct_PtrContainer`).

2.8.3 Operationen mit Collections

Objekte einfügen, kopieren und löschen

Das folgende Programmbeispiel demonstriert das Einfügen, Kopieren und Löschen von Objekten in einer Collection. Die Klasse `ct_Int` wird im Abschnitt 'Beispielprogramme' beschrieben.

```

ct_Int co_int = 1;
ct_Int * pco_int;
ct_AnyCollection co_collection;
ct_AnyCollection::t_Position o_pos;

// Neues Objekt in der Collection mit Defaultkonstruktor erzeugen
o_pos = co_collection. AddPtr (new ct_Int);

// Auf das Objekt zugreifen und es und initialisieren
pco_int = (ct_Int *) co_collection. GetPtr (o_pos);
(* pco_int) = 2;

// Vorhandenes Objekt in die Collection kopieren
o_pos = co_collection. AddPtr (new ct_Int (co_int));

// Objekt aus der Collection nehmen und löschen

```

```
co_collection. DelPtrAndObj (o_pos);
```

Vorwärts iterieren

Zum Iterieren einer Collection in aufsteigender Reihenfolge der Einträge wird eine for-Schleife nach folgendem Muster empfohlen:

```
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);
    // ...
}
```

Rückwärts iterieren

Zum Iterieren einer Collection in absteigender Reihenfolge der Einträge wird eine for-Schleife nach folgendem Muster empfohlen:

```
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);
    // ...
}
```

Iterieren und verändern

Zum Iterieren und Verändern einer Collection wird eine for-Schleife nach folgendem Muster empfohlen:

```
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);
    // ...
}
```

Statt der for-Schleife kann auch eine while-Schleife nach folgendem Muster verwendet werden:

```
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 Abstrakte Refcollection (tuning/refcollection.hpp)

Die Klasse `ct_RefCollection` erweitert die Collectionschnittstelle um Methoden zum Verarbeiten von Referenzzählern. Eine Refcollection wird mit Hilfe eines Refzeigercontainers, z. B. `gct_Chn_RefPtrDList<ct_Object>`, implementiert (siehe Abschnitt 'Refliste', Template `gct_RefDList`). Die erweiterte Schnittstelle entspricht syntaktisch und semantisch der des zugehörigen Containers.

Basisklassen

`ct_Object` (siehe Abschnitt 'Abstraktes Objekt')
`ct_Collection` (siehe Abschnitt 'Abstrakte Collection')

Klassendeklaration

```

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;
};

```

In einer Refcollection wird jedem einzelnen Node ein Referenzzähler zugeordnet. Dieser ermöglicht die Implementierung von sicheren Zeigern auf Collectioneinträge. Ein sicherer Zeiger erhöht den Referenzzähler des Eintrags, auf den er verweist.

Ein Positionszeiger einer Refcollection behält seine Gültigkeit, solange der Eintrag nicht (z. B. mit `DelPtr`) gelöscht wurde oder der Referenzzähler ungleich Null ist. Wurde das Element mit `DelPtr` aus der Collection entfernt, liefert `IsAlloc` den Wert `false`, und es kann nicht mehr mit `GetPtr` auf das Objekt zugegriffen werden. Erreicht der Referenzzähler mit `DecRef` den Wert Null, wird auch der zugehörige Speicher freigegeben, und der Positionszeiger verliert seine Gültigkeit.

Methoden

`void IncRef (t_Position o_pos);`

Erhöht den zum Collectioneintrag `o_pos` gehörenden Referenzzähler. `o_pos` muß eine gültige Position sein.

`void DecRef (t_Position o_pos);`

Verkleinert den zum Collectioneintrag `o_pos` gehörenden Referenzzähler. `o_pos` muß eine gültige Position sein.

`t_RefCount GetRef (t_Position o_pos) const;`

Liefert den Wert des zum Collectioneintrag `o_pos` gehörenden Referenzzählers. `o_pos` muß eine gültige Position sein.

`bool IsAlloc (t_Position o_pos) const;`

Liefert `true`, wenn der zum Positionszeiger `o_pos` gehörende Collectioneintrag nicht (z. B. mit `DelPtr`) gelöscht wurde und mit `GetPtr` darauf zugegriffen werden kann. `o_pos` muß eine gültige Position sein.

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

Diese Methode ist die logische Negation von `IsAlloc`. `o_pos` muß eine gültige Position sein.

2.8.5 Konkrete Collections

Zur Erleichterung des Umgangs mit der Collectionschnittstelle werden in der Bibliothek **Spirick Tuning** einige konkrete Collections vordefiniert. Das Makro `COLLMAP_DCL` deklariert eine Collectionklasse. Ihre Implementierung erfolgt mit Hilfe des Makros `COLLMAP_DEF`. Beide Makros werden in der Datei **'tuning/collmap.hpp'** definiert.

Die Deklaration einer Collectionklasse wurde so gestaltet, daß keine Einbeziehung von Headerdateien mit Templates notwendig ist. Damit erhöht sich die Übersetzungsgeschwindigkeit gegenüber der Verwendung von Containern.

Verwaltungsart	Implementierung	Übersetzungszeit	Laufzeit
Container	Templates, Inline-Methoden	langsam	schnell
Collection	virtuelle Methoden	schnell	langsam

Zum Deklarieren einer konkreten Collectionklasse wird in einer Headerdatei das Makro `COLLMAP_DCL` plazierte. Z. B. expandiert die Makroverwendung

```
COLLMAP_DCL (Array)
```

zu folgendem Text (der Makroparameter ist fett hervorgehoben):

```
class ct_Array: public ct_Collection
{
    // ...
};
```

Die Implementierung erfolgt in einer anderen Datei. Dabei ist die Einbeziehung des zugehörigen Zeigercontainers notwendig. Dieser wird beim Makro `COLLMAP_DEF` als zweiter Parameter angegeben.

```
#include "tuning/chn/ptrarray.h"
COLLMAP_DEF (Array, gct_Chn_PtrArray)
```

Zum Deklarieren und Implementieren einer Refcollection dienen die Makros `REFCOLLMAP_DCL` und `REFCOLLMAP_DEF` aus der Datei **'tuning/refcollmap.hpp'**. Die vordefinierten Collection- und Refcollectionklassen werden auf Zeigercontainer des Typs `gct_Chn_...` zurückgeführt.

In der Datei 'tuning/array.hpp' wird deklariert:

```
class ct_Array: public ct_Collection { /*...*/ };
```

In der Datei 'tuning/dlist.hpp' wird deklariert:

```
class ct_DList: public ct_Collection { /*...*/ };
```

In der Datei 'tuning/sortedarray.hpp' wird deklariert:

```
class ct_SortedArray: public ct_Collection { /*...*/ };
```

In der Datei 'tuning/blockdlist.hpp' wird deklariert:

```
■ class ct_BlockDList: public ct_Collection { /*...*/ };
```

In der Datei 'tuning/refdlist.hpp' wird deklariert:

```
■ class ct_RefDList: public ct_RefCollection { /*...*/ };
```

In der Datei 'tuning/blockrefdlist.hpp' wird deklariert:

```
■ class ct_BlockRefDList: public ct_RefCollection { /*...*/ };
```

3 ZEICHENKETTEN UND SYSTEMDIENSTE

3.1 Systemschnittstelle

3.1.1 Ressourcenfehler (tuning/sys/creserror.hpp)

Die Datei 'tuning/sys/creserror.hpp' enthält Fehlercodes, die bei der Verwendung systemnaher Ressourcen auftreten können.

Aufzählung

```
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

Es ist kein Fehler aufgetreten.

ec_ResUnknownError

Es ist ein unbekannter Fehler aufgetreten.

ec_ResUninitialized

Es wurde versucht, ein nicht initialisiertes Objekt zu verwenden.

ec_ResAlreadyInitialized

Es wurde versucht, ein bereits initialisiertes Objekt erneut zu initialisieren.

ec_ResInvalidKey

Der Schlüssel ist ungültig.

ec_ResInvalidValue

Ein Funktionsparameter ist ungültig.

ec_ResNoKey

Es wurde versucht, ein Objekt ohne Schlüssel zu verwenden.

ec_ResAlreadyExists

Beim Erzeugen eines Objektes wurde festgestellt, daß bereits ein Objekt mit demselben Schlüssel existiert.

ec_ResAccessDenied

Beim Erzeugen oder Öffnen eines Objektes wurde der Zugriff verweigert.

ec_ResNotFound

Beim Öffnen eines Objektes wurde keine Ressource mit dem angegebenen Schlüssel gefunden.

ec_ResLockCountMismatch

Bei einem Mutexobjekt wurden Sperr-/Freigabe-Aufrufe nicht paarweise verwendet.

ec_ResLockFailed

Das Sperren eines Mutexobjektes ist fehlgeschlagen.

ec_ResUnlockFailed

Das Freigeben eines Mutexobjektes ist fehlgeschlagen.

ec_ResMemMapFailed

Das Zuordnen von Shared Memory in den lokalen Speicher ist fehlgeschlagen.

ec_ResUnmapFailed

Das Freigeben von Shared Memory ist fehlgeschlagen.

ec_ResQuerySizeFailed

Die Abfrage der Größe von Shared Memory ist fehlgeschlagen.

3.1.2 Zeichen und Zeichenketten (tuning/sys/cstring.hpp)

Die Systemschnittstelle für Zeichenketten enthält Funktionen zum Umwandeln von Zeichenketten sowie zur Längen- und Hashwertberechnung. Von allen Funktionen existiert jeweils eine Version für die Datentypen `char` und `wchar_t`. Alle Längenangaben beziehen sich auf die Anzahl der Zeichen und nicht auf die Anzahl der Bytes.

Die Umwandlung von Klein- in Großbuchstaben ist auf eine hohe Rechengeschwindigkeit ausgelegt und verwendet keine Systemrufe, die vom aktiven Locale abhängen. Sie arbeitet mit einem reinen 8-Bit-Zeichensatz und ist nicht auf Unicodezeichen anwendbar (UTF-8 oder UTF-16). Es wird je eine Tabelle nach der Windows-1252 Kodierung verwendet. Das ist eine Obermenge von ISO 8859-1 (Latin-1).

Die zweite Version dieser Umwandlungsfunktionen (`t1_ToUpper2/t1_ToLower2`) existiert zunächst nur für `wchar_t`. Die Implementierung verwendet performante, systemnahe Funktionen, die in den meisten Fällen auch für Unicodezeichen ein korrektes Resultat liefern (MS Windows: `CharUpperW`, Linux: `towupper`). Auf der Grundlage der Widecharacter-Funktionen wurden auch zwei Umwandlungsfunktionen für UTF-8-Strings implementiert. Dabei wird der String intern temporär in einen Widecharacter-String umgewandelt.

Für die Umwandlungen zwischen Multibytecharacters (UTF-8) und Widecharacters (MS Windows: UTF-16, Linux: UTF-16 oder UTF-32) existieren zunächst einmal die Richtungen `char` in `wchar_t` und `wchar_t` in `char`. Wegen der späteren Verwendung in `char`- und `wchar_t`-basierten Templates sind auch die Richtungen `char` in `char` und `wchar_t` in `wchar_t` als reine Kopieroperationen implementiert.

Funktionen

```
char t1_ToUpper (char c);  
wchar_t t1_ToUpper (wchar_t c);
```

Wandelt das Zeichen `c` in einen Großbuchstaben um.

```
char t1_ToLower (char c);  
wchar_t t1_ToLower (wchar_t c);
```

Wandelt das Zeichen `c` in einen Kleinbuchstaben um.

```
bool t1_ToUpper (char * pc_str);  
bool t1_ToUpper (wchar_t * pc_str);
```

Wandelt die nullterminierte Zeichenkette `pc_str` in Großbuchstaben um. Die Umwandlung erfolgt inplace, also in der übergebenen Zeichenkette selbst.

```
bool t1_ToLower (char * pc_str);  
bool t1_ToLower (wchar_t * pc_str);
```

Wandelt die nullterminierte Zeichenkette `pc_str` in Kleinbuchstaben um. Die Umwandlung erfolgt inplace, also in der übergebenen Zeichenkette selbst.

```
wchar_t t1_ToUpper2 (wchar_t c);
```

Wandelt das Zeichen `c` in einen Großbuchstaben um (Unicode-konform, s.o.).

```
wchar_t t1_ToLower2 (wchar_t c);
```

Wandelt das Zeichen `c` in einen Kleinbuchstaben um (Unicode-konform, s.o.).

```
bool t1_ToUpper2 (char * pc_str);  
bool t1_ToUpper2 (wchar_t * pc_str);
```

Wandelt die nullterminierte Zeichenkette `pc_str` in Großbuchstaben um. Die Umwandlung erfolgt inplace, also in der übergebenen Zeichenkette selbst (Unicode-konform, s.o.).

```
bool t1_ToLower2 (char * pc_str);  
bool t1_ToLower2 (wchar_t * pc_str);
```

Wandelt die nullterminierte Zeichenkette `pc_str` in Kleinbuchstaben um. Die Umwandlung erfolgt inplace, also in der übergebenen Zeichenkette selbst (Unicode-konform, s.o.).

```
t_UInt t1_StringLength (const char * pc);  
t_UInt t1_StringLength (const wchar_t * pc);
```

Ermittelt die Länge der nullterminierten Zeichenkette `pc`.

```
unsigned t1_StringHash (const char * pc, t_UInt u_length);  
unsigned t1_StringHash (const wchar_t * pc, t_UInt u_length);
```

Berechnet einen Hashwert für die Zeichenkette.

```
t_UInt t1_MbConvertCount (wchar_t *, const char * pc_src);
```

Berechnet die Anzahl der Widecharacters im Zielspeicher inklusive des abschließenden Nullzeichens für die Umwandlung der nullterminierten Zeichenkette `pc_src` in Widecharacters.

```
bool t1_MbConvert (wchar_t * pc_dst, const char * pc_src, t_UInt u_count);
```

Wandelt die nullterminierten Multibytecharacters `pc_src` in die Widecharacters `pc_dst` inklusive des abschließenden Nullzeichens um. Der Parameter `u_count` gibt die Anzahl der Widecharacters in `pc_dst` an.

```
t_UInt t1_MbConvertCount (char *, const wchar_t * pc_src);
```

Berechnet die Anzahl der Bytes im Zielspeicher inklusive des abschließenden Nullzeichens für die Umwandlung der nullterminierten Widecharacters `pc_src` in Multibytecharacters.

```
bool t1_MbConvert (char * pc_dst, const wchar_t * pc_src, t_UInt u_count);
```

Wandelt die nullterminierten Widecharacters `pc_src` in die Multibytecharacters `pc_dst` inklusive des abschließenden Nullzeichens um. Der Parameter `u_count` gibt die Anzahl der Bytes in `pc_dst` an.

Zugehörige Klassen

Die globalen Funktionen dieser Schnittstelle dienen als Grundlage der Klassen `ct_String` und `ct_WString`.

3.1.3 Unicode (UTF) (tuning/sys/cutf.hpp)

Die Systemschnittstelle für Unicode enthält Funktionen zum Umwandeln von Zeichenketten sowie zur Längenberechnung und zur Groß-/Kleinschreibung. Bei den meisten Funktionen werden nullterminierte Zeichenketten anders behandelt als nicht-nullterminierte. Ist der Parameter `b_null` gleich `true`, dann wird am Ende der Zeichenkette ein Nullzeichen erwartet, und innerhalb der Zeichenkette dürfen sich keine Nullzeichen befinden. Andernfalls werden Nullzeichen wie normale Steuerzeichen behandelt.

Die UTF-Funktionen liefern im Fehlerfall einen genauen Fehlercode. In einigen Funktionen wird der Zeiger auf die Quelldaten als Referenz übergeben. Im Fehlerfall verweist dieser Zeiger dann auf die betroffene Stelle in der Zeichenkette.

Aufzählung

```
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
};
```

Für UTF-8 wird der Datentyp `t_UInt8` verwendet, für UTF-16 `t_UInt16` und für UTF-32 `t_UInt32`. Alle Längenangaben beziehen sich auf den jeweiligen Datentyp und nicht auf die Größe in Bytes.

Zum Umwandeln verschiedener UTF-Kodierungen gibt es die Richtungen UTF-8 <-> UTF-32, UTF-16 <-> UTF-32 und UTF-8 <-> UTF-16. Für jede Richtung gibt es jeweils eine Count- und eine Convert-Funktion. Die Count-Funktion berechnet die Größe des Zielpuffers für die Umwandlung, in der Convert-Funktion wird die Umwandlung ausgeführt (inkl. Nullzeichen, falls der Parameter `b_null` gleich `true` ist).

Für die Count-Funktion wird eigentlich kein Zeiger auf den Zielpuffer benötigt. Zur Unterscheidung der überladenen Funktionen muß aber der Datentyp dieses Zeigers angegeben werden. Der Zeiger selbst wird in der Count-Funktion nicht verwendet, es kann z.B. ein Null-Zeiger angegeben werden.

Die Length-Funktionen berechnen die Anzahl der Unicode-Zeichen in einer UTF-kodierten Zeichenkette. Ist der Parameter `b_null` gleich `true`, dann wird das abschließende Nullzeichen mitgezählt. Bei UTF-32 ist diese Anzahl gleich der Länge der Zeichenkette. Die Length-Funktion für UTF-32 prüft jedoch, ob die Zeichenkette fehlerfrei ist, und liefert nur dann diese Anzahl.

Die Upper/Lower-Funktionen wandeln eine UTF-kodierte Zeichenkette in Groß- bzw. Kleinbuchstaben um. Die Konvertierung erfolgt direkt in der Zeichenkette. Es werden nur Zeichen aus der Basic Multilingual Plane (< 0x10000) umgewandelt, und die Konvertierung erfolgt auch nur dann, wenn sich dadurch die Länge der Zeichenkette nicht verändert.

Funktionen

```
et_UtfError t1_UtfConvertCount (t_UIntY *, t_UInt & u_dstLen, const t_UIntX * & pu_src, t_UInt u_srcLen, bool b_null = true);
```

Liefert im Parameter `u_dstLen` die Größe des Puffers für die Umwandlung der Zeichenkette `pu_src` (UTF-X) mit der Länge `u_srcLen` in eine Zeichenkette vom Typ UTF-Y.

```
et_UtfError t1_UtfConvert (t_UIntY * pu_dst, t_UInt u_dstLen, const t_UIntX * pu_src, t_UInt u_srcLen, bool b_null = true);
```

Konvertiert die Zeichenkette `pu_src` (UTF-X) mit der Länge `u_srcLen` in den Zielpuffer `pu_dst` (UTF-Y) mit der Länge `u_dstLen`.

```
et_UtfError t1_UtfLength (t_UInt & u_len, const t_UIntX * & pu_src, t_UInt u_srcLen, bool b_null = true);
```

Liefert im Parameter `u_len` die Anzahl der Unicode-Zeichen der Zeichenkette `pu_src` (UTF-X) mit der Länge `u_srcLen`.

```
et_UtfError t1_UtfToUpper (t_UIntX * & pu_src, t_UInt u_srcLen);
```

Wandelt die Zeichenkette `pu_src` (UTF-X) mit der Länge `u_srcLen` in Großbuchstaben um.

```
et_UtfError t1_UtfToLower (t_UIntX * & pu_src, t_UInt u_srcLen);
```

Wandelt die Zeichenkette `pu_src` (UTF-X) mit der Länge `u_srcLen` in Kleinbuchstaben um.

3.1.4 Unicode-Const-Iterator (tuning/utfcit.h)

Mit dem UTF-Const-Iterator kann man konstante UTF-Strings zeichenweise iterieren. Das Template kann für UTF-8, UTF-16 und UTF-32 verwendet werden und liefert jedes einzelne Zeichen als UTF-32. Für UTF-8 wird der Datentyp `t_UInt8` verwendet, für UTF-16 `t_UInt16` und für UTF-32 `t_UInt32`. Alle Längenangaben beziehen sich auf den jeweiligen Datentyp und nicht auf die Größe in Bytes. Nullzeichen werden vom UTF-Const-Iterator wie normale Steuerzeichen behandelt. Das Verändern der Zeichenkette ist während des Iterierens nicht möglich.

Templatedeklaration

```
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;
};
```


Methoden

`gct_UtfCit ()`;

Initialisiert ein leeres Objekt.

`gct_UtfCit (const t_UIntX * pu_src, t_UInt u_srcLen)`;

Initialisiert das Objekt und liest aus der Zeichenkette `pu_src` (UTF-X) mit der Länge `u_srcLen` das erste UTF-Zeichen.

`void First (const t_UIntX * pu_src, t_UInt u_srcLen)`;

Liest aus der Zeichenkette `pu_src` (UTF-X) mit der Länge `u_srcLen` das erste UTF-Zeichen.

`bool Ready () const`;

Liefert `true`, wenn aus der Zeichenkette erfolgreich ein UTF-Zeichen gelesen wurde.

`void Next ()`;

Liest aus der Zeichenkette das nächste UTF-Zeichen.

`t_UInt32 GetChar () const`;

Liefert das aktuelle UTF-Zeichen im Format UTF-32.

`t_UInt GetCharPos () const`;

Liefert die fortlaufende Nummer vom aktuellen UTF-Zeichen.

`t_UInt GetRawPos () const`;

Liefert die Position vom aktuellen UTF-Zeichen im Format `t_UIntX`.

`t_UInt GetRawLen () const`;

Liefert die Länge vom aktuellen UTF-Zeichen im Format `t_UIntX`.

`et_UtfError GetError () const`;

Liefert den Fehlercode vom aktuellen Lesevorgang. Dabei gibt es die folgenden Möglichkeiten:

`ec_UtfOK`: Das UTF-Zeichen wurde erfolgreich gelesen.

`ec_UtfEOS`: Das Ende der Zeichenkette wurde erreicht.

Anderer Fehlercode: In der Zeichenkette befindet sich ein fehlerhaftes UTF-Zeichen. Das Iterieren kann nicht fortgesetzt werden.

Beispiel

Zum Iterieren einer Zeichenkette wird eine `for`-Schleife nach folgendem Muster empfohlen:

```
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 Präzisionszeit (tuning/sys/ctimedate.hpp)

Die Systemuhrzeit ist im Millisekundenbereich meistens ungenau. Deshalb unterstützen einige Betriebssysteme zusätzlich eine Präzisionszeit. Diese leistet bei Zeitmessungen z. B. für das Performancetuning gute Dienste.

Datentypen

```
typedef t_Int64 t_MicroTime;
```

t_MicroTime ist ein Datentyp zur Zeitmessung in Mikrosekunden.

Funktionen

```
t_MicroTime t1_QueryPrecisionTime ();
```

Liefert die Anzahl der seit dem ersten Aufruf dieser Funktion verstrichenen Mikrosekunden.

3.1.6 Uhrzeit und Datum (tuning/sys/ctimedate.hpp)

In dieser Rubrik befinden sich Funktionen zur Abfrage und Umrechnung von Zeitwerten. Die Zeit wird in Mikrosekunden seit dem 01.01.1970 0 Uhr angegeben. Es kann sowohl die koordinierte Weltzeit (UTC) als auch die lokale Zeit verwendet werden, die der im Betriebssystem eingestellten Zeitzone entspricht.

Datentypen, Konstanten

```
typedef t_Int64 t_MicroTime;
```

t_MicroTime ist ein Datentyp zur Zeitmessung in Mikrosekunden.

```
const t_MicroTime co_MicroSecondFactor = 111;  
const t_MicroTime co_MilliSecondFactor = 100011;  
const t_MicroTime co_SecondFactor = 100000011;  
const t_MicroTime co_MinuteFactor = 6000000011;  
const t_MicroTime co_HourFactor = 360000000011;  
const t_MicroTime co_DayFactor = 8640000000011;
```

Diese Konstanten dienen der Umrechnung von Mikrosekunden in Millisekunden, Sekunden, Minuten, Stunden und Tage.

Funktionen

```
t_MicroTime t1_QueryUTCTime ();
```

Liefert die aktuelle UTC Systemzeit.

```
t_MicroTime t1_QueryLocalTime ();
```

Liefert die aktuelle lokale Systemzeit.

```
t_MicroTime t1_UTCToLocalTime (t_MicroTime i_time);
```

Rechnet eine UTC Zeit in lokale Zeit um.

```
t_MicroTime t1_LocalToUTCTime (t_MicroTime i_time);
```

Rechnet eine lokale Zeit in UTC Zeit um.

Zugehörige Klasse

Die globalen Funktionen dieser Schnittstelle dienen als Grundlage der Klasse ct_TimeDate.

3.1.7 Prozessorzeit (tuning/sys/ctimedata.hpp)

In dieser Rubrik befinden sich zwei Funktionen zur Abfrage der Zeit, die ein Thread oder Prozeß auf einem Prozessor aktiv gewesen ist. Die Zeit wird in Mikrosekunden angegeben.

Strukturdeklaration

```
struct st_UserKernelTime
{
    t_MicroTime    o_UserTime;
    t_MicroTime    o_KernelTime;
};
```

Die Struktur `st_UserKernelTime` enthält je eine Zeitangabe über verstrichene Mikrosekunden im Usermode und im Kernelmode.

Funktionen

```
bool tl_QueryProcessTimes (st_UserKernelTime * pso_times);
```

Ermittelt die Zeit, die der aktuelle Prozeß (inklusive aller Threads) auf einem Prozessor aktiv gewesen ist, und liefert bei Erfolg `true`.

```
bool tl_QueryThreadTimes (st_UserKernelTime * pso_times);
```

Ermittelt die Zeit, die der aktuelle Thread aktiv gewesen ist, und liefert bei Erfolg `true`.

3.1.8 Taskumgebung (tuning/sys/cprocess.hpp)

In dieser Rubrik befinden sich Abfrage- und Steuerungsfunktionen für Threads und Prozesse.

Funktionen

```
t_Int32 tl_InterlockedRead (volatile t_Int32 * pi_value);
```

Liest einen `t_Int32` Wert aus dem Speicher und liefert diesen Wert. Es wird eine atomare Hardwareoperation ausgeführt. Dadurch können mehrere Threads und Prozesse ohne Synchronisation auf denselben Speicher zugreifen.

```
t_Int32 tl_InterlockedWrite (volatile t_Int32 * pi_value, t_Int32 i_new);
```

Schreibt einen `t_Int32` Wert in den Speicher und liefert den alten Wert. Es wird eine atomare Hardwareoperation ausgeführt.

```
t_Int32 tl_InterlockedAdd (volatile t_Int32 * pi_value, t_Int32 i_add);
```

Addiert zu einem `t_Int32` Wert im Speicher einen anderen und liefert den neuen Wert. Es wird eine atomare Hardwareoperation ausgeführt.

```
t_Int32 tl_InterlockedIncrement (volatile t_Int32 * pi_value);
```

```
t_Int32 tl_InterlockedDecrement (volatile t_Int32 * pi_value);
```

Vergrößert bzw. verkleinert einen `t_Int32` Wert im Speicher um Eins und liefert den neuen Wert. Es wird eine atomare Hardwareoperation ausgeführt.

`void t1_Delay (int i_milliSec);`

Unterbricht die Ausführung des aktuellen Threads für `i_milliSec` Millisekunden. Andere Threads können jedoch weiterarbeiten.

`void t1_RelinquishTimeSlice ();`

Beendet die Zeitscheibe des aktuellen Threads. Das führt zur unmittelbaren Aktivierung eines anderen Threads.

`ct_String t1_GetEnv (const char * pc_name);`

Liefert den Wert der Umgebungsvariablen `pc_name` als ein Stringobjekt.

`ct_String t1_GetTempPath ();`

Liefert den Pfad für temporäre Dateien als ein Stringobjekt.

3.1.9 Threads (tuning/sys/cthread.hpp)

In dieser Rubrik befinden sich Funktionen für Threads.

Datentypen

`typedef void (* ft_ThreadFunc) (void *);`

`ft_ThreadFunc` ist ein Zeiger auf eine Hauptfunktion eines Threads. Die Funktion erwartet einen Parameter vom Typ `void *` und besitzt keinen Rückgabewert.

Funktionen

`bool t1_BeginThread (ft_ThreadFunc fo_func, void * pv_param, t_UInt u_stackSize = 8u * 1024u);`

Beginnt einen Thread mit der Hauptfunktion `fo_func`. Der Parameter `pv_param` wird an die Hauptfunktion weitergeleitet. Optional kann die Stackgröße des neuen Threads angegeben werden. Der Thread wird durch einen Aufruf von `t1_EndThread` oder das Ende der Hauptfunktion abgeschlossen. Der Rückgabewert ist `false`, wenn er nicht gestartet werden konnte.

`void t1_EndThread ();`

Beendet die Ausführung des aktuellen Threads. Die MS Windows Implementierung ruft keine Destruktoren von lokalen Objekten auf, die sich auf dem Stack des Threads befinden.

`t_UInt64 t1_ThreadId ();`

Liefert die systemabhängige Id des aktuellen Threads.

3.1.10 Prozesse (tuning/sys/cprocess.hpp)

In dieser Rubrik befinden sich Funktionen für Prozesse.

Funktionen

`void t1_EndProcess (unsigned u_exitCode);`

Beendet den aktuellen Prozeß, ohne Destruktoren globaler Objekte aufzurufen. Der Parameter `u_exitCode` wird an das Betriebssystem übergeben.

`int t1_ProcessId ();`

Liefert die systemabhängige Id des aktuellen Prozesses.

```
bool tl_IsProcessRunning (int i_processId);
```

Liefert true, wenn der Prozeß mit der Id i_processId gestartet und noch nicht beendet wurde.

```
int tl_Exec (const char * pc_path, unsigned u_params, const char * * ppc_params, bool b_wait = false);
```

Startet einen neuen Prozeß mit der ausführbaren Datei pc_path. Optional können u_params String-Parameter an den neuen Prozeß übergeben werden, wobei ppc_params auf ein Array mit u_params Zeigern verweist. Der Zeiger auf einen String-Parameter muß gleich Null sein oder auf eine nullterminierte Zeichenkette verweisen. Ist der Zeiger auf einen String-Parameter gleich Null, wird er durch eine leere Zeichenkette ersetzt. Ein String-Parameter kann Leerzeichen enthalten und optional mit dem Zeichen '"' beginnen und enden. Der Rückgabewert ist gleich -1, wenn der Prozeß nicht gestartet werden konnte. Ist der Parameter b_wait gleich false, kehrt die Funktion unmittelbar zurück und liefert die systemabhängige Id des neuen Prozesses. Andernfalls wartet die Funktion auf das Beenden des Prozesses und liefert dessen Exitcode.

3.1.11 Thread-Mutex (tuning/sys/cthmutex.hpp)

In dieser Rubrik befinden sich eine Klasse und globale Funktionen zum Synchronisieren von Threads.

Klassendeklaration

```
class ct_ThMutex
{
public:
    bool          GetInitSuccess () const;
    et_ResError   TryLock (bool & b_success);
    et_ResError   Lock ();
    et_ResError   Unlock ();
};
```

Die Klasse ct_ThMutex implementiert ein Verfahren zum wechselseitigen Ausschluß (mutual exclusion). Die Implementierung ist rekursiv, d. h. ein Thread kann ein bereits gesperrtes Mutexobjekt erneut sperren. Mutexobjekte können nicht mit Konstruktor oder Gleichoperator kopiert werden und dürfen auch nicht mit memcpy kopiert werden.

Methoden

```
bool GetInitSuccess ();
```

Liefert true, wenn das Mutexobjekt fehlerfrei initialisiert wurde.

```
et_ResError TryLock (bool & b_success);
```

Versucht, das Mutexobjekt zu sperren, und setzt bei Erfolg b_success auf true.

```
et_ResError Lock ();
```

Hält die Ausführung des Threads an, bis das Mutexobjekt gesperrt wurde, d. h. daß gleichzeitig kein anderer Thread dieses Mutexobjekt sperrt.

```
et_ResError Unlock ();
```

Gibt das Mutexobjekt wieder frei, d. h. anschließend können andere Threads dieses Mutexobjekt sperren.

Funktionen

Die folgenden globalen Funktionen implementieren ein Verfahren für kritische Abschnitte. Sie verwenden dafür ein globales Mutexobjekt.

```
bool tl_CriticalSectionInitSuccess ();
```

Liefert true, wenn das globale Mutexobjekt fehlerfrei initialisiert wurde.

```
void tl_DeleteCriticalSection ();
```

Zerstört das globale Mutexobjekt. Diese Funktion kann optional am Programmende aufgerufen werden, wenn sichergestellt ist, daß das Objekt nicht mehr verwendet wird.

```
et_ResError tl_TryEnterCriticalSection (bool & b_success);
```

Versucht, das globale Mutexobjekt zu sperren, und setzt bei Erfolg b_success auf true.

```
et_ResError tl_EnterCriticalSection ();
```

Hält die Ausführung des Threads an, bis das globale Mutexobjekt gesperrt wurde, d. h. daß gleichzeitig kein anderer Thread dieses Mutexobjekt sperrt.

```
et_ResError tl_LeaveCriticalSection ();
```

Gibt das globale Mutexobjekt wieder frei, d. h. anschließend können andere Threads dieses Mutexobjekt sperren.

3.1.12 Thread-Semaphor (tuning/sys/cthsemaphore.hpp)

In dieser Rubrik befindet sich eine weitere Klasse für die Thread-Synchronisation.

Klassendeklaration

```
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 ();
};
```

Die Klasse ct_ThSemaphore wird ähnlich wie ct_ThMutex zum Synchronisieren von Threads verwendet. Es existieren jedoch zwei wesentliche Unterschiede:

1. Wurde ein Mutex von einem Thread gesperrt, kann es nur vom selben Thread wieder freigegeben werden. Semaphoren können jedoch von mehreren Threads in beliebiger Reihenfolge angefordert und freigegeben werden.
2. Ein Semaphor kann mehrmals hintereinander freigegeben werden. Dadurch erhöht sich ein interner Zähler. Ein Thread wird beim Anfordern des Semaphors nur dann blockiert, wenn dieser interne Zähler Null erreicht hat.

Steht der interne Zähler anfangs auf Eins und werden Anfordern und Freigeben immer paarweise im selben Thread aufgerufen, so ist die Wirkung wie bei einem Mutex. Ein Semaphor kann jedoch auch anders verwendet werden. Z. B. kann damit eine Message-Queue mit mehreren Sender-Threads und einem Empfänger-Thread implementiert werden. Semaphorobjekte können nicht mit Konstruktor oder Gleichoperator kopiert werden und dürfen auch nicht mit memcpy kopiert werden.

Methoden

`ct_ThSemaphore (t_Int32 i_initValue = 1);`

Konstruiert ein Semaphorobjekt und setzt den internen Zähler auf `i_initValue`.

`bool GetInitSuccess ();`

Liefert `true`, wenn das Semaphorobjekt fehlerfrei initialisiert wurde.

`et_ResError TryAcquire (bool & b_success, t_UInt32 u_milliSec = 0);`

Versucht, das Semaphorobjekt anzufordern, und setzt bei Erfolg `b_success` auf `true`. Die Methode wartet maximal `u_milliSec` Millisekunden.

`et_ResError Acquire ();`

Das Semaphorobjekt wird angefordert. Die Methode wartet, falls der interne Zähler gleich Null ist. Andernfalls wird vom internen Zähler Eins subtrahiert.

`et_ResError Release ();`

Das Semaphorobjekt wird freigegeben. Dabei wird zum internen Zähler Eins addiert. War der Zähler vorher gleich Null, dann wird ein eventuell wartender Thread aufgeweckt.

3.1.13 Gemeinsame Ressource (tuning/sys/csharedres.hpp)

Die Klasse `ct_SharedResource` implementiert die Basisfunktionalität für Objekte, die von mehreren Prozessen gemeinsam verwendet werden können. Die gemeinsame Ressource wird über einen String-Schlüssel identifiziert.

Vor der Verwendung muß ein Schlüssel gesetzt und das Objekt initialisiert werden. Die Initialisierung erfolgt durch Öffnen oder Erzeugen (Methoden `Open` oder `Create` in den abgeleiteten Klassen). Nach der Initialisierung kann der Schlüssel nicht mehr geändert werden.

Klassendeklaration

```
class ct_SharedResource
{
public:
    ct_SharedResource ();
    ct_SharedResource (const char * pc_key);
    ct_SharedResource (const char * pc_key, unsigned u_idx);
    ~ct_SharedResource ();

    virtual
        GetInitSuccess () const;
        GetKey () const;
        SetKey (const char * pc_key);
        SetKey (const char * pc_key, unsigned u_idx);
};
```

Methoden

`ct_SharedResource ();`

Konstruiert eine gemeinsame Ressource ohne Schlüssel.

`ct_SharedResource (const char * pc_key);`

Konstruiert eine gemeinsame Ressource mit dem Schlüssel `pc_key`.

```
ct_SharedResource (const char * pc_key, unsigned u_idx);
```

Konstruiert eine gemeinsame Ressource mit dem Schlüssel `pc_key`. Der Index `u_idx` wird in eine Zeichenkette umgewandelt und an `pc_key` angehängt.

```
virtual ~ct_SharedResource ();
```

Der virtuelle Destruktor ruft den Destruktor des abgeleiteten Objekts auf.

```
bool GetInitSuccess ();
```

Liefert `true`, wenn die gemeinsame Ressource fehlerfrei initialisiert wurde.

```
const char * GetKey () const;
```

Liefert den Schlüssel.

```
et_ResError SetKey (const char * pc_key);
```

Setzt den Schlüssel `pc_key`. Liefert `ec_ResOK`, wenn der Schlüssel gültig und das Objekt noch nicht initialisiert war.

```
et_ResError SetKey (const char * pc_key, unsigned u_idx);
```

Setzt den Schlüssel `pc_key`. Der Index `u_idx` wird in eine Zeichenkette umgewandelt und an `pc_key` angehängt. Liefert `ec_ResOK`, wenn der Schlüssel gültig und das Objekt noch nicht initialisiert war.

3.1.14 Prozeß-Mutex (tuning/sys/cprmutex.hpp)

In dieser Rubrik befinden sich eine Klasse und globale Funktionen zum Synchronisieren von Prozessen.

Basisklasse

`ct_SharedResource` (siehe Abschnitt 'Gemeinsame Ressource')

Klassendeklaration

```
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 ();
};
```

Die Klasse `ct_PrMutex` implementiert ein Verfahren zum wechselseitigen Ausschluß (mutual exclusion). Das Mutexobjekt ist vollständig initialisiert, wenn ein Schlüssel gesetzt wurde und die Methoden `Open` oder `Create` den Wert `ec_ResOK` geliefert haben. Die MS Windows Implementierung ist rekursiv, d. h. ein Prozeß kann ein bereits gesperrtes Mutexobjekt erneut sperren. Die Linux Implementierung ist nicht rekursiv, d. h. wenn ein Prozeß ein bereits gesperrtes Mutexobjekt erneut sperrt, blockiert er sich selbst. Die Methoden `TryLock`, `Lock` und `Unlock` sind gegen den konkurrierenden Zugriff mehrerer Threads geschützt, d. h. nach dem Initialisieren kann ein `PrMutex`objekt auch zum Synchronisieren von Threads verwendet werden.

Methoden

`ct_PrMutex ()`;

Konstruiert ein Mutexobjekt mit einem globalen Schlüssel.

`ct_PrMutex (const char * pc_key)`;

Konstruiert ein Mutexobjekt mit dem Schlüssel `pc_key`.

`ct_PrMutex (const char * pc_key, unsigned u_idx)`;

Konstruiert ein Mutexobjekt mit dem Schlüssel `pc_key`. Der Index `u_idx` wird in eine Zeichenkette umgewandelt und an `pc_key` angehängt.

`~ct_PrMutex ()`;

Der Destruktor schließt das Mutexobjekt, falls es geöffnet war.

`et_ResError Open ()`;

Versucht, sich mit einem bestehenden Mutexobjekt, das denselben Schlüssel verwendet, zu verbinden, und liefert bei Erfolg `ec_ResOK`. Der Vorgang ist vergleichbar mit dem Öffnen einer Datei.

`et_ResError Create (bool b_createNew = false)`;

Versucht, ein neues Mutexobjekt zu erzeugen, und liefert bei Erfolg `ec_ResOK`. Liefert `ec_ResAlreadyExists`, wenn `b_createNew` gleich `true` ist und ein Mutexobjekt, das denselben Schlüssel verwendet, bereits existiert. Der Vorgang ist vergleichbar mit dem Erzeugen einer Datei.

`et_ResError Close ()`;

Versucht, ein geöffnetes Mutexobjekt zu schließen, und liefert bei Erfolg `ec_ResOK`. Der Vorgang ist vergleichbar mit dem Schließen einer Datei.

`et_ResError TryLock (bool & b_success, t_UInt32 u_milliSec = 0)`;

Versucht, das Mutexobjekt zu sperren, und setzt bei Erfolg `b_success` auf `true`. Die Methode wartet maximal `u_milliSec` Millisekunden.

`et_ResError Lock ()`;

Hält die Ausführung des Prozesses an, bis das Mutexobjekt gesperrt wurde, d. h. daß gleichzeitig kein anderer Prozeß ein Mutexobjekt mit demselben Schlüssel sperrt.

`et_ResError Unlock ()`;

Gibt das Mutexobjekt wieder frei, d. h. anschließend können andere Prozesse Mutexobjekte mit demselben Schlüssel sperren.

Funktionen

Die folgenden globalen Funktionen implementieren ein Verfahren für kritische Abschnitte. Sie verwenden dafür ein globales `PrMutexobjekt`.

`bool t1_CriticalPrSectionInitSuccess ()`;

Liefert `true`, wenn das globale Mutexobjekt fehlerfrei initialisiert wurde.

`void t1_DeleteCriticalPrSection ()`;

Zerstört das globale Mutexobjekt. Diese Funktion kann optional am Programmende aufgerufen werden, wenn sichergestellt ist, daß das Objekt nicht mehr verwendet wird.

```
et_ResError tl_TryEnterCriticalSection (bool & b_success, t_UInt32 u_milliSec = 0);
```

Versucht, das globale Mutexobjekt zu sperren, und setzt bei Erfolg `b_success` auf `true`. Die Methode wartet maximal `u_milliSec` Millisekunden.

```
et_ResError tl_EnterCriticalSection ();
```

Hält die Ausführung des Prozesses an, bis das globale Mutexobjekt gesperrt wurde, d. h. daß gleichzeitig kein anderer Prozeß ein Mutexobjekt mit dem globalen Schlüssel sperrt.

```
et_ResError tl_LeaveCriticalSection ();
```

Gibt das globale Mutexobjekt wieder frei, d. h. anschließend können andere Prozesse Mutexobjekte mit dem globalen Schlüssel sperren.

3.1.15 Prozeß-Semaphor (tuning/sys/cprsemaphore.hpp)

In dieser Rubrik befindet sich eine weitere Klasse für die Prozeß-Synchronisation.

Klassendeklaration

```
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 ();
};
```

Die Klasse `ct_PrSemaphore` implementiert ein Semaphor zum Synchronisieren von Prozessen (siehe Klasse `ct_ThSemaphore` für Threads). Das Semaphorobjekt ist vollständig initialisiert, wenn ein Schlüssel gesetzt wurde und die Methoden `Open` oder `Create` den Wert `ec_ResOK` geliefert haben. Die Methoden `TryAcquire`, `Acquire` und `Release` sind gegen den konkurrierenden Zugriff mehrerer Threads geschützt, d. h. nach dem Initialisieren kann ein `PrSemaphore`objekt auch zum Synchronisieren von Threads verwendet werden.

Methoden

```
ct_PrSemaphore ();
```

Konstruiert ein Semaphorobjekt mit einem globalen Schlüssel.

```
ct_PrSemaphore (const char * pc_key);
```

Konstruiert ein Semaphorobjekt mit dem Schlüssel `pc_key`.

```
ct_PrSemaphore (const char * pc_key, unsigned u_idx);
```

Konstruiert ein Semaphorobjekt mit dem Schlüssel `pc_key`. Der Index `u_idx` wird in eine Zeichenkette umgewandelt und an `pc_key` angehängt.

```
~ct_PrSemaphore ();
```

Der Destruktor schließt das Semaphorobjekt, falls es geöffnet war.

et_ResError Open ();

Versucht, sich mit einem bestehenden Semaphorobjekt, das denselben Schlüssel verwendet, zu verbinden, und liefert bei Erfolg ec_ResOK. Der Vorgang ist vergleichbar mit dem Öffnen einer Datei.

et_ResError Create (t_Int32 i_initValue = 1, bool b_createNew = false);

Versucht, ein neues Semaphorobjekt zu erzeugen. Liefert bei Erfolg ec_ResOK und setzt den internen Zähler auf i_initValue. Liefert ec_ResAlreadyExists, wenn b_createNew gleich true ist und ein Semaphorobjekt, das denselben Schlüssel verwendet, bereits existiert. Der Vorgang ist vergleichbar mit dem Erzeugen einer Datei.

et_ResError Close ();

Versucht, ein geöffnetes Semaphorobjekt zu schließen, und liefert bei Erfolg ec_ResOK. Der Vorgang ist vergleichbar mit dem Schließen einer Datei.

et_ResError TryAcquire (bool & b_success, t_UInt32 u_milliSec = 0);

Versucht, das Semaphorobjekt anzufordern, und setzt bei Erfolg b_success auf true. Die Methode wartet maximal u_milliSec Millisekunden.

et_ResError Acquire ();

Das Semaphorobjekt wird angefordert. Die Methode wartet, falls der interne Zähler gleich Null ist. Andernfalls wird vom internen Zähler Eins subtrahiert.

et_ResError Release ();

Das Semaphorobjekt wird freigegeben. Dabei wird zum internen Zähler Eins addiert. War der Zähler vorher gleich Null, dann wird ein eventuell wartender Thread/Prozeß aufgeweckt.

3.1.16 Gemeinsamer Speicher (tuning/sys/csharedmem.hpp)

Die Klasse ct_SharedMemory implementiert die gemeinsame Nutzung von Hauptspeicher durch mehrere Prozesse. Das Sharedmemoryobjekt ist vollständig initialisiert, wenn ein Schlüssel gesetzt wurde und die Methoden Open oder Create den Wert ec_ResOK geliefert haben.

Basisklasse

ct_SharedResource (siehe Abschnitt 'Gemeinsame Ressource')

Klassendeklaration

```
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;
};
```

Methoden

`ct_SharedMemory ()`;

Konstruiert ein Sharedmemoryobjekt mit einem globalen Schlüssel.

`ct_SharedMemory (const char * pc_key)`;

Konstruiert ein Sharedmemoryobjekt mit dem Schlüssel `pc_key`.

`ct_SharedMemory (const char * pc_key, unsigned u_idx)`;

Konstruiert ein Sharedmemoryobjekt mit dem Schlüssel `pc_key`. Der Index `u_idx` wird in eine Zeichenkette umgewandelt und an `pc_key` angehängt.

`~ct_SharedMemory ()`;

Der Destruktor schließt das Sharedmemoryobjekt, falls es geöffnet war.

`et_ResError Open (bool b_readOnly)`;

Versucht, sich mit einem bestehenden Sharedmemoryobjekt, das denselben Schlüssel verwendet, zu verbinden, und liefert bei Erfolg `ec_ResOK`. Wenn `b_readOnly` gleich `true` ist, kann auf den gemeinsamen Speicher nur lesend zugegriffen werden. Der Vorgang ist vergleichbar mit dem Öffnen einer Datei.

`et_ResError Create (t_UInt u_size, bool b_createNew = false)`;

Versucht, ein neues Sharedmemoryobjekt mit der Größe `u_size` Bytes zu erzeugen, und liefert bei Erfolg `ec_ResOK`. Liefert `ec_ResAlreadyExists`, wenn `b_createNew` gleich `true` ist und ein Sharedmemoryobjekt, das denselben Schlüssel verwendet, bereits existiert. Der Vorgang ist vergleichbar mit dem Erzeugen einer Datei.

`et_ResError Close ()`;

Versucht, ein geöffnetes Sharedmemoryobjekt zu schließen, und liefert bei Erfolg `ec_ResOK`. Der Vorgang ist vergleichbar mit dem Schließen einer Datei.

`t_UInt GetSize () const`;

Liefert die Größe in Bytes des gemeinsamen Speichers.

`void * GetData () const`;

Liefert einen Zeiger auf das erste Byte des gemeinsamen Speichers.

3.1.17 Datei (tuning/sys/cfile.hpp)

Innerhalb der Bibliothek **Spirick Tuning** werden Pfad- und Dateinamen als UTF-8-Strings interpretiert. Unter Linux werden die Strings unverändert an die Systemfunktionen übergeben. Unter MS Windows werden Pfad- und Dateinamen intern in UTF-16 umgewandelt.

Die Systemschnittstelle für Dateien ist auf das blockweise Verarbeiten großer Datenmengen ausgerichtet. Die Funktionen bauen direkt auf der ungepufferten Dateiein- und -ausgabe des Betriebssystems auf. Für eine optimale Geschwindigkeit sollte als Blockgröße ein Vielfaches von vier KB verwendet werden. Die Funktionen `tl_OpenFile` und `tl_CreateFile` sind gegen das gleichzeitige Aufrufen durch mehrere Prozesse geschützt (race conditions).

Sämtliche Funktionen liefern bei erfolgreicher Ausführung den Wahrheitswert `true`. Im Fehlerfall wird keine C++-Exception ausgelöst, sondern `false` zurückgegeben. Die Funktionen können somit auch in einer Programmierumgebung genutzt werden, in der keine Exceptions zur Verfügung stehen oder diese mit einer Compileroption deaktiviert sind.

Datentypen, Konstanten

```
typedef ... t_FileId;  
const t_FileId co_InvalidFileId = ...;  
typedef t_Int64 t_FileSize;
```

Eine FileId enthält einen systemabhängigen Code für eine geöffnete Datei. Die Konstante co_InvalidFileId ist eine garantiert ungültige FileId. Der Datentyp t_FileSize wird für Größen- und Positionsangaben verwendet.

Funktionen

```
bool t1_OpenFile (const char * pc_name, t_FileId & o_file, bool b_readOnly = true, bool b_sequential = true);
```

Öffnet die bestehende Datei pc_name abhängig vom Parameter b_readOnly zum Lesen oder Schreiben. Der optionale Parameter b_sequential beeinflusst die Arbeitsweise des Cachemanagers. Wird die Datei sequentiell bearbeitet, sollte er auf true gesetzt werden. Der Parameter o_file muß vor dem Aufruf auf co_InvalidFileId gesetzt werden. Bei Erfolg liefert die Funktion true, und o_file enthält die FileId der geöffneten Datei.

```
bool t1_CreateFile (const char * pc_name, t_FileId & o_file, bool b_createNew = false);
```

Erzeugt die neue Datei pc_name und öffnet sie zum Schreiben. Eine eventuell vorhandene Datei gleichen Namens wird überschrieben. Liefert false, wenn b_createNew gleich true ist und eine Datei mit demselben Namen bereits existiert. Der Parameter o_file muß vor dem Aufruf auf co_InvalidFileId gesetzt werden. Bei Erfolg liefert die Funktion true, und o_file enthält die FileId der geöffneten Datei.

```
bool t1_CloseFile (t_FileId o_file);
```

Versucht, die geöffnete Datei o_file zu schließen, und liefert bei Erfolg true.

```
bool t1_ExistsFile (const char * pc_name);
```

Liefert true, wenn die Datei pc_name existiert.

```
bool t1_MoveFile (const char * pc_old, const char * pc_new);
```

Verschiebt die Datei pc_old nach pc_new. Befinden sich alter und neuer Name innerhalb desselben Verzeichnisses, wird nur der Name des Eintrags geändert.

```
bool t1_CopyFile (const char * pc_old, const char * pc_new, bool b_overwrite = true);
```

Kopiert die Datei pc_old nach pc_new. Ist der optionale Parameter b_overwrite gleich true, wird eine eventuell vorhandene Datei gleichen Namens überschrieben.

```
bool t1_DeleteFile (const char * pc_name);
```

Löscht die Datei pc_name.

```
bool t1_QuerySize (t_FileId o_file, t_FileSize & o_size);
```

Ermittelt die aktuelle Größe der geöffneten Datei o_file.

```
bool t1_QueryPos (t_FileId o_file, t_FileSize & o_pos);
```

Ermittelt die aktuelle Position des Zugriffszeigers der geöffneten Datei o_file.

```
bool t1_SeekAbs (t_FileId o_file, t_FileSize o_pos);
```

Positioniert den Zugriffszeiger der geöffneten Datei o_file absolut auf die Position o_pos.

```
bool t1_SeekRel (t_FileId o_file, t_FileSize o_pos);
```

Positioniert den Zugriffszeiger der geöffneten Datei o_file relativ auf die Position o_pos.

```
bool t1_Truncate (t_FileId o_file, t_FileSize o_size);
```

Verändert die Größe der geöffneten Datei o_file auf o_size Bytes.

```
bool t1_Read (t_FileId o_file, void * pv_dst, t_FileSize o_len);
```

Liest `o_len` Bytes aus der geöffneten Datei `o_file` nach `pv_dst` und verschiebt den Zugriffszeiger.

```
bool t1_Write (t_FileId o_file, const void * pv_src, t_FileSize o_len);
```

Schreibt `o_len` Bytes von `pv_src` in die geöffnete Datei `o_file` und verschiebt den Zugriffszeiger.

Zugehörige Klasse

Die globalen Funktionen dieser Schnittstelle dienen als Grundlage der Klasse `ct_File`.

3.1.18 Verzeichnis (tuning/sys/cdir.hpp)

Innerhalb der Bibliothek **Spirick Tuning** werden Pfad- und Dateinamen als UTF-8-Strings interpretiert. Unter Linux werden die Strings unverändert an die Systemfunktionen übergeben. Unter MS Windows werden Pfad- und Dateinamen intern in UTF-16 umgewandelt.

Die Systemschnittstelle für Verzeichnisse enthält einige elementare Funktionen, die häufig benötigt werden, die jedoch in der C-Standardbibliothek noch nicht compiler- und systemunabhängig definiert sind.

Sämtliche Funktionen liefern bei erfolgreicher Ausführung den Wahrheitswert `true`. Im Fehlerfall wird keine C++-Exception ausgelöst, sondern `false` zurückgegeben. Die Funktionen können somit auch in einer Programmierumgebung genutzt werden, in der keine Exceptions zur Verfügung stehen oder diese mit einer Compileroption deaktiviert sind.

Funktionen

```
bool t1_QueryCurrentDirectory (const char * pc_drive, t_UInt u_driveLen, ct_String & co_currentDirectory);
```

Ermittelt das aktuelle Verzeichnis des Laufwerks `pc_drive` und schreibt das Resultat nach `co_currentDirectory`. Die Laufwerksangabe muß nicht nullterminiert sein. Statt des Nullzeichens wird die Länge `u_driveLen` angegeben. Ist `u_driveLen` gleich Null, wird das aktuelle Laufwerk verwendet. Die Linux Implementierung ignoriert die Parameter `pc_drive` und `u_driveLen`.

```
bool t1_CreateDirectory (const char * pc_name);
```

Erzeugt das neue Verzeichnis `pc_name`.

```
bool t1_MoveDirectory (const char * pc_old, const char * pc_new);
```

Verschiebt das Verzeichnis `pc_old` nach `pc_new`. Befinden sich alter und neuer Name innerhalb desselben übergeordneten Verzeichnisses, wird nur der Name des Eintrags geändert.

```
bool t1_DeleteDirectory (const char * pc_name);
```

Löscht das leere Verzeichnis `pc_name`.

Zugehörige Klasse

Die globalen Funktionen dieser Schnittstelle dienen als Grundlage der Klasse `ct_Directory`.

3.1.19 Systemnahe Informationen (tuning/sys/cinfo.hpp)

In dieser Rubrik befinden sich mehrere Strukturen und Funktionen zur Abfrage systemnaher Informationen. Zeichenketten werden in statisch allokiertem Speicher abgelegt.

Strukturdeklaration

```
struct st_FileSystemInfo
{
    t_UInt64          u_TotalBytes;
    t_UInt64          u_FreeBytes;
    t_UInt64          u_AvailableBytes;
};
```

Die Struktur `st_FileSystemInfo` stellt wichtige Informationen über ein Dateisystem zur Verfügung. Enthalten sind die Gesamtgröße (`u_TotalBytes`), der insgesamt freie Speicher (`u_FreeBytes`) sowie der für den aktuellen Prozeß/Nutzer verfügbare Speicher (`u_AvailableBytes`).

Strukturdeklaration

```
struct st_HardwareInfo
{
    t_UInt64          u_TotalBytes;
    t_UInt64          u_AvailableBytes;
    unsigned          u_TotalProcessors;
    unsigned          u_AvailableProcessors;
    const char *      pc_CPUName;
};
```

Die Struktur `st_HardwareInfo` stellt wichtige Informationen über die Computerhardware zur Verfügung. Enthalten sind die Gesamtgröße (`u_TotalBytes`) und die verfügbaren Bytes (`u_AvailableBytes`) des Arbeitsspeichers, die Gesamtzahl (`u_TotalProcessors`) und die für den aktuellen Prozeß/Nutzer verfügbare Anzahl (`u_AvailableProcessors`) der Prozessorkerne sowie der Name des Prozessors als Zeichenkette (`pc_CPUName`).

In einer 32-Bit-Umgebung kann ein Prozeß je nach Architektur nur max. 2-4 GB Arbeitsspeicher verwenden. Wenn mehr als 4 GB physisch vorhanden sind, kann es sein, daß bei Gesamtgröße und verfügbaren Bytes Werte größer als 4 GB geliefert werden.

Strukturdeklaration

```
struct st_ProcessMemoryInfo
{
    t_UInt            u_VMBytes;
    t_UInt            u_RSSBytes;
};
```

Die Struktur `st_ProcessMemoryInfo` stellt Informationen über den Speicherverbrauch des aktuellen Prozesses zur Verfügung. Enthalten sind die gesamten Bytes (`u_VMBytes`, virtual memory size) und die residenten Bytes (`u_RSSBytes`, resident set size). Die Gesamtgröße umfaßt den Speicher, der sich im Arbeitsspeicher oder im Pagefile befindet. Die residenten Bytes umfassen nur die Bereiche, die sich aktuell im Arbeitsspeicher befinden. Die Art und Weise, wie die beiden Speichergrößen berechnet werden, unterscheidet sich von Betriebssystem zu Betriebssystem, z. B. ob Speicher, der von mehreren Prozessen gemeinsam genutzt wird, eingerechnet wird oder nicht.

Strukturdeklaration

```
enum et_Compiler
{
    ec_CompilerMSVC,
    ec_CompilerGCC
};

struct st_CompilerInfo
{
```

```

et_Compiler      eo_Compiler;
const char *     pc_CompilerVersion;
const char *     pc_RuntimeVersion;
};

```

Die Struktur `st_CompilerInfo` stellt wichtige Informationen über den verwendeten Compiler und das Laufzeitsystem zur Verfügung. Enthalten sind der Compilertyp (`eo_Compiler`) sowie die Versionen von Compiler (`pc_CompilerVersion`) und Laufzeitsystem (`pc_RuntimeVersion`) als Zeichenkette.

Strukturdeklaration

```

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;
};

```

Die Struktur `st_SystemInfo` stellt wichtige Informationen über das Betriebssystem zur Verfügung. Enthalten sind der Betriebssystemtyp (`eo_System`) sowie Zeichenketten für die Betriebssystemversion (`pc_SystemVersion`), den Computernamen (`pc_ComputerName`) und den Namen des aktuellen Nutzers (`pc_UserName`).

Strukturdeklaration

```

struct st_BatteryInfo
{
    bool          b_ACLine;
    bool          b_BatteryFound;
    int           i_LifePercent;
};

```

Die Struktur `st_BatteryInfo` stellt wichtige Informationen über die Stromversorgung des Computers zur Verfügung. Die Membervariable `b_ACLine` ist gleich `true`, wenn der Computer am Stromnetz angeschlossen ist. Die Membervariable `b_BatteryFound` ist gleich `true`, wenn sich im Computer eine Batterie befindet. Die Membervariable `i_LifePercent` enthält den Füllstand der Batterie in Prozent.

Funktionen

```
bool t1_QueryFileSystemInfo (const char * pc_path, st_FileSystemInfo * pso_info);
```

Speichert in `pso_info` Informationen über das Dateisystem, auf das der Parameter `pc_path` verweist, und liefert bei Erfolg den Wert `true`.

```
bool t1_QueryHardwareInfo (st_HardwareInfo * pso_info);
```

Speichert in `pso_info` Informationen über die Computerhardware und liefert bei Erfolg den Wert `true`.

```
bool t1_QueryProcessMemoryInfo (st_ProcessMemoryInfo * pso_info);
```

Speichert in `pso_info` Informationen über den Speicherverbrauch und liefert bei Erfolg den Wert `true`.

```
bool t1_QueryCompilerInfo (st_CompilerInfo * pso_info);
```

Speichert in `pso_info` Informationen über den verwendeten Compiler und das Laufzeitsystem und liefert bei Erfolg den Wert `true`.


```
bool t1_QuerySystemInfo (st_SystemInfo * pso_info);
```

Speichert in pso_info Informationen über das Betriebssystem und liefert bei Erfolg den Wert true.

```
bool t1_QueryBatteryInfo (st_BatteryInfo * pso_info);
```

Speichert in pso_info Informationen über die Stromversorgung des Computers und liefert bei Erfolg den Wert true.

3.2 Zeichenketten und Dateinamen

3.2.1 Stringtemplate (tuning/string.h)

Die Stringklassen in der Bibliothek **Spirick Tuning** enthalten nullterminierte Zeichenketten und zusätzlich eine Längenangabe. Das abschließende Nullzeichen ist eine verbreitete Konvention und sichert die Kompatibilität mit zahlreichen anderen Bibliotheken. Die zusätzliche Längenangabe dient der Beschleunigung von Rechenvorgängen. Ohne sie müsste häufig die Länge der Zeichenkette durch Suche nach dem Nullzeichen ermittelt werden. Positionsangaben innerhalb einer Zeichenkette beginnen mit dem Wert Null. Diese Zählung entspricht ebenfalls einer verbreiteten Konvention.

Das Klassentemplate `gct_String` dient als Basisklasse für alle weiteren Stringklassen. Der Parameter `t_block` ist eine Blockklasse mit Zeichenblock-Schnittstelle, z. B. `gct_CharBlock <ct_Chn32Block, char>`, und dient dem String als Basisklasse. Um Speicherplatz bei leeren Strings zu sparen, wird empfohlen, das Template `gct_NullDataBlock` zu verwenden, z. B. `gct_CharBlock <gct_NullDataBlock <ct_Chn32Block, char>, char>`. Der zweite Templateparameter `t_staticStore` ist eine statische Storeklasse, z. B. `ct_Chn32Store`. Sie wird in der Methode `ReplaceAll` als temporärer Zwischenspeicher verwendet.

Basisklassen

`gct_CharBlock` (siehe Abschnitt 'Zeichenblock')

Templatedeklaration

```
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    GetHashCode () 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);
};

```

Parameterarten

Für das Zuweisen und Einfügen von Zeichenketten existieren die folgenden Parameterarten:

1. **Einzelnes Zeichen (t_Char c):** Das Zeichen wird als Zeichenkette der Länge Eins betrachtet.
2. **Zeichen mit Längenangabe (t_Char c, t_Size o_len):** Die Parameterliste wird als Zeichenkette der Länge o_len betrachtet, die mit dem Zeichen c gefüllt ist.
3. **Nullterminierte Zeichenkette (const t_Char * pc):** Die Zeichenkette wird bis zu ihrem Nullzeichen verarbeitet.
4. **Zeichenkette mit Längenangabe (const t_Char * pc, t_Size o_len):** Es werden die ersten o_len Zeichen der Zeichenkette pc verarbeitet. Darin darf kein Nullzeichen vorkommen.
5. **Stringobjekt (const gct_String & co):** Es wird die gesamte Zeichenkette des Stringobjekts co verarbeitet. Die Länge wird vom Stringobjekt abgefragt und muß nicht berechnet werden.
6. **Formatierte Zeichenkette (const t_Char * pc_format, ...):** Die Parameterliste wird wie eine formatierte Zeichenkette im printf-Format behandelt. Diese Parameterart kann nicht in überladenen Methoden verwendet werden, da sie sich nicht eindeutig von 3. und 4. unterscheiden läßt.

Selbstzuweisung

Nicht alle Methoden einer Stringklasse enthalten eine Sonderbehandlung für Selbstzuweisung.

Eine Selbstzuweisung liegt vor, wenn als Parameter ein Zeiger auf die eigene Zeichenkette (GetStr () bzw. this) übergeben wird. Sie tritt in der Praxis selten auf, ihre Behandlung kostet jedoch Rechenzeit. Eine Selbstzuweisung kann z. B. beim Iterieren eines Containers auftreten, wenn allen Elementen der Wert eines Elements desselben Containers zugewiesen wird. Wird die Selbstzuweisung innerhalb einer Zuweisungsmethode nicht gesondert behandelt, kommt es zu unerwarteten und fehlerhaften Resultaten.

Datentypen

`typedef t_block::t_Size t_Size;`

Der geschachtelte Größentyp einer Stringklasse bestimmt den Wertebereich der Größen- und Positionsangaben. Ist z. B. der Größentyp auf `t_UInt8` definiert, kann die Zeichenkette maximal 255 Bytes umfassen (einschließlich des Nullzeichens). Der Größentyp beeinflusst auch die Größe des Stringobjekts, denn die meisten Stringklassen enthalten ein Attribut des Typs `t_Size`.

Konstruktoren

`gct_String ();`

Der normale Konstruktor initialisiert ein leeres Stringobjekt. Die Zeichenkette besteht nur aus dem abschließenden Nullzeichen.

`gct_String (t_Char c_init);`

Initialisiert ein Stringobjekt der Länge Eins. Das Zeichen `c_init` wird übernommen und darf kein Nullzeichen sein.

`gct_String (t_Char c_init, t_Size o_len);`

Initialisiert ein Stringobjekt der Länge `o_len`. Die Zeichenkette wird mit dem Zeichen `c_init` gefüllt. Es darf kein Nullzeichen sein.

`gct_String (const t_Char * pc_init);`

Initialisiert ein Stringobjekt durch Kopieren der nullterminierten Zeichenkette `pc_init`. Es wird eine echte Kopie (deep copy) angefertigt. Der Inhalt von `pc_init` wird in einen eigenen Speicherbereich kopiert.

`gct_String (const t_Char * pc_init, t_Size o_len);`

Initialisiert ein Stringobjekt durch Kopieren der ersten `o_len` Zeichen der Zeichenkette `pc_init`. Es wird eine echte Kopie (deep copy) angefertigt. Der Inhalt von `pc_init` wird in einen eigenen Speicherbereich kopiert.

`gct_String (const gct_String & co_init);`

Initialisiert ein Stringobjekt durch Kopieren des Inhalts von `co_init`. Es wird eine echte Kopie (deep copy) angefertigt. Der Inhalt von `co_init` wird in einen eigenen Speicherbereich kopiert.

Zugriff auf Länge und Zeichenkette

`t_UInt GetHashCode () const;`

Liefert einen Hashwert der Zeichenkette.

`bool IsEmpty () const;`

Liefert `true`, wenn die Zeichenkette leer ist.

`t_Size GetMaxLen () const;`

Liefert die maximale Länge der Zeichenkette (ohne abschließendes Nullzeichen).

`t_Size GetLen () const;`

Liefert die Länge der Zeichenkette (ohne abschließendes Nullzeichen).

`const t_Char * GetStr () const;`

`const t_Char * operator () () const;`

Liefert einen Zeiger auf das erste Zeichen. Bei einer leeren Zeichenkette zeigt er auf das abschließende Nullzeichen.

```
const t_Char * GetStr (t_Size o_pos) const;  
const t_Char * operator () (t_Size o_pos) const;
```

Liefert einen Zeiger auf das Zeichen an der Position `o_pos`. Bei `o_pos == GetLen ()` zeigt er auf das abschließende Nullzeichen. Es muß `o_pos <= GetLen ()` gelten.

```
t_Char & GetChar (t_Size o_pos) const;  
t_Char & operator [] (t_Size o_pos) const;
```

Liefert eine Referenz auf das Zeichen an der Position `o_pos`. Diesem Zeichen darf kein Nullzeichen zugewiesen werden. Es muß `o_pos < GetLen ()` gelten.

```
t_Char & GetRevChar (t_Size o_pos) const;
```

Liefert eine Referenz auf das Zeichen an der Position `GetLen () - 1 - o_pos`. Bei `o_pos == 0` ist es das letzte Zeichen, bei `o_pos == 1` das vorletzte usw. Diesem Zeichen darf kein Nullzeichen zugewiesen werden. Es muß `o_pos < GetLen ()` gelten.

```
gct_String SubStr (t_Size o_len) const;
```

Liefert ein String-Objekt, das die ersten `o_len` Zeichen der eigenen Zeichenkette enthält. Es muß `o_len <= GetLen ()` gelten.

```
gct_String RevSubStr (t_Size o_len) const;
```

Liefert ein String-Objekt, das die letzten `o_len` Zeichen der eigenen Zeichenkette enthält. Es muß `o_len <= GetLen ()` gelten.

```
gct_String SubStr (t_Size o_pos, t_Size o_len) const;  
gct_String operator () (t_Size o_pos, t_Size o_len) const;
```

Liefert ein String-Objekt, das beginnend bei `o_pos` die nächsten `o_len` Zeichen der eigenen Zeichenkette enthält. Es muß `o_pos + o_len <= GetLen ()` gelten.

Suche nach Zeichen und Teilzeichenketten

```
t_Int First (t_Char c_search, t_Size o_pos = 0) const;
```

Liefert die Position des ersten Auftretens des Zeichens `c_search` ab der Position `o_pos` oder einen negativen Wert, wenn das Zeichen nicht gefunden wurde.

```
t_Int First (const t_Char * pc_search, t_Size o_pos = 0) const;
```

Liefert die Position des ersten Auftretens der Zeichenkette `pc_search` ab der Position `o_pos` oder einen negativen Wert, wenn die Zeichenkette nicht gefunden wurde.

```
t_Int First (const gct_String & co_search, t_Size o_pos = 0) const;
```

Liefert die Position des ersten Auftretens der Zeichenkette `co_search` ab der Position `o_pos` oder einen negativen Wert, wenn die Zeichenkette nicht gefunden wurde.

```
t_Int Last (t_Char c_search, t_Size o_pos = 0) const;
```

Liefert die Position des letzten Auftretens des Zeichens `c_search` ab der Position `o_pos` oder einen negativen Wert, wenn das Zeichen nicht gefunden wurde.

```
t_Int Last (const t_Char * pc_search, t_Size o_pos = 0) const;
```

Liefert die Position des letzten Auftretens der Zeichenkette `pc_search` ab der Position `o_pos` oder einen negativen Wert, wenn die Zeichenkette nicht gefunden wurde.

```
t_Int Last (const gct_String & co_search, t_Size o_pos = 0) const;
```

Liefert die Position des letzten Auftretens der Zeichenkette `co_search` ab der Position `o_pos` oder einen negativen Wert, wenn die Zeichenkette nicht gefunden wurde.

Teilvergleich

Die folgenden Methoden liefern einen Wert kleiner Null, wenn die eigene Zeichenkette kleiner als der Parameter ist, gleich Null bei Gleichheit mit dem Parameter und einen Wert größer Null, wenn die eigene Zeichenkette größer als der Parameter ist.

Es wird nur eine Teilzeichenkette verglichen. Der Vergleich beginnt an der Position `o_pos`. Im Gegensatz zum vollständigen (s. u.) endet der Teilvergleich spätestens am Ende des Parameters. Wurde bis dorthin kein Unterschied festgestellt, gelten die Zeichenketten als gleich. Eventuell folgende Zeichen werden nicht berücksichtigt.

```
int CompSubStr (t_Size o_pos, t_Char c_comp) const;
```

Vergleicht die eigene Zeichenkette ab der Position `o_pos` mit dem Zeichen `c_comp`. Dieses gilt als Zeichenkette der Länge Eins.

```
int CompSubStr (t_Size o_pos, const t_Char * pc_comp) const;
```

Vergleicht die eigene Zeichenkette ab der Position `o_pos` mit der nullterminierten Zeichenkette `pc_comp`.

```
int CompSubStr (t_Size o_pos, const t_Char * pc_comp, t_Size o_len) const;
```

Vergleicht die eigene Zeichenkette ab der Position `o_pos` mit den ersten `o_len` Zeichen der Zeichenkette `pc_comp`.

```
int CompSubStr (t_Size o_pos, const gct_String & co_comp) const;
```

Vergleicht die eigene Zeichenkette ab der Position `o_pos` mit der Zeichenkette `co_comp`.

Vollständiger Vergleich

Die folgenden Methoden liefern einen Wert kleiner Null, wenn die eigene Zeichenkette kleiner als der Parameter ist, gleich Null bei Gleichheit mit dem Parameter und einen Wert größer Null, wenn die eigene Zeichenkette größer als der Parameter ist.

Die beiden Zeichenketten werden vollständig miteinander verglichen. Wurde bis zum Ende einer der beiden Zeichenketten kein Unterschied festgestellt, gilt die längere als größer.

```
int CompTo (t_Char c_comp) const;
```

Vergleicht die eigene Zeichenkette vollständig mit dem Zeichen `c_comp`. Dieses gilt als Zeichenkette der Länge Eins.

```
int CompTo (const t_Char * pc_comp) const;
```

Vergleicht die eigene Zeichenkette vollständig mit der nullterminierten Zeichenkette `pc_comp`.

```
int CompTo (const t_Char * pc_comp, t_Size o_len) const;
```

Vergleicht die eigene Zeichenkette vollständig mit den ersten `o_len` Zeichen der Zeichenkette `pc_comp`.

```
int CompTo (const gct_String & co_comp) const;
```

Vergleicht die eigene Zeichenkette vollständig mit der Zeichenkette `co_comp`.

Zuweisen

Die folgenden Methoden weisen der eigenen Zeichenkette einen neuen Wert zu. Eine Prüfung auf Selbstzuweisung (s. o.) erfolgt nicht bei allen Methoden.

```
void Clear ();
```

Setzt die Länge auf Null.

```
void Assign (t_Char c_asgn);
```

Setzt die Länge auf Eins und übernimmt das Zeichen `c_asgn`. Es darf kein Nullzeichen sein.

`void Assign (t_Char c_asgn, t_Size o_len);`

Setzt die Länge auf `o_len` und füllt die Zeichenkette mit dem Zeichen `c_asgn`. Es darf kein Nullzeichen sein.

`void Assign (const t_Char * pc_asgn);`

Übernimmt die nullterminierte Zeichenkette `pc_asgn` vollständig. Es wird eine echte Kopie (deep copy) angefertigt. Der Inhalt von `pc_asgn` wird in den eigenen Speicherbereich kopiert (mit Prüfung auf Selbstzuweisung).

`void Assign (const t_Char * pc_asgn, t_Size o_len);`

Übernimmt die ersten `o_len` Zeichen der Zeichenkette `pc_asgn`. Es wird eine echte Kopie (deep copy) angefertigt. Der Inhalt von `pc_asgn` wird in den eigenen Speicherbereich kopiert (ohne Prüfung auf Selbstzuweisung).

`void Assign (const gct_String & co_asgn);`

Übernimmt die Zeichenkette `co_asgn` vollständig. Es wird eine echte Kopie (deep copy) angefertigt. Der Inhalt von `co_asgn` wird in den eigenen Speicherbereich kopiert (mit Prüfung auf Selbstzuweisung).

Anfügen

Die folgenden Methoden fügen an das Ende der eigenen Zeichenkette einen neuen Wert an. Eine Prüfung auf Selbstzuweisung (s. o.) erfolgt nicht bei allen Methoden.

`void Append (t_Char c_app);`

Fügt an das Ende das einzelne Zeichen `c_app` an. Es darf kein Nullzeichen sein.

`void Append (t_Char c_app, t_Size o_len);`

Fügt an das Ende die `o_len`-fache Wiederholung des Zeichens `c_app` an. Es darf kein Nullzeichen sein.

`void Append (const t_Char * pc_app);`

Fügt an das Ende die nullterminierte Zeichenkette `pc_app` an (mit Prüfung auf Selbstzuweisung).

`void Append (const t_Char * pc_app, t_Size o_len);`

Fügt an das Ende die ersten `o_len` Zeichen der Zeichenkette `pc_app` an (ohne Prüfung auf Selbstzuweisung).

`void Append (const gct_String & co_app);`

Fügt an das Ende die Zeichenkette `co_app` an (mit Prüfung auf Selbstzuweisung).

Einfügen

Die folgenden Methoden fügen an der Position `o_pos` eine Zeichenkette ein. Eine Prüfung auf Selbstzuweisung (s. o.) erfolgt nicht.

`void Insert (t_Size o_pos, t_Char c_ins);`

Fügt an der Position `o_pos` das einzelne Zeichen `c_ins` ein. Es darf kein Nullzeichen sein. Es muß `o_pos <= GetLen ()` gelten.

`void Insert (t_Size o_pos, t_Char c_ins, t_Size o_len);`

Fügt an der Position `o_pos` die `o_len`-fache Wiederholung des Zeichens `c_ins` ein. Es darf kein Nullzeichen sein. Es muß `o_pos <= GetLen ()` gelten.

`void Insert (t_Size o_pos, const t_Char * pc_ins);`

Fügt an der Position `o_pos` die nullterminierte Zeichenkette `pc_ins` ein. Es muß `o_pos <= GetLen ()` gelten.

`void Insert (t_Size o_pos, const t_Char * pc_ins, t_Size o_len);`

Fügt an der Position `o_pos` die ersten `o_len` Zeichen der Zeichenkette `pc_ins` ein. Es muß `o_pos <= GetLen ()` gelten.

`void Insert (t_Size o_pos, const gct_String & co_ins);`

Fügt an der Position `o_pos` die Zeichenkette `co_ins` ein. Es muß `o_pos <= GetLen ()` gelten.

Löschen

`void Delete (t_Size o_pos);`

Löscht alle Zeichen ab der Position `o_pos`. Es muß `o_pos <= GetLen ()` gelten.

`void Delete (t_Size o_pos, t_Size o_len);`

Löscht `o_len` Zeichen ab der Position `o_pos`. Es muß `o_pos + o_len <= GetLen ()` gelten.

`void DeleteRev (t_Size o_len);`

Löscht die letzten `o_len` Zeichen. Es muß `o_len <= GetLen ()` gelten.

Ersetzen

Die folgenden Methoden ersetzen `o_delLen` Zeichen an der Position `o_pos` durch eine andere Zeichenkette. Eine Prüfung auf Selbstzuweisung (s. o.) erfolgt nicht.

`void Replace (t_Size o_pos, t_Size o_delLen, t_Char c_ins);`

Ersetzt `o_delLen` Zeichen an der Position `o_pos` durch das einzelne Zeichen `c_ins`. Es darf kein Nullzeichen sein. Es muß `o_pos + o_delLen <= GetLen ()` gelten.

`void Replace (t_Size o_pos, t_Size o_delLen, t_Char c_ins, t_Size o_insLen);`

Ersetzt `o_delLen` Zeichen an der Position `o_pos` durch die `o_insLen`-fache Wiederholung des Zeichens `c_ins`. Es darf kein Nullzeichen sein. Es muß `o_pos + o_delLen <= GetLen ()` gelten.

`void Replace (t_Size o_pos, t_Size o_delLen, const t_Char * pc_ins);`

Ersetzt `o_delLen` Zeichen an der Position `o_pos` durch die nullterminierte Zeichenkette `pc_ins`. Es muß `o_pos + o_delLen <= GetLen ()` gelten.

`void Replace (t_Size o_pos, t_Size o_delLen, const t_Char * pc_ins, t_Size o_insLen);`

Ersetzt `o_delLen` Zeichen an der Position `o_pos` durch die ersten `o_insLen` Zeichen der Zeichenkette `pc_ins`. Es muß `o_pos + o_delLen <= GetLen ()` gelten.

`void Replace (t_Size o_pos, t_Size o_delLen, const gct_String & co_ins);`

Ersetzt `o_delLen` Zeichen an der Position `o_pos` durch die Zeichenkette `co_ins`. Es muß `o_pos + o_delLen <= GetLen ()` gelten.

Alles ersetzen

`t_Size ReplaceAll (const gct_String & co_search, const gct_String & co_replace);`

Ersetzt alle Teilstrings, die gleich `co_search` sind, durch `co_replace` und liefert die Anzahl der Ersetzungen. Es wird ein optimierter Algorithmus mit minimalen Reallokationen verwendet.

Formatierte Zeichenketten

Die folgenden Methoden verhalten sich wie `Assign`, `Append`, `Insert` bzw. `Replace`. Sie behandeln jedoch ihre Parameterliste als eine formatierte Zeichenkette im `printf`-Format und liefern deren Länge. Ein negativer Rückgabewert deutet darauf hin, daß bei einem Parameter ein Formatierungsfehler auftrat (siehe Abschnitt 'Zeichenketten formatieren').

```
int AssignF (const t_Char * pc_format, ...);
```

Übernimmt die formatierte Zeichenkette `pc_format` vollständig.

```
int AppendF (const t_Char * pc_format, ...);
```

Fügt an das Ende die formatierte Zeichenkette `pc_format` an.

```
int InsertF (t_Size o_pos, const t_Char * pc_format, ...);
```

Fügt an der Position `o_pos` die formatierte Zeichenkette `pc_format` ein. Es muß `o_pos <= GetLen ()` gelten.

```
int ReplaceF (t_Size o_pos, t_Size o_delLen, const t_Char * pc_format, ...);
```

Ersetzt `o_delLen` Zeichen an der Position `o_pos` durch die formatierte Zeichenkette `pc_format`. Es muß `o_pos + o_delLen <= GetLen ()` gelten.

Klein-/Großbuchstaben

Die folgenden Methoden nutzen globale Funktionen der Systemschnittstelle (siehe Abschnitt 'Zeichen und Zeichenketten', Funktionen `tl_ToUpper` und `tl_ToLower`).

```
bool ToUpper ();
```

Wandelt die gesamte Zeichenkette in Großbuchstaben um (Windows-1252).

```
bool ToLower ();
```

Wandelt die gesamte Zeichenkette in Kleinbuchstaben um (Windows-1252).

```
bool ToUpper2 ();
```

Wandelt die gesamte Zeichenkette in Großbuchstaben um (teilw. UTF-kompatibel).

```
bool ToLower2 ();
```

Wandelt die gesamte Zeichenkette in Kleinbuchstaben um (teilw. UTF-kompatibel).

Vergleichsoperatoren

Die folgenden Operatoren vergleichen die beiden Zeichenketten vollständig miteinander. Wurde bis zum Ende einer der beiden Zeichenketten kein Unterschied festgestellt, gilt die längere als größer. Die Operatoren verhalten sich semantisch wie `CompTo`, liefern jedoch als Rückgabewert `true` oder `false`.

```
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;
```

Zuweisungsoperatoren

Die folgenden Operatoren weisen der eigenen Zeichenkette einen neuen Wert zu. Es erfolgt eine Prüfung auf Selbstzuweisung (s. o.).

```
gct_String & operator = (t_Char c_asgn);
```

Setzt die Länge auf Eins und übernimmt das Zeichen `c_asgn`. Es darf kein Nullzeichen sein.

```
gct_String & operator = (const t_Char * pc_asgn);
```

Übernimmt die nullterminierte Zeichenkette `pc_asgn` vollständig. Es wird eine echte Kopie (deep copy) angefertigt. Der Inhalt von `pc_asgn` wird in den eigenen Speicherbereich kopiert.

```
gct_String & operator = (const gct_String & co_asgn);
```

Übernimmt die Zeichenkette `co_asgn` vollständig. Es wird eine echte Kopie (deep copy) angefertigt. Der Inhalt von `co_asgn` wird in den eigenen Speicherbereich kopiert.

Anfügeoperatoren

Die folgenden Operatoren fügen an das Ende der eigenen Zeichenkette einen neuen Wert an. Es erfolgt eine Prüfung auf Selbstzuweisung (s. o.).

```
gct_String & operator += (t_Char c_app);
```

Fügt an das Ende das einzelne Zeichen `c_app` an. Es darf kein Nullzeichen sein.

```
gct_String & operator += (const t_Char * pc_app);
```

Fügt an das Ende die nullterminierte Zeichenkette `pc_app` an.

```
gct_String & operator += (const gct_String & co_app);
```

Fügt an das Ende die Zeichenkette `co_app` an.

Temporäres Anfügen

Die folgenden Operatoren erzeugen ein temporäres String-Objekt, das eine Kopie der eigenen Zeichenkette und den angefügten Wert enthält. Die eigene Zeichenkette bleibt unverändert.

```
gct_String operator + (t_Char c_app) const;
```

Liefert ein String-Objekt mit angefügtem einzelnen Zeichen `c_app`. Es darf kein Nullzeichen sein.

```
gct_String operator + (const t_Char * pc_app) const;
```

Liefert ein String-Objekt mit angefügter nullterminierter Zeichenkette `pc_app`.

```
gct_String operator + (const gct_String & co_app) const;
```

Liefert ein String-Objekt mit angefügter Zeichenkette `co_app`.

```
friend gct_String operator + (t_Char c_init, const gct_String & co_app);
```

Liefert ein String-Objekt, das aus der Zusammensetzung von `c_init` und `co_app` besteht.

```
friend gct_String operator + (const t_Char * pc_init, const gct_String & co_app);
```

Liefert ein String-Objekt, das aus der Zusammensetzung von `pc_init` und `co_app` besteht.

Konvertieren

Die folgenden Methoden dienen der Konvertierung zwischen `char`- und `wchar_t`-basierten Stringobjekten. Es erfolgt keine Prüfung auf Selbstzuweisung (s. o.).

```
template <class t_string> void Convert (const t_string & co_asgn);
```

Übernimmt die Zeichenkette `co_asgn` ohne Berücksichtigung von Multibytecharacters.

```
template <class t_string> bool MbConvert (const t_string & co_asgn);
```

Übernimmt die Zeichenkette `co_asgn` mit Berücksichtigung von Multibytecharacters (siehe Abschnitt 'Zeichen und Zeichenketten').

```
template <class t_asgnChar> bool MbConvert (const t_asgnChar * po_asgn);
```

Übernimmt die nullterminierte Zeichenkette `po_asgn` mit Berücksichtigung von Multibytecharacters (siehe Abschnitt 'Zeichen und Zeichenketten').

3.2.2 String-Instanzen (tuning/xxx/[w]string.h)

Zur Erleichterung des Umgangs mit der Stringschnittstelle werden in der Bibliothek **Spirick Tuning** einige Standardinstanzen des Klassentemplates `gct_String` vordefiniert. Die Makros `STRING_DCL(t_Block, StoreSpec)` und `WSTRING_DCL(t_Block, StoreSpec)` generieren für eine Wrapperklasse eines globalen Storeobjekts eine Stringklasse. Die Makroverwendung

```
STRING_DCL (gct_AnyBlock, ct_Any32)
```

expandiert zu folgendem Text (die Makroparameter sind fett hervorgehoben):

```
typedef gct_String <gct_CharBlock <gct_NullDataBlock  
  <gct_AnyBlock <ct_Any32Store>, char>, char>, ct_Any32Store> ct_Any32String;
```

Die Makroverwendung

```
WSTRING_DCL (gct_AnyBlock, ct_Any32)
```

expandiert zu folgendem Text (die Makroparameter sind fett hervorgehoben):

```
typedef gct_String <gct_CharBlock <gct_NullDataBlock  
  <gct_AnyBlock <ct_Any32Store>, wchar_t>, wchar_t>, ct_Any32Store> ct_Any32WString;
```

Stringklassen werden nicht wie Blockklassen oder Arraycontainer im 'Vierer-Block' generiert, sondern einzeln. Eine Stringklasse enthält vergleichsweise sehr viele Methoden. Das Erzeugen mehrerer Klassen in einem einzigen Makro würde die Übersetzungsdauer unnötig erhöhen.

Jedes Verzeichnis eines dynamischen Stores enthält die zwei Stringdateien 'string.h' und 'wstring.h'. In jeder Datei wird mit Hilfe des Makros `STRING_DCL` bzw. `WSTRING_DCL` je eine Stringklasse deklariert. Z. B. enthält die Datei 'tuning/std/string.h' die Klasse `ct_Std_String`. Sie besitzt den Größentyp `t_UInt` und fordert den Speicher für den dynamischen Block vom globalen Standardstoreobjekt an.

In der Datei 'tuning/std/[w]string.h' wird deklariert:

```
typedef ... ct_Std_[W]String;
```

In der Datei 'tuning/rnd/[w]string.h' wird deklariert:

```
typedef ... ct_Rnd_[W]String;
```

In der Datei 'tuning/chn/[w]string.h' wird deklariert:

```
typedef ... ct_Chn_[W]String;
```

3.2.3 Polymorphe Stringklassen (tuning/[w]string.hpp)

Neben den vordefinierten Instanzen des Templates `gct_String` enthält die Bibliothek **Spirick Tuning** die beiden Stringklassen `ct_String` und `ct_WString`, die von polymorphen Collections verwaltet werden können. Das Makro `OBJ_STRING_DCL(StoreSpec)` generiert eine Stringklasse, die von einer vordefinierten Templateinstanz erbt, die wiederum von `ct_Object` abgeleitet ist. Darin werden Konstruktoren, Gleichoperatoren und `ct_Object`-spezifische Methoden definiert. Die Makroverwendung

```
OBJ_STRING_DCL(ct_Chn_Obj)
```

expandiert zu folgendem Text (die Makroparameter sind fett hervorgehoben):

```
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_ObjectString & 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 GetHashCode () 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);
};
...
```

Zusätzliche Methoden

```
bool operator < (const ct_Object & co_comp) const;
```

Dieser Vergleichsoperator wird aufgerufen, wenn ein Stringobjekt in eine sortierte Arraycollection eingefügt wird. Er prüft, ob das übergebene Objekt vom Typ `ct_String` (oder abgeleitet) ist und führt dann einen Zeichenkettenvergleich aus. Andernfalls wird der Vergleichsoperator der Basisklasse aufgerufen.

In der Datei 'tuning/string.hpp' wird deklariert:

```
OBJ_STRING_DCL(ct_Chn_Obj)
typedef ct_Chn_ObjectString ct_String;
```

In der Datei 'tuning/wstring.hpp' wird deklariert:

```
OBJ_STRING_DCL(ct_Chn_WObj)
typedef ct_Chn_WObjectString ct_WString;
```

3.2.4 Dateiname (tuning/filename.hpp)

Die Klasse `ct_FileName` bietet zahlreiche Möglichkeiten zur Bearbeitung von Dateinamen. Ein Dateiname wird als zusammenhängende nullterminierte Zeichenkette gespeichert. Auf seine Komponenten kann mit Hilfe ihrer Position (des Offsets) zugegriffen werden. Einzelne Komponenten können zwar nicht als nullterminierte Zeichenketten abgefragt, aber von einem anderen Dateinamenobjekt kopiert werden.

Ein Dateiname wird in vier Komponenten unterteilt: Laufwerk (Drive), Pfad (Path), Name (Name) und Erweiterung (Ext). Laufwerk und Pfad werden zusammengenommen `DrivePath` genannt, Name und Erweiterung `NameExt`. Die Pfadkomponente enthält stets einen abschließenden [Back]slash. Der Pfad ohne diesen [Back]slash wird `PurePath` genannt, Laufwerk und Pfad ohne [Back]slash `PureDrivePath`.

Die Klasse `ct_FileName` unterstützt auch die Universal Naming Convention (UNC). Anstatt einer Laufwerksbezeichnung (z. B. A:) kann ein Netzwerkname (z. B. \\server\\share) stehen. In beiden Fällen wird mit der Bezeichnung `Drive` auf die Komponente zugegriffen. Nur in den Methoden `HasDriveOrUNC`, `HasDrive` und `HasUNC` wird zwischen Laufwerksbezeichnung und Netzwerkname unterschieden.

Bei der Zuweisung einer Pfadkomponente werden unter MS Windows automatisch Slash-Zeichen durch einen Backslash ersetzt (unter Linux umgekehrt). Wird die Pfadkomponente einzeln zugewiesen, ist der abschließende [Back]slash optional und wird ggf. ergänzt. Der trennende Punkt wird weder dem Namen noch der Erweiterung zugeordnet. Bei der Zuweisung einer einzelnen Erweiterung ist die Angabe des Punktes optional.

Nach dem Zuweisen einer vollständigen Zeichenkette werden die Positionen der einzelnen Komponenten berechnet. Dafür existieren zwei Möglichkeiten. Die 'Zuweisung als Name' versucht, am Ende der Zeichenkette den Namen und die Erweiterung zu erkennen. Nur wenn die Zeichenkette mit einem [Back]slash endet, sind Namens- und Erweiterungskomponente leer. Bei der 'Zuweisung als Pfad' werden stets die letzten Zeichen der Pfadkomponente zugeordnet und ggf. ein [Back]slash angehängt.

Basisklassen

`ct_Object` (siehe Abschnitt 'Abstraktes Objekt')
`ct_String` (siehe Abschnitt 'Polymorphe Stringklasse')

Klassendeklaration

```
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);
};

```

Methoden

`ct_FileName ()`;

Initialisiert ein leeres Dateinamenobjekt.

`ct_FileName (const char * pc_init)`;

Initialisiert das Objekt mit der Methode `AssignAsName`.

```
ct_FileName & operator = (const char * pc_asgn);
```

Ruft die Methode AssignAsName auf.

```
ct_FileName & operator = (const ct_FileName & co_asgn);
```

Übernimmt alle Angaben des Objektes 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);
```

Diese Methoden weisen dem Objekt eine neue Zeichenkette zu und berechnen die Positionen der Komponenten. Die letzten Zeichen werden dem Pfad zugeordnet. Name und Erweiterung sind leer.

```
void AssignAsName (const char * pc_name);  
void AssignAsName (const char * pc_name, t_Size u_len);  
void AssignAsName (const ct_String & co_name);
```

Diese Methoden weisen dem Objekt eine neue Zeichenkette zu und berechnen die Positionen der Komponenten. Es wird versucht, am Ende Name und Erweiterung zu erkennen. Nur wenn die Zeichenkette mit einem Backslash endet, sind Name und Erweiterung leer.

```
bool HasDriveOrUNC () const;  
bool HasDrive () const;  
bool HasUNC () const;  
bool HasPath () const;  
bool HasName () const;  
bool HasExt () const;
```

Diese Methoden liefern bei Vorhandensein einzelner Komponenten den Wert true.

```
bool HasDot () const;
```

Liefert true, wenn zwischen Name und Erweiterung ein Punkt vorhanden ist.

```
bool HasWildCards () const;
```

Liefert true, wenn in Name oder Erweiterung Wildcards ('*' und '?') vorkommen.

```
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;
```

Diese Methoden liefern die Längen einzelner Komponenten.

```
t_Size GetDotLen () const;
```

Liefert den Wert Eins, wenn zwischen Name und Erweiterung ein Punkt vorhanden ist, sonst Null.

```
t_Size GetAllLen () const;
```

Liefert die Länge des gesamten Dateinamens.

```
t_Size GetDriveOffs () const;  
t_Size GetPathOffs () const;  
t_Size GetNameOffs () const;  
t_Size GetExtOffs () const;
```

Diese Methoden liefern die Positionen einzelner Komponenten.

```

const char * GetDriveStr () const;
const char * GetPathStr () const;
const char * GetNameStr () const;
const char * GetExtStr () const;
const char * GetAllStr () const;

```

Diese Methoden liefern Zeiger auf den Anfang einzelner Komponenten.

```

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;

```

Diese Methoden liefern einzelne Komponenten als temporäre Stringobjekte.

```

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);

```

Mit diesen Methoden können einzelne Komponenten geändert werden.

```

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);

```

Diese Methoden kopieren einzelne Komponenten von einem anderen Objekt.

```

void InsertPath (const char * pc_path);
void InsertPath (const char * pc_path, t_Size u_len);
void InsertPath (const ct_String & co_path);

```

Diese Methoden fügen am Anfang der Pfadkomponente einen Teilpfad ein.

```

void InsertDrivePath (const char * pc_path);
void InsertDrivePath (const char * pc_path, t_Size u_len);
void InsertDrivePath (const ct_String & co_path);

```

Diese Methoden fügen am Anfang der Pfadkomponente einen Teilpfad ein und ersetzen die Laufwerkskomponente.

```

void AppendPath (const char * pc_path);
void AppendPath (const char * pc_path, t_Size u_len);
void AppendPath (const ct_String & co_path);

```

Diese Methoden fügen am Ende der Pfadkomponente einen Teilpfad an.


```
void CompressPath ();
```

Diese Methode entfernt sich aufhebende Teilpfade (`.\` und `path\.\`) aus der Pfadkomponente. Z. B. wird `A:\SRC\.\SPIRICK.TXT` zu `A:\SRC\SPIRICK.TXT` und `A:\SRC\.\SPIRICK.TXT` zu `A:\SPIRICK.TXT` komprimiert.

```
bool IsAbs () const;
```

Liefert `true`, wenn der Pfad absolut ist, also mit einem Backslash beginnt.

```
bool IsRel () const;
```

Liefert `true`, wenn der Pfad relativ ist.

```
void ToAbs (const char * pc_currDrivePath, bool b_withDrive = true);
```

Wandelt den vorhandenen relativen Pfad in einen absoluten bzgl. des Verzeichnisses `pc_currDrivePath` um. Ist `b_withDrive` gleich `true`, wird das Laufwerk von `pc_currDrivePath` übernommen, andernfalls wird die Laufwerkskomponente gelöscht.

```
void ToRel (const char * pc_currDrivePath, bool b_withDrive = false);
```

Wandelt den vorhandenen absoluten Pfad in einen relativen bzgl. des Verzeichnisses `pc_currDrivePath` um. Ist `b_withDrive` gleich `true`, wird das Laufwerk von `pc_currDrivePath` übernommen, andernfalls wird die Laufwerkskomponente gelöscht.

3.2.5 Zeichenketten formatieren (tuning/printf.hpp)

In dieser Schnittstelle befindet sich die für `char` und `wchar_t` überladene Funktionen `t1_Vsprintf`. Sie ermöglicht das Formatieren von Zeichenketten mit variabler Länge und variabler Anzahl von Parametern. Es wird empfohlen, diese Funktionen nicht direkt zu verwenden, sondern über die Stringmethoden `AssignF`, `AppendF`, `InsertF` und `ReplaceF`. Im Beispielprogramm `TString` befinden sich auch Demonstrationsbeispiele für `t1_Vsprintf`.

Funktionen

```
int t1_Vsprintf (char * * ppc_buffer, const char * pc_format, va_list o_argList);  
int t1_Vsprintf (wchar_t * * ppc_buffer, const wchar_t * pc_format, va_list o_argList);
```

Formatiert die Zeichenkette `pc_format` mit den Parametern `o_argList` und schreibt das Ergebnis in einen Puffer, der mit `malloc` allokiert wurde. Bei Erfolg wird die Anzahl der Zeichen (ohne das abschließende Nullzeichen) zurückgegeben, und `* ppc_buffer` enthält einen Zeiger auf den Puffer, der mit `free` freigegeben werden muß. Andernfalls wird eine Zahl kleiner als Null zurückgegeben, und der Puffer muß nicht freigegeben werden.

3.2.6 Zeichenketten sortieren (tuning/stringsort.hpp)

Die Bibliothek **Spirick Tuning** enthält einen optimierten Sortieralgorithmus. Er ist auf Zeichenketten spezialisiert. Diese besitzen die Eigenschaft, aus einzelnen Zeichen zu bestehen. Ein Zeichen wiederum besitzt einen Wertebereich von 0 bis 255. Um Werte in diesem Bereich zu sortieren, müssen sie nicht miteinander verglichen werden, sondern ihr Wert kann als Index zum Eintrag in eine Tabelle genutzt werden. Anschließend wird die Tabelle von 0 bis 255 durchlaufen. Dabei erscheinen die Werte in sortierter Reihenfolge.

Wurde dieser Schritt für das erste Zeichen durchgeführt, können jeder Stelle in der Tabelle mehrere Zeichenketten zugeordnet worden sein. Deshalb legt der Algorithmus eine Kette an. Diese wird anschließend mit demselben Verfahren, aber dem nächsten Zeichen (dem zweiten, dritten usw.) sortiert.

Der Eintrag in die Tabelle erfolgt indirekt über eine `SortPage`. Damit kann die natürliche Sortierreihenfolge geändert werden. Sollen z. B. Klein- und Großbuchstaben gleichberechtigt behandelt werden, enthält die

SortPage an der Stelle mit dem Index 'a' den Wert 'A'. Soll zusätzlich das 'Ä' unter 'A' einsortiert werden, muß auch an der Stelle mit dem Index 'Ä' ein 'A' eingetragen werden.

Die private Methode `GetDefaultSortPage` liefert als Voreinstellung eine SortPage mit natürlicher Sortierreihenfolge. Damit werden die Strings nach aufsteigender Wertigkeit ihrer Zeichen sortiert. Soll in umgekehrter Reihenfolge sortiert werden, ist beim Index `i` der Wert `256 - i` einzutragen. Das erste Zeichen (Index Null) einer SortPage muß stets den Wert Null besitzen.

Der Algorithmus erwartet als Parameter ein C++-Array von Zeigern (`const char * * ppc_strings`) und schreibt seine Resultate in ein C++-Array von `t_Int`-Werten (`t_Int * pi_sortedIndex`). Der Speicher für beide Arrays muß vom Anwender verwaltet werden. Das `t_Int`-Array enthält am Ende in aufsteigender Reihenfolge die Indizes der sortierten Strings. Das Stringarray selbst wird nicht geändert.

Das Sortierverfahren benötigt folgenden Speicher:

1. Den Parameter `char * apc [n]` und das Resultat `t_Int ai [n]`, wobei `n` die Anzahl der zu sortierenden Strings ist.
2. Das Array `t_Int ai_temp [n]`, in dem die Ketten gespeichert werden.
3. `x * 256 * sizeof (t_Int)` für die Reihenfolgetabellen, wobei `x` die maximale Anzahl der Zeichen ist, in denen zwei Zeichenketten am Anfang übereinstimmen.

Die Rechenzeit ist nicht von der Länge der Zeichenketten abhängig, sondern von der Länge, mit der zwei Strings am Anfang übereinstimmen. Diese Abhängigkeit besteht bei `qsort ()` in Verbindung mit `strcmp ()` auch, denn `strcmp ()` bricht an der Stelle der ersten Nichtübereinstimmung ab. Im Gegensatz zu `qsort ()` ist der neue Sortieralgorithmus jedoch nicht von einer eventuellen Vorsortierung abhängig. Er benötigt für ein vollkommen unsortiertes Array genauso lange wie für ein vollständig sortiertes und ist im Durchschnitt doppelt so schnell wie `qsort ()` in Verbindung mit `strcmp ()`.

Klassendeklaration

```
class ct_StringSort
{
public:
    bool Sort (const char * * ppc_strings, t_Int * pi_sortedIndex, t_Int i_numOfStrings,
               const char * pc_sortPage = GetDefaultSortPage ());
};
```

Methoden

`bool Sort (const char * * ppc_strings, t_Int * pi_sortedIndex, t_Int i_numOfStrings, const char * pc_sortPage = GetDefaultSortPage ());`

Speichert die sortierten Indizes des Zeichenketten-Arrays `ppc_strings` in `pi_sortedIndex`. Die Arrays `ppc_strings` und `pi_sortedIndex` müssen vom Anwender bereitgestellt und freigegeben werden. Temporäre Zwischenspeicher werden automatisch angefordert und freigegeben. Der Rückgabewert `false` deutet auf Speichermangel oder einen Fehler in der SortPage (erstes Zeichen ungleich Null) hin.

3.2.7 Zahlen sortieren (tuning/stringsort.hpp)

Der Algorithmus zum Sortieren von Zeichenketten (siehe voriger Abschnitt) kann auch auf Zahlen angewendet werden, indem z. B. eine `t_UInt32` Zahl als Folge von vier Zeichen betrachtet wird. Diese Idee wurde in der Klasse `ct_UInt32Sort` umgesetzt. Der Algorithmus wurde für little-endian Hardware implementiert.

Klassendeklaration

```
class ct_UInt32Sort
{
```

```

public:
    bool                Sort (const t_UInt32 * pu_ints, t_Int * pi_sortedIndex,
                             t_Int i_numOfInts);
};

```

Methoden

`bool Sort (const t_UInt32 * pu_ints, t_Int * pi_sortedIndex, t_Int i_numOfInts);`

Speichert die sortierten Indizes des `t_UInt32`-Arrays `pu_ints` in `pi_sortedIndex`. Die Arrays `pu_ints` und `pi_sortedIndex` müssen vom Anwender bereitgestellt und freigegeben werden. Temporäre Zwischenspeicher werden automatisch angefordert und freigegeben. Der Rückgabewert `false` deutet auf Speichermangel hin.

3.3 Dateien und Verzeichnisse

3.3.1 Datei (tuning/file.hpp)

Innerhalb der Bibliothek **Spirick Tuning** werden Pfad- und Dateinamen als UTF-8-Strings interpretiert. Unter Linux werden die Strings unverändert an die Systemfunktionen übergeben. Unter MS Windows werden Pfad- und Dateinamen intern in UTF-16 umgewandelt.

Die Klasse `ct_File` hüllt die globalen Funktionen der Systemschnittstelle in ein objektorientiertes Gewand und enthält einige Zusatzfunktionen, z. B. das Schließen der Datei im Destruktor. `ct_File` erbt von `ct_FileName`. Damit stehen zahlreiche Methoden zum Bearbeiten des Namens der Datei zur Verfügung. Die Methoden `TryOpen`, `Open`, `Create`, `Load`, `Save`, `Exists`, `Move`, `Copy` und `Delete` dürfen nur auf eine nicht geöffnete Datei angewendet werden.

Basisklassen

`ct_Object` (siehe Abschnitt 'Abstraktes Objekt')
`ct_String` (siehe Abschnitt 'Polymorphe Stringklasse')
`ct_FileName` (siehe Abschnitt 'Dateiname')

Klassendeklaration

```

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;
};

```

Methoden

ct_File ();

Initialisiert das Dateiojekt.

ct_File (const char * pc_init);

Initialisiert das Dateiojekt und ruft ct_FileName::AssignAsName (pc_init) auf.

ct_File (const ct_FileName & co_init);

Initialisiert das Dateiojekt mit dem Dateinamen co_init.

~ct_File ();

Schließt die Datei, wenn sie noch geöffnet ist.

ct_File & operator = (const char * pc_asgn);

Ruft ct_FileName::AssignAsName (pc_asgn) auf.

ct_File & operator = (const ct_FileName & co_asgn);

Weist dem Dateiojekt den neuen Dateinamen co_asgn zu.

bool TryOpen (bool b_readOnly = true, bool b_sequential = true, t_UInt32 u_milliSec = 0);

Versucht, eine bestehende Datei abhängig vom Parameter b_readOnly zum Lesen oder Schreiben zu öffnen. Der optionale Parameter b_sequential beeinflusst die Arbeitsweise des Cachemanagers. Wird die Datei sequentiell bearbeitet, sollte er auf true gesetzt werden. Liefert false, wenn das Öffnen innerhalb von u_milliSec Millisekunden nicht gelingt.

bool Open (bool b_readOnly = true, bool b_sequential = true);

Öffnet eine bestehende Datei abhängig vom Parameter b_readOnly zum Lesen oder Schreiben. Der optionale Parameter b_sequential beeinflusst die Arbeitsweise des Cachemanagers. Wird die Datei sequentiell bearbeitet, sollte er auf true gesetzt werden.

bool Create (bool b_createNew = false);

Erzeugt eine neue Datei und öffnet sie zum Schreiben. Eine eventuell vorhandene Datei gleichen Namens wird überschrieben. Liefert false, wenn b_createNew gleich true ist und eine Datei mit demselben Namen bereits existiert.

bool Close ();

Schließt die geöffnete Datei.

bool Load (ct_String * pco_str);

Läd den gesamten Inhalt der Datei (öffnen, lesen, schließen) in das Stringobjekt pco_str. Die Datei darf keine Nullzeichen enthalten.

```
bool Save (const ct_String * pco_str);
```

Sichert den gesamten Inhalt des Stringobjekts `pco_str` in die Datei (öffnen, schreiben, schließen).

```
bool Exists ();
```

Liefert `true`, wenn die Datei existiert.

```
bool Move (const char * pc_new);
```

Verschiebt die Datei nach `pc_new`. Befinden sich alter und neuer Name innerhalb desselben Verzeichnisses, wird nur der Name des Eintrags geändert. Bei Erfolg wird auch der interne Name (Basisklasse `ct_FileName`) aktualisiert.

```
bool Copy (const char * pc_new, bool b_overwrite = true);
```

Kopiert die Datei nach `pc_new`. Ist der optionale Parameter `b_overwrite` gleich `true`, wird eine eventuell vorhandene Datei gleichen Namens überschrieben.

```
bool Delete ();
```

Löscht die Datei.

```
bool QuerySize (t_FileSize & o_size) const;
```

Ermittelt die aktuelle Größe der geöffneten Datei.

```
bool QueryPos (t_FileSize & o_pos) const;
```

Ermittelt die aktuelle Position des Zugriffszeigers der geöffneten Datei.

```
bool EndOfFile (bool & b_eof) const;
```

Setzt `b_eof` auf `true`, wenn sich der Zugriffszeiger am Ende der Datei befindet.

```
bool SeekAbs (t_FileSize o_pos) const;
```

Positioniert den Zugriffszeiger der geöffneten Datei absolut auf die Position `o_pos`.

```
bool SeekRel (t_FileSize o_pos) const;
```

Positioniert den Zugriffszeiger der geöffneten Datei relativ auf die Position `o_pos`.

```
bool Truncate (t_FileSize o_size);
```

Verändert die Größe der geöffneten Datei auf `o_size` Bytes.

```
bool Read (void * pv_dst, t_FileSize o_len) const;
```

Liest `o_len` Bytes aus der geöffneten Datei nach `pv_dst` und verschiebt den Zugriffszeiger.

```
bool Write (const void * pv_src, t_FileSize o_len) const;
```

Schreibt `o_len` Bytes von `pv_src` in die geöffnete Datei und verschiebt den Zugriffszeiger.

3.3.2 Verzeichnis (tuning/dir.hpp)

Innerhalb der Bibliothek **Spirick Tuning** werden Pfad- und Dateinamen als UTF-8-Strings interpretiert. Unter Linux werden die Strings unverändert an die Systemfunktionen übergeben. Unter MS Windows werden Pfad- und Dateinamen intern in UTF-16 umgewandelt.

Die Klasse `ct_Directory` hüllt die globalen Funktionen der Systemschnittstelle in ein objektorientiertes Gewand und enthält einige Zusatzfunktionen, z. B. das Zerlegen des aktuellen Verzeichnisses in seine Komponenten. `ct_Directory` erbt von `ct_FileName`. Damit stehen zahlreiche Methoden zum Bearbeiten des Namens des Verzeichnisses zur Verfügung. Es werden jedoch nur die Laufwerks- und Pfadkomponente verarbeitet (`PureDrivePath`). Name und Erweiterung werden nicht berücksichtigt.

Basisklassen

ct_Object (siehe Abschnitt 'Abstraktes Objekt')
ct_String (siehe Abschnitt 'Polymorphe Stringklasse')
ct_FileName (siehe Abschnitt 'Dateiname')

Klassendeklaration

```
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 ();
};
```

Methoden

ct_Directory ();

Initialisiert das Verzeichnisobjekt.

ct_Directory (const char * pc_init);

Initialisiert das Verzeichnisobjekt und ruft ct_FileName::AssignAsPath (pc_init) auf.

ct_Directory (const ct_FileName & co_init);

Initialisiert das Verzeichnisobjekt mit dem Dateinamen co_init.

ct_Directory & operator = (const char * pc_asgn);

Ruft ct_FileName::AssignAsPath (pc_asgn) auf.

ct_Directory & operator = (const ct_FileName & co_asgn);

Weist dem Verzeichnisobjekt den neuen Dateinamen co_asgn zu.

bool QueryCurrentDrive ();

Setzt die Laufwerkskomponente auf das aktuelle Laufwerk.

bool QueryCurrentDirectory ();

Setzt die Pfadkomponente auf das aktuelle Verzeichnis der Laufwerkskomponente, verändert die Laufwerkskomponente nicht.

bool QueryCurrentDriveDirectory ();

Setzt die Laufwerkskomponente, wenn sie noch leer ist, und setzt die Pfadkomponente auf das aktuelle Verzeichnis.

```
bool Create ();
```

Erzeugt ein Verzeichnis.

```
bool Exists ();
```

Liefert true, wenn das Verzeichnis existiert.

```
bool Move (const char * pc_new);
```

Verschiebt das Verzeichnis nach `pc_new`. Befinden sich alter und neuer Name innerhalb desselben übergeordneten Verzeichnisses, wird nur der Name des Eintrags geändert. Bei Erfolg wird auch der interne Name (Basisklasse `ct_FileName`) aktualisiert.

```
bool Delete ();
```

Löscht das leere Verzeichnis.

3.3.3 Verzeichnis durchlaufen (tuning/dirscan.hpp)

Innerhalb der Bibliothek **Spirick Tuning** werden Pfad- und Dateinamen als UTF-8-Strings interpretiert. Unter Linux werden die Strings unverändert an die Systemfunktionen übergeben. Unter MS Windows werden Pfad- und Dateinamen intern in UTF-16 umgewandelt.

Die Klasse `ct_DirScan` erbt von `ct_Directory` und indirekt von `ct_FileName`. Die Laufwerks- und Pfadkomponente des Dateinamens bestimmen das zu durchlaufende Verzeichnis. Die Namens- und Erweiterungskomponente dienen als Parameter (siehe unten) und als Resultat. Während des Durchlaufens des Verzeichnisses enthalten sie den Namen des aktuellen Eintrags.

Die Funktionen zum Durchlaufen von Verzeichnissen können auch zum Ermitteln der Eigenschaften einer einzelnen Datei verwendet werden. In diesem Fall wird in der Methode `FindFirst` der Dateiname ohne Jokerzeichen verwendet. Liefert die Funktion true, wurden mit einem einzigen Systemruf sämtliche Attribute und Zeitangaben ermittelt. Das abschließende `FindNext` oder `AbortFind` darf nicht vergessen werden.

Die `FindOnce`-Methoden fassen drei Arbeitsschritte in einem Aufruf zusammen: Zuerst wird ein evtl. aktiver Suchvorgang beendet, danach wird ein neuer Name zugewiesen, und mit `FindFirst` wird ein neuer Suchvorgang gestartet.

Basisklassen

<code>ct_Object</code>	(siehe Abschnitt 'Abstraktes Objekt')
<code>ct_String</code>	(siehe Abschnitt 'Polymorphe Stringklasse')
<code>ct_FileName</code>	(siehe Abschnitt 'Dateiname')
<code>ct_Directory</code>	(siehe Abschnitt 'Verzeichnis')

Datentypen, Konstanten

```
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;
```

Mit dem Datentyp `t_FileAttributes` und den zugehörigen Konstanten können mehrere Dateiattribute in einen einzigen Wert durch Oder-Verknüpfung zusammengefaßt werden.

Klassendeklaration

```
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;
};
```

Methoden

ct_DirScan ();

Initialisiert ein leeres Objekt.

ct_DirScan (const char * pc_init);

Initialisiert das Objekt und ruft ct_FileName::AssignAsName (pc_init) auf.

ct_DirScan (const ct_FileName & co_init);

Initialisiert das Objekt mit dem Dateinamen co_init.

~ct_DirScan ();

Gibt die evtl. vorhandene Betriebssystem-Ressource frei.

ct_DirScan & operator = (const char * pc_asgn);

Ruft ct_FileName::AssignAsName (pc_asgn) auf.

ct_DirScan & operator = (const ct_FileName & co_asgn);

Weist dem Objekt den neuen Dateinamen co_asgn zu.

`bool FindOnce ();`

Beginnt mit dem aktuellen Dateinamen einen neuen Suchvorgang.

`bool FindOnce (const char * pc_find);`

Ruft `ct_FileName::AssignAsName (pc_find)` auf und beginnt einen neuen Suchvorgang.

`bool FindOnce (const ct_FileName & co_find);`

Ruft `ct_FileName::AssignAsName (co_find)` auf und beginnt einen neuen Suchvorgang.

`bool FindOncePath ();`

Ruft `ct_FileName::AssignAsName (GetPureDrivePath ())` auf und beginnt einen neuen Suchvorgang, d. h. es wird nicht im aktuellen Verzeichnis gesucht, sondern das Verzeichnis selbst wird gesucht.

`bool FindOncePath (const ct_FileName & co_find);`

Ruft `ct_FileName::AssignAsName (co_find.GetPureDrivePath ())` auf und beginnt einen neuen Suchvorgang, d. h. es wird nicht im Verzeichnis von `co_find` gesucht, sondern das Verzeichnis selbst wird gesucht.

`bool FindFirst ();`

Sucht den ersten Verzeichniseintrag und liefert bei Erfolg `true`. Anschließend können Eigenschaften des Verzeichniseintrages abgefragt und mit `FindNext` der nächste gesucht werden.

`bool FindFirstFile ();`

Sucht den ersten Verzeichniseintrag, der kein Unterverzeichnis ist, und liefert bei Erfolg `true`. Anschließend können Eigenschaften der Datei abgefragt und mit `FindNextFile` die nächste gesucht werden.

`bool FindFirstDirectory ();`

Sucht den ersten Verzeichniseintrag, der ein Unterverzeichnis ist, und liefert bei Erfolg `true`. Anschließend können Eigenschaften des Unterverzeichnisses abgefragt und mit `FindNextDirectory` das nächste gesucht werden.

`bool FindNext ();`

Sucht den nächsten Verzeichniseintrag und liefert bei Erfolg `true`. Anschließend können Eigenschaften des Verzeichniseintrages abgefragt und weitere gesucht werden. Beim Rückgabewert `false` ist die Suche beendet, und die Betriebssystem-Ressource wurde automatisch freigegeben.

`bool FindNextFile ();`

Sucht den nächsten Verzeichniseintrag, der kein Unterverzeichnis ist, und liefert bei Erfolg `true`. Anschließend können Eigenschaften der Datei abgefragt und weitere gesucht werden. Beim Rückgabewert `false` ist die Suche beendet, und die Betriebssystem-Ressource wurde automatisch freigegeben.

`bool FindNextDirectory ();`

Sucht den nächsten Verzeichniseintrag, der ein Unterverzeichnis ist, und liefert bei Erfolg `true`. Anschließend können Eigenschaften des Unterverzeichnisses abgefragt und weitere gesucht werden. Beim Rückgabewert `false` ist die Suche beendet, und die Betriebssystem-Ressource wurde automatisch freigegeben.

`void AbortFind ();`

Beendet die Suche vorzeitig und gibt die Betriebssystem-Ressource frei. Vor dem Aufruf der Methode muß mindestens ein Verzeichniseintrag gefunden worden sein.

`bool Found ();`

Liefert `true`, wenn der vorhergehende Aufruf von `FindFirst` oder `FindNext` den Wert `true` geliefert hat.

```
t_MicroTime GetCreationTime () const;
```

Liefert die Zeit, an der der Verzeichniseintrag erzeugt wurde, als UTC Mikrosekunden.

```
t_MicroTime GetLastAccessTime () const;
```

Liefert die Zeit, an der zuletzt auf den Verzeichniseintrag lesend oder schreibend zugegriffen wurde, als UTC Mikrosekunden.

```
t_MicroTime GetLastWriteTime () const;
```

Liefert die Zeit, an der zuletzt auf den Verzeichniseintrag schreibend zugegriffen wurde, als UTC Mikrosekunden.

```
t_FileSize GetSize () const;
```

Liefert die Größe des Verzeichniseintrages.

```
t_FileAttributes GetAttributes () const;
```

Liefert alle Attribute des Verzeichniseintrags in einen einzigen Wert.

```
bool IsArchive () const;
```

```
bool IsDirectory () const;
```

```
bool IsHidden () const;
```

```
bool IsReadOnly () const;
```

```
bool IsSystem () const;
```

Diese Methoden ermitteln einzelne Attribute des Verzeichniseintrages.

Parameterarten für Verzeichnisse

Die Klasse `ct_DirScan` erbt von `ct_Directory` und indirekt von `ct_FileName`. Die Laufwerks- und Pfadkomponente des Dateinamens bestimmen das zu durchlaufende Verzeichnis. Mit Hilfe der Methode `ct_Directory::Exists` kann geprüft werden, ob das Verzeichnis existiert.

```
ct_DirScan co_dirScan;  
co_dirScan. SetDrivePath ("c:\\spirick\\tuning");  
  
if (co_dirScan. Exists ())  
    // ...
```

Die Namens- und Erweiterungskomponente des Dateinamens bestimmen den Inhalt, nach dem gesucht werden soll. Dafür existieren drei Parameterarten:

```
co_dirScan. SetNameExt ("*");
```

Die Zeichenkette "*" führt zum ungefilterten Durchlaufen des gesamten Verzeichnisses.

```
co_dirScan. SetNameExt ("*.?pp");
```

Befinden sich in Name oder Erweiterung die Wildcards '*' oder '?', liefert die Klasse `ct_DirScan` nur die zutreffenden Verzeichniseinträge. Wildcards sind auf UNIX-Systemen nicht verfügbar.

```
co_dirScan. SetNameExt ("dirscan.hpp");
```

Name und Erweiterung können auch einen eindeutigen Dateinamen enthalten. Liefert die Methode `FindFirst` den Wert `true`, existiert dieser Verzeichniseintrag. Anschließend können alle zugehörigen Informationen abgefragt werden.

Während des Durchlaufens des Verzeichnisses werden Name und Erweiterung des aktuellen Eintrags im `ct_DirScan`-Objekt gespeichert. Der ursprüngliche Inhalt von Name und Erweiterung geht dabei verloren.

Verzeichnis vollständig durchlaufen

Zum vollständigen Durchlaufen eines Verzeichnisses wird eine `for`-Schleife nach folgendem Muster empfohlen:

```
ct_DirScan co_dirScan ("c:\\spirick\\tuning\\*");

for (co_dirScan. FindFirst ();
     co_dirScan. Found ();
     co_dirScan. FindNext ())
{
    // ...
}
```

Verzeichnis durchlaufen, nur Dateien

Zum Durchlaufen aller Dateien eines Verzeichnisses (ohne Unterverzeichniseinträge) wird eine `for`-Schleife nach folgendem Muster empfohlen:

```
ct_DirScan co_dirScan ("c:\\spirick\\tuning\\*");

for (co_dirScan. FindFirstFile ();
     co_dirScan. Found ();
     co_dirScan. FindNextFile ())
{
    // ...
}
```

Verzeichnis durchlaufen, nur Unterverzeichnisse

Zum Durchlaufen aller Unterverzeichniseinträge eines Verzeichnisses (ohne Dateien) wird eine `for`-Schleife nach folgendem Muster empfohlen:

```
ct_DirScan co_dirScan ("c:\\spirick\\tuning\\*");

for (co_dirScan. FindFirstDirectory ();
     co_dirScan. Found ();
     co_dirScan. FindNextDirectory ())
{
    // ...
}
```

3.4 Weitere Werkzeuge

3.4.1 Uhrzeit und Datum (tuning/timedata.hpp)

Die Klasse `ct_TimeDate` hüllt die globalen Funktionen der Systemschnittstelle in ein objektorientiertes Gewand und ermöglicht den Zugriff auf die einzelnen Datums- und Zeit-Komponenten. Die Zeit wird in Mikrosekunden seit dem 1. 1. 1970 0 Uhr angegeben. Es kann sowohl die koordinierte Weltzeit (UTC) als auch die lokale Zeit verwendet werden, die der im Betriebssystem eingestellten Zeitzone entspricht.

Klassendeklaration

```
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;
};

```

Methoden

ct_TimeDate ();

Setzt alle Komponenten auf den Wert Null.

ct_TimeDate (t_MicroTime i_time);

Berechnet aus einem Wert in Mikrosekunden die einzelnen Komponenten.

void Clear ();

Setzt alle Komponenten auf den Wert Null.

t_MicroTime GetTime () const;

Berechnet aus den einzelnen Komponenten einen Wert in Mikrosekunden.

void SetTime (t_MicroTime i_time);

Berechnet aus einem Wert in Mikrosekunden die einzelnen Komponenten.

void QueryUTCTime ();

Fragt die aktuelle UTC Systemzeit ab.

void QueryLocalTime ();

Fragt die aktuelle lokale Systemzeit ab.

```

unsigned GetYear () const;
unsigned GetMonth () const;
unsigned GetDay () const;
unsigned GetDayOfWeek () const;
unsigned GetHour () const;
unsigned GetMinute () const;
unsigned GetSecond () const;
unsigned GetMicroSecond () const;

```

Diese Methoden liefern die Werte einzelner Komponenten.

```

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);

```

Mit diesen Methoden können einzelne Komponenten geändert werden.

```

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;

```

Diese Operatoren vergleichen zwei ct_TimeDate-Objekte miteinander.

3.4.2 MD5 Summe (tuning/md5.hpp)

Die Klasse ct_MD5 ist für die einmalige Berechnung eines MD5-Hashwertes vorgesehen. Die Daten, aus denen der Hashwert berechnet wird, können komplett im Konstruktor oder in mehreren Blöcken übergeben werden. Am Ende der Berechnung kann das Ergebnis binär oder textuell abgefragt werden. Für eine neue Berechnung muß ein neues ct_MD5-Objekt verwendet werden.

Klassendeklaration

```

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;
};

```

Methoden

ct_MD5 ();

Initialisiert das Objekt.

```
ct_MD5 (const t_MD5Result & ac_init);
```

Initialisiert das Objekt mit einem vorhandenen binären Rechenergebnis.

```
ct_MD5 (const void * pv_data, t_UInt u_len);
```

Initialisiert das Objekt und ruft die Methoden `Update` und `Finalize` auf.

```
void Update (const void * pv_data, t_UInt u_len);
```

Stehen die Originaldaten nicht in einem zusammenhängenden Speicherbereich zur Verfügung, so können mit der `Update`-Methode nacheinander einzelne Teilblöcke zur Berechnung übergeben werden. Die Parameter `pv_data` und `u_len` beschreiben Position und Länge eines Teilblocks.

```
void Finalize ();
```

Beendet die Hashwert-Berechnung. Anschließend kann das Rechenergebnis abgefragt werden.

```
const t_MD5Result & GetResult () const;
```

Liefert das binäre Rechenergebnis.

```
const char * GetResultStr ();
```

Liefert das textuelle Rechenergebnis. Es besteht aus 32 Hexadezimalziffern mit Kleinbuchstaben und einem abschließenden Nullzeichen.

```
bool operator == (const ct_MD5 & co_comp) const;
```

Vergleicht zwei `ct_MD5`-Objekte, bei denen die Berechnung abgeschlossen ist.

3.4.3 Universally Unique Identifier (tuning/uuid.hpp)

Mit der Klasse `ct_UUID` können Universally Unique Identifier erzeugt und verarbeitet werden.

Klassendeklaration

```
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;
};
```

Methoden

```
ct_UUID ();
```

Initialisiert alle Elemente mit Nullen.

```
ct_UUID (const ct_UUID & co_init);
```

Initialisiert das Objekt durch Kopie von einem anderen Objekt.

```
ct_UUID (const t_UUID & ao_init);
```

Initialisiert das Objekt mit einem vorhandenen binären Rechenergebnis.

```
ct_UUID & operator = (const ct_UUID & co_asgn);
```

Übernimmt den Inhalt des Objektes `co_asgn`.

```
bool IsEmpty () const;
```

Liefert `true`, wenn alle Elemente gleich Null sind.

```
t_UInt GetHashCode () const;
```

Liefert einen Hashwert, der z. B. für den Eintrag in eine Hashtabelle verwendet werden kann.

```
const t_UUID & GetUUID () const;
```

Liefert das binäre Rechenergebnis.

```
void Clear ();
```

Setzt alle Elemente auf Null.

```
bool Create ();
```

Erzeugt einen neuen Universally Unique Identifier.

```
bool ToStr (char * pc_dst, t_UInt u_len, bool b_upperCase) const;
```

Berechnet eine textuelle Repräsentation der UUID. Das Ergebnis besteht aus 36 Zeichen und enthält kein abschließendes Nullzeichen. Position und Länge des Resultatpuffers werden durch `pc_dst` und `u_len` bestimmt. Die Länge des Puffers muß mindestens 36 Zeichen betragen. Ist der Parameter `b_upperCase` gleich `true`, dann werden Großbuchstaben verwendet.

```
bool FromStr (const char * pc_src, t_UInt u_len);
```

Berechnet eine UUID aus einer textuellen Repräsentation. Es werden die ersten 36 Zeichen des Puffers verwendet, der durch `pc_src` und `u_len` bestimmt ist.

```
bool operator == (const ct_UUID & co_comp) const;
```

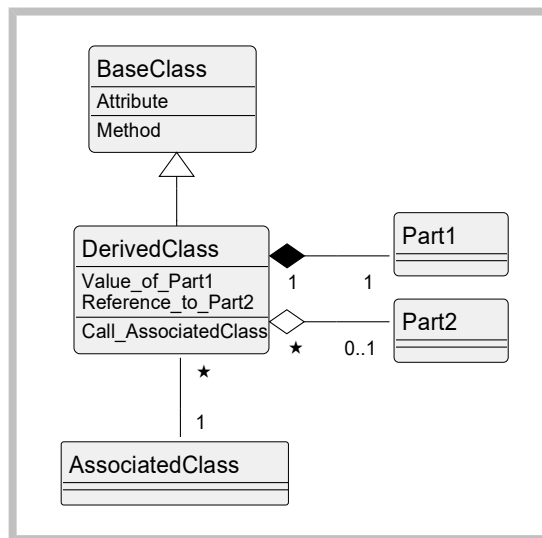
```
bool operator != (const ct_UUID & co_comp) const;
```

Diese Operatoren vergleichen zwei `ct_UUID`-Objekte miteinander.





4 DESIGNDIAGRAMME

4.1 Zur Notation

Die folgenden Abschnitte enthalten einige objektorientierte Designdiagramme. Sie veranschaulichen das Zusammenspiel der Komponenten innerhalb der Bibliothek **Spirick Tuning**. Zur ihrer Erstellung wurde die Notation 'Unified Modeling Language' (UML) verwendet. Die folgende Abbildung zeigt einen Teil der grafischen Elemente der UML.



Klassen werden als umrandete Vierecke dargestellt. Jedes Viereck enthält drei Bereiche: den Klassennamen, die Liste der Attribute und die Liste der Methoden. In den Designdiagrammen der folgenden Abschnitte werden vier Verbindungsarten verwendet:

- die Vererbung, 
- die Aggregation als Wert, 
- die Aggregation als Referenz und 
- die Assoziation. 

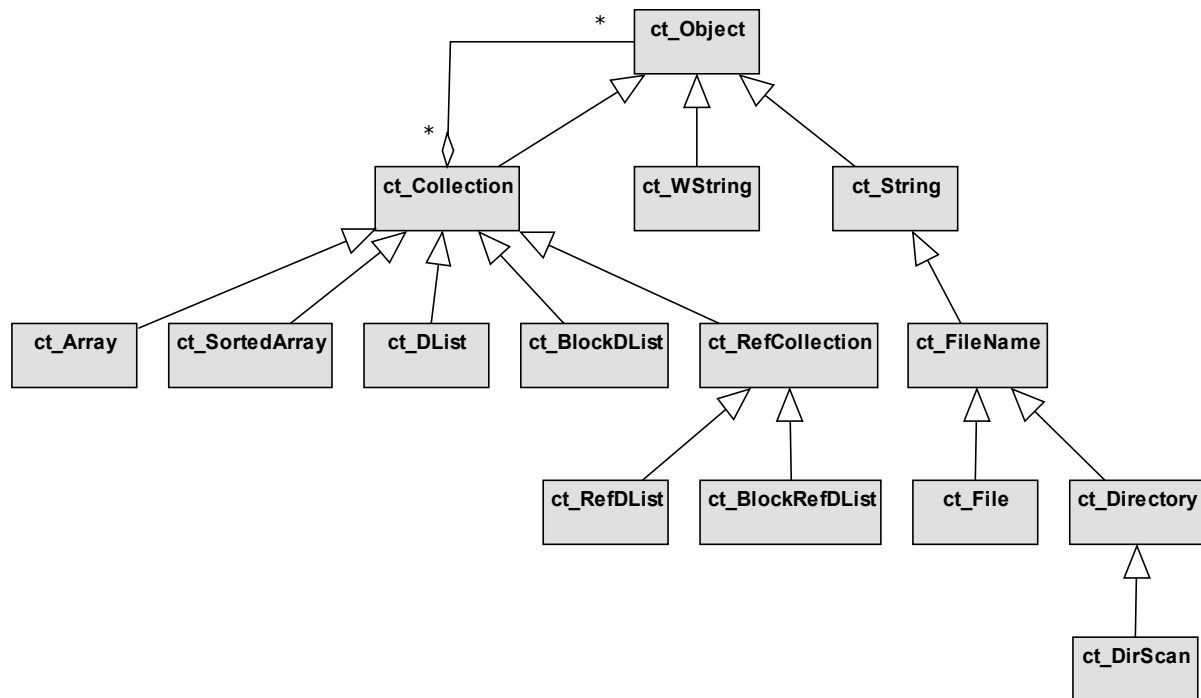
Aggregationen (Teil-Ganzes-Relationen) können zusätzlich mit Kardinalitäten versehen werden. Im Beispiel bedeuten sie: Ein Objekt der Klasse **DerivedClass** enthält genau ein Objekt der Klasse **Part1**, und ein Objekt der Klasse **Part1** gehört (im Sinne dieser Relation) zu genau einem Objekt der Klasse **DerivedClass**. Ein Objekt der Klasse **DerivedClass** enthält optional eine Referenz auf ein Objekt der Klasse **Part2**, und ein Objekt der Klasse **Part2** kann von beliebig vielen Objekten der Klasse **DerivedClass** referenziert werden.

Zwischen **DerivedClass** und **AssociatedClass** besteht eine Assoziation. Auch diese kann mit Kardinalitäten näher beschrieben werden. Die Bedeutung im Beispiel ist: Zu einem Objekt der Klasse **DerivedClass** existiert genau ein Objekt der Klasse **AssociatedClass**, und zu einem Objekt der Klasse **AssociatedClass** existieren beliebig viele Objekte der Klasse **DerivedClass**.

4.2 Polymorphe Klassenhierarchie

In der folgenden Abbildung wird die polymorphe Klassenhierarchie der Bibliothek **Spirick Tuning** zusammengefaßt. Sie zeigt sämtliche Klassen, die direkt oder indirekt von `ct_Object` erben. Zur Erhöhung der Übersichtlichkeit wurde die große Zahl der Attribute und Methoden ausgeblendet.

Die Klasse `ct_Collection` enthält Zeiger auf Instanzen des Typs `ct_Object`. Eine Collection kann mehrere Zeiger auf `ct_Object`-Instanzen besitzen. Im Sinne dieser Referenz-Aggregation ist ein `ct_Object` in beliebig vielen Collections enthalten.



4.3 Ein Array

Die folgende Abbildung enthält sämtliche Klassen, die bei der Implementierung eines Array-Containers verwendet werden. Neben den öffentlichen Methoden werden auch einige private Attribute dargestellt. Sie erleichtern das Verständnis der Verbindungen zwischen den Klassen. Die Containerklasse wurde mit Hilfe der folgenden Anweisungen generiert:

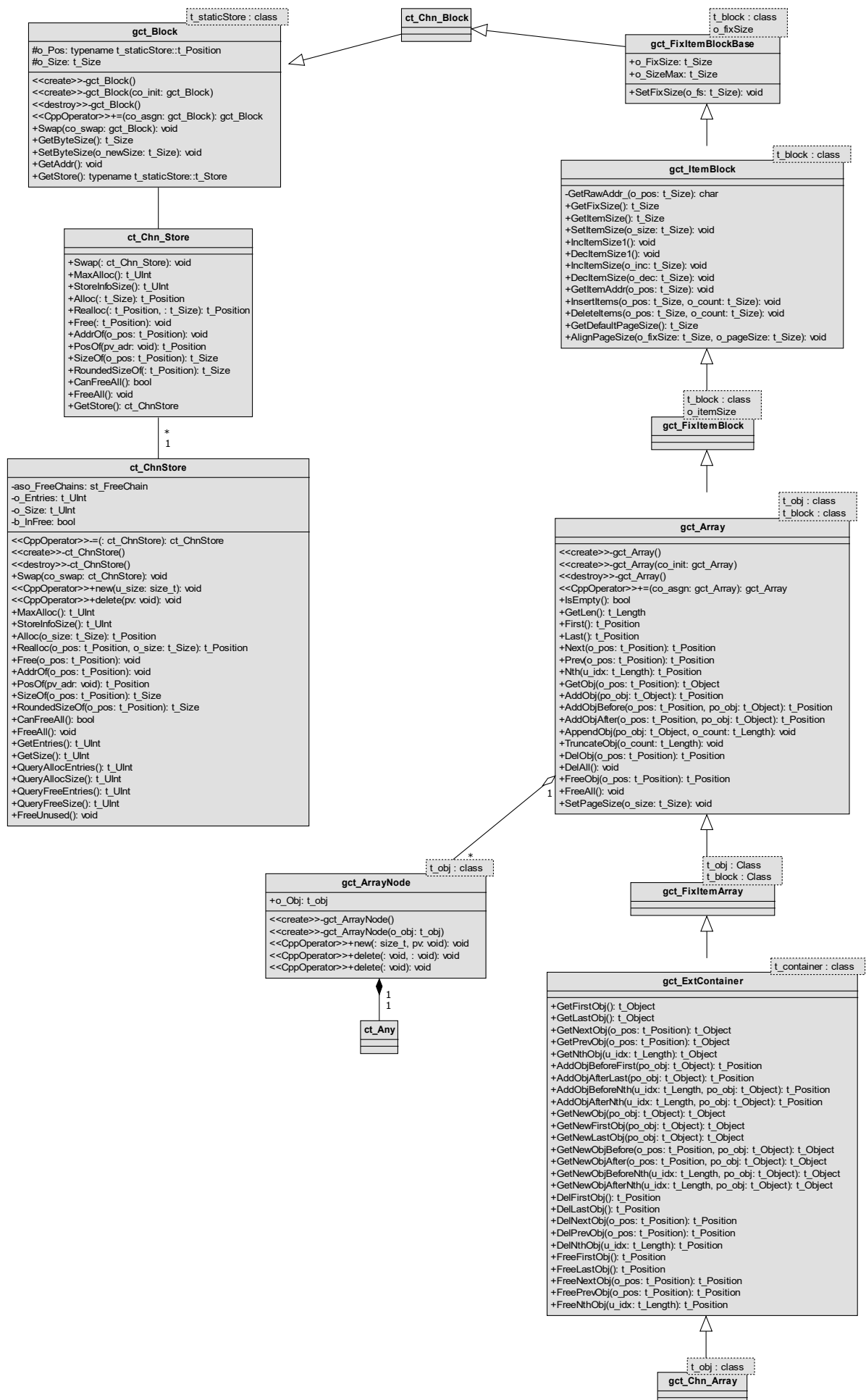
```
#include "tuning/chn/array.h"
class ct_Any { /* ... */ };
gct_Chn_Array <ct_Any> co_AnyArray;
```

Die Implementierung des Array-Containers beginnt auf der untersten Stufe mit der dynamischen Storeklasse `ct_ChnStore`. Von ihr wird eine globale Instanz gebildet. C++-Templates können als Parameter keine Objekte besitzen, sondern nur Typen und Konstanten. Deshalb mappt die Wrapperklasse `ct_Chn_Store` die Methoden des globalen Storeobjekts auf statische Methoden einer Klasse. Von ihr können beliebig viele Instanzen gebildet werden, die jedoch stets auf dasselbe globale Storeobjekt zugreifen. Das Kürzel `_` innerhalb des Namens weist auf den geschachtelten Größentyp `t_UInt` hin.

`ct_Chn_Store` dient dem Template `gct_Block` als Parameter. Zwischen `gct_Block <ct_Chn_Store>` und `ct_Chn_Store` besteht eine Assoziation, denn das Blocktemplate ruft die Methoden der Wrapperklasse auf. Die Klasse `ct_Chn_Block` enthält keine eigenen Attribute und Methoden. Sie dient nur der kürzeren Schreibweise des Namens `gct_Block <ct_Chn_Store>`. Die Blockschnittstelle wird mit Hilfe des Templates `gct_ItemBlock` um Methoden zum Zugriff auf Elemente erweitert. Die Hilfstemplates `gct_FixItemBlockBase` und `gct_FixItemBlock` stellen die feste Elementgröße für `gct_ItemBlock` bereit.

Das Containertemplate `gct_Array` besitzt als Parameter den Typ der verwalteten Objekte `ct_Any` und die Blockklasse `gct_FixItemBlock <t_block, sizeof (gct_ArrayNode <ct_Any>)>`. Der Array-Container erbt von der Blockklasse und nutzt den dynamischen Speicherblock zur kompakten Unterbringung der verwalteten Objekte. Das Hilfstemplate `gct_ArrayNode` dient dem Erzeugen und Löschen der Objekte. Es enthält als Attribut `o_Obj` je ein Objekt und besitzt eigene Operatoren `new` und `delete`. Das Hilfstemplate `gct_FixItemArray` stellt die Parameter für `gct_FixItemBlock` bereit.

Die Containerschnittstelle wird mit Hilfe des Templates `gct_ExtContainer` um nützliche Methoden erweitert. Das Template `gct_Chn_Array` enthält keine eigenen Attribute und Methoden. Es dient nur der kürzeren Schreibweise des Namens `gct_ExtContainer <gct_FixItemArray <t_obj, ct_Chn_Block> >`.



4.4 Ein Zeigerarray

Die folgende Abbildung enthält sämtliche Klassen, die bei der Implementierung eines Zeigerarray-Containers verwendet werden. Neben den öffentlichen Methoden werden auch einige private Attribute dargestellt. Sie erleichtern das Verständnis der Verbindungen zwischen den Klassen. Die Containerklasse wurde mit Hilfe der folgenden Anweisungen generiert:

```
#include "tuning/chn/ptrarray.h"
class ct_Any { /* ... */ };
gct_Chn_PtrArray <ct_Any> co_AnyPtrArray;
```

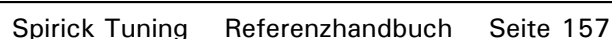
Die Implementierung des Zeigerarray-Containers beginnt auf der untersten Stufe mit der dynamischen Storeklasse `ct_ChnStore`. Von ihr wird eine globale Instanz gebildet. C++-Templates können als Parameter keine Objekte besitzen, sondern nur Typen und Konstanten. Deshalb mappt die Wrapperklasse `ct_Chn_Store` die Methoden des globalen Storeobjekts auf statische Methoden einer Klasse. Von ihr können beliebig viele Instanzen gebildet werden, die jedoch stets auf dasselbe globale Storeobjekt zugreifen. Das Kürzel `_` innerhalb des Namens weist auf den geschachtelten Größentyp `t_UInt` hin.

`ct_Chn_Store` dient dem Template `gct_Block` als Parameter. Zwischen `gct_Block <ct_Chn_Store>` und `ct_Chn_Store` besteht eine Assoziation, denn das Blocktemplate ruft die Methoden der Wrapperklasse auf. Die Klasse `ct_Chn_Block` enthält keine eigenen Attribute und Methoden. Sie dient nur der kürzeren Schreibweise des Namens `gct_Block <ct_Chn_Store>`. Die Blockschnittstelle wird mit Hilfe des Templates `gct_ItemBlock` um Methoden zum Zugriff auf Elemente erweitert. Die Hilfstemplates `gct_FixItemBlockBase` und `gct_FixItemBlock` stellen die feste Elementgröße für `gct_ItemBlock` bereit.

Ein Zeigercontainer baut auf einem Container auf, der untypisierte C++-Zeiger verwaltet. Das Containertemplate `gct_Array` besitzt als Parameter den Typ der verwalteten Objekte `void *` und die Blockklasse `gct_FixItemBlock <t_block, sizeof(gct_ArrayNode <void *>>)`. Der Array-Container erbt von der Blockklasse und nutzt den dynamischen Speicherblock zur kompakten Unterbringung der C++-Zeiger. Das Hilfstemplate `gct_ArrayNode` dient dem Erzeugen und Löschen der Zeiger. Es enthält als Attribut `o_Obj` je einen untypisierten C++-Zeiger und besitzt eigene Operatoren `new` und `delete`. Da der Basiscontainer zur Verwaltung typisierter Zeiger verwendet wird, besitzt `gct_ArrayNode` eine Referenz-Aggregation zur Klasse `ct_Any`. Das Hilfstemplate `gct_FixItemArray` stellt die Parameter für `gct_FixItemBlock` bereit.

Die Containerschnittstelle wird mit Hilfe des Templates `gct_ExtContainer` um nützliche Methoden erweitert. Das Template `gct_Chn_Array` enthält keine eigenen Attribute und Methoden. Es dient nur der kürzeren Schreibweise des Namens `gct_ExtContainer <gct_FixItemArray <t_obj, ct_Chn_Block> >`.

Für C++-Zeiger existiert der Gleichheitsoperator `operator ==`. Deshalb kann der bisherige Container um die Schnittstelle des Templates `gct_CompContainer` erweitert werden. Der davon abgeleitete Zeigercontainer nutzt die Funktionalität seiner Basisklassen und wandelt untypisierte C++-Zeiger in Zeiger auf die Klasse `ct_Any` um. Das Template `gct_Chn_PtrArray` enthält keine eigenen Attribute und Methoden. Es dient nur der kürzeren Schreibweise des Namens `gct_PtrContainer <ct_Any, gct_Chn_Array <void *> >`.



4.5 Eine Liste

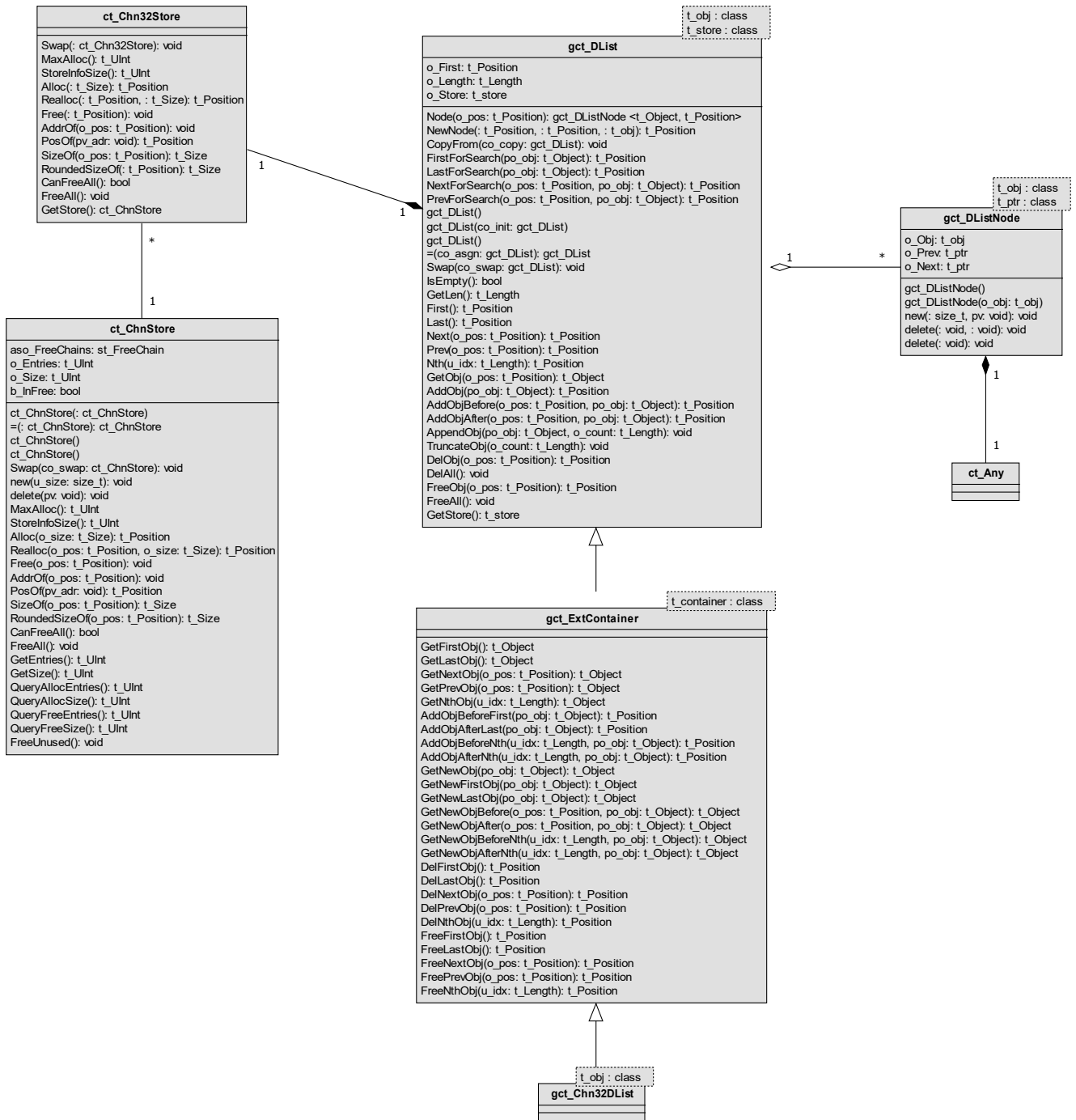
Die folgende Abbildung enthält sämtliche Klassen, die bei der Implementierung eines Listen-Containers verwendet werden. Neben den öffentlichen Methoden werden auch einige private Attribute dargestellt. Sie erleichtern das Verständnis der Verbindungen zwischen den Klassen. Die Containerklasse wurde mit Hilfe der folgenden Anweisungen generiert:

```
#include "tuning/chn/dlist.h"
class ct_Any { /* ... */ };
gct_Ch32DList <ct_Any> co_AnyDList;
```

Die Implementierung des Listen-Containers beginnt auf der untersten Stufe mit der dynamischen Storeklasse `ct_Ch32Store`. Von ihr wird eine globale Instanz gebildet. C++-Templates können als Parameter keine Objekte besitzen, sondern nur Typen und Konstanten. Deshalb mappt die Wrapperklasse `ct_Ch32Store` die Methoden des globalen Storeobjekts auf statische Methoden einer Klasse. Von ihr können beliebig viele Instanzen gebildet werden, die jedoch stets auf dasselbe globale Storeobjekt zugreifen. Das Kürzel 32 innerhalb des Namens weist auf den geschachtelten Größentyp `t_UInt32` hin.

Das Containertemplate `gct_DList` besitzt als Parameter den Typ der verwalteten Objekte `ct_Any` und die Storeklasse `ct_Ch32Store`. Der Listen-Container enthält die Storeklasse als Attribut `o_Store` und nutzt deren Methoden zum Verwalten des Speichers für die Knoten (Nodes). Das Hilfstemplate `gct_DListNode` dient dem Erzeugen und Löschen der Objekte. Es enthält als Attribut `o_Obj` je ein Objekt und besitzt eigene Operatoren `new` und `delete`. Listen-Nodes sind durch Positionszeiger in beiden Richtungen miteinander verbunden.

Die Containerschnittstelle wird mit Hilfe des Templates `gct_ExtContainer` um nützliche Methoden erweitert. Das Template `gct_Ch32DList` enthält keine eigenen Attribute und Methoden. Es dient nur der kürzeren Schreibweise des Namens `gct_ExtContainer <gct_DList <ct_Any, ct_Ch32Store> >`.



4.6 Eine Blockliste

Die folgende Abbildung enthält sämtliche Klassen, die bei der Implementierung eines Blocklisten-Containers verwendet werden. Neben den öffentlichen Methoden werden auch einige private Attribute dargestellt. Sie erleichtern das Verständnis der Verbindungen zwischen den Klassen. Die Containerklasse wurde mit Hilfe der folgenden Anweisungen generiert:

```
#include "tuning/chn/blockdlist.h"
class ct_Any { /* ... */ };
gct_Ch32BlockDList <ct_Any> co_AnyBlockDList;
```

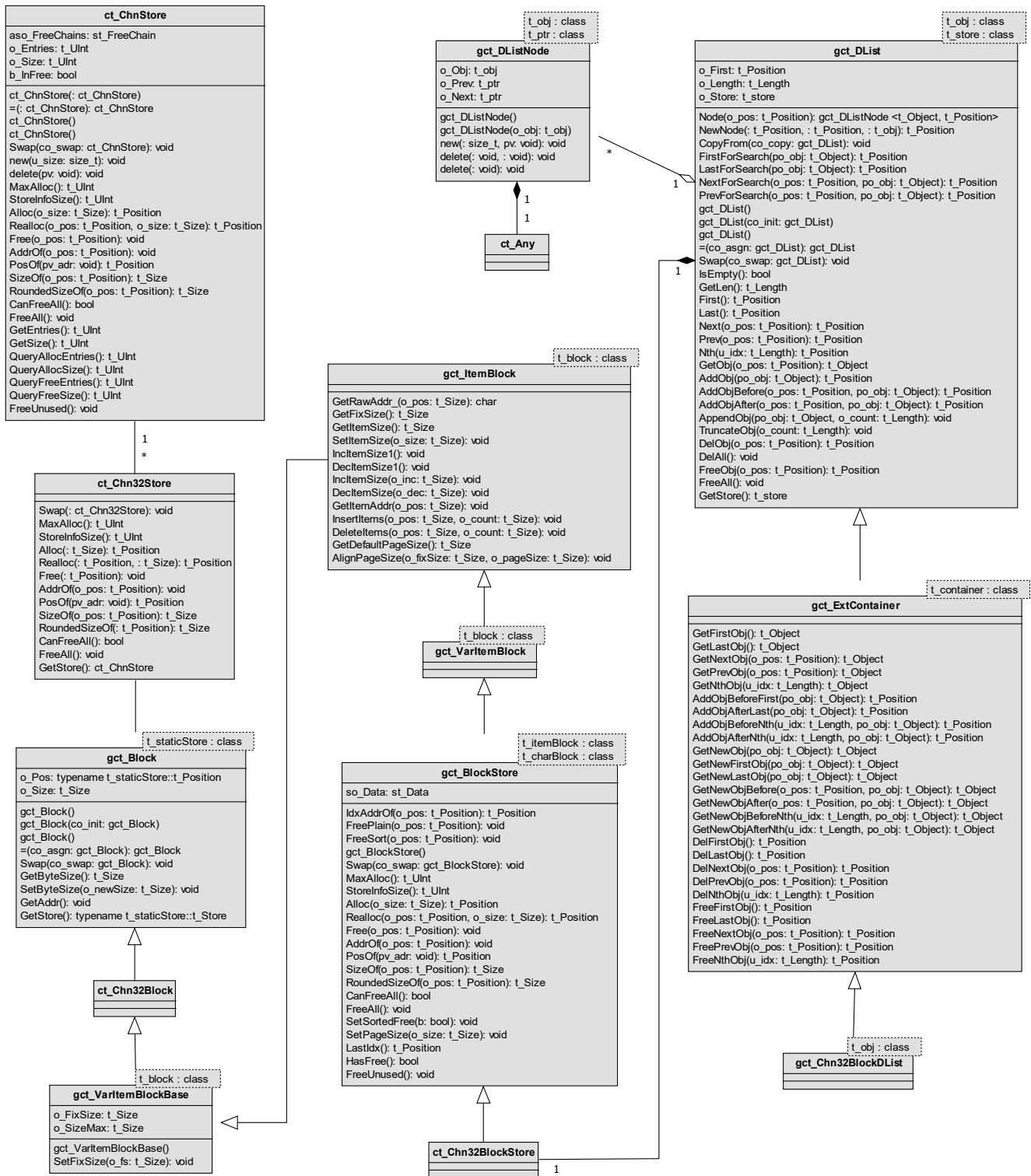
Die Implementierung des Blocklisten-Containers beginnt auf der untersten Stufe mit der dynamischen Storeklasse `ct_Ch32Store`. Von ihr wird eine globale Instanz gebildet. C++-Templates können als Parameter keine Objekte besitzen, sondern nur Typen und Konstanten. Deshalb mappt die Wrapperklasse `ct_Ch32Store` die Methoden des globalen Storeobjekts auf statische Methoden einer Klasse. Von ihr können beliebig viele Instanzen gebildet werden, die jedoch stets auf dasselbe globale Storeobjekt zugreifen. Das Kürzel 32 innerhalb des Namens weist auf den geschachtelten Größentyp `t_UInt32` hin.

`ct_Ch32Store` dient dem Template `gct_Block` als Parameter. Zwischen `gct_Block <ct_Ch32Store>` und `ct_Ch32Store` besteht eine Assoziation, denn das Blocktemplate ruft die Methoden der Wrapperklasse auf. Die Klasse `ct_Ch32Block` enthält keine eigenen Attribute und Methoden. Sie dient nur der kürzeren Schreibweise des Namens `gct_Block <ct_Ch32Store>`. Die Blockschnittstelle wird mit Hilfe des Templates `gct_ItemBlock` um Methoden zum Zugriff auf Elemente erweitert. Die Hilfstemplates `gct_VarItemBlockBase` und `gct_VarItemBlock` stellen die feste Elementgröße für `gct_ItemBlock` bereit.

Das Storetemplate `gct_BlockStore` besitzt als Parameter die Elementblockklasse `gct_VarItemBlock <ct_Ch32Block>` und die Zeichenblockklasse `gct_CharBlock <ct_Ch32Block, char>`. Der Blockstore erbt von der Elementblockklasse und nutzt den dynamischen Speicherblock zur kompakten Verwaltung der eigenen Speicherblöcke. Die Klasse `ct_Ch32BlockStore` enthält keine eigenen Attribute und Methoden. Sie dient nur der kürzeren Schreibweise des Namens `gct_BlockStore <gct_Var..., gct_Char...>`.

Das Containertemplate `gct_DList` besitzt als Parameter den Typ der verwalteten Objekte `ct_Any` und die Storeklasse `ct_Ch32BlockStore`. Der Listen-Container enthält die Storeklasse als Attribut `o_Store` und nutzt deren Methoden zum Verwalten des Speichers für die Knoten (Nodes). Das Hilfstemplate `gct_DListNode` dient dem Erzeugen und Löschen der Objekte. Es enthält als Attribut `o_Obj` je ein Objekt und besitzt eigene Operatoren `new` und `delete`. Listen-Nodes sind durch Positionszeiger in beiden Richtungen miteinander verbunden.

Die Containerschnittstelle wird mit Hilfe des Templates `gct_ExtContainer` um nützliche Methoden erweitert. Das Template `gct_Ch32BlockDList` enthält keine eigenen Attribute und Methoden. Es dient nur der kürzeren Schreibweise des Namens `gct_ExtContainer <gct_DList <ct_Any, ct_Ch32BlockStore> >`.



5 INSTALLATION UND BEISPIELE

5.1 Hinweise zur Installation

5.1.1 Verfügbare Plattformen

Die Klassenbibliothek **Spirick Tuning** ist zur Zeit für die Betriebssysteme MS Windows XP, MS Windows 7, MS Windows 10 sowie Linux x86/x86-64 mit Kernel 2.6.32 bis 5.13.0 verfügbar. Sie wurde mit 32- und 64-Bit-Speichermodellen entwickelt und getestet. Die Klassen können sowohl in einer singlethreaded als auch in einer multithreaded Umgebung eingesetzt werden. Der Quelltext ist an die Compiler MS Visual C++ 8.0 (2005) bis 17.0 (2022) sowie g++ 4.4.5 bis 11.2.0 angepasst.

Die Verfügbarkeit für andere Umgebungen kann beim Hersteller erfragt werden. Die Klassen zur Speicher- und Objektverwaltung und zur Zeichenkettenverarbeitung sind größtenteils system- und compilerunabhängig. Sie lassen sich mit geringem Aufwand portieren. Bei globalen Funktionen und Klassen, die Systemdienste zur Verfügung stellen, ist jedoch eine detaillierte Anpassung erforderlich.

5.1.2 Abhängigkeiten

Die Klassenbibliothek **Spirick Tuning** verwendet das Laufzeitsystem des Compilers und des Betriebssystems. Auf Linux Plattformen werden zusätzlich PThreads verwendet. Darüber hinaus bestehen keine Abhängigkeiten zu anderen Bibliotheken. Die Klassenbibliothek **Spirick Tuning** kann ohne Wechselwirkungen mit anderen Bibliotheken kombiniert werden, z. B. mit BOOST oder der C++ Standardbibliothek.

5.1.3 Installation

Die Klassenbibliothek **Spirick Tuning** wird im Quelltext ausgeliefert. Zum Installieren werden alle Verzeichnisse in ein Verzeichnis auf der Festplatte, z. B. c:\spirick, kopiert. Dabei entstehen folgende Unterverzeichnisse:

Verzeichnis	Inhalt
c:\spirick\tuning	C++-Templates und -Klassen
c:\spirick\tuning\sys	Systemschnittstelle in C
c:\spirick\tuning\std	Standardstore
c:\spirick\tuning\rnd	Roundstore
c:\spirick\tuning\chn	Chainstore
c:\spirick\samples	Beispielprogramme

Im Lieferumfang befinden sich Makefiles, die sowohl mit `nmake` als auch mit `gmake` verwendet werden können. Sie erzeugen eine binäre Bibliothek '**tuning.lib**' bzw. '**tuning.a**'. Die Klassenbibliothek kann auch in ein anderes Buildsystem eingebunden werden. Vor dem Compilieren ist zu prüfen, ob sich das Verzeichnis oberhalb von `tuning` (im Beispiel `c:\spirick`) im Includepfad befindet.

Die mitgelieferten Makefiles verwenden die Umgebungsvariablen `TL_PROJECT_TARGETDIR`, `TL_COMPILER` und `TL_RELEASE`. Mit `TL_PROJECT_TARGETDIR` kann das Zielverzeichnis für Compiler und Linker angegeben werden. Die Variable `TL_COMPILER` sollte Infos über Compilerversion und Architektur enthalten, z.B. "`msc192164`" für den MS Compiler 19.21 64-Bit. Mit dieser Variablen wird im Zielverzeichnis ein Unterverzeichnis angelegt. Die Variable `TL_RELEASE` steuert, ob eine Debug- oder eine Release-Version kompiliert wird.

5.1.4 Performance-Tests

Mit Hilfe der Präzisionszeit und der Heapinformationen kann die Performance einzelner Klassen und Templates genau gemessen werden. Zum Erreichen einer bestmöglichen Rechengeschwindigkeit sollten Bausteine eingesetzt werden, die mit dem globalen Chainstore arbeiten, z. B. `ct_Chn_String` oder `gct_Chn_Array`. Eine bestmögliche Speicherauslastung wird mit Array- und Blocklisten-Containern erzielt. Belegen alle Nodes einer Blockliste zusammengenommen nicht mehr als 64 KB, wird das Template `gct_Chn16BlockDList` empfohlen.

In der Bibliothek **Spirick Tuning** werden zahlreiche Parameter und Zwischenergebnisse mit `ASSERT`-Makros überprüft. Diese Prüfungen befinden sich auch an rechenzeitkritischen Stellen. Deshalb sollte zum Performance-Test statt einer Debug- eine Release-Version verwendet werden. Dazu ist ein Compilerschalter zu setzen oder das Makro `NDEBUG` zu definieren.

5.1.5 Inline-Methoden

Beim Expandieren von Inline-Methoden der Bibliothek **Spirick Tuning** ist zu beachten, daß Inline-Methoden z. T. sehr tief geschachtelt sind. Die Schachtelung ist eine Folge des schichtweisen Aufbaus der zahlreichen Schnittstellen. Die Standardeinstellung der meisten Compiler für die Schachtelungstiefe wird dabei häufig überschritten. Deshalb enthält die Datei '**tuning/defs.hpp**' folgende Präprozessoranweisung:

```
#pragma inline_depth (32)
```

5.1.6 DLL's

Alle globalen Funktionen und Klassen sind für den Einsatz in DLL's vorgesehen. Sie wurden mit dem Makro `TL_EXPORT` versehen. Sollen die Deklarationen exportiert werden, ist global das Makro `TL_BUILD_DLL` zu setzen.

5.1.7 Globale Objekte

Die Bibliothek **Spirick Tuning** enthält einige globale Storeobjekte (je eine globale Instanz des Standard-, Round- und Chainstores). Sie werden von zahlreichen anderen Klassen und Templates, z. B. `ct_Chn32String`, direkt oder indirekt verwendet. Obwohl die Reihenfolge der Initialisierung globaler Objekte nicht standardisiert ist, können globale Anwenderobjekte sicher auf globale Stores zugreifen. Globale Stores werden automatisch erzeugt, wenn zum ersten mal darauf zugegriffen wird oder wenn der erste Thread gestartet wird (siehe Abschnitt 'Globale Stores').

Globale Anwenderobjekte können auch in ihrem Destruktor (am Programmende) sicher auf globale Stores zugreifen, da diese zwar automatisch erzeugt, aber nicht zerstört werden. Das bedeutet jedoch, daß ein Heapwalker die globalen Storeobjekte am Programmende als Memoryleak melden kann. Dieser Effekt

läßt sich nur vermeiden, indem die vordefinierten globalen Storeobjekte nicht verwendet oder manuell gelöscht werden. Je nach Bedarf kann auch ein eigener Mechanismus (z. B. mit Referenzzählern) implementiert werden.

In der Datei '**tuning/sys/cprocess.cpp**' werden auf ähnliche Weise zwei globale Mutexobjekte bei der ersten Verwendung oder beim Starten des ersten Threads automatisch erzeugt, aber nicht automatisch zerstört. Auch diese Objekte können am Programmende manuell gelöscht werden, wenn sichergestellt ist, daß sie nicht mehr verwendet werden (z. B. im Destruktor eines globalen Anwenderobjekts).

5.1.8 Multithreading

Bei der Entwicklung der Klassenbibliothek **Spirick Tuning** wurde auf Sicherheit beim Multithreading geachtet. Weder Funktionen noch Klassen enthalten lokale statische Variable. Globale Variable im Filescope sind selten und sind entweder konstant (z. B. Umrechnungstabellen) oder schützen sich selbst (Reservespeicher, Roundstore, Chainstore).

In einer singlethreaded Umgebung können die Funktionen `tl_EnterCriticalSection` und `tl_LeaveCriticalSection` zwar aufgerufen werden, bleiben aber ohne Wirkung. Die Funktionen `tl_BeginThread` und `tl_EndThread` stehen singlethreaded nicht zur Verfügung.

5.1.9 Exception Handling

C++ Exceptions sind ein allgemeiner Mechanismus zur Fehlerbehandlung. Sie besitzen jedoch nicht nur Vorteile. Nach dem Auslösen einer Exception werden zwar die Destruktoren aller auf dem Stack befindlichen Objekte aufgerufen. Damit bleibt die Konsistenz der Daten gewahrt. Der Compiler muß jedoch zur Laufzeit über alle fertig konstruierten Objekte Buch führen. Damit verlangsamt sich die Geschwindigkeit.

Da sich C++ Exceptions mit einer Compileroption ausschalten lassen und die Bibliothek **Spirick Tuning** auf eine bestmögliche Geschwindigkeit ausgelegt ist, verwendet sie selbst keine Exceptions. Sie kann jedoch in Umgebungen mit oder ohne Exception Handling eingesetzt werden. Die Containerklassen verbleiben in einem konsistenten Zustand, auch wenn im Konstruktor oder Destruktor eines enthaltenen Objekts eine Exception auftritt.

5.2 Beispielprogramme

Im Lieferumfang der Bibliothek **Spirick Tuning** befinden sich einige Beispielprogramme. Sie sind kommandozeilenorientiert und übertragen textuelle Informationen zur Standardausgabe. In jeder der folgenden Dateien (außer `int.cpp`) befindet sich eine `main`-Funktion.

5.2.1 Protokollklasse (`samples/int.cpp`)

Die Klasse `ct_Int` protokolliert Konstruktoren und Destruktoren zur Standardausgabe und wird zum Testen von Containern und Collections verwendet. `ct_Int` enthält einen Wert des Typs `int`. Er wird indirekt in einem dynamisch erzeugten `int`-Objekt gespeichert. Wird 'vergessen', den Destruktor eines `ct_Int`-Objekts aufzurufen, wird das dynamische `int`-Objekt nicht freigegeben. Dieser Fehler tritt bei einer Heapanalyse zutage.

Klassendeklaration

```
class ct_Int: public ct_Object
{
    int *          pi_Value;
public:
                ct_Int ();
                ct_Int (int i);
                ct_Int (const ct_Int & co_init);
                ~ct_Int ();

    virtual bool  operator < (const ct_Object & co_comp) const;
    virtual int   GetHashCode () const;

    ct_Int &      operator = (int i);
    ct_Int &      operator = (const ct_Int & co_asgn);
    bool         operator == (const ct_Int & co_comp) const;
    bool         operator < (const ct_Int & co_comp) const;
    int          GetValue () const;
};
```

5.2.2 Speicherüberlauf (`samples/talloc.cpp`)

Das Beispielprogramm `TALloc` demonstriert die Verwendung der Techniken zur Erkennung und Behandlung eines Speicherüberlaufs. Die Funktionen `MyReserveHandler` und `MyOverflowHandler` dienen als Reservehandler und Overflowhandler.

```
void MyReserveHandler ()
{
    printf ("ReserveHandler HasReserve = %d ReserveSize = %d\n",
        tl_HasReserve (), tl_GetReserveSize ());
}

void MyOverflowHandler ()
{
    printf ("OverflowHandler\n");
}
```

Am Beginn der `main`-Funktion werden die beiden Handler registriert und die Größe des Reservespeichers auf 1 MB festgelegt. Das erfolgreiche Setzen des Reservespeichers muß vom Reservehandler protokolliert werden.

```
void main ()
{
    tl_SetReserveHandler (MyReserveHandler);
    tl_SetOverflowHandler (MyOverflowHandler);
    tl_SetReserveSize (1024 * 1024);
```

Anschließend wird in einer Schleife fortlaufend 1 MB Speicher angefordert.

```
for (unsigned u = 0; u < tl_MaxAlloc () / (1024 * 1024); u++)
{
    printf ("%d\n", u);
    tl_Alloc (1024 * 1024);
}
```

Wird das Ende des verfügbaren virtuellen Speichers erreicht, liefert die C-Standardfunktion `malloc` den Wert Null. Dann gibt die Bibliothek **Spirick Tuning** den Reservespeicher frei und versucht die Speicheranforderung erneut. Das Freigeben des Reservespeichers muß vom Reservehandler protokolliert werden. Da der Reservespeicher dieselbe Größe wie die zyklische Speicheranforderung besitzt, ist ein Schleifendurchlauf später der Speicher endgültig verbraucht. Dann wird der Overflowhandler aufgerufen und das Programm beendet.

Das Beispielprogramm `TAlloc` kann z. B. folgende Ausgabe erzeugen:

```
ReserveHandler  HasReserve = 1  ReserveSize = 1048576
0
1
2
....
95
96
ReserveHandler  HasReserve = 0  ReserveSize = 1048576
97
OverflowHandler
```

5.2.3 Alignment (samples/talign.cpp)

Das kleine Beispielprogramm `TAlign` gibt die Größe der Datentypen `t_RefCount`, `ct_RefCount` und `t_ChnInfo` aus. Die Größe dieser Datentypen beeinflusst das Alignment im Chainstore und den verschiedenen Refstores. Unabhängig vom gewählten Speichermodell (32 Bit oder 64 Bit) sollte folgende Ausgabe erscheinen:

```
sizeof (t_RefCount)  = 4
sizeof (ct_RefCount) = 8
sizeof (t_ChnInfo)   = 8
```

5.2.4 Globale Stores (samples/tstore.cpp)

Im Beispielprogramm `TStore` wird jeweils ein globaler Store einem Härtestest unterworfen. Relevante Testergebnisse sind die Gesamtrechenzeit und der Speicherbedarf des Heaps. Am Beginn der Datei kann mit Präprozessoranweisungen ein globaler Store ausgesucht werden. Zum Testen des Standardstores dienen die folgenden Zeilen.

```
#include "tuning/std/store.hpp"
#define GetStore GetStdStore
```

Die Funktionen `PrintLong` und `HeapInfo` dienen dem Protokollieren des Heaps. Am Beginn der `main`-Funktion wird ein Array mit untypisierten Zeigern angelegt. Anschließend wird der aktuelle Zustand des Heaps ausgegeben und der Wert des Präzisionstimers gespeichert. Im Härtestest wird jedem Zeiger des Arrays Speicher zugewiesen. Die Größe des Speichers liegt zufällig zwischen 10 und 110 Bytes und wird mehrmals geändert. Am Ende wird der Speicher wieder freigegeben.

Nach dem Härtestest werden die verbrauchte Zeit und der Zustand des Heaps protokolliert. Nach dem Freigeben ungenutzten Speichers wird noch einmal der Zustand des Heaps ausgegeben. Beim Aufruf dieses Beispielprogramms ist zu beachten, daß der Compiler u. U. keine Informationen über die Freiliste liefert. Die folgenden Ausgaben wurden auf einem Testsystem erzielt.

Standardstore:	Roundstore:	Chainstore:
Heap info AllocEntries 0.000.012 FreeEntries 0.000.001 AllocSize 0.001.960 FreeSize 0.063.528 HeapSize 0.065.488 Start Ready 181 ms	Heap info AllocEntries 0.000.012 FreeEntries 0.000.001 AllocSize 0.001.960 FreeSize 0.063.528 HeapSize 0.065.488 Start Ready 135 ms	Heap info AllocEntries 0.000.012 FreeEntries 0.000.001 AllocSize 0.001.960 FreeSize 0.063.528 HeapSize 0.065.488 Start Ready 47 ms
Heap info AllocEntries 0.000.013 FreeEntries 0.000.008 AllocSize 0.001.976 FreeSize 0.521.928 HeapSize 0.523.904 Free unused	Heap info AllocEntries 0.000.013 FreeEntries 0.000.010 AllocSize 0.001.976 FreeSize 0.652.904 HeapSize 0.654.880 Free unused	Heap info AllocEntries 0.005.023 FreeEntries 0.000.008 AllocSize 0.470.232 FreeSize 0.053.672 HeapSize 0.523.904 Free unused
Heap info AllocEntries 0.000.013 FreeEntries 0.000.001 AllocSize 0.001.976 FreeSize 0.063.512 HeapSize 0.065.488	Heap info AllocEntries 0.000.013 FreeEntries 0.000.001 AllocSize 0.001.976 FreeSize 0.063.512 HeapSize 0.065.488	Heap info AllocEntries 0.000.013 FreeEntries 0.000.001 AllocSize 0.002.264 FreeSize 0.063.224 HeapSize 0.065.488

5.2.5 Block (samples/tblock.cpp)

Im Beispielprogramm `TBlock` wird mit einer selbstdefinierten Blockbasisklasse die Nutzung von `Paddingbytes` demonstriert. Mit einem Teststore, der alle Anforderungen und Freigaben protokolliert, wird das Verhalten des Templates `gct_ResBlock` bei verschiedenen Minimalgrößen überprüft.

5.2.6 Block- und Packstore (samples/tblockstore.cpp)

Das Beispielprogramm `TBlockstore` prüft die Funktionsweise von Block- und Packstore. Der Blockstore wird mit einem Elementblock und einem Pageblock überprüft. Die Storeklassen auf der untersten Ebene protokollieren alle Anforderungen und Freigaben, die von den übergeordneten Block- und Packstores an sie weitergereicht werden.

Im Hauptprogramm werden nacheinander Anforderungen und Freigaben an Block- und Packstore erzeugt und protokolliert. Anhand der zugehörigen Ausgaben der Protokollstores wird ersichtlich, ob sich Block- und Packstore wie erwartet verhalten.

5.2.7 Container (samples/tcontainer.cpp)

Das Beispielprogramm TContainer demonstriert einige elementare Containeroperationen. Mit Templatefunktionen werden verschiedene Objekt- und Zeiger-Containerarten getestet. Wichtig ist, daß zu jedem Konstruktor der Klasse `ct_Int` ein Destruktor aufgerufen wird. Die Funktionen `PrintContainer` und `PrintPtrContainer` protokollieren den Inhalt eines Containers auf die Standardausgabe. Container können unterschiedliche Datentypen für Positionszeiger verwenden. Deshalb werden sie mit eigenen Templatefunktionen ausgegeben.

```
void PrintPos (t_UInt u)
{
    printf ("%d", u);
}

void PrintPos (void * p)
{
    printf ("%p", p);
}

template <class t_container>
void PrintContainer (t_container * po_cont)
{
    printf ("Container:");

    for (typename t_container::t_Position o_pos = po_cont-> First ();
        o_pos != 0;
        o_pos = po_cont-> Next (o_pos))
    {
        printf (" Entry[");
        PrintPos (o_pos);
        printf ("]=%d", po_cont-> GetObj (o_pos)-> GetValue ());
    }

    printf ("\n");
}
```

In der Testfunktion werden ein Container erzeugt und verschiedene Methoden aufgerufen. Zwischen den Containeroperationen wird mehrmals der aktuelle Inhalt des Containers ausgegeben, um das erwartete Ergebnis mit dem tatsächlichen zu vergleichen.

```
template <class t_container>
void Test ()
{
    gct_CompContainer <t_container> * po_cont = new gct_CompContainer <t_container>;
    typename t_container::t_Position o_pos1;
    ct_Int co_int (1);
    PrintContainer (po_cont);
    po_cont-> AddObjCond (& co_int);
    ...
    po_cont-> DelObj (po_cont-> First ());
    PrintContainer (po_cont);
    delete po_cont;
}
```

5.2.8 Collections (samples/tcollection.cpp)

Das Beispielprogramm TCollection demonstriert einige elementare Zeigercontainer- und Collectionoperationen. Es ist ähnlich wie TContainer aufgebaut und erzeugt eine ähnliche Ausgabe. Wichtig ist, daß zu jedem Konstruktor der Klasse `ct_Int` ein Destruktor aufgerufen wird. Mit Templatefunktionen werden je ein Zeigercontainer und eine Collection getestet.

5.2.9 [Zeiger]Mapcontainer (samples/t[ptr]map.cpp)

Die Beispielprogramme TMap und TPtrMap demonstrieren einige elementare Operationen mit Mapcontainern. Wichtig ist, daß zu jedem Konstruktor der Klasse `ct_Int` ein Destruktor aufgerufen wird und daß eingefügte Schlüssel auch gefunden werden.

Für Mapcontainer existieren keine vordefinierten Standardinstanzen. Deshalb dienen die Beispielprogramme TMap und TPtrMap auch als Vorlage für mögliche Parameter der Templates `gct_Map` und `gct_PtrMap`.

5.2.10 Zugriffsbeschleunigung (samples/taccess.cpp)

Das Beispielprogramm TAccess demonstriert die Zugriffsbeschleunigung von speziellen Containern. Mit Templatefunktionen werden je ein Array, ein sortiertes Array und eine Hashtabelle getestet. In der Testfunktion werden ein Container und einige Hilfsvariable erzeugt. Der Container wird mit zufällig erzeugten Zeichenketten der Länge 1 bis 40 gefüllt. Die für den Aufbau des Containers benötigte Zeit wird protokolliert. Anschließend wird jede einzelne Zeichenkette mit der Methode `SearchFirstObj` im Container gesucht. Die gesamte für das Suchen benötigte Zeit wird protokolliert. Das Programm kann für die verschiedenen Containerarten folgende Ausgaben erzeugen:

Array:	Sortiertes Array:	Hashtabelle:
Begin construction . . .	Begin construction . . .	Begin construction . . .
Ready: 89 ms	Ready: 1224 ms	Ready: 111 ms
Begin searching . . .	Begin searching . . .	Begin searching . . .
Ready: 49667 ms	Ready: 215 ms	Ready: 94 ms

5.2.11 Exceptions in Containern (samples/texception.cpp)

Das Beispielprogramm TException demonstriert das Verhalten von Containern beim Auftreten von Exceptions im Konstruktor oder Destruktor enthaltener Objekte. Relevante Testergebnisse sind das Zerstören vollständig konstruierter Objekte und das Erhalten der Konsistenz im Container. Zum Testen der Container wird die Klasse `ct_Throw` verwendet.

```
bool b_Throw = true;

class ct_Throw
{
    int i;
public:
    static int      i_Num;
    static int      i_Throw;

    ct_Throw ();
    ct_Throw (const ct_Throw &);
    ~ct_Throw ();
    ct_Throw &      operator = (const ct_Throw &);

    int             GetHashCode () const { return i_Num; }
};
```

Alle Methoden der Klasse übertragen eine Meldung auf die Standardausgabe und lösen unter bestimmten Bedingungen eine Exception aus. Als Beispiel folgt der Default-Konstruktor.

```
ct_Throw::ct_Throw ()
{
    printf ("%2d ct_Throw ()\n", ++ i_Num);
```

```

    if ((b_Throw) || (i_Num == i_Throw))
    {
        i_Num--;
        throw 1;
    }
}

```

Von mehreren Containerarten werden Instanzen erzeugt.

```

gct_Std32DList <ct_Throw> co_DList;
gct_Std32BlockDList <ct_Throw> co_BDList;
gct_Std32Array <ct_Throw> co_Array;
gct_Std32Array <ct_Throw> co_Array2;
gct_Std32HashTable <ct_Throw> co_HashTable;

```

Die Funktion TArrayConstructor testet den Konstruktor eines Arraycontainers.

```

void TArrayConstructor ()
{
    b_Throw = false;
    co_Array. AddObj ();
    co_Array. AddObj ();
    co_Array. AddObj ();
    co_Array. AddObj ();
    ct_Throw::i_Throw = ct_Throw::i_Num + 3;
    gct_Std32Array <ct_Throw> co_array2 = co_Array;
}

```

Die Funktion TArrayDestructor testet den Destruktor eines Arraycontainers.

```

void TArrayDestructor ()
{
    b_Throw = false;
    gct_Std32Array <ct_Throw> co_array;
    co_array. AddObj ();
    co_array. AddObj ();
    co_array. AddObj ();
    co_array. AddObj ();
    b_Throw = true;
}

```

In der main-Funktion werden die verschiedenen Containerarten überprüft. Für jeden einzelnen Container werden mehrere Methoden aufgerufen, die Exceptionhandler enthalten. Zum Beispiel wird die Methode AddObj eines Arraycontainers getestet.

```

void main ()
{
    ....
    try
    {
        co_Array. AddObj ();
    }
    catch (int i)
    {
        printf ("Exception %d from co_Array. AddObj ()\n", i);
        printf ("Array length %d\n", co_Array. GetLen ());
    }
}

```

Nach dem Prüfen einzelner Methoden werden Konstruktor und Destruktor des Containers getestet.

```

try
{
    TArrayConstructor ();
}
catch (int i)

```

```

{
    printf ("Exception %d from TArrayConstructor ()\n", i);
    ct_Throw:: i_Throw = 1000;
    co_Array. DeTA11 ();
}

```

Das Programm erzeugt unter anderem folgende Ausgaben:

```

1 ct_Throw ()
Exception 1 from co_Array. AddObj ()
Array length 0
....
2 ct_Throw ()
3 ct_Throw ()
4 ct_Throw ()
5 ct_Throw ()
6 ct_Throw (copy)
7 ct_Throw (copy)
8 ct_Throw (copy)
7 ~ct_Throw ()
6 ~ct_Throw ()
Exception 2 from TArrayConstructor ()
5 ~ct_Throw ()
4 ~ct_Throw ()
3 ~ct_Throw ()
2 ~ct_Throw ()

```

5.2.12 Interlocked (samples/tinterlocked.cpp)

Das Beispielprogramm TInterlocked prüft das Verhalten der Funktionen `tl_InterlockedIncrement` und `tl_InterlockedDecrement`. Es werden fünf Threads gestartet, die gleichzeitig und ohne Synchronisierung mit den Interlocked-Funktionen auf die eine Variable und mit den Operatoren ++ und -- auf eine andere Variable zugreifen. Am Ende wird erwartet, daß die Interlocked-Variable den Wert der Testkonstanten enthält und die andere Variable einen zufälligen Wert.

5.2.13 Threads (samples/tthread.cpp)

Das Beispielprogramm TThread prüft das Starten und Beenden von Threads sowie die Thread-synchronisation. Zum Protokollieren des Programmablaufs werden von der `main`-Funktion und den Threadfunktionen Informationen zur Standardausgabe übertragen. Das Programm wird an verschiedenen Stellen mit `tl_Delay` unterbrochen. Die Länge der Pausen ist so gewählt, daß nie zwei Threads gleichzeitig versuchen etwas auszugeben. Dadurch erscheinen die asynchronen Ausgaben in einer geordneten Reihenfolge.

Zu Beginn der `main`-Funktion werden drei Threads gestartet und deren Ende abgewartet. Danach werden zwei kritische Abschnitte geschachtelt. Dabei darf sich das Programm nicht selbst blockieren. Anschließend wird ein Thread gestartet, der zehnmal in einen kritischen Abschnitt eintritt und nach dem Verlassen eine Pause einlegt. In der Mitte dieser Schleife tritt der Hauptthread in einen kritischen Abschnitt ein und wartet eine Sekunde. In dieser Zeit dürfen vom zweiten Thread keine Ausgaben erscheinen. Anschließend wird die Schleife fortgesetzt. Am Ende der `main`-Funktion werden dieselben Tests mit kritischen Abschnitten für Prozeßsynchronisation durchgeführt. Dabei wird auch das versuchsweise Sperren des Mutexobjekts mit einem Timeout getestet.

5.2.14 Semaphoren (samples/tsemaphore.cpp)

Im Beispielprogramm TSemaphore wird die Funktionsweise von Thread- und Prozeß-Semaphoren geprüft. Zunächst werden die Semaphoren wie Mutexobjekte verwendet. Die Ausgaben erscheinen ähnlich wie vom Programm TThread. Anschließend wird mit einem Semaphor eine einfache Message-Queue getestet.

5.2.15 Prozesse (samples/texec.cpp)

Das Beispielprogramm TExec prüft das Verhalten der Funktionen `tl_Exec` und `tl_IsProcessRunning`. Der zweite Prozeß wird einmal asynchron und einmal synchron gestartet. Nach dem asynchronen Aufruf wartet das Hauptprogramm mit `tl_IsProcessRunning`, bis der zweite Prozeß beendet ist. Nach dem synchronen Aufruf gibt das Hauptprogramm den Rückgabecode aus.

Es werden auch die verschiedenen Arten der Parameterübergabe geprüft. Ein Prozeßparameter ist im einfachsten Fall eine nullterminierte Zeichenkette ohne Leerzeichen und Anführungsstriche. Es kann jedoch auch eine leere Zeichenkette, ein Nullzeiger, eine Zeichenkette mit Leerzeichen und eine Zeichenkette mit Anführungsstrichen verwendet werden. Der zweite Prozeß überträgt die Parameter zur Kontrolle auf die Standardausgabe.

5.2.16 Starthilfe (samples/texechelper.cpp)

Hintergrund: Bei UNIX-ähnlichen Betriebssystemen werden neue Prozesse meist mit `fork` oder davon abgeleiteten Funktionen gestartet. Dabei kann es zu Ressourcenproblemen kommen, wenn der Prozeß mehrere Threads gestartet, mehrere Dateien geöffnet und/oder viel Arbeitsspeicher belegt hat. Diese Probleme kann man umgehen, indem man relativ früh in der Startphase des Prozesses einen Hilfsprozeß startet, der nur dazu dient, weitere Prozesse zu starten. Haupt- und Hilfsprozeß kommunizieren über zwei Semaphoren und Sharedmemory miteinander.

Das Beispielprogramm TExecHelper enthält dieselben Schritte wie TExec. Statt der Funktion `tl_Exec` wird aber die Klasse `ct_ExecHelper` verwendet.

5.2.17 Gemeinsame Ressourcen (samples/tshared.cpp)

Das Beispielprogramm TShared prüft das Verhalten der gemeinsamen Ressourcen Mutex und Sharedmemory. Für jeden der beiden Tests wird mit `tl_Exec` ein zweiter Prozeß gestartet. Im ersten Test sperrt der zweite Prozeß das gemeinsame Mutexobjekt zehnmal in einer Schleife. Nach der Freigabe wird jeweils eine Pause eingelegt. In der Mitte dieser Schleife sperrt der Hauptprozeß das Mutexobjekt und wartet eine Sekunde. In dieser Zeit dürfen vom zweiten Prozeß keine Ausgaben erscheinen. Anschließend wird die Schleife fortgesetzt. Im zweiten Test wird der Zugriff auf den gemeinsamen Speicher geprüft und protokolliert.

5.2.18 Zeichenketten (samples/tstring.cpp)

Das Beispielprogramm TString prüft einige elementare Operationen für Zeichenketten. Die Tests werden parallel mit einer `char`- und einer `wchar_t`-Klasse durchgeführt, z. B. `ct_String` und `ct_WString` oder `ct_Rnd_String` und `ct_Rnd_WString`. Bei der Standardausgabe eines Programms können `printf` und `wprintf` nicht gemischt verwendet werden. Deshalb wird bei `wchar_t`-Zeichenketten jedes Zeichen einzeln mit `printf` ausgegeben. Die meisten Teiltests werden mit `char` und `wchar_t` durchgeführt und protokolliert. Anschließend wird das erwartete Ergebnis als eine Stringkonstante ausgegeben, d. h. auf der Standardausgabe erscheint dreimal hintereinander dieselbe Ausgabe.

Im einzelnen werden die folgenden Tests durchgeführt: Suche nach Zeichen und Zeichenketten, Vergleich von Zeichen und Zeichenketten, Einfügen, Löschen, Ersetzen, Austauschen, temporäre Stringobjekte, Zeichenketten formatieren, Konstruktoren und Umwandeln von `char`- und `wchar_t`-Zeichenketten.

5.2.19 Zeichenketten sortieren (samples/tsort.cpp)

Das Beispielprogramm TSort demonstriert das Sortieren von Zeichenketten mit Hilfe der Klasse `ct_StringSort`. Der Sortieralgorithmus wird mit der Standardfunktion `qsort()` im Zusammenspiel mit `strcmp()` und `stricmp()` verglichen. Es werden 1 000 000 Zeichenketten zufällig erzeugt und sortiert. Beim Sortieren wird die Zeit in Millisekunden gemessen und ausgegeben. Das Programm kann folgende Ausgabe erzeugen:

```
StrCmp 937
StrICmp 1176
StringSort 212
```

Im zweiten Teil des Programms werden Zahlen mit Hilfe der Klasse `ct_UInt32Sort` sortiert. Dieselbe Sortierung wird noch einmal mit `qsort()` durchgeführt, und die Rechenergebnisse werden miteinander verglichen.

5.2.20 Dateiname (samples/tfilename.cpp)

Das Beispielprogramm TFileName demonstriert elementare Operationen der Klasse `ct_FileName`. Der Zugriff auf die einzelnen Komponenten wird überprüft. Am Beginn der `main`-Funktion wird ein Dateinamen-Objekt angelegt und mit einer Zeichenkette versehen. Alle Komponenten werden einzeln abgefragt und ausgegeben.

```
void main ()
{
    ct_FileName co_name;
    co_name.AssignAsName ("A:\\PATH\\NAME.EXT");
    printf ("\n");
    printf ("Drive       : \"%s\\\"\\n", co_name.GetDrive ().GetStr ());
    printf ("Path         : \"%s\\\"\\n", co_name.GetPath ().GetStr ());
    printf ("PurePath      : \"%s\\\"\\n", co_name.GetPurePath ().GetStr ());
    printf ("DrivePath    : \"%s\\\"\\n", co_name.GetDrivePath ().GetStr ());
    printf ("PureDrivePath: \"%s\\\"\\n", co_name.GetPureDrivePath ().GetStr ());
    printf ("Name         : \"%s\\\"\\n", co_name.GetName ().GetStr ());
    printf ("Ext          : \"%s\\\"\\n", co_name.GetExt ().GetStr ());
    printf ("NameExt      : \"%s\\\"\\n", co_name.GetNameExt ().GetStr ());
    printf ("All          : \"%s\\\"\\n", co_name.GetAllStr ());
```

Anschließend wird die Umwandlung in relative und absolute Dateinamen geprüft.

```
co_name.ToRel ("A:\\PATH\\X");
printf ("ToRel       : \"%s\\\"\\n", co_name.GetAllStr ());
co_name.ToAbs ("A:\\PATH\\X");
printf ("ToAbs       : \"%s\\\"\\n", co_name.GetAllStr ());
co_name.ToLower ();
printf ("ToLower      : \"%s\\\"\\n", co_name.GetAllStr ());
printf ("Wildc       : %d\\n", co_name.HasWildCards ());
printf ("Abs         : %d\\n", co_name.IsAbs ());
printf ("Rel         : %d\\n", co_name.IsRel ());
```

Am Ende wird eine Methode überprüft, die ein temporäres `ct_String`-Objekt liefert.

```
if (co_name.GetExt () == "ext")
    printf ("\nGetExt () == \"ext\"\n");
```

Das Programm erzeugt folgende Ausgabe:

```
Drive       : "A:"
Path        : "\\PATH\"
PurePath    : "\\PATH"
DrivePath   : "A:\\PATH\"
PureDrivePath: "A:\\PATH"
Name        : "NAME"
Ext         : "EXT"
NameExt     : "NAME.EXT"
All         : "A:\\PATH\\NAME.EXT"
ToRel       : "..\\NAME.EXT"
ToAbs       : "A:\\PATH\\NAME.EXT"
ToLower     : "a:\\path\\name.ext"
Wildc       : 0
Abs         : 1
Rel         : 0

GetExt () == "ext"
```

5.2.21 Datei (samples/tfile.cpp)

Das Beispielprogramm TFile prüft im Verzeichnis für temporäre Dateien einige elementare Operationen der Klasse `ct_File`. An zwei Stellen im Programm versucht ein zweiter Prozeß auf die Datei zuzugreifen, die im Hauptprozeß zum Lesen oder Schreiben geöffnet ist. Im einzelnen werden die folgenden Tests durchgeführt: Erzeugen, Öffnen, Schließen, Lesen, Schreiben, Positionieren, Ändern der Größe, Verschieben und Löschen.

5.2.22 Verzeichnis (samples/tdir.cpp)

Das Beispielprogramm TDir prüft im Verzeichnis für temporäre Dateien einige elementare Operationen der Klasse `ct_Directory`. Im einzelnen werden die folgenden Tests durchgeführt: Abfrage des aktuellen Verzeichnisses, Erzeugen, Verschieben und Löschen.

5.2.23 Verzeichnis durchlaufen (samples/tdirscan.cpp)

Das Beispielprogramm TDirScan demonstriert elementare Operationen der Klasse `ct_DirScan`. Der Inhalt eines Verzeichnisses wird gelesen und ähnlich dem MS-DOS-Kommando `dir` auf die Standardausgabe übertragen. Die Funktion `PrintEntry` gibt die Daten eines einzelnen Verzeichniseintrags aus. In der `main`-Funktion wird eine `ct_DirScan`-Variable angelegt und überprüft, ob das Verzeichnis existiert. In der ersten Schleife werden alle Verzeichniseinträge ungefiltert durchlaufen. In der zweiten Schleife werden nur Dateien und in der dritten Schleife nur Unterverzeichnisse ausgegeben.

5.2.24 Verzeichnisbaum (samples/ttree.cpp)

Das Beispielprogramm TTree demonstriert ähnlich wie TDirScan elementare Operationen der Klasse `ct_DirScan`. Das aktuelle Verzeichnis wird rekursiv durchlaufen. Alternativ kann auch ein anderes Verzeichnis als Kommandozeilenparameter übergeben werden. Der Verzeichnisbaum wird ähnlich dem MS-DOS-Kommando `tree` auf die Standardausgabe übertragen.

5.2.25 Uhrzeit und Datum (samples/ttimedate.cpp)

Das Beispielprogramm TTimeDate vergleicht die Genauigkeit der Systemzeit mit der Präzisionszeit. In der `main`-Funktion werden zwei Objekte der Klasse `ct_TimeDate` angelegt. In einer Schleife werden fortlaufend die aktuelle Systemzeit und die Präzisionszeit ausgegeben. Im zweiten Teil des Programms wird die Präzisionszeit in Millisekunden mit der Präzisionszeit in Mikrosekunden verglichen.

5.2.26 Systemnahe Informationen (samples/tinfo.cpp)

Das Beispielprogramm TInfo fragt der Reihe nach alle systemnahen Informationen ab, die in der Datei `'tuning/sys/cinfo.hpp'` bereit gestellt werden, und überträgt sie auf die Standardausgabe.

5.2.27 MD5 und UUID (samples/tmd5.cpp und tuuid.cpp)

Die Beispielprogramme TMD5 und TUUID enthalten kleine Testsequenzen für die Klassen `ct_MD5` und `ct_UUID`. Dabei werden die textuellen Repräsentationen der Rechenergebnisse ausgegeben.

Index

A

AbortFind.....	145
Acquire.....	111, 115
AddKeyAndValPtr.....	85
AddKeyAndValPtrCond.....	85
AddKeyAndValue.....	81
AddKeyAndValueCond.....	82
AddObj.....	48
AddObjAfter.....	48
AddObjAfterLast.....	52
AddObjAfterLastCond.....	68
AddObjAfterNth.....	52
AddObjBefore.....	48
AddObjBeforeFirst.....	52
AddObjBeforeFirstCond.....	68
AddObjBeforeNth.....	52
AddObjCond.....	68
AddPtr.....	72
AddPtrAfter.....	72
AddPtrAfterLast.....	72
AddPtrAfterLastCond.....	74
AddPtrAfterNth.....	72
AddPtrBefore.....	72
AddPtrBeforeFirst.....	72
AddPtrBeforeFirstCond.....	74
AddPtrBeforeNth.....	72
AddPtrCond.....	74
AddRefAfterLastCond.....	78
AddRefBeforeFirstCond.....	78
AddRefCond.....	78
AddrOf.....	12
AlignPageSize.....	30, 31, 34
Alles ersetzen.....	128
Alloc.....	12
AllocData.....	35, 44
AllocPtr.....	35
Anfügen.....	127
Anfügen und Löschen mehrerer Objekte.....	48
Anfügeoperatoren.....	130
Anzahl der Objekte.....	47
Append.....	127
AppendChars.....	29
AppendF.....	129
AppendObj.....	48
AppendPath.....	136
ARRAY_DCLS.....	56
Assign.....	126
AssignAsName.....	135
AssignAsPath.....	135
AssignChars.....	30
AssignF.....	129

B

Bedingtes Einfügen.....	68, 78
Bedingtes Einfügen von Zeigern.....	74
Bedingtes Löschen gefundener Objekte.....	69
Bedingtes Löschen gefundener Paare.....	82, 85

Bedingtes Löschen gefundener Paare und referenzierter Objekte.....	86
Bedingtes Löschen gefundener Zeiger.....	75
Bedingtes Löschen gefundener Zeiger und referenzierter Objekte.....	75, 79
Bedingtes Löschen von Zeigern gefundener Objekte.....	79
Before.....	60
BLOCK_DCLS.....	35
BLOCK_DLIST_DCLS.....	63
BLOCK_STORE_DCLS.....	38
BLOCKPTR_DLIST_DCLS.....	89
BLOCKREF_DLIST_DCLS.....	66
BLOCKREF_STORE_DCLS.....	42
BLOCKREFPTR_DLIST_DCLS.....	91

C

CanFreeAll.....	13
Clear.....	126, 148, 151
Close.....	113, 115, 116, 140
co_AttrArchive.....	143
co_AttrDirectory.....	143
co_AttrHidden.....	143
co_AttrReadOnly.....	143
co_AttrSystem.....	143
co_DayFactor.....	106
co_HourFactor.....	106
co_InvalidFileld.....	117
co_MicroSecondFactor.....	106
co_MilliSecondFactor.....	106
co_MinuteFactor.....	106
co_SecondFactor.....	106
COLLMAP_DCL.....	98
COLLMAP_DEF.....	98
CompressPath.....	137
CompSubStr.....	126
CompTo.....	126
ContainsKey.....	81, 84
ContainsObj.....	67
ContainsPtr.....	74
ContainsRef.....	78
Convert.....	131
Copy.....	141
CopyDriveFrom.....	136
CopyDrivePathFrom.....	136
CopyExtFrom.....	136
CopyNameExtFrom.....	136
CopyNameFrom.....	136
CopyPathFrom.....	136
CountKeys.....	81, 84
CountObjs.....	67
CountPtrs.....	74
CountRefs.....	78
Create.....	113, 115, 116, 140, 143, 151
CreateChnStore.....	20
CreateRndStore.....	17
CreateStdStore.....	16

ct_AnyBlock.....	21	ct_Rnd8Block.....	35
ct_AnyStore.....	11	ct_Rnd8BlockRefStore.....	42
ct_Array.....	98	ct_Rnd8BlockStore.....	38
ct_BlockDList.....	99	ct_Rnd8RefStore.....	42
ct_BlockRefDList.....	99	ct_Rnd8Store.....	17
ct_Chn_[W]String.....	131	ct_RndStore.....	16
ct_Chn_Block.....	35	ct_SharedMemory.....	115, 116
ct_Chn_BlockRefStore.....	42	ct_SharedResource.....	111
ct_Chn_BlockStore.....	38	ct_SortedArray.....	98
ct_Chn_RefStore.....	42	ct_Std_[W]String.....	131
ct_Chn_Store.....	20	ct_Std_Block.....	35
ct_Chn16Block.....	36	ct_Std_BlockRefStore.....	42
ct_Chn16BlockRefStore.....	42	ct_Std_BlockStore.....	38
ct_Chn16BlockStore.....	38	ct_Std_RefStore.....	41
ct_Chn16RefStore.....	42	ct_Std_Store.....	16
ct_Chn16Store.....	20	ct_Std16Block.....	35
ct_Chn32Block.....	36	ct_Std16BlockRefStore.....	42
ct_Chn32BlockRefStore.....	42	ct_Std16BlockStore.....	38
ct_Chn32BlockStore.....	38	ct_Std16RefStore.....	41
ct_Chn32RefStore.....	42	ct_Std16Store.....	16
ct_Chn32Store.....	20	ct_Std32Block.....	35
ct_Chn8Block.....	36	ct_Std32BlockRefStore.....	42
ct_Chn8BlockRefStore.....	42	ct_Std32BlockStore.....	38
ct_Chn8BlockStore.....	38	ct_Std32RefStore.....	41
ct_Chn8RefStore.....	42	ct_Std32Store.....	16
ct_Chn8Store.....	20	ct_Std8Block.....	35
ct_ChnStore.....	18	ct_Std8BlockRefStore.....	42
ct_Collection.....	94	ct_Std8BlockStore.....	38
ct_Directory.....	142	ct_Std8RefStore.....	41
ct_DirScan.....	144	ct_Std8Store.....	16
ct_DList.....	98	ct_StdStore.....	15
ct_File.....	139, 140	ct_String.....	132
ct_FileName.....	133, 134	ct_StringSort.....	138
ct_MD5.....	149	ct_ThMutex.....	109
ct_Object.....	93	ct_ThSemaphore.....	110, 111
ct_PackStore.....	44	ct_TimeDate.....	147, 148
ct_PackStoreBase.....	43	ct_UInt32Sort.....	138
ct_PageBlock.....	34	ct_UUID.....	150
ct_PageBlockBase.....	33	ct_WString.....	132
ct_PrMutex.....	112, 113	cu_HashPrime1.....	62
ct_PrSemaphore.....	114	cu_HashPrime16.....	62
ct_RefCollection.....	97	cu_HashPrime2.....	62
ct_RefCount.....	39	cu_HashPrime4.....	62
ct_RefDList.....	99	cu_HashPrime8.....	62
ct_Rnd_[W]String.....	131		
ct_Rnd_Block.....	35	D	
ct_Rnd_BlockRefStore.....	42	DecCharSize.....	29
ct_Rnd_BlockStore.....	38	DecltemSize.....	31
ct_Rnd_RefStore.....	42	DecltemSize1.....	31
ct_Rnd_Store.....	17	DecRef.....	39, 41, 65, 97
ct_Rnd16Block.....	35	DelAll.....	48
ct_Rnd16BlockRefStore.....	42	DelAllKey.....	85
ct_Rnd16BlockStore.....	38	DelAllKeyAndValue.....	82, 86
ct_Rnd16RefStore.....	42	DelAllPtr.....	73
ct_Rnd16Store.....	17	DelAllPtrAndObj.....	73
ct_Rnd32Block.....	35	Delete.....	128, 141, 143
ct_Rnd32BlockRefStore.....	42	DeleteChars.....	30
ct_Rnd32BlockStore.....	38	DeleteChnStore.....	20
ct_Rnd32RefStore.....	42	Deleteltems.....	31
ct_Rnd32Store.....	17	DeleteRev.....	128
		DeleteRndStore.....	17

DeleteStdStore.....	16	et_UtfError.....	103
DelFirstEqualObj.....	69	Exception.....	8, 34, 49
DelFirstEqualObjCond.....	69	Exists.....	141, 143
DelFirstEqualPtr.....	74	F	
DelFirstEqualPtrAndObj.....	75	FillChars.....	30
DelFirstEqualPtrAndObjCond.....	75	Finalize.....	150
DelFirstEqualPtrCond.....	75	FindFirst.....	145
DelFirstEqualRef.....	79	FindFirstDirectory.....	145
DelFirstEqualRefAndObj.....	79	FindFirstFile.....	145
DelFirstEqualRefAndObjCond.....	79	FindNext.....	145
DelFirstEqualRefCond.....	79	FindNextDirectory.....	145
DelFirstKey.....	85	FindNextFile.....	145
DelFirstKeyAndValue.....	82, 86	FindOnce.....	145
DelFirstKeyAndValueCond.....	82, 86	FindOncePath.....	145
DelFirstKeyCond.....	85	First.....	47, 105, 125
DelFirstObj.....	53	Formatierte Zeichenketten.....	129
DelFirstPtr.....	72	Found.....	145
DelFirstPtrAndObj.....	73	Free.....	12
DelKey.....	85	FreeAll.....	13, 49
DelKeyAndValue.....	82, 86	FreeData.....	35, 44
DelLastEqualObj.....	69	FreeFirstObj.....	54
DelLastEqualObjCond.....	69	FreeLastObj.....	54
DelLastEqualPtr.....	75	FreeNextObj.....	54
DelLastEqualPtrAndObj.....	75	FreeNthObj.....	54
DelLastEqualPtrAndObjCond.....	75	FreeObj.....	49
DelLastEqualPtrCond.....	75	FreePrevObj.....	54
DelLastEqualRef.....	79	FreeUnused.....	19, 38
DelLastEqualRefAndObj.....	79	FromStr.....	151
DelLastEqualRefAndObjCond.....	79	ft_ThreadFunc.....	108
DelLastEqualRefCond.....	79	G	
DelLastKey.....	85	gct_AnyContainer.....	45, 47
DelLastKeyAndValue.....	82, 86	gct_Array.....	55
DelLastKeyAndValueCond.....	82, 86	gct_Block.....	23
DelLastKeyCond.....	86	gct_BlockBase.....	22
DelLastObj.....	54	gct_BlockStore.....	36
DelLastPtr.....	73	gct_CharBlock.....	28
DelLastPtrAndObj.....	73	gct_Chn_Array.....	56
DelNextObj.....	54	gct_Chn_BlockDList.....	64
DelNextPtr.....	73	gct_Chn_BlockPtrDList.....	90
DelNextPtrAndObj.....	73	gct_Chn_BlockRefDList.....	66
DelNthObj.....	54	gct_Chn_BlockRefPtrDList.....	91
DelNthPtr.....	73	gct_Chn_DList.....	58
DelNthPtrAndObj.....	73	gct_Chn_HashTable.....	63
DelObj.....	48	gct_Chn_PtrArray.....	87
DelPrevObj.....	54	gct_Chn_PtrDList.....	88
DelPrevPtr.....	73	gct_Chn_PtrHashTable.....	89
DelPrevPtrAndObj.....	73	gct_Chn_PtrSortedArray.....	88
DelPtr.....	72	gct_Chn_RefDList.....	66
DelPtrAndObj.....	73	gct_Chn_RefPtrDList.....	90
DLIST_DCLS.....	58	gct_Chn_SortedArray.....	60
E		gct_Chn16Array.....	56
Einfügen.....	127	gct_Chn16BlockDList.....	64
Einfügen von Objekten.....	48, 52	gct_Chn16BlockPtrDList.....	90
Einfügen von Paaren.....	81, 85	gct_Chn16BlockRefDList.....	67
Einfügen von Zeigern.....	72	gct_Chn16BlockRefPtrDList.....	91
EndOfFile.....	141	gct_Chn16DList.....	58
Ersetzen.....	128	gct_Chn16HashTable.....	63
et_Compiler.....	119	gct_Chn16PtrArray.....	87
et_ResError.....	100	gct_Chn16PtrDList.....	88
et_System.....	120	gct_Chn16PtrHashTable.....	89

gct_Chn16PtrSortedArray.....	88	gct_Rnd_BlockDList.....	64
gct_Chn16RefDList.....	66	gct_Rnd_BlockPtrDList.....	90
gct_Chn16RefPtrDList.....	90	gct_Rnd_BlockRefDList.....	66
gct_Chn16SortedArray.....	61	gct_Rnd_BlockRefPtrDList.....	91
gct_Chn32Array.....	56	gct_Rnd_DList.....	58
gct_Chn32BlockDList.....	64	gct_Rnd_HashTable.....	63
gct_Chn32BlockPtrDList.....	90	gct_Rnd_PtrArray.....	87
gct_Chn32BlockRefDList.....	67	gct_Rnd_PtrDList.....	87
gct_Chn32BlockRefPtrDList.....	91	gct_Rnd_PtrHashTable.....	89
gct_Chn32DList.....	58	gct_Rnd_PtrSortedArray.....	88
gct_Chn32HashTable.....	63	gct_Rnd_RefDList.....	66
gct_Chn32PtrArray.....	87	gct_Rnd_RefPtrDList.....	90
gct_Chn32PtrDList.....	88	gct_Rnd_SortedArray.....	60
gct_Chn32PtrHashTable.....	89	gct_Rnd16Array.....	56
gct_Chn32PtrSortedArray.....	88	gct_Rnd16BlockDList.....	64
gct_Chn32RefDList.....	66	gct_Rnd16BlockPtrDList.....	90
gct_Chn32RefPtrDList.....	91	gct_Rnd16BlockRefDList.....	66
gct_Chn32SortedArray.....	61	gct_Rnd16BlockRefPtrDList.....	91
gct_Chn8Array.....	56	gct_Rnd16DList.....	58
gct_Chn8BlockDList.....	64	gct_Rnd16HashTable.....	63
gct_Chn8BlockPtrDList.....	90	gct_Rnd16PtrArray.....	87
gct_Chn8BlockRefDList.....	66	gct_Rnd16PtrDList.....	87
gct_Chn8BlockRefPtrDList.....	91	gct_Rnd16PtrHashTable.....	89
gct_Chn8DList.....	58	gct_Rnd16PtrSortedArray.....	88
gct_Chn8HashTable.....	63	gct_Rnd16RefDList.....	66
gct_Chn8PtrArray.....	87	gct_Rnd16RefPtrDList.....	90
gct_Chn8PtrDList.....	88	gct_Rnd16SortedArray.....	60
gct_Chn8PtrHashTable.....	89	gct_Rnd32Array.....	56
gct_Chn8PtrSortedArray.....	88	gct_Rnd32BlockDList.....	64
gct_Chn8RefDList.....	66	gct_Rnd32BlockPtrDList.....	90
gct_Chn8RefPtrDList.....	90	gct_Rnd32BlockRefDList.....	66
gct_Chn8SortedArray.....	60	gct_Rnd32BlockRefPtrDList.....	91
gct_CompContainer.....	67	gct_Rnd32DList.....	58
gct_DList.....	57	gct_Rnd32HashTable.....	63
gct_EmptyBaseBlock.....	23	gct_Rnd32PtrArray.....	87
gct_EmptyBaseMiniBlock.....	25	gct_Rnd32PtrDList.....	87
gct_EmptyBaseResBlock.....	27	gct_Rnd32PtrHashTable.....	89
gct_ExtContainer.....	51	gct_Rnd32PtrSortedArray.....	88
gct_FixBlock.....	27	gct_Rnd32RefDList.....	66
gct_FixItemArray.....	56	gct_Rnd32RefPtrDList.....	90
gct_FixItemBlock.....	32	gct_Rnd32SortedArray.....	60
gct_FixItemSortedArray.....	60	gct_Rnd8Array.....	56
gct_HashTable.....	61	gct_Rnd8BlockDList.....	64
gct_ItemBlock.....	30	gct_Rnd8BlockPtrDList.....	90
gct_Key.....	80, 83	gct_Rnd8BlockRefDList.....	66
gct_Map.....	80	gct_Rnd8BlockRefPtrDList.....	91
gct_MiniBlock.....	24	gct_Rnd8DList.....	58
gct_MiniBlockBase.....	24	gct_Rnd8HashTable.....	63
gct_NullDataBlock.....	28	gct_Rnd8PtrArray.....	87
gct_ObjectBaseBlock.....	23	gct_Rnd8PtrDList.....	87
gct_ObjectBaseMiniBlock.....	25	gct_Rnd8PtrHashTable.....	89
gct_ObjectBaseResBlock.....	27	gct_Rnd8PtrSortedArray.....	88
gct_PtrCompContainer.....	77	gct_Rnd8RefDList.....	66
gct_PtrContainer.....	70	gct_Rnd8RefPtrDList.....	90
gct_PtrMap.....	83	gct_Rnd8SortedArray.....	60
gct_RefDList.....	64	gct_SortedArray.....	59
gct_RefStore.....	40	gct_Std_Array.....	56
gct_ResBlock.....	26	gct_Std_BlockDList.....	64
gct_ResBlockBase.....	26	gct_Std_BlockPtrDList.....	90
gct_Rnd_Array.....	56	gct_Std_BlockRefPtrDList.....	66

gct_Std_BlockRefPtrDList.....	91	GetByteSize.....	22
gct_Std_DList.....	58	GetChar.....	105, 125
gct_Std_HashTable.....	63	GetCharAddr.....	29
gct_Std_PtrArray.....	87	GetCharPos.....	105
gct_Std_PtrDList.....	87	GetCharSize.....	29
gct_Std_PtrHashTable.....	89	GetChnStore.....	20
gct_Std_PtrSortedArray.....	88	GetCreationTime.....	146
gct_Std_RefDList.....	66	GetData.....	116
gct_Std_RefPtrDList.....	90	GetDay.....	149
gct_Std_SortedArray.....	60	GetDayOfWeek.....	149
gct_Std16Array.....	56	GetDefaultPageSize.....	30, 31, 34
gct_Std16BlockDList.....	64	GetDotLen.....	135
gct_Std16BlockPtrDList.....	90	GetDrive.....	136
gct_Std16BlockRefDList.....	66	GetDriveLen.....	135
gct_Std16BlockRefPtrDList.....	91	GetDriveOffs.....	135
gct_Std16DList.....	58	GetDrivePath.....	136
gct_Std16HashTable.....	63	GetDrivePathLen.....	135
gct_Std16PtrArray.....	87	GetDriveStr.....	136
gct_Std16PtrDList.....	87	GetEntries.....	19
gct_Std16PtrHashTable.....	89	GetError.....	105
gct_Std16PtrSortedArray.....	88	GetExt.....	136
gct_Std16RefDList.....	66	GetExtLen.....	135
gct_Std16RefPtrDList.....	90	GetExtOffs.....	135
gct_Std16SortedArray.....	60	GetExtStr.....	136
gct_Std32Array.....	56	GetFirstEqualObj.....	68
gct_Std32BlockDList.....	64	GetFirstEqualRef.....	78
gct_Std32BlockPtrDList.....	90	GetFirstObj.....	52
gct_Std32BlockRefDList.....	66	GetFirstPtr.....	71
gct_Std32BlockRefPtrDList.....	91	GetFirstValPtr.....	84
gct_Std32DList.....	58	GetFirstValue.....	81
gct_Std32HashTable.....	63	GetFixPagePtrs.....	34
gct_Std32PtrArray.....	87	GetFixSize.....	31
gct_Std32PtrDList.....	87	GetHash.....	94, 124, 151
gct_Std32PtrHashTable.....	89	GetHashSize.....	62
gct_Std32PtrSortedArray.....	88	GetHour.....	149
gct_Std32RefDList.....	66	GetInitSuccess.....	109, 111, 112
gct_Std32RefPtrDList.....	90	GetItemAddr.....	31
gct_Std32SortedArray.....	60	GetItemSize.....	31
gct_Std8Array.....	56	GetKey.....	81, 84, 112
gct_Std8BlockDList.....	64	GetLastAccessTime.....	146
gct_Std8BlockPtrDList.....	90	GetLastEqualObj.....	68
gct_Std8BlockRefDList.....	66	GetLastEqualRef.....	78
gct_Std8BlockRefPtrDList.....	91	GetLastObj.....	52
gct_Std8DList.....	58	GetLastPtr.....	71
gct_Std8HashTable.....	63	GetLastValPtr.....	85
gct_Std8PtrArray.....	87	GetLastValue.....	81
gct_Std8PtrDList.....	87	GetLastWriteTime.....	146
gct_Std8PtrHashTable.....	89	GetLen.....	47, 124
gct_Std8PtrSortedArray.....	88	GetMaxByteSize.....	22
gct_Std8RefDList.....	66	GetMaxChainExp.....	19
gct_Std8RefPtrDList.....	90	GetMaxCharSize.....	29
gct_Std8SortedArray.....	60	GetMaxItemSize.....	31
gct_String.....	121	GetMaxLen.....	55, 59, 124
gct_UtfCit.....	104	GetMicroSecond.....	149
gct_VarItemBlock.....	32	GetMinByteSize.....	26
GetAddr.....	22	GetMinSize.....	17
GetAllLen.....	135	GetMinute.....	149
GetAllocByteSize.....	26	GetMonth.....	149
GetAllStr.....	136	GetName.....	136
GetAttributes.....	146	GetNameExt.....	136

GetNameExtLen.....	135	HasWildCards.....	135
GetNameLen.....	135	I	
GetNameOffs.....	135	IncCharSize.....	29
GetNameStr.....	136	IncltItemSize.....	31
GetNewFirstObj.....	53	IncltItemSize1.....	31
GetNewLastObj.....	53	IncRef.....	39, 41, 65, 97
GetNewObj.....	53	Init.....	44
GetNewObjAfter.....	53	Initialize.....	39
GetNewObjAfterNth.....	53	Insert.....	127
GetNewObjBefore.....	53	InsertChars.....	29
GetNewObjBeforeNth.....	53	InsertDrivePath.....	136
GetNextObj.....	52	InsertF.....	129
GetNextPtr.....	71	InsertItems.....	31
GetNthObj.....	52	InsertPath.....	136
GetNthPtr.....	72	IsAbs.....	137
GetObj.....	48	IsAlloc.....	39, 41, 65, 97
GetPageSize.....	34	IsArchive.....	146
GetPath.....	136	IsDirectory.....	146
GetPathLen.....	135	IsEmpty.....	47, 124, 151
GetPathOffs.....	135	IsFree.....	39, 41, 65, 98
GetPathStr.....	136	IsHidden.....	146
GetPrevObj.....	52	IsNull.....	39
GetPrevPtr.....	72	IsReadOnly.....	146
GetPtr.....	71	IsRel.....	137
GetPureDrivePath.....	136	IsSystem.....	146
GetPureDrivePathLen.....	135	Iterieren des Containers.....	47
GetPurePath.....	136	Iterieren und verändern.....	50, 76, 96
GetPurePathLen.....	135	K	
GetRawAddr.....	29	Klein-/Großbuchstaben.....	129
GetRawLen.....	105	Konvertieren.....	130
GetRawPos.....	105	L	
GetRef.....	39, 41, 65, 97	Last.....	47, 125
GetResult.....	150	LastIdx.....	37
GetResultStr.....	150	LastPageError.....	34
GetRevChar.....	125	LastPageWarning.....	34
GetRndStore.....	17	Load.....	140
GetRoundedSize.....	34	Lock.....	109, 113
GetSecond.....	149	Löschen.....	128
GetSize.....	19, 116, 146	Löschen gefundener Objekte.....	69
GetStdStore.....	16	Löschen gefundener Paare.....	82, 85
GetStepDiv.....	17	Löschen gefundener Paare und referenzierter Objekte.....	86
GetStore.....	41	Löschen gefundener Zeiger.....	74
GetStr.....	124	Löschen gefundener Zeiger und referenzierter Objekte.....	75, 79
GetTime.....	148	Löschen von Objekten.....	48, 53
GetUUID.....	151	Löschen von Paaren.....	82, 85
GetValPtr.....	84	Löschen von Paaren und referenzierten Objekten.....	86
GetValue.....	81	Löschen von Zeigern.....	72
GetYear.....	149	Löschen von Zeigern gefundener Objekte....	79
GLOBAL_STORE_DCLS.....	14	Löschen von Zeigern und referenzierten Objekten.....	73
GLOBAL_STORE_DEFS.....	14	M	
H		MaxAlloc.....	12
HasDot.....	135	MaxDataAlloc.....	44
HasDrive.....	135	MbConvert.....	131
HasDriveOrUNC.....	135	Move.....	141, 143
HasExt.....	135	MS Visual C + +	9
HasFree.....	37	N	
HASHTABLE_DCLS.....	62		
HasName.....	135		
HasPath.....	135		
HasUNC.....	135		

	Next.....	47, 105
	Nth.....	47
O		
	Open.....	113, 115, 116, 140
	operator !=	129, 149, 151
	operator ().....	124, 125
	operator [].....	125
	operator +	130
	operator + =	130
	operator <	94, 129, 132, 149
	operator < =	129, 149
	operator =	21, 47, 130, 135, 140, 142, 144, 151
	operator = =	129, 149, 150, 151
	operator >	129, 149
	operator > =	129, 149
	operator delete.....	20
	operator delete [].....	20
	operator new.....	20
	operator new [].....	20
P		
	Parameterarten für Verzeichnisse.....	146
	PosOf.....	13
	Prev.....	47
	PTR_ARRAY_DCLS.....	86
	PTR_DLIST_DCLS.....	87
	PTR_HASHTABLE_DCLS.....	88
	PTR_SORTEDARRAY_DCLS.....	88
Q		
	QueryAllocEntries.....	19
	QueryAllocSize.....	19
	QueryCurrentDirectory.....	142
	QueryCurrentDrive.....	142
	QueryCurrentDriveDirectory.....	142
	QueryFreeEntries.....	19
	QueryFreeSize.....	19
	QueryLocalTime.....	148
	QueryPos.....	141
	QuerySize.....	141
	QueryUTCTime.....	148
R		
	Read.....	141
	Ready.....	105
	Realloc.....	12
	ReallocPtr.....	35, 44
	REF_DLIST_DCLS.....	65
	REF_STORE_DCLS.....	41
	REFCOLLMAP_DCL.....	98
	REFCOLLMAP_DEF.....	98
	REFPTR_DLIST_DCLS.....	90
	Release.....	111, 115
	Replace.....	128
	ReplaceAll.....	128
	ReplaceChars.....	30
	ReplaceF.....	129
	RevSubStr.....	125
	RoundedSizeOf.....	13
	Rückgabewert von Löschmethoden....	48, 53, 68, 72, 78, 82, 85
	Rückwärts iterieren.....	50, 76, 96

S		
	Save.....	141
	SearchFirstKey.....	81, 84
	SearchFirstObj.....	68
	SearchFirstPtr.....	74
	SearchFirstRef.....	78
	SearchLastKey.....	81, 84
	SearchLastObj.....	68
	SearchLastPtr.....	74
	SearchLastRef.....	78
	SearchNextKey.....	81, 84
	SearchNextObj.....	68
	SearchNextPtr.....	74
	SearchNextRef.....	78
	SearchPrevKey.....	81, 84
	SearchPrevObj.....	68
	SearchPrevPtr.....	74
	SearchPrevRef.....	78
	SeekAbs.....	141
	SeekRel.....	141
	Selbstzuweisung.....	123
	SetAlloc.....	39
	SetByteSize.....	22
	SetCharSize.....	29
	SetDay.....	149
	SetDayOfWeek.....	149
	SetDrive.....	136
	SetDrivePath.....	136
	SetExt.....	136
	SetFixPagePtrs.....	34
	SetFree.....	39
	SetHashSize.....	62
	SetHour.....	149
	SetItemSize.....	31
	SetKey.....	112
	SetMaxChainExp.....	19
	SetMicroSecond.....	149
	SetMinByteSize.....	26
	SetMinSize.....	17
	SetMinute.....	149
	SetMonth.....	149
	SetName.....	136
	SetNameExt.....	136
	SetPageSize.....	37, 55, 60
	SetPath.....	136
	SetSecond.....	149
	SetSortedFree.....	37
	SetStepDiv.....	17
	SetTime.....	148
	SetYear.....	149
	SizeOf.....	13
	Sort.....	138, 139
	SORTEDARRAY_DCLS.....	60
	Speicherüberlauf.....	7
	st_BatteryInfo.....	120
	st_CompilerInfo.....	119
	st_FileSystemInfo.....	119
	st_HardwareInfo.....	119
	st_HeapInfo.....	9
	st_ProcessMemoryInfo.....	119

st_SystemInfo.....	120	tl_FillMemory.....	10
st_UserKernelTime.....	107	tl_FirstChar.....	10
StoreInfoSize.....	12	tl_FirstMemory.....	10
STRING_DCL.....	131	tl_Free.....	9
SubStr.....	125	tl_FreeReserve.....	8
Suche nach Objekten.....	67	tl_FreeUnused.....	10
Suche nach Paaren.....	81, 84	tl_GetEnv.....	108
Suche nach referenzierten Objekten.....	78	tl_GetReserveSize.....	8
Suche nach Zeichen und Teilzeichenketten		tl_GetTempPath.....	108
.....	125	tl_HasReserve.....	8
Suche nach Zeigern.....	74	tl_InterlockedAdd.....	107
Swap.....	12, 21, 47	tl_InterlockedDecrement.....	107
T		tl_InterlockedIncrement.....	107
t_FileAttributes.....	143	tl_InterlockedRead.....	107
t_FileId.....	117	tl_InterlockedWrite.....	107
t_FileSize.....	117	tl_IsProcessRunning.....	109
t_Int.....	7	tl_LastChar.....	11
t_Int16.....	7	tl_LastMemory.....	11
t_Int32.....	7	tl_LeaveCriticalPrSection.....	114
t_Int8.....	7	tl_LeaveCriticalSection.....	110
t_Key.....	80, 84	tl_LocalToUTCTime.....	106
t_Length.....	46	tl_MaxAlloc.....	9
t_MD5Result.....	149	tl_MbConvert.....	102
t_MicroTime.....	106	tl_MbConvertCount.....	102
t_Object.....	47	tl_MoveDirectory.....	118
t_Position.....	12, 46	tl_MoveFile.....	117
t_RefCount.....	39	tl_MoveMemory.....	10
t_RefObject.....	71	tl_OpenFile.....	117
t_Size.....	12, 21, 124	tl_ProcessId.....	108
t_UInt.....	7	tl_QueryBatteryInfo.....	121
t_UInt16.....	7	tl_QueryCompilerInfo.....	120
t_UInt32.....	7	tl_QueryCurrentDirectory.....	118
t_UInt8.....	7	tl_QueryFileSystemInfo.....	120
t_UUID.....	150	tl_QueryHardwareInfo.....	120
t_Value.....	80, 84	tl_QueryHeapInfo.....	9
Teilvergleich.....	126	tl_QueryLocalTime.....	106
Temporäres Anfügen.....	130	tl_QueryPos.....	117
tl_Alloc.....	9	tl_QueryPrecisionTime.....	106
tl_AllocReserve.....	8	tl_QueryProcessMemoryInfo.....	120
tl_BeginThread.....	108	tl_QueryProcessTimes.....	107
tl_CloseFile.....	117	tl_QuerySize.....	117
tl_CompareChar.....	10	tl_QuerySystemInfo.....	121
tl_CompareMemory.....	10	tl_QueryThreadTimes.....	107
tl_CopyFile.....	117	tl_QueryUTCTime.....	106
tl_CopyMemory.....	10	tl_Read.....	118
tl_CreateDirectory.....	118	tl_Realloc.....	9
tl_CreateFile.....	117	tl_RelinquishTimeSlice.....	108
tl_CriticalPrSectionInitSuccess.....	113	tl_SeekAbs.....	117
tl_CriticalSectionInitSuccess.....	110	tl_SeekRel.....	117
tl_Delay.....	108	tl_SetOverflowHandler.....	8, 34
tl_DeleteCriticalPrSection.....	113	tl_SetReserveHandler.....	8
tl_DeleteCriticalSection.....	110	tl_SetReserveSize.....	8
tl_DeleteDirectory.....	118	tl_StoreInfoSize.....	9
tl_DeleteFile.....	117	tl_StringHash.....	102
tl_EndProcess.....	108	tl_StringLength.....	102
tl_EndThread.....	108	tl_SwapMemory.....	11
tl_EnterCriticalPrSection.....	114	tl_SwapObj.....	11
tl_EnterCriticalSection.....	110	tl_ThreadId.....	108
tl_Exec.....	109	tl_ToLower.....	102
tl_ExistsFile.....	117	tl_ToLower2.....	102

tl_ToUpper.....	102	Verzeichnis durchlaufen, nur Dateien.....	147
tl_ToUpper2.....	102	Verzeichnis durchlaufen, nur	
tl_Truncate.....	117	Unterverzeichnisse.....	147
tl_TryEnterCriticalPrSection.....	114	Verzeichnis vollständig durchlaufen.....	147
tl_TryEnterCriticalSection.....	110	Vollständiger Vergleich.....	126
tl_UTCToLocalTime.....	106	Vorwärts iterieren.....	50, 76, 96
tl_UtfConvert.....	104	W	
tl_UtfConvertCount.....	104	Write.....	141
tl_UtfLength.....	104	WSTRING_DCL.....	131
tl_UtfToLower.....	104	Z	
tl_UtfToUpper.....	104	Zugriff auf gefundene Objekte..	68, 78, 81, 84
tl_VSprintf.....	137	Zugriff auf Länge und Zeichenkette.....	124
tl_Write.....	118	Zugriff auf neue Objekte.....	53
ToAbs.....	137	Zugriff auf Objekte.....	48, 52
ToLower.....	129	Zugriff auf referenzierte Objekte.....	71
ToLower2.....	129	Zugriff auf Schlüssel und Wert.....	81, 84
ToRel.....	137	Zuweisen.....	126
ToStr.....	151	Zuweisungsoperatoren.....	130
ToUpper.....	129	~	
ToUpper2.....	129	~ ct_AnyBlock.....	21
tpf_AllocHandler.....	8	~ ct_DirScan.....	144
Truncate.....	141	~ ct_File.....	140
TruncateObj.....	48	~ ct_Object.....	93
TryAcquire.....	111, 115	~ ct_PackStore.....	44
TryLock.....	109, 113	~ ct_PageBlock.....	35
TryOpen.....	140	~ ct_PrMutex.....	113
U		~ ct_PrSemaphore.....	114
Unlock.....	109, 113	~ ct_SharedMemory.....	116
Update.....	150	~ ct_SharedResource.....	112
V		~ gct_AnyContainer.....	47
Vergleich im Zeigercontainer.....	73	~ gct_PtrContainer.....	71
Vergleichsoperatoren.....	129		