1 Motivation

Oft steht der Programmierer vor Problemen der folgenden Art:

- ein Programm benötigt für seine Arbeit eine Menge von Werten (Zahlen, Dateinamen, etc.), die nicht bei jedem Programmlauf manuell neu eingegeben werden sollen oder können (z.B. wegen Batchbetrieb)
- mehrere Programme sollen untereinander einfache Daten austauschen, ohne komplizierte Protokolle verwenden zu müssen.

Die Lösung dieser Probleme besteht meist in der Implementierung einer individuellen Datenschnittstelle, mittels derer Daten entsprechend vorgegebener Konventionen gelesen und geschrieben werden können. Der Aufwand für Programmierung und Verifikation einer solchen Schnittstelle ist gewöhnlich immens und eine Wiederverwendbarkeit für andere Programme nur schwer zu erreichen.

Aus diesem Grund wurde eine allgemeine Datenschnittstelle entwickelt, die ausgehend von einem einfachen Konzept einen großen Bereich von Anwendungsfällen abdecken kann. Sie besitzt die folgenden Charakteristika:

- es können sowohl numerische als auch Zeichenketten-Daten verwendet werden
- Daten können logisch zu Vektoren und Matrizen zusammengefaßt werden, deren Dimensionen sich dynamisch ändern können
- die interne Datenrepräsentation erfolgt in Speicherbereichen, die wie normale Programmvariablen gelesen und modifiziert werden können
- die externe Datenrepräsentation erfolgt zeilenorientiert in ASCII-Textdateien, die mit beliebigen Texteditoren erzeugt, modifiziert und angezeigt werden können
  (→ Transparenz durch Nutzerlesbarkeit)
- die Spezifikation konkreter Datenschnittstellen erfolgt durch einen einfachen Beschreibungsmechanismus
- es existieren Realisierungen für verschiedene Programmiersprachen und Betriebssysteme (zur Zeit C, C++ und PASCAL für MS-DOS und UNIX)

Das Modul, das diese Schnittstelle implementiert, heißt ***i_face***. Im folgenden wird die C++-Variante dieser Schnittstelle beschrieben. Analoge Beschreibungen existieren auch für C und PASCAL.

2 Die Datenschnittstelle

Die durch das Modul *i_face* realisierte allgemeine Datenschnittstelle bietet eine einfache Möglichkeit, abstrakte Datenmengen zu beschreiben und deren konkrete Wertausprägungen zwischen einer *internen* Repräsentation (Speicherbereich im Programm) und einer *externen* Repräsentation (textuelle Darstellung in ASCII-Datei) auszutauschen. Die entsprechenden Aktivitäten werden (in Anlehnung an die "Sicht" des Programmes) als

- *Lesen* (Textdatei → Speicherbereich) und
- *Schreiben* (Speicherbereich → Textdatei)

bezeichnet.

Eine konkrete Datenschnittstelle wird durch eine *Transfertabelle* beschrieben, die Informationen zum Aufbau der abstrakten Datenmenge und zur eindeutigen Abbildung ihrer konkreten Wertausprägungen in der internen und externen Repräsentation beinhaltet (siehe Bild 1).

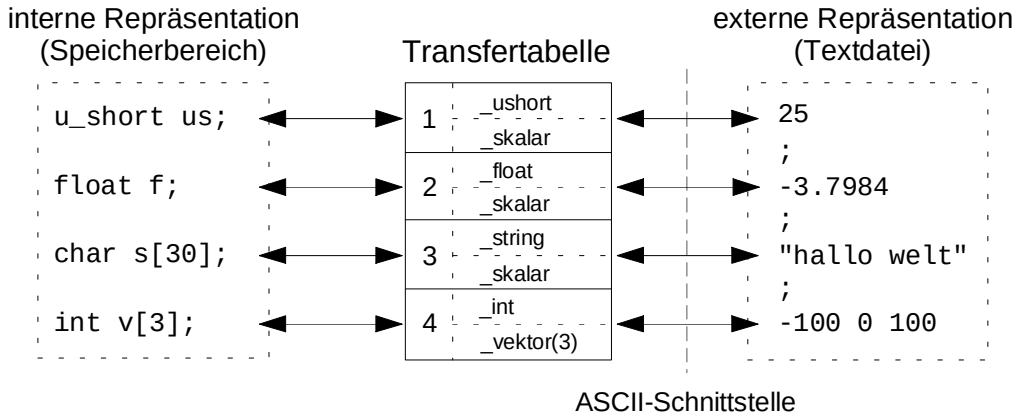

Bild 1. Aufbau einer konkreten Datenschnittstelle

2.1 Aufbau von Transfertabellen

2.1.1 Beschreibung der abstrakten Datenmenge

Transfertabellen bestehen aus einer Reihe von Einträgen, wobei jeder Eintrag ein Datenobjekt der abstrakten Datenmenge beschreibt. *Datenobjekte* (DO) entstehen durch logische Zusammenfassung einer Menge von gleichartigen Daten und werden durch einen Datentyp und eine Datenstruktur gekennzeichnet.

Der *Datentyp* gibt an, welche konkreten Datenwerte erlaubt sind und wie diese in der internen und externen Repräsentation dargestellt werden. Die folgende Tabelle nennt alle verwendbaren Datentypen und ordnet ihnen äquivalente C++-Datentypen zu. Letztere bestimmen entsprechend der Sprachdefinition von C++ sowohl die Wertebereiche als auch die Darstellung konkreter Datenwerte (textuell und im Speicher).

Datentyp	äquiv. C++-Datentyp	Wertebereich[1)]
`_char`	`signed char`	-128 .. 127
`_uchar`	`unsigned char`	0 .. 255
`_short`	`short`	-32768 .. 32767
`_ushort`	`unsigned short`	0 .. 65535
`_int`	`int`	wie `_short` bzw. `_long`
`_uint`	`unsigned int`	wie `_ushort` bzw. `_ulong`
`_long`	`long`	-2147483648 .. 2147483647
`_ulong`	`unsigned long`	0 .. 4294967295
`_float`	`float`	$0.0 \cup 1.18*10^{-38} \le \|x\| \le 3.40*10^{38}$

`_double`	`double`	$0.0 \cup 2.23*10^{-308} \le \|x\| \le 1.79*10^{308}$
`_string`	`char[]`	beliebige Zeichenketten

[1] zum Teil implementationsabhängig

Die *Datenstruktur* gibt an, wie viele Daten auf welche Art zu einem Datenobjekt zusammengefaßt werden. Die folgende Tabelle nennt alle verwendbaren Datenstrukturen und jeweils ein Beispiel für eine äquivalente Struktur in C++.

Datenstruktur	C++-Beispiel	Bedeutung
`_skalar`	`int s;`	ein Datenwert
`_vektor(N)`	`int v[10];`	eindimensionales Feld von Datenwerten die Dimension `N` > 0 spezifiziert die Anzahl Werte im Vektor
`_matrix(`N_1`,`N_2`)`	`int m[3][5];`	zweidimensionales Feld von Datenwerten die Dimensionen N_1 > 0 und N_2 > 0 spezifizieren die Anzahl Zeilen und Spalten in der Matrix (Anzahl Werte in der Matrix = $N_1 * N_2$)

2.1.2 Abbildung der abstrakten Datenmenge in der internen Repräsentation

In der internen Repräsentation ist jedem Datenobjekt ein zusammenhängender Speicherbereich zugeordnet, der dessen aktuelle Wert(e) enthält. Der Speicherbereich wird als eindimensionales Feld von Elementen des jeweils äquivalenten C++-Datentyps angesehen. Die Reihenfolge der Werte in diesem Feld ist

- `_skalar`: wert
- `_vektor(N)`: $wert_1$, .., $wert_N$
- `_matrix(`N_1`,`N_2`)`: $wert_{1,1}$, .., $wert_{1,N2}$, .., $wert_{N1,1}$, .., $wert_{N1,N2}$

und entspricht den Konventionen von C++. Somit kann der lesende und schreibende Zugriff auf die Daten nach den in C++ üblichen Verfahren erfolgen[1].

Bsp.:
```
short s;          // sei Speicherbereich für skalares DO
int v[5];         // sei Speicherbereich für Vektor-DO
double m[3][4];   // sei Speicherbereich für Matrix-DO
```

`s` referenziert den Wert des skalaren DO
`v[2]` referenziert den Wert (3) des Vektor-DO
`m[2][0]` referenziert den Wert (3,1) des Matrix-DO

Die Position eines Datenobjekts in der internen Repräsentation wird durch die Anfangsadresse des zugeordneten Speicherbereichs beschrieben.

[1] dabei ist zu beachten, daß die Feldindizierung in C++ stets mit 0 beginnt

2.1.3 Abbildung der abstrakten Datenmenge in der externen Repräsentation

In der externen Repräsentation (Textdatei) ist jedem Datenobjekt eine Reihe von aufeinanderfolgenden Datenzeilen zugeordnet (siehe Tabelle).

Datenstruktur	Anzahl Zeilen	Anzahl Datenwerte je Zeile
`_skalar`	`1`	`1`
`_vektor(N)`	`1`	`N`
`_matrix(N1,N2)`	N_1	N_2

Befinden sich mehrere Datenwerte in einer Zeile, dann müssen diese durch mindestens ein Whitespace-Zeichen (Leerzeichen bzw. Tabulator) voneinander getrennt sein. Zeilen, die mit dem frei definierbaren Kommentarzeichen beginnen (Standard: '`;`'), sind keine Daten- sondern Kommentarzeilen.

Datenwerte werden in der externen Repräsentation textuell dargestellt, wobei das textuelle Format vom zugeordneten Datentyp abhängig ist. Die Beschreibung der Formate unter Verwendung der EBNF[1)] ist in der folgenden Tabelle zusammengefaßt.

Spezielle Symbole:
<ziffer> = "0" | "1" | .. | "9"
<zfolge> = <ziffer> { <ziffer> }
<ws> = " " | <tab> (<tab> - Tabulator)
<q1> = "'" (Quotierungszeichen 1)
<q2> = "" (Quotierungszeichen 2)

Datentyp	Format

`_uchar`
`_ushort`
`_uint`
`_ulong`
	["+"] <zfolge>

`_char`
`_short`
`_int`
`_long`
	["+" | "-"] <zfolge>

`_float` `_double`	["+" | "-"] <zfolge> ["." <zfolge>] [("e" | "E") ["+" | "-"] <zfolge>]
`_string`[1)]	<zeichenfolge ohne <ws>, <q1> und <q2>> | (<q1> <zeichenfolge ohne <q1>> <q1>) | (<q1> <q1>) | (<q2> <zeichenfolge ohne <q2>> <q2>) | (<q2> <q2>)

[1)] etwas informelle Formatbeschreibung, da durch Quotierungsvarianten sonst zu kompliziert

[1)] *Erweiterte Backus-Naur-Form* - Beschreibungsformalismus für syntaktische Definitionen

Bsp. für eine externe Repräsentation:

```
; ein Skalar
5
; ein Vektor (5 Werte)
1 2  3 +4  -5
; eine Matrix (3x4 Werte)
1 2 3 4
0.07 -0.01 1.23 0.123
1.2e+3 -1.2E3 0.7e-2 -0.7e-2
;
```

Die Position eines Datenobjekts in der externen Repräsentation entspricht der Position des Datenobjekteintrags in der verwendeten Transfertabelle, d.h. die Reihenfolge der Einträge in der Tabelle legt die Reihenfolge der Datenzeilen in der Textdatei fest (siehe Bild 1).

2.1.4 Zur Problematik *dynamischer* abstrakter Datenmengen

Insbesondere bei komplexeren Datenschnittstellen kann der Fall auftreten, daß zur eindeutigen Beschreibung des Aufbaus der abstrakten Datenmenge die Kenntnis des aktuellen Wertes bestimmter darin enthaltener Datenobjekte (der sich dynamisch ändern kann) notwendig ist.

Bsp. 1: die abstrakte Datenmenge besteht aus einem `_skalar`-Datenobjekt des Typs `_ushort` sowie einem `_vektor`-Datenobjekt des Typs `_int`, dessen Dimension `N` durch den Wert des `_skalar`-Datenobjekts angegeben wird

textuell z.B.

```
3
1 2 3
```

oder

```
5
1 2 3 4 5
```

Die in diesem Beispiel beschriebene Problematik kann gelöst werden, indem erlaubt wird, Dimensionen von `_vektor`- und `_matrix`-Datenobjekten nicht nur statisch als konstante Zahlen sondern auch dynamisch in Form von C++-Funktionen anzugeben (siehe Bild 2). Derartige Funktionen liefern bei Aufruf den aktuellen Dimensionswert (> 0), der z.B. anhand der aktuellen Werte anderer Datenobjekte bestimmt wird.

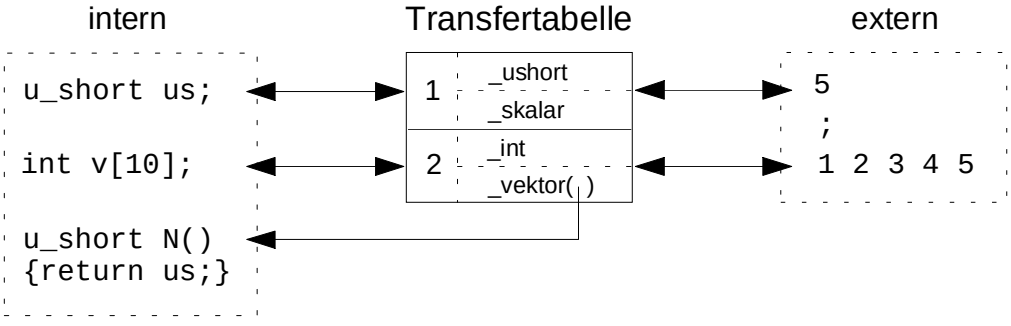

Bild 2. Dynamische Dimensionen von Datenobjekten

Bsp. 2: die abstrakte Datenmenge besteht aus einem `_skalar`-Datenobjekt des Typs `_ushort` sowie einem `_skalar`-Datenobjekt des Typs `_double` oder `_string`, je nachdem, ob der Wert des ersten Datenobjekts gerade oder ungerade ist

```
textuell z.B.   2          oder   3
                3.456             "hallo welt"
```

Die in diesem Beispiel beschriebene Problematik kann gelöst werden, indem die Definition von "Pseudo"-Datenobjekten erlaubt wird, die als Platzhalter für abstrakte Datenmengen fungieren. Derartige Datenobjekte werden durch C++-Funktionen repräsentiert, die bei Aufruf die Beschreibung einer abstrakten Datenmenge in Form einer Transfertabelle liefern, deren Aufbau z.B. vom Wert anderer Datenobjekte abhängig ist (siehe Bild 3). Die Pseudo-Datenobjekte werden wie normale Datenobjekte durch Einträge in der die Datenschnittstelle beschreibenden Transfertabelle repräsentiert und erlauben somit ein gewisses "dynamisches Include" von abstrakten (Teil-)Datenmengen.

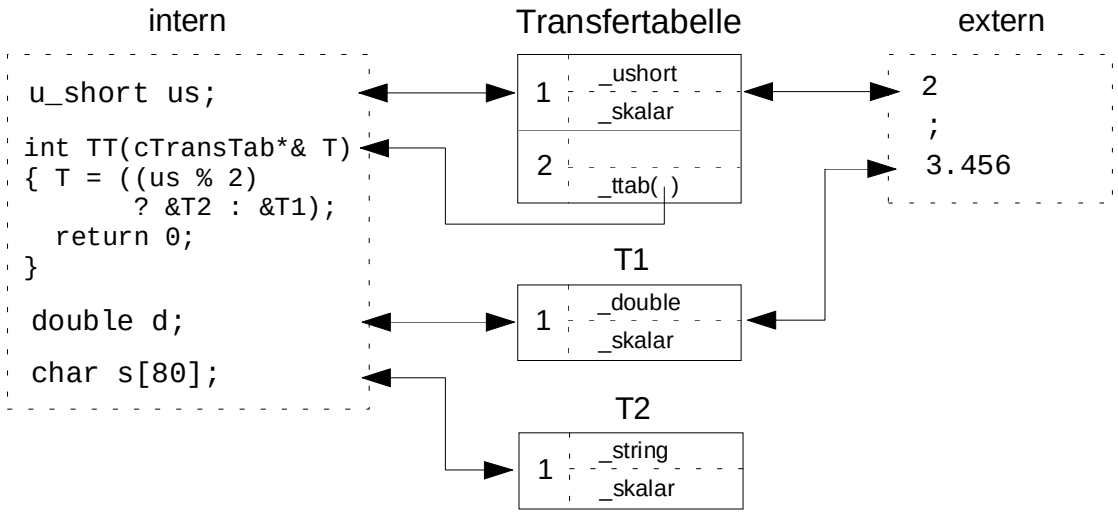

Bild 3. Dynamisches Include von Transfertabellen

2.2 Externe Repräsentation → interne Repräsentation (Lesen)

Die Textdatei wird zum Lesen geöffnet und für die Datenobjekte[1)] der verwendeten Transfertabelle werden die in der Datei enthaltenen Datenwerte gelesen, in die rechnerinterne Darstellung konvertiert und in den dem jeweiligen Datenobjekt zugeordneten Speicherbereich geschrieben (mit automatischem Test von Format, Wertebereich und nutzerdefinierten Constraints[2)]). Wie viele Datenwerte je Datenobjekt gelesen werden müssen und wie sich diese auf die Textzeilen verteilen, ergibt sich aus der Datenstruktur und der Dimension[3)] des Datenobjekts. Existieren für ein Datenobjekt keine korrekten Datenwerte in der Textdatei (z.B. falsches Format, falscher Wertebereich, zu wenig Werte), dann wird ein Fehlerstatus gesetzt und das Einlesen abgebrochen.

Wird der Einlesevorgang fehlerfrei abgeschlossen, dann enthält die interne Repräsentation für alle beteiligten Datenobjekte garantiert Werte, die den geforderten Datentypen und -strukturen sowie den nutzerdefinierten Constraints entsprechen.

---

1) für Pseudo-Datenobjekte werden keine Datenwerte gelesen, sondern die zugeordnete C++-Funktion ausgeführt, die einen Verweis auf eine (ggf. dynamisch erzeugte) Transfertabelle liefert
danach werden erst für alle Datenobjekte dieser Tabelle Datenwerte gelesen, ehe der Einlesevorgang mit dem nächsten Datenobjekt der zuvor verwendeten Tabelle fortgesetzt wird

2) siehe Abschnitt 2.4 "Nutzerdefinierte Constraints"

3) bei Verwendung dynamischer Dimensionen werden erst, wenn ein Datenobjekt "an der Reihe ist", die zugeordneten C++-Funktionen aufgerufen, die den aktuellen Dimensionswert liefern

2.3 Interne Repräsentation → externe Repräsentation (Schreiben)

Die Textdatei wird zum Schreiben geöffnet und für die Datenobjekte[1)] der verwendeten Transfertabelle werden die im zugeordneten Speicherbereich enthaltenen Datenwerte gelesen (mit automatischem Test von Wertebereich und nutzerdefinierten Constraints[2)]), in die textuelle Darstellung konvertiert und zeilenweise in die Datei geschrieben[3)]. Wie viele Datenwerte je Datenobjekt geschrieben werden müssen und wie sich diese auf die Textzeilen verteilen, ergibt sich aus der Datenstruktur und der Dimension[4)] des Datenobjekts. Zur Verbesserung der Nutzerlesbarkeit der Textdatei kann vor die Werte eines Datenobjekts eine beliebige, dem Datenobjekt zugeordnete Kommentarzeile geschrieben werden.

Wird der Schreibvorgang fehlerfrei abgeschlossen, dann kann die Textdatei unter Verwendung derselben Transfertabelle jederzeit wieder eingelesen werden (Ausnahme: siehe Fußnote 3).

2.4 Nutzerdefinierte Constraints

Neben dem formalen Test von Datenwerten auf die Einhaltung der entsprechenden Wertebereiche ist es oft notwendig, die Erfüllung spezieller Constraints durch die Datenwerte zu überprüfen.

Bsp.: die Datenwerte eines `_vektor`-Datenobjekts repräsentieren eine Wahrscheinlichkeitsverteilung und müssen deshalb zwischen 0.0 .. 1.0 liegen und sich zu 1.0 summieren

Um solche Constraints automatisch beim Lesen bzw. Schreiben testen zu können, kann einer Transfertabelle eine C++-Prüffunktion zugeordnet werden, die bei Angabe der Eintragsnummer eines Datenobjekts prüft, ob dessen aktuelle Datenwerte in der internen Repräsentation alle geforderten Constraints erfüllen. Beim Lesen wird diese Prüffunktion automatisch für jedes Datenobjekt aufgerufen, unmittelbar nachdem dessen Datenwerte in die interne Repräsentation überführt wurden. Beim Schreiben wird diese Prüffunktion automatisch für jedes Datenobjekt aufgerufen, unmittelbar bevor dessen Datenwerte in die externe Repräsentation überführt werden. Existieren nicht erfüllte Constraints, dann wird das Lesen bzw. Schreiben mit einem Fehlerstatus abgebrochen.

---

1) für Pseudo-Datenobjekte werden keine Datenwerte geschrieben, sondern die zugeordnete C++-Funktion ausgeführt, die einen Verweis auf eine (ggf. dynamisch erzeugte) Transfertabelle liefert danach werden erst für alle Datenobjekte dieser Tabelle Datenwerte geschrieben, ehe der Schreibvorgang mit dem nächsten Datenobjekt der zuvor verwendeten Tabelle fortgesetzt wird

2) siehe Abschnitt 2.4 "Nutzerdefinierte Constraints"

3) leider ist es in der aktuellen Version des Schnittstellen-Moduls nicht möglich, `_vektor`- und `_matrix`-Datenobjekte des Typs `_string` korrekt in Textdateien auszugeben; statt dessen wird für jeden Wert die leere Zeichenkette `''` ausgegeben

4) bei Verwendung dynamischer Dimensionen werden erst, wenn ein Datenobjekt "an der Reihe ist", die zugeordneten C++-Funktionen aufgerufen, die den aktuellen Dimensionswert liefern

3 Die Programmierschnittstelle

Das Schnittstellen-Modul besteht aus den beiden Dateien *i_face++.h* und *i_face++.cc* (UNIX) bzw. *i_facexx.h* und *i_facexx.cc* (DOS). Sie definieren eine Reihe von symbolischen Konstanten, Datentypen sowie die Transfertabellen-Klasse `cTransTab` und versetzen den Programmierer in die Lage, eigene applikationsspezifische Datenschnittstellen zu implementieren. Die folgenden Abschnitte stellen die dafür notwendigen Informationen zur Verfügung.

3.1 Wichtige symbolische Konstanten

Das Schnittstellen-Modul kann durch eine Reihe symbolischer Konstanten an verschiedene Gegebenheiten (Aufgabenstellungen, Compiler, Betriebssysteme, etc.) angepaßt werden. Sie können bei Bedarf entweder direkt in der Headerdatei *i_face++.h* modifiziert oder bei der Übersetzung mittels der `-D`-Option (z.B. `-DINT_16BIT`, `-DMAX_ZLEN=100`) überschrieben werden (siehe Tabelle).

Name	Standard	Erläuterung
`__MSDOS__`	nicht definiert	muß beim Übersetzen auf einem DOS-System definiert sein (erfolgt i.allg. automatisch)
`IF_NO_DELETE_A`	nicht definiert	muß beim Übersetzen mit Compilern, die kein `delete[]` kennen, definiert sein
`IF_NO_DEF_ARGS`	nicht definiert	muß beim Übersetzen mit Compilern, die keine Defaultargumente kennen, definiert sein
`IF_UF_NO_ERROR`	nicht definiert	muß beim Übersetzen definiert sein, wenn der Underflow von `_float`- und `_double`-DO nicht als Fehler gelten soll (Underflow → 0.0)
`INT_16BIT`	nicht definiert	muß beim Übersetzen mit Compilern, bei denen `int` 2 Byte statt 4 Byte lang ist, definiert sein
`MAX_ZLEN`	`5000`	maximale Zeilenlänge der verwendeten Dateien
`MAX_DOBJEKTE`	`1000`	maximale Anzahl DO-Einträge je Transfertabelle
`MAX_VELEM`	`500`	maximale Dimension `N` von `_vektor`-DO
`MAX_MZELEM`	`500`	maximale Zeilenanzahl N_1 von `_matrix`-DO
`MAX_MSPELEM`	`500`	maximale Spaltenanzahl N_2 von `_matrix`-DO
`MAX_SIGFLOAT`	`8`	maximale Anzahl signifikanter Ziffern bei der Ausgabe von `_float`-DO
`MAX_SIGDOUBLE`	`8`	maximale Anzahl signifikanter Ziffern bei der Ausgabe von `_double`-DO

3.2 Wichtige Datentypen

Im Schnittstellen-Modul wird eine Reihe spezieller Datentypen verwendet (siehe Tabelle).

Name	Erläuterung
`boolean`	Boolescher Typ mit den Werten `TRUE` und `FALSE`
`s_char`	Synonym für `signed char`
`u_char`	Synonym für `unsigned char`
`u_short`	Synonym für `unsigned short`
`u_int`	Synonym für `unsigned int`
`u_long`	Synonym für `unsigned long`
`tDTyp`	Aufzählungstyp aller DO-Typen (`_uchar`, ...)

3.3 Die `cTransTab`-Klasse und deren Funktionen

Die Repräsentation konkreter Datenschnittstellen erfolgt durch Objekte der Klasse `cTransTab`. Die Elementfunktionen dieser Klasse lassen sich grundsätzlich einteilen in solche zur Fehlerbehandlung, zur Definition von Transfertabellen und zum Datenlesen und -schreiben. Die wichtigsten Funktionen werden im folgenden erläutert.

3.3.1 Fehlerbehandlung

Die meisten Elementfunktionen liefern einen `int`-Wert zurück, der den Ausführungsstatus der Funktion beschreibt. Der Wert `0` bedeutet, daß die Funktion fehlerfrei ausgeführt wurde; der Wert `-1` bedeutet, daß ein Fehler aufgetreten ist.

Die Funktion

```
int GetLastError(const char*& Text, u_long& ZNr,
                 u_short& EntryNr) const;
```

liefert den konkreten Fehlercode `IF_ERR_`...[1] zum letzten aufgetretenen Fehler sowie einen erklärenden Fehlertext `Text`, ggf. die Nummer `ZNr` (`1`..) der zuletzt gelesenen Textzeile (falls von Bedeutung, sonst `0`) und ggf. die Nummer `EntryNr` (`1`..) des zuletzt verwendeten Datenobjekts (falls von Bedeutung, sonst `0`). Der Programmierer kann diese Informationen zur Realisierung einer eigenen Fehlerbehandlung verwenden.

Die Ausgabe einer Fehlermeldung zum zuletzt aufgetretenen Fehler in die zum Schreiben geöffnete Datei `OutFile` (Default: Standardfehlerausgabe) kann durch die Funktion

```
void PrintLastError(FILE* OutFile = stderr) const;
```

erreicht werden[2].

---

[1] siehe *i_face++.h* für eine Beschreibung aller existierenden Fehlercodes

[2] eine automatische Ausgabe auf die Standardfehlerausgabe bei Auftreten eines Fehlers erfolgt dann, wenn bei der Definition der Transfertabelle `ErrorWrite` = `TRUE` angegeben wurde

3.3.2 Definition von Transfertabellen

Bei der Erzeugung von Transfertabellen-Objekten wird der Konstruktor

```
cTransTab(u_short Anz, boolean ErrorWrite = TRUE,
          const char *Header = NULL, char KZeichen = ';');
```

abgearbeitet. `Anz` gibt dabei die genaue Anzahl enthaltener Datenobjekteinträge an, `ErrorWrite` gibt an, ob bei Fehlern automatisch eine Fehlermeldung ausgegeben werden soll, `Header` ist ein Headerstring[1] (oder `NULL`) und `KZeichen` das Zeichen, das Kommentarzeilen in Textdateien einleitet.

Unmittelbar nach der Erzeugung ist eine Transfertabelle noch "leer". Bevor sie verwendet werden kann, müssen erst alle Datenobjekteinträge mit korrekten Informationen gefüllt werden. Dazu stehen je nach Art des jeweiligen Datenobjekts die folgenden Funktionen zur Verfügung:

```
int SetDObj_Skalar(u_short Nr, tDTyp DTyp, void *DAdr,
                   const char *TAdr);

int SetDObj_Vektor(u_short Nr, tDTyp DTyp, void *DAdr,
                   const char *TAdr, u_short N);

int SetDObj_Vektor(u_short Nr, tDTyp DTyp, void *DAdr,
                   const char *TAdr, u_short (*NFunc)(void));

int SetDObj_Matrix(u_short Nr, tDTyp DTyp, void *DAdr,
                   const char *TAdr, u_short Z, u_short Sp);

int SetDObj_Matrix(u_short Nr, tDTyp DTyp, void *DAdr,
                   const char *TAdr, u_short (*ZFunc)(void),
                   u_short Sp);

int SetDObj_Matrix(u_short Nr, tDTyp DTyp, void *DAdr,
                   const char *TAdr, u_short Z,
                   u_short (*SpFunc)(void));

int SetDObj_Matrix(u_short Nr, tDTyp DTyp, void *DAdr,
                   const char *TAdr, u_short (*ZFunc)(void),
                   u_short (*SpFunc)(void));
```

`Nr` gibt den Index des zu füllenden Eintrags an (im Bereich `1`..`Anz`), `DTyp` den Datentyp (`_uchar`, ...), `DAdr` die Adresse des zugeordneten Speicherbereichs und `TAdr` ggf. einen Zeiger auf einen Kommentarstring[2] (oder `NULL`). Die Angabe der Dimensionen kann entweder in Form konkreter Zahlen (`N`, `Z`, `Sp`) oder in Form von Zeigern auf C++-Funktionen (`NFunc`, `ZFunc`, `SpFunc`) des allgemeinen Typs `u_short foo(void)`, die bei Aufruf den aktuellen Dimensionswert liefern, erfolgen.

Zum Füllen des Eintrags eines Pseudo-Datenobjekts (siehe Abschnitt 2.1.4) wird die Funktion

```
int SetTmpTTab(u_short Nr, int (*GetTTFkt)(cTransTab*&));
```

verwendet. `Nr` gibt den Index des zu füllenden Eintrags an (im Bereich `1`..`Anz`) und `GetTTFkt` einen Zeiger auf eine C++-Funktion der Form `int foo(cTransTab*&)`, die bei

[1] der Headerstring wird stets als erstes in eine Ausgabedatei geschrieben; er darf maximal `MAX_ZLEN` Zeichen lang sein und kann auch Newlines ('`\n`') enthalten

[2] der Kommentarstring darf maximal `MAX_ZLEN` Zeichen lang sein und kann auch Newlines ('`\n`') enthalten

Aufruf im Referenzparameter einen Zeiger auf die temporär zu verwendende Transfertabelle bereitstellt und einen der Statuswerte

- `0` Transfertabelle nach Gebrauch nicht automatisch löschen
- `1` Transfertabelle nach Gebrauch automatisch löschen
- `-1` Fehler aufgetreten

liefert.

Zur Zuordnung einer Prüffunktion (siehe Abschnitt 2.4) zu einer Transfertabelle wird die Funktion

```
int SetPruefFkt(int (*PruefFkt)(u_short));
```

verwendet. `PruefFkt` ist ein Zeiger auf eine C++-Funktion der Form `int foo(u_short)`, die bei Aufruf mit Parameter i (im Bereich `1..ANZ`) die Erfüllung aller für Datenobjekt i geforderten Constraints überprüft und eine `0` für "erfüllt" bzw. eine `-1` für "nicht erfüllt" liefert. Ist `PruefFkt` = `NULL`, dann wird eine bestehende Zuordnung wieder aufgehoben.

3.3.3 Daten lesen und schreiben

Nachdem eine Transfertabelle vollständig definiert wurde, kann sie zum Datenaustausch verwendet werden.

Die Umwandlung externer Daten in interne Daten (Lesen) wird durch die Funktion

```
int LeseDaten(const char *DName)
```

realisiert. `DName` ist dabei der Name der Textdatei[1)] (ggf. inklusive Pfad), die die externen Datenwerte enthält. Nach erfolgreicher Ausführung dieser Funktion enthalten die den beteiligten Datenobjekten zugeordneten Speicherbereiche die interne Repräsentation der gelesenen Datenwerte.

Achtung: bevor die Funktion `LeseDaten(.)` ausgeführt werden kann, müssen (leider) die sämtlichen beteiligten `_string`-Datenobjekten zugeordneten Speicherbereiche explizit mit einer Zeichenkette der jeweils maximalen Länge belegt werden[2)]. Dazu kann die Funktion

```
char* FillString(char *String, char C, int Length)
```

verwendet werden, wobei `String` die Anfangsadresse des Speicherbereichs angibt, `C` ein beliebiges Zeichen (außer '`\0`'), aus dem sich die Zeichenkette zusammensetzen soll und `Length` die Länge des Speicherbereichs.

---

1) die Textdatei muß existieren und für den Nutzer, der das Programm ausführt, lesbar sein

2) Grund hierfür ist, daß das Modul bei `_string`-Datenobjekten gegenüber allen anderen Datentypen nicht selbständig ermitteln kann, wie groß der für einen Datenwert (maximal) notwendige Speicherplatz ist. Bei dem geschilderten Vorgehen hingegen kann diese Größe einfach durch Anwendung der `strlen(.)`-Funktion auf die Adresse des Speicherbereichs ermittelt werden.

Die Umwandlung interner Daten in externe Daten (Schreiben) wird durch die Funktion

```
int SchreibeDaten(const char *DName, const char *Header = NULL)
```

realisiert. `DName` ist dabei der Name der Textdatei[1] (ggf. inklusive Pfad), in die die Daten geschrieben werden sollen und `Header` ggf. einen Headerstring[2] (oder `NULL`). Nach erfolgreicher Ausführung dieser Funktion enthält die Textdatei die externe Repräsentation der Datenwerte der beteiligten Datenobjekte.

3.4 Einbindung des Schnittstellen-Moduls in eigene Programme

Um das Schnittstellen-Modul in eigenen C++-Programmen verwenden zu können, muß die Zeile

```
#include "i_face++.h"   (UNIX)  bzw.
#include "i_facexx.h"   (DOS)
```

in den Quelltext des jeweiligen Programms eingefügt werden. Die Übersetzung des Programms kann z.B. folgendermaßen erfolgen (`gcc` - GNU C/C++ Compiler):

```
gcc -o my_prog my_prog.cc i_face++.cc     (UNIX)  bzw.
gcc -o my_prog my_prog.cc i_facexx.cc     (DOS)
```

3.5 Ein Beispielprogramm

Das folgende einfache Beispielprogramm *my_prog* illustriert die Verwendung des Schnittstellen-Moduls in eigenen Programmen. Nach dem Start definiert das Programm eine Transfertabelle `ttab`, liest entsprechend dieser Tabelle Daten aus der Textdatei *my_prog.in* und schreibt diese anschließend in die Textdatei *my_prog.out* .

3.5.1 Quelltext von *my_prog.cc*:

```
#include <stdio.h>
#ifdef __MSDOS__
#include "i_facexx.h"   /* bei Uebersetzung unter DOS */
#else
#include "i_face++.h"   /* bei Uebersetzung unter UNIX */
#endif


/* Speicherbereiche */

u_char uc_s;
short  s_v[10];
float  f_m[2][4];
char   st_s[80];


/* Header/Kommentare */

char header[] = "das ist der Dateiheader\n  ";
char komm[]   = "das ist ein Kommentar !!!";
```

[1] die Textdatei muß für den Nutzer, der das Programm ausführt, schreibbar sein

[2] Der Headerstring wird stets als erstes in eine Ausgabedatei geschrieben. Er darf maximal `MAX_ZLEN` Zeichen lang sein und kann auch Newlines ('\n') enthalten. Ein bei dieser Funktion angegebener Headerstring wird statt des bei der Konstruktion evtl. bereits definierten Headerstring verwendet.

```
/* Hilfsfunktion */

u_short DimVek(void)
/* aktuelle Anzahl Elemente von 's_v' */
{ return uc_s; }


/* Hauptprogramm */

int main()
{ cTransTab ttab(4,FALSE,header);

  /* Transfertabelle fuellen */
  if ((ttab.SetDObj_Skalar(1,_uchar,&uc_s,NULL) != 0) ||
      (ttab.SetDObj_Vektor(2,_short,s_v,NULL,DimVek) != 0) ||
      (ttab.SetDObj_Matrix(3,_float,f_m,komm,2,4) != 0) ||
      (ttab.SetDObj_Skalar(4,_string,st_s,NULL) != 0))
  { ttab.PrintLastError(); return (-1); }

  /* alle '_string'-Datenobjekte initialisieren */
  cTransTab::FillString(st_s,' ',80);

  /* Daten aus 'my_prog.in' lesen */
  if (ttab.LeseDaten("my_prog.in") != 0)
  { ttab.PrintLastError(); return (-1); }

  /* jetzt stehen die Daten in den entspr. Programmvariablen zur Verfuegung
     und koennen im Programm verarbeitet (gelesen, modifiziert, etc.)
     werden
  */

  /* Daten in 'my_prog.out' schreiben */
  if (ttab.SchreibeDaten("my_prog.out") != 0)
  { ttab.PrintLastError(); return (-1); }

  return 0;
}
```

3.5.2 Inhalt der Eingabedatei *my_prog.in*:

```
; skalarer '_uchar'-Wert (= DIM)
5
; Vektor mit '_short'-Werten (Dimension DIM)
-12345 -123 0  123 12345
; Matrix mit '_float'-Werten (Dimension 2 x 4)
  1.2345  1.2345e-10  1.2345e+10  0.000012345
 -1.2345 -1.2345e-10 -1.2345e+10 -0.000012345
; skalarer '_string'-Wert
'das ist ein String !!!'
```

3.5.3 Inhalt der Ausgabedatei *my_prog.out* (nach Ausführung von *my_prog*):

```
; das ist der Dateiheader
;
;
5
;
-12345 -123 0 123 12345
;
; das ist ein Kommentar !!!
1.2345001 1.2345e-10 1.2345e+10 1.2345e-05
-1.2345001 -1.2345e-10 -1.2345e+10 -1.2345e-05
;
'das ist ein String !!!'
```

Anhang A: Datenaustausch zwischen Programmen mittels *i_face*

In diesem Abschnitt soll das prinzipielle Vorgehen beim Datenaustausch zwischen zwei Programmen beschrieben werden. Dabei wird eines der Programme als *primär* (aufrufendes, aktives Programm) bezeichnet, das andere als *sekundär* (aufgerufenes, passives Programm). O.B.d.A. soll das primäre Programm dem sekundären einen Vektor von Parameterwerten übergeben, aus denen dieses einen Funktionswert bestimmt (z.B. durch Simulation) und an das primäre Programm zurückliefert. Dieses Schema wird u.a. bei der Kopplung von Simulations- und Optimierungssystemen eingesetzt.

Im Primär- und Sekundärprogramm muß je eine Transfertabelle zur Beschreibung der Parameter- und der Funktionswert-Schnittstelle definiert sein.

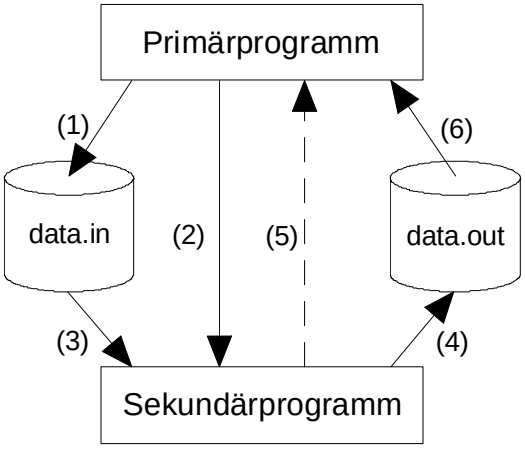

Bild 4. Datenaustausch zwischen Primär- und Sekundärprogramm

Der Datenaustausch erfolgt in sechs Schritten (siehe Bild 4):

1) das Primärprogramm schreibt die aktuellen Parameterwerte in die Textdatei *data.in*
2) das Primärprogramm startet[1)] das Sekundärprogramm (z.B. mit `system(.)`)
3) das Sekundärprogramm liest die Parameterwerte aus der Textdatei *data.in* und bestimmt den zugehörigen Funktionswert
4) das Sekundärprogramm schreibt den Funktionswert in die Textdatei *data.out*
5) das Sekundärprogramm beendet seine Arbeit und gibt die Steuerung an das Primärprogramm zurück
6) das Primärprogramm liest den Funktionswert aus der Textdatei *data.out* und setzt seine Arbeit fort

---

[1)] Bei Verwendung von Multitasking-Betriebssystemen ist es auch möglich, daß Primär- und Sekundärprogramm gleichzeitig (parallel) abgearbeitet werden. In diesem Fall ist jedoch eine Synchronisation notwendig, z.B.

:
2) das Primärprogramm sendet eine "Starte Berechnung"-Nachricht an das Sekundärprogramm
:
5) das Sekundärprogramm sendet eine "Berechnung beendet"-Nachricht an das Primärprogramm
: