Zugriff auf Texte und Tabellen

Horst Hollatz

1. September 2005

Zusammenfassung

Das folgende kleine Text/Tabellen-Zugriffssystem ist aus dem Bedürfnis entstanden, die Möglichkeit zu haben, in einfacher Weise auf Texte und Tabellen zugreifen zu können, wobei sequentielles wort- und zeilenweises, direktes wort- und zeilenweises und Tabellen-Verarbeiten möglich sein sollten. Die Tatsache, dass nicht `C++`, sondern `C` benutzt wird, ist dem Umstand geschuldet, dass ein `C++`-Programm mit gleicher Funktionalität wie das entsprechende `C`-Programm einen erheblich längeren Programm-Code besitzt. Trotzdem sind die vorliegenden Programme mittels `C++`-Übersetzer zu verwenden, da sie Standard-Parameter-Werte beim Aufruf verwenden und zur Programm-Identifikation auch die Daten-Typen des Aufrufs gehören.

1. Texte

Für Textbearbeitung wird die folgende Daten-Struktur verwendet (siehe `defText.h`):

```
typedef struct TextStruct
        { size_t lText,         // Textlänge
                 lFree;         // Freiplätze
            char *Text,         // Text-Anfangszeiger
                 *Word,         // aktueller Wort-Zeiger
                 *Line,         // aktueller Zeilen-Zeiger
                 *Output,       // Ausgabe-Feld für Zeile/Wort
                 *Ignore;       // Feld mit Wort-Trennzeichen
        } Text;
```

Während der Zeilen-Zeiger stets auf einen Zeilen-Anfang zeigt, wandert der Wort-Zeiger von einem Wort zum nächsten. Nach dem Setzen des Zeilen-Zeigers hat der Wort-Zeiger den gleichen Wert. Die Indizierung der Zeilen und Wörter beginnt mit 0 und gilt stets relativ zur Position des Zeilen- bzw. Wort-Zeigers. Falls der Rückkehr-Wert einer Routine vom Typ `unsigned char` ist, gibt dieser im Erfolgsfalle den Wert 1, sonst 0 zurück; sollte der Rückkehr-Wert vom Typ `size_t` sein, wird die Wort-Länge, eine Zeilen-Anzahl oder ein Index zurückgegeben.
Es gibt die folgenden Routinen.

Text-Struktur erzeugen:

```
Text OpenText(char *name);
Text OpenText(Table *f);
```

Es wird eine Text-Struktur mit dem Inhalt der Datei `name` geliefert; auf dem Feld `Ignore` sind die Wort-Trennzeichen (=Ignorier-Zeichen) abgelegt. Jede Text-Struktur ist mit einem Aufruf von `OpenText` zu initialisieren. Im Falle `name=0` wird eine leere Text-Struktur erzeugt; ist `name` das Leer-Wort, wird der Text von der Standard-Eingabe gelesen. Das neue-Zeile-Zeichen sollte stets zu den Ignorier-Zeichen gehören; standardmäßig gelten die Werte (auch für Tabellen):

```
lOUTPUT 256,   // max. Zeilen-Länge
COLMAX  128,   // Spalten-Anzahl
EXTRA   2048,  // Länge Zuordnungseinheit
char *IGNORE=" \n\t.,;{}()<>!?:+§|[]=\"@\\";
```

Sollte der 1. Parameter ein Zeiger auf eine Tabelle sein, wird aus dieser ein Text erzeugt. Der Zeilen-Zeiger zeigt auf den Text-Anfang.

Text-Struktur schließen:

```
void CloseText(Text *f);
```

Die Struktur wird geschlossen; nach dem Aufruf darf die Struktur mittels `OpenText` wieder initialisiert werden.

Zeilen-Zeiger zurücksetzen:

```
unsigned char ResetText(Text *f);
```

Der Zeilen- und der Wort-Zeiger werden auf den Text-Anfang gesetzt.

Zeile liefern:

```
size_t GetLine(Text *f, size_t i=0);
```

Es wird die `i`-te Zeile nach der aktuellen auf `Output` bereitgestellt und der Zeilen-Zeiger auf den Anfang der nächsten Zeile gesetzt. Die Routine gibt die Zeilen-Länge zurück.

Zeilen entsprechend einer Maske liefern:

```
Text GetLine(Text *f, char *s);
```

Es werden alle jene Zeilen (ab der aktuellen) als neue Struktur bereitgestellt, die das auf `s` angegebene Merkmal erfüllen (analog zum `ls`- bzw. `dir`-Kommando) Steht auf `s` das Leer-Wort, werden alle Zeilen (ab der aktuellen) übernommen; ist `s` das Null-Wort, wird die aktuelle Zeile übernommen. In dem folgenden Aufruf-Beispiel werden alle Zeilen gefiltert, die den Buchstaben `m` enthalten und mit der Kombination `.c` enden:

```
GetLine(f,"*m*.c");
```

Auf die nächste Zeile positionieren:

```
size_t NextLine(Text *f);
```

Es wird auf die nächste Zeile positioniert und deren Zeilen-Länge zurückgegeben.

Auf die vorherige Zeile positionieren:

```
size_t PreviousLine(Text *f);
```

Es wird auf die vorherige Zeile positioniert und deren Zeilen-Länge zurückgegeben.

Wort liefern:

```
size_t GetWordinLine(Text *f, size_t i=0);
size_t GetWord(Text *f, size_t i=0);
```

Es wird das `i`-te Wort aus der aktuellen Zeile relativ zum Wert des Wortzeigers auf `Output` bereitgestellt und die Wort-Länge zurückgegeben. Beim 2. Aufruf wird der gesamte Text wortweise verarbeitet; es wird relativ zum Wort-Zeiger indiziert.

Wörter entsprechend einer Maske liefern:

```
size_t GetWordinLine(Text *f, char *s);
size_t GetWord(Text *f, char *s);
```

Es wird das 1. Wort aus der aktuellen Zeile auf `Output` bereitgestellt, das das auf `s` angegebene Merkmal erfüllt und die Wort-Länge zurückgegeben. Steht auf `s` das Leer-Wort, wird die Rest-Zeile bereitgestellt; ist `s` das Null-Wort, wird das aktuelle Wort geliefert. Der Wort-Zeiger steht danach hinter dem ausgegebenen Wort. Beim 2. Aufruf wird der gesamte Text wortweise verarbeitet; es wird relativ zum Wort-Zeiger indiziert.

Anhängen einer Zeile:

```
unsigned char AppendLine(Text *f, char *s);
```

Anhängen eines Textes:

```
unsigned char AppendText(Text *f, char *s);
```

Einfügen einer Zeile vor der aktuellen:

```
 unsigned char InsertLine(Text *f, char *s)
```

Einfügen eines Textes vor der aktuellen Zeile:

```
unsigned char InsertText(Text *f, char *s);
```

Einfügen einer Datei vor der aktuellen Zeile:

```
unsigned char InsertFile(Text *f, char *name);
```

Die Datei mit dem Namen `name` wird vor der aktuellen Zeile eingefügt.

Streichen von Text:

```
unsigned char DeleteText(Text *f,char *a=0,char *e=0);
```

Es wird der gesamte Text ab der Adresse `a` bis zur Adresse `b` gelöscht. Im Falle, dass `a` gleich `NULL` ist, wird ab Text-Anfang gelöscht; im Falle, dass `e` gleich `NULL` ist, wird bis Text-Ende gelöscht.

Streichen von `anz` Zeilen ab aktueller:

```
unsigned char DeleteLine(Text *f, size_t anz=1);
```

Streichen von Zeilen mittels Merkmal ab aktueller:

```
size_t DeleteLine(Text *f, char *s);
```

Es werden alle jene Zeilen (ab der aktuellen) gestrichen, die das auf `s` angegebene Merkmal erfüllen (analog zum `ls`- bzw. `dir`-Kommando) Steht auf s das Leer-Wort, werden alle Zeilen (ab der aktuellen) gestrichen; ist `s` das Null-Wort, wird die aktuelle Zeile gestrichen. Die Routine gibt die Anzahl der gestrichenen Zeilen zurück.

Streichen von `anz` Wörtern ab aktuellem:

```
unsigned char DeleteWord(Text *f, size_t anz=1);
```

Streichen von Wörtern mittels Merkmal ab aktuellem:

```
unsigned char DeleteWord(Text *f, char *s);
```

Es wird das 1. Wort aus der aktuellen Zeile gestrichen, das das auf `s` angegebene Merkmal erfüllt. Steht auf `s` das Leer-Wort, wird die Rest-Zeile gestrichen; ist `s` das Null-Wort, wird das aktuelle Wort gestrichen. Der Wort-Zeiger steht danach hinter dem gestrichenen Wort.

Ersetzen von Zeichen-Ketten:

```
size_t ReplaceString(Text *f, char *z="  ", char *s=" ");
```

Jede gefundene Zeichen-Kette `z` wird durch die auf `s` stehenden Zeichen-Kette ersetzt. Standardmäßig werden alle Mehrfach-Leerzeichen durch eines ersetzt. Im Falle `s=""` wird jede gefundene Zeichen-Kette gelöscht. Die Routine gibt die Anzahl der gefundenen Zeichen-Ketten zurück.

Schreiben eines Textes in eine Datei:

```
unsigned char PrintText
             (Text *f, char *name="", char *mode="w+");
```

Der vorhandene Text wird in die Datei `name` geschrieben; auf `mode` ist der Öffnungsmodus für die Datei anzugeben. Falls `name` das Leer-Wort ist, wird die Standard-Ausgabe verwendet.

2. Tabellen

Für die Tabellen-Bearbeitung wird die folgende Struktur verwendet (`defTable.h`):

```
typedef struct TableStruct
        { size_t lTable,       // Zeilen-Anzahl
                 lFree,        // Zeiger-Freiplätze
                 asp,          // Spalten-Anzahl
                 *Col;         // relative Spalten-Anfänge
             char **Table,     // Zeilen-Anfangszeiger
                  *Output;     // Ausgabe für Zeile/Eintrag
        } Table;
```

Tabelle erzeugen:

```
Table OpenTable(char *name=0);
Table OpenTable(Text *f, char *head="");
Table CreateTable(char *head);
Table CopyTable(Table *f);
```

Es wird eine Tabellen-Struktur erzeugt und eine Tabelle aus der Datei `name` eingelesen; die 1. gelesene Zeile wird als Tabellen-Kopf verwendet. Ist `name=""`, wird von der Standard-Eingabe gelesen; im Falle `name=0` wird eine leere Tabelle erzeugt. Die eigentliche Zeilen-Zählung beginnt mit `1`. Die Zeile 0 ist der Tabellen-Kopf und enthält die Spalten-Namen. Die Spalten-Zählung beginnt ebenfalls mit `1`. In den Aufrufen der einzelnen Funktionen sind oft ein Zeilen-, Spalten-Index, Zeilen- oder Spalten-Anzahl anzugeben. In allen Fällen, in denen im Aufruf ein Spalten-Index anzugeben ist, darf auch der Spalten-Name angegeben werden. Falls der Spalten-Eintrag in der 1. Spalte jede Zeile eindeutig identifiziert (Zeilen-Id), darf anstelle eines Zeilen-Indexes auch ein Spalten-Eintrag aus der 1. Spalte angegeben werden. Bei dieser alphanumerischen Indizierung ist die Zeilen- bzw. Spaltenanzahl stets gleich 1.
Beim Aufruf von `CreateTable` wird nur der Tabellen-Kopf übergeben. Ist der 1. Parameter im Aufruf der Zeiger auf eine Text-Struktur, wird aus dieser eine Tabelle erzeugt; auf `head` wird der Tabellen-Kopf erwartet; im Falle `head=""` wird der Tabellen-Kopf mit den Spalten-Indices gefüllt; Die Spalten-Breiten richtet sich nach dem längsten Wort, das in die betreffende Spalte einzutragen ist. Die Spalten-Einträge werden als Wörter linksbündig abgespeichert. Im Gegensatz zur Text-Struktur haben alle Zeilen die gleiche Länge und für jedes Wort steht in einer Spalte die gleiche Byte-Anzahl zur Verfügung. Im Feld `Col` findet man die relative Adresse jedes Spalten-Eintrags, so dass durch `*(Table+i)+*(Col+j-1)` der Zeiger auf den Anfang des in Zeile `i` und Spalte `j` stehenden Eintrags gegeben ist. Standardmäßig wird eine leere Tabelle erzeugt.

Tabellen-Struktur beenden:

```
void CloseTable(Table *f);
```

Die Struktur wird geschlossen; alle angeforderten Felder werden freigegeben. Nach dem Aufruf darf die Struktur mittels `OpenTable` wieder initialisiert werden.

Zeilen ausgeben:

```
Table GetRow(Table *f, size_t i, size_t anz=1);
```

Die zurückgegebene Tabelle enthält neben der Kopf-Zeile die ab Zeile `i` folgenden `anz` Zeilen aus der Tabelle.

Zeilen nach Merkmal ausgeben:

```
Table SearchRow(Table *f, size_t j, char *s);
```

Es werden alle jene Zeilen in die neue Tabelle übernommen, die in der Spalte `j` das auf `s` angegebene Merkmal haben (analog zum `ls`- bzw. `dir`-Kommando).

Spalten liefern:

```
Table GetColumn(Table *f, size_t j, size_t anz=1);
```

Die zurückgegebene Tabelle enthält die ab Spalte `j` folgenden `anz` Spalten aus der Tabelle `f`.

Spalten-Index liefern:

```
size_t SearchColumnIndex(Table *f, char *s);
```

Es wird der Index jener Spalte ermittelt, die den Namen `s` hat. Falls der Spalten-Name nicht existiert, ist `0` der Rückkehrwert.

Zeilen-Index liefern:

```
size_t SearchRowIndex(Table *f, char *s);
```

Es wird der Index jener Zeile ermittelt, die in der 1. Spalte den Eintrag `s` hat. Falls der Eintrag nicht existiert, ist `0` der Rückkehrwert.

Eintrag liefern:

```
char* GetEntry(Table *f, size_t i, size_t j);
```

Der sich in Zeile `i` und Spalte `j` befindliche Eintrag wird bereitgestellt.

Eintrag an eine Stelle speichern:

```
unsigned char PutEntry
              (Table *f, size_t i, size_t j, char *s);
```

In Zeile `i` und Spalte `j` wird das Wort `s` rechtsbündig eingetragen; gegebenenfalls wird die Spalten-Breite korrigiert.

Spalten überschreiben:

```
 unsigned char PutColumn
(Table *f, size_t j, Table *fi, size_t k=0, size_t anz=1);
 unsigned char PutColumn(Table *f, size_t j, char *s);
```

Ab der Spalte `j` in der Tabelle `f` werden die Einträge von `anz` Spalten mit den Einträgen ab der `k`-ten Spalte aus der Tabelle `fi` überschrieben. Die Operation wird nur ausgeführt, wenn die Spalten-Namen übereinstimmen. Beim 2. Aufruf werden die Wörter des Strings `s` in die Spalte `j` eingetragen; jedes Wort in dem String muss durch ein Leerzeichen vom folgenden getrennt sein. Der Spalten-Name darf nicht im String auftreten.

Zeilen überschreiben:

```
  unsigned char PutRow
 (Table *f, size_t i, Table *fi, size_t k=1, size_t anz=1);
```

Ab der Zeile `i` in der Tabelle `f` werden die Einträge von `anz` Zeilen mit den Einträgen ab der `k`-ten Zeile aus der Tabelle `fi` überschrieben.

Zeilen anfügen:

```
unsigned char AppendRow
             (Table *f, Table *fi, size_t i, size_t anz=1);
  unsigned char AppendRow(Table *f, char *s);
```

Aus der Tabelle `fi` werden ab Zeile `i` `anz` Zeilen an die Tabelle `f` angefügt. Die Spalten-Strukturen beider Tabellen müssen gleich sein. Bei der 2. Variante wird der auf `s` stehende String an die Tabelle angefügt. Die Tabelle muss mindestens den Tabellen-Kopf enthalten. Falls eine Spalte für einen Eintrag zu schmal ist, gehen führende Zeichen verloren.

Spalten anfügen:

```
  Table AppendColumn
        (Table *f, Table *fi, size_t j, size_t anz=1);
```

Aus der Tabelle `fi` werden ab `j`-ter Spalte `anz` Spalten an die Tabelle `f` angefügt. Beide Tabellen müssen gleiche Zeilen-Anzahl haben. Doppelspalten (Spalten mit gleichen Spalten-Namen) werden gestrichen.

Einfügen einer Zeile:

```
  unsigned char InsertRow(Table *f, size_t i, char *s);
```

Der angegebene String wird als `i`-te Zeile eingefügt.

Einfügen von Spalten:

```
  unsigned char InsertColumn
  (Table *f, size_t j, Table *fi, size_t k, size_t anz=1);
  unsigned char InsertColumn(Table *f, size_t j, char *s);
```

In die Tabelle `f` werden ab Spalte `j` aus der Tabelle `fi` ab Spalte `k` `anz` Spalten eingefügt, wobei Doppelspalten vermieden werden. Beim 2. Aufruf wird aus den Wörtern des Strings `s` eine Spalte gebildet und diese als Spalte `j` eingefügt. Als erstes Wort muss der String den Spalten-Namen enthalten.

Tabellen verbinden:

```
Table JoinTable
   (Table *f, size_t j, Table *ff, size_t k, size_t anz=1);
```

Aus den Tabellen `f` und `ff` wird eine neue Tabelle `fo` gebildet; wenn Tabelle `f` genau `r` Spalten hat, Tabelle `ff` genau `s` Spalten, so hat die neue Tabelle `fo` genau `r+s-anz` Zeilen. Zunächst müssen ab Spalte `j` in Tabelle `f` `anz` Spalten-Namen mit den Spalten-Namen ab Spalte `k` in der Tabelle `ff` übereinstimmen. Eine Zeile `u` aus der Tabelle `ff` wird unter Vermeidung von Doppelspalten an eine Zeile `v` aus der Tabelle `f` angehängt, falls die Einträge in den `anz` Spalten ab Spalte `k` in `v` mit den Einträgen in den `anz` Spalten ab Spalte `j` in `u` übereinstimmen.

Streichen von Zeilen:

```
unsigned char DeleteRow(Table *f, size_t i, size_t anz=1);
unsigned char DeleteDoubleRow(Table *f, size_t j);
```

Ab der Zeile `i` werden `anz` Zeilen gestrichen. Beim 2. Aufruf werden alle Doppelzeilen bezüglich Spalte `j` gestrichen.

Streichen von Spalten:

```
unsigned char DeleteColumn
         (Table *f, size_t j, size_t anz=1);
unsigned char DeleteDoubleColumn(Table *f);
```

Es werden ab Spalte `j` in der Tabelle `f` genau `anz` Spalten gestrichen. Beim 2. Aufruf werden alle Doppelspalten (Spalten mit gleichem Spalten-Namen) gestrichen.

Tabelle sortieren:

```
unsigned char SortTable(Table *f, size_t j=1);
```

Die Tabelle wird bezüglich der angegebenen Spalte aufsteigend sortiert.

Vertauschen von Zeilen:

```
unsigned char SwapRow(Table *f, size_t i, size_t k);
```

In der Tabelle werden die Zeilen `i` und `k` getauscht.

Vertauschen von Spalten:

```
unsigned char SwapColumn(Table *f, size_t j, size_t k);
```

In der Tabelle werden die Spalten `j` und `k` getauscht.

Spalte in einen String umwandeln:

```
size_t ColumnToLine(Table *f, size_t j);
```

Aus der Spalte `j` wird (ohne den Spalten-Namen) ein String erstellt und dieser auf `Output` ausgegeben.

Zeile in einen String umwandeln:

```
size_t RowToLine(Table *f, size_t i);
```

Aus der Zeile `i` wird ein String erstellt und dieser auf `Output` ausgegeben.

Spalten-Breite ändern:

```
unsigned char ChangeColumnWidth
              (Table *f, size_t j, size_t nwidth);
unsigned char SetColumnWidth(Table *f);
```

Die Breite der Spalte `j` wird auf den angebenen Wert `nwidth` gebracht; gegebenenfalls können dabei führende Zeichen verschwinden. Beim Aufruf von `SetColumnWidth` werden alle Spalten auf ihre optimale Breite korrigiert.

Spalten-Breite liefern:

```
size_t GetColumnWidth(Table *f, size_t j);
```

Es wird die optimale Spalten-Breite für die Spalte `j` ausgegeben.

Schreiben einer Tabelle in eine Datei:

```
unsigned char PrintTable
              (Table *f, char *name="", char *mode="w+");
```

Die in der Struktur vorhandene Tabelle wird in die Datei `name` geschrieben; auf `mode` ist der Öffnungsmodus für die Datei anzugeben. Falls `name` das Leer-Wort ist, wird die Standard-Ausgabe verwendet. Die Spalten-Einträge werden rechtsbündig ausgegeben.

3. Hilfsroutinen

Der während der Laufzeit eines Programms angeforderte Speicher-Platz wird beim Programm-Ende nicht freigegeben; aus diesem Grunde wurde eine kleine Speicher-Platz-Verwaltung geschaffen, die dieses kleine, aber wichtige Problem behebt.

Anfordern von Speicher-Platz:

```
char* Alloc(long int lc);
```

Es wird ein genulltes Feld mit `lc` Bytes zurückgegeben (oder `exit(1)` bei Mißerfolg); alle angeforderten Felder sind vorwärts verkettet.

Freigeben von Speicher-Platz:

```
void Free(void *c);
```

Das angegebene Feld wird ausgekettet und freigegeben; im Falle `c=0` werden alle Felder freigegeben.

Programm beenden:

```
void Return(int l);
```

Es werden alle Felder freigegeben; danach wird `exit(l)` aufgerufen.

Neue Tabelle:

```
char** NewTable(Table *f, size_t l);
```

Das Feld `Table` wird um `l` Zeilen-Einträge erweitert; der Rückkehrwert steht am Tabellen-Ende.

Groß- und Kleinschreibung ignorieren:

```
char* sstrstr(const char*, const char*);
int sstrcmp(const char*, const char*);
int sstrncmp(const char*, const char*, size_t);
```

Diese Routinen entsprechend den `C`-Routinen `strstr`, `strcmp` und `strncmp`, wobei hier die Groß- und Kleinschreibung unberücksichtigt bleibt; sie werden standardmäßig verwendet. Falls man dies nicht wünscht, ist zu setzen:

```
Strstr=&strstr; Strcmp=&strcmp; Strncmp=&strncmp;
```

4. Nutzungshinweise

In drei `h`-Dateien sind die für die Übersetzung des Systems nötigen Vorbereitungen getroffen: `defTable.h`, `defText.h` und `defTabText.h`; letztere ist für die Anwendung des Systems zu verwenden. Es ist zweckmäßig, eine Objektmodul-Bibliothek zu erzeugen; dazu dienen die beiden Dateien `all.bat` und `all.bash`. Unter `WINDOWS` wurde der `g++`-Compiler aus `Dev-Cpp` verwendet und eine Kopie unter dem Namen `CC.exe` im Ordner `Dev-Cpp\bin` angelegt. Hier ein kleines Beispiel, das sich in der Datei `tt_test.c` befindet:

```
/*:UNIX
 CC -O -o tt_test tt_test.c TabText.a
*/
#include "defTabText.h"
int main()
{ Text f=OpenText("aa.txt"); // Text einlesen
  Table fi, ti, si, ui;
  DeleteLine(&f);            // 1.Zeile streichen
  ReplaceString(&f,",");     // alle Kommata streichen
  PrintText(&f,"b.txt");     // Ausgabe auf Datei
  fi=OpenTable(&f);          // Tabelle aus Text erzeugen
  CloseText(&f);             // Text schließen
  ui=CopyTable(&fi);         // Tabelle kopieren
```

```
  DeleteColumn(&fi,1,4);     // erste 4 Spalten streichen
  Strstr=&strstr;            // Groß- und Kleinschribung
  Strcmp=&strcmp;            // beachten
  Strncmp=&strncmp;
  ti=GetRow(&fi,5,"*tT*.c"); // Zeilen-Auswahl nach Maske
  PrintTable(&ti);           // neue Tabelle anzeigen
  printf("****************************************\n");
  si=JoinTable(&ui,9,&ti,5); // Tabellen verbinden
  PrintTable(&si);
  Return(0);                 // alles freigeben, beenden
  return 0;                  // verhindert Warnung
}
```