# Teamarbeit (Team 2)

☞ **Analysieren Sie den Datensatzaufbau der XML - Datendatei!**

☞ **Stellen Sie Ihre Ergebnisse den anderen Teams vor!**

## Tankstellenkassensystem: Beschreibung zum Datensatzaufbau

Die Daten zum Tankstellenkassensystem liegen im XML Format als Datei vor. Die Datenbanktreiberklasse muss die Datensätze für die Artikel daher auslesen.

Die Daten liegen als Textdatei Daten.xml in folgendem Format vor:
(Das Textfile wird vom Tankstellenbetreiber gestellt!).

Kraftstoffe beginnen mit einer 100.000er Artikelnummer, Waren mit der Artikelnummer 200.100.

Die Artikel sind in einer Preisliste erfasst. Einzeldaten sind mit Tags umschlossen.

Als Einzeldaten werden Artikelnummer, Bezeichnung, Nettoeinkaufspreis und Handelsspanne (in Prozent) abgespeichert.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<Preisliste>
   <Artikel>
      <Artikelnummer>100000</Artikelnummer>
      <Bezeichnung>Super E10</Bezeichnung>
      <Preis>1.40</Preis>
      <Handelsspanne>30</Handelsspanne>
   </Artikel>
   …
   <Artikel>
      <Artikelnummer>100003</Artikelnummer>
      <Bezeichnung>Diesel</Bezeichnung>
      <Preis>1.40</Preis>
      <Handelsspanne>40</Handelsspanne>
   </Artikel>
   <Artikel>
      <Artikelnummer>200100</Artikelnummer>
      <Bezeichnung>Wrigleys Extra</Bezeichnung>
      <Preis>0.29</Preis>
      <Handelsspanne>50</Handelsspanne>
   </Artikel>
   …
   <Artikel>
      <Artikelnummer>200169</Artikelnummer>
      <Bezeichnung>Eisschokis</Bezeichnung>
      <Preis>2.40</Preis>
      <Handelsspanne>50</Handelsspanne>
   </Artikel>
</Preisliste>
```

☞ **Welche Daten liefert die hierarchische XML Datei?**
ArtNummer, Bezeichnung, Nettoeinkaufspreis, Handelsspanne in %

☞ **Nach welchem Schlüssel werden die Datensätze der Artikel gesucht?**
Artikelnummer

☞ **Wie kann zwischen Kraftstoff und Ware unterschieden werden?**
Anhand der Artikelnummer

☞ **Erläutern Sie Vor- und Nachteile der XML-Datenspeicherung !**

Da vom Kassensystem aus in der Regel auf einen ganzen Datensatz zugegriffen wird, benötigt die sequenzielle Datenbank einen Zwischenspeicher im Programm, in den der gerade angeforderte Datensatz eingelesen wird. Dazu bietet sich eine C++ Structure an:

```
struct DatenSatz
{
        long artNummer;
        char bez[25];

        double preis;

        int handelsspanne;

} ihrDatensatz;
```

☞ **Ergänzen Sie die C++ Structure!**

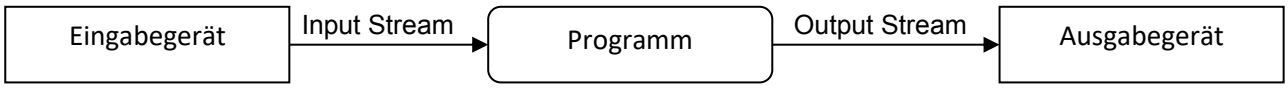
☞ **Erklären Sie den Zweck der C++ Structure!**

## Hilfestellung Arbeiten mit Ein- und Ausgabestreams in C++

## 1. Die Philosophie der Streamverarbeitung

**☞ Was versteht man unter einem Stream?**

Im Gegensatz zu vielen anderen Programmiersprachen stellt C/ C++ keine Sprachelemente zur Ein- und Ausgabe zur Verfügung. Unter C (und natürlich auch C++) erfolgt die Ein- und Ausgabe gemäß dem ANSI-Standard über Streams und entsprechende Funktionen (z.B. printf(...) oder scanf(...)) der Standard-Bibliotheken.
**Unter einem Stream ist dabei der Datenfluss zwischen einem Ein- oder Ausgabegerät und dem Programm zu verstehen.** Entsprechend der UNIX-Philosophie wird auch hier nicht zwischen sequentiellen Geräten (Tastatur, Monitor usw.) und filestrukturierten Datenträgern (Festplatte usw.) unterschieden.




## 2. Klassen für die Ein – und Ausgabe in C++

Grundlage für die Streamverarbeitung bildet die Standard Input/ Output Library mit den template Basisklassen **ios_base** und **basic_streambuf**. Mit Hilfe dieser und davon abgeleiteter Klassen werden folgende Ein- und Ausgabefunktionen ermöglicht:

- Ein- und Ausgabe eingebauter und benutzerdefinierter Datentypen,
- Verknüpfung von strings und streams,
- Formatsteuerung und Manipulatoren,
- Dateibearbeitung sowie
- Statusabfrage und Fehlerbehandlung.

Wichtige Streamklassen:

| **Eingabe** | **Ausgabe** |
|---|---|
| basic_istream (allgem. Input Stream) | basic_ostream (allgem. Output Stream) |
| cin (Standardeingabe) | cout (Standardausgabe)<br>cerr (Standardfehlerausgabe) |
| basic_ifstream (Input File Stream) | basic_ofstream (Output File Stream) |

**Kombination für Ein- und Ausgabe**

| | |
|---|---|
| basic_iostream | basic_fstream |

## 3. Ein- und Ausgabe mit Streams

### 3.1 Eingabe:

In der Klasse basic_istream ist für die formatierte Eingabe der Input-Operator **>>,** den man mit lies von übersetzten könnte, für folgende Datentypen überladen:
unsigned und signed char*, unsigned und signed char, short und unsigned short, int und unsigned int, long und unsigned long, float, double und long double.
Zur unformatierten Eingabe sind folgende Funktionalitäten in basic_istream definiert:

| **Methode** | **Beschreibung** |
|---|---|
| get | liest Daten vom stream bis zum Begrenzungszeichen |
| getline | liest bis zum Zeilenende und entfernt Begrenzungszeichen |
| gcount | ermittelt, wieviel Zeichen bisher gelesen wurden |
| read | liest Daten und speichert sie in ein char-array (binäres Lesen) |
| peek | liest ein Zeichen |
| ignore | überspringt n Zeichen |
| tellg | liefert aktuellen Wert des Lesezeigers |
| seekg | setzt den Lesezeiger |
| putback | setzt gelesenes Zeichen wieder in den stream ein |
| sync | synchronisiert Puffer und Zeichenquelle |

Alle Methoden können mit dem Punktoperator aufgerufen werden (z.B. cin.get(...)).

### 3.2 Ausgabe:

Für die formatierte Ausgabe ist der Output-Operator **<<**, den man mit schreib in übersetzten könnte, in der Klasse basic_ostream für die folgenden eingebauten Datentypen überladen:
unsigned und signed char*, unsigned und signed char, short und unsigned short, int und unsigned int, long und unsigned long, float, double und long double sowie void*
Zur unformatierten Ausgabe sind folgende Funktionalitäten in basic_ostream definiert:

| Methode | Beschreibung |
|---|---|
| write | schreibt Zeichen in den stream |
| put | schreibt ein Zeichen in den stream |
| tellp | liefert aktuellen Wert des Schreibzeigers |
| seekp | setzt den Schreibzeiger |
| flush | entleert den dem stream zugewiesenen Puffer |

Die Methoden können wieder mit dem Punktoperator aufgerufen werden (z.B. cout.put(...)).

## 4. Arbeiten mit Dateien

### 4.1 Öffnen und Schließen von Streams

Zum Arbeiten mit Dateien (Files) bietet die Standardheaderdatei <fstream> eine Reihe von Funktionalitäten und Möglichkeiten. Grundlage der Dateibearbeitung sind dabei die vordefinierten Klassen vom Typ **ifstream** (Eingabestrom) bzw. **ofstream** (Ausgabestrom).

Um auf eine Datei zuzugreifen erzeugt man einfach ein Objekt vom Typ ifstream (z.B. ifstream quelle;) oder ofstream (ofstream ziel;).
Mit der Methode open(…) kann dann eine Quell- oder Zieldatei geöffnet werden.
Die Methode close( ) zum Schließen der Datei sollte explizit aufgerufen werden, damit die Daten aus dem Streambuffer sofort in die Datei geschrieben werden. Ansonsten erfolgt der Aufruf erst beim Destruktoraufruf des Fileobjekts!

Zum Arbeiten mit Filestreams sind in C++ folgende Modi vorhanden.

| Modus | Beschreibung |
|---|---|
| ios_base::in | Datei für Lesezugriff öffnen |
| ios_base::out | Datei für Schreibzugriff öffnen (Überschreibmodus, ggf. neues Anlegen der Datei) |
| ios_base::ate | Datei öffnen und an Dateiende gehen |
| ios_base::app | Datei öffnen und hinten anhängen |
| ios_base::trunc | vorhandene Datei öffnen und auf 0 Byte kürzen |
| ios_base::binary | Datei binär ohne Interpretation des Inhalts öffnen (Escape Sequenzen werden ignoriert) |

**Beispiel:**
```
ifstream quelle;
quelle.open("C:\\Testordner\\Test.txt", ios_base::in | ios_base::binary);
…
quelle.close( );
```

Die Datei Test.txt wird zum lesen geöfnet. Der bitweise ODER Operator (Pipezeichen) kombiniert den Lesemodus mit dem Binärmodus.

### 4.2 Statusabfrage und Fehlerbehandlung

Zur Kontrolle und zur Steuerung aller bisher vorgestellten Funktionen bezüglich der Streamverarbeitung stellt die Klasse basic_ios spezielle Funktionen bereit. Mit Hilfe der Abfrage der entsprechenden Flags lassen sich Fehler und Zustände bei der Arbeit mit Streams erkennen:

Methoden zum Abfragen und Setzen der Statusbits:

| Methode | Beschreibung | Bitwert |
|---|---|---|
| good( ) | liefert TRUE (!=0), falls alles in Ordnung, d.h. keines der Fehlerflags gesetzt ist | goodbit = 0 |
| eof( ) | TRUE falls Dateiende (EOF) erreicht ist | eofbit = 1 |
| fail( ) | TRUE, wenn letzte Operation fehlerhaft,<br>= badbit \| failbit | failbit = 2 |
| bad( ) | TRUE, wenn schwerwiegender Fehler aufgetreten ist | badbit = 4 |
| rdstate( ) | liefert Status des streams als Integer zurück,<br>= eofbit \| failbit \| badbit (Kombination der drei Werte) | |
| clear( ) | setzt gezielt ein oder alle drei Bits gleichzeitig zurück | |

Beispiel: `if(quelle.good( )) // liefert != 0 wenn kein Fehlerbit des Streams quelle gesetzt ist.`

**Hilfestellung zu bool SucheDatensatz(long no)**

Zuerst benötigt man drei lokale Variablen zur Zwischenspeicherung einer ganzen Zeile sowie als Puffer für die eigentlichen Daten der XML Datei. Artnum dient als Puffer für die Artikelnummer.
Die Funktion strtok_s(…) wird später zum Teilen der gelesenen Zeile genutzt und benötigt einen Zwischenpuffer next_token.

Vor der eigentlichen Suchfunktion müssen die Fehlerflags des Streams zurückgesetzt und der Dateizeiger an den Dateianfang gesetzt werden.

Mit ignore(…) überspringt man dann zunächst den Kopf und die Start Tags der XML Datei bis zum ersten Artikel. ignore(…) überspringt maximal 128 Zeichen, stoppt aber bei einem Zeilenumbruch.

getline(…) liest die Zeile mit der Artikelnummer in den Puffer.

Die Suche nach der Artikelnummer beginnt in der while – Schleife (solange das Dateiende nicht erreicht ist… siehe eof()).
Mit strtok(…) zerlegt man die Datenzeile bis zum Ende der ersten Tag Markierung („>“). In der folgenden Zeile kopiert man die Artikelnummer in den Datenpuffer (bis zum ersten Tag Ende Zeichen („<“).
Die Artikelnummer befindet sich nun als String in dataBuffer. Um einen Vergleich mit der Artikelnummer no im Format long anstellen zu können, wird dataBuffer mit atol( ) aus der standard Header – Datei <cstdlib> in einen long umgewandelt. Ist ein Artikel gefunden (artnum == no), so werden die Daten in die Structure geschrieben.

Die Bezeichnung ist auch in der Structure ein String und muss nicht umgewandelt werden.
Anders sieht es beim Nettopreis aus. Dieser wird zunächst in einen Puffer geschrieben und mit atof( ) in einen double verwandelt.
Das Einlesen der HSP verläuft analog zum Nettopreis.
Wird die Artikelnummer gefunden, so verlässt man mit return true die Methode SucheDatenSatz(...).

Stimmen artnum und no nicht überein, so wird mit ignore(...) bis zum Datensatzende weitergesprungen. getline(…) liest das nächste Artikeldaten tag ein, oder überschreitet das Dateiende.
Die while – Schleife beginnt erneut, solange das dateiende nicht erreicht ist und sucht im nächsten Datensatz der xml Datei.

Wird kein Artikel mit der richtigen Artikelnummer gefunden, so startet man eine Warnmeldung und gibt false zurück.

```
bool ArtDatenBank::SucheDatenSatz(long no)
{
   char buffer[128]; // Puffer für eine Zeile
   char dataBuffer[25];// Puffer für die Teildaten
   long artnum = 0;

   // Puffer für den Reststring nach strtok_s(...)
   char* next_token = NULL;


   // Fehlerflags zurücksetzen
   ihreDaten.clear( );

   // Streamzeiger auf Dateianfang
   ihreDaten.seekg(0, ios_base::beg);

   ihreDaten.ignore(128, '\n'); // XML Kopf
   ihreDaten.ignore(…); // Preisliste Tag
   ihreDaten. …; // Artikel Tag



   // Zeile mit Artikelnummer Tag lesen
   ihreDaten.getline(buffer, 128);

   // bis zum Dateiende
   while (ihreDaten. ...)
   {
     // Start Tag extrahieren
     strcpy_s(dataBuffer, strtok_s(buffer, ">", &next_token));
     // Artikelnummer extrahieren
     strcpy_s(dataBuffer, strtok_s(NULL, "<", &next_token));


     artnum = atol(dataBuffer);


     // Artikel gefunden
     if (artnum == no)
     {
        ihrDatensatz.artNummer = artnum;


       // Bezeichnung
       ihreDaten.getline(buffer, 128);
       strcpy_s(dataBuffer, strtok_s(buffer, ">", &next_token));
       strcpy_s(… , strtok_s( … ));
       // Preis
       ihreDaten.getline(…);
       strcpy_s( … );
       …
        ihrDatensatz.netto = atof(dataBuffer);
        // Handelsspanne
        …
        return true;
     }
     ihreDaten.ignore(…); // Artikelende Tag
     ihreDaten.ignore(…); // Artikel Tag oder Preisliste Ende
     ihreDaten.getline(…);// Artikeldaten Tag oder EOF erreicht
   }

   cout << "Artikel nicht gefunden!\n";

   return false;
}
```