Notizen AIK ProgTech 3: File-I/O

Klaus Kusche

Klassen

Die für uns relevanten Klassen sind **`ifstream`** (Input File Stream) und **`ofstream`** (Output File Stream), ev. auch **`fstream`** (beide Richtungen), alle drei aus dem Header **`fstream`**.

Alle drei sind Ableitungen speziell für Dateien von den Klassen **`istream`**, **`ostream`** bzw. **`iostream`** (alle drei aus dem Header **`iostream`**), den allgemeinen Klassen für Ein- und Ausgabe, die auch das Terminal, Stringstreams (siehe unten) usw. umfassen.

`cin` ist ein vordefiniertes **`istream`**-Objekt, **`cout`** und **`cerr`** sind **`ostream`**-Objekte.

File öffnen und schließen

Vor der Ein- oder Ausgabe muss ein File *geöffnet* werden. Normalerweise geschieht das gleich bei der *Deklaration* durch einen *Konstruktor-Aufruf* mit dem *Filenamen*.

Beispiel: **`ifstream inFile(argv[1]);`**

Alternativ ist es möglich, explizit die Methode **`open`** aufzurufen:
`inFile.open(argv[1]);`

In beiden Fällen sollte man unmittelbar danach *auf Fehler prüfen*
(Datei nicht vorhanden, fehlende Rechte, ...):
`if (!inFile) {` ...

Optional kann man in beiden Fällen als zweiten Parameter *Flags* angeben
(als Textfile mit **`\r\n`**-Konvertierung oder als Binärfile öffnen, beim Schreiben: Überschreiben oder anhängen, ...) (nicht Prüfungsstoff).

Beim *Destruktor* wird der File implizit *geschlossen*. Man kann auch explizit die Methode **`close()`** aufrufen (z.B. wenn man sicher sein will, dass alles geschrieben ist, oder wenn man für dasselbe File-Objekt ein **`open`** auf eine andere Datei machen möchte).

Ein- und Ausgabe

1.) **Formatiertes File-I/O mit Konvertierung: `<<` und `>>`**

Funktion und Verwendung wie bisher von **`cin`** und **`cout`** bekannt.

Achtung beim Lesen:

- *Zwischenräume* und *Zeilenvorschübe* werden *ignoriert* bzw. überlesen.
- Jedes **`>>`** verarbeitet *genau ein Wort*: Es ist nicht möglich, mit **`>>`** eine ganze Zeile oder einen String mit Zwischenräumen zu lesen.
- Das Zeichen, das die Eingabe beendet (Zeilenvorschub, Leerzeichen oder das erste nicht konvertierbare Zeichen), bleibt in der Eingabedatei *erhalten* und muss von der nächsten Input-Operation verarbeitet oder überlesen werden!

2.) **Zeichenweises File-I/O**: **`get`** und **`put`**

Beide lesen bzw. schreiben *genau ein Zeichen*,
incl. Zwischenräume, Zeilenvorschübe usw..

Beispiele: **`c = inFile.get();`** (oder **`inFile.get(c);`**)
und **`outFile.put(c);`**

3.) **Zeilenweises File-I/O**: **`getline`**

Es gibt zwei **`getline`**, beide lesen *eine* Zeile bis zum **`\n`**:

- Die *Methode* **`getline`** liest eine Zeile in einen *C-String* (**`char`**-Array) mit der angegebenen Maximal-Länge. Sie hängt automatisch ein **`\0`** an.

  Beispiel:
  ```
  char zeile[81];
  inFile.getline(zeile, 81);
  ```

- Die globale *Funktion* **`getline`** liest eine Zeile in ein *C++-***`string`***-Objekt*.

  Beispiel:
  ```
  string zeile;
  getline(inFile, zeile);
  ```

Das **`\n`** am Ende wird von beiden gelesen,
aber *nicht* in den Ergebnis-String gespeichert.

Für die *zeilenweise Ausgabe* gibt es keine eigene Methode,
es wird einfach **`<<`** verwendet.

4.) **Binäres File-I/O**: **`read`** und **`write`**

`read` und **`write`** lesen bzw. schreiben eine *vorgegebene Anzahl von Bytes* in ein bzw. aus einem *Array*, ohne Rücksicht auf Zwischenräume, Zeilen oder Nullbytes.

Nicht Prüfungsstoff.

File-Positionierung

Jedes File-Objekt merkt sich intern, bis zum *wievielten Byte* die Datei schon gelesen wurde bzw. beim wievielten Byte der Datei das Schreiben gerade steht. Diese Positionen lassen sich getrennt für Lesen und Schreiben auch abfragen (**`tellg`** und **`tellp`**) und (wenn es sich um eine "echte" Datei handelt und nicht um das Terminal) auch ändern (**`seekg`** und **`seekp`**).

Man kann dadurch *an einer beliebigen anderen Stelle* der Datei (weiter vorne oder weiter hinten) weiterlesen oder weiterschreiben oder z.B. die Datei nochmals von Anfang an lesen.

Nicht Prüfungsstoff.

Fehlerbehandlung

- Die meisten I/O-Methoden zeigen das Auftreten eines Fehlers oder das Fileende mittels *Returnwert* an. Fast alle Methoden geben einfach *das File-Objekt selbst* zurück, das dann mit **`!`** ("nicht") auf *Fehler* geprüft bzw. *direkt als Bedingung* verwendet werden kann (siehe unten).
- Jedes File-Objekt hat intern *3 Statusbits*,
  in denen es sich die bisher aufgetretenen Fehler merkt.

Es gibt *3 Methoden*, um jedes dieser Statusbits explizit zu prüfen:

- **`eof()`** … bei einer der *vorangehenden* Eingabe-Operationen wurde das *Fileende* erreicht.
- **`fail()`** … bei einer der vorangehenden Eingabe-Operationen ist ein *Konvertierungsfehler* aufgetreten (z.B. Zahl erwartet und Text gefunden, oder Zeile vom File zu lang für den String, …).

  Weiters wird **`fail`** gesetzt, wenn beim letzten Lesen schon das *Fileende* erreicht war, *bevor* irgendetwas gelesen und konvertiert werden konnte.
- **`bad()`** … es ist ein *echter I/O-Fehler* aufgetreten (Platte voll, Daten defekt, USB-Stick entfernt, …).

- Daneben gibt es 2 "kombinierte" Prüf-Methoden:
  - **`good()`** … keiner der 3 Fehler **`eof`**, **`fail`** oder **`bad`** .
  - **`!`** (der "not"-Operator) … **`fail`** oder **`bad`** .

    `!` ist der übliche, am meisten verwendete Test!
  - Weiters gibt es *implizite Typumwandlungen*, die dafür sorgen, dass ein File-Objekt (bzw. eine Funktion oder Methode, die ein File-Objekt als Returnwert liefert) *direkt als Bedingung* in einem **`if`**, **`while`** usw. verwendet werden kann. Das Ergebnis ist die Negation des **`!`** , d.h. ist **`fail`** oder **`bad`** gesetzt, ist die Bedingung "falsch", sonst "wahr".

    Beispiel: **`while (inFile)`** { …
    oder **`while (inFile.getline(zeile, 81))`** { …
    oder **`while (inFile >> txt) {`** …
- Optional kann beim Auftreten eines Fehlers eine *Exception* ausgelöst werden.
- Einmal gesetzte Fehlerbits **bleiben erhalten** (auch über nachfolgende *erfolgreiche* I/O-Aufrufe hinweg!!!), bis man sie durch Aufruf der Methode **`clear()`** explizit *zurücksetzt*.
- *Korrekte Fehlerbehandlung* in C++ ist nicht ganz einfach:
  - **`eof`** ***ist unpraktisch***: Man kann mit **`eof`** allein *nicht* feststellen, ob die letzte Leseoperation noch *erfolgreich* alle Daten gelesen hat und erst *danach* der File zu Ende war, oder ob der File schon zu Ende war, *bevor* die gewünschen Daten gelesen werden konnten.

    Korrekterweise prüft man ***nicht*** **auf `eof`**, sondern macht seine Input-Schleife, bis **`fail`** oder **`bad`** auftritt (z.B. **`if (!inFile) break;`**).

    Erst *nach der Schleife* prüft man **`eof`**: Ist es gesetzt, war der File zu Ende, ist es nicht gesetzt, ist ein "echter" Fehler aufgetreten.
  - Geht eine Leseoperation (**`>>`** oder **`getline`**) wegen **nicht konvertierbarem oder zu langem Input** schief, so bleibt der ***defekte Input erhalten***: Das nächste Lesen versucht wieder, denselben falschen Input zu konvertieren, und schlägt wieder fehl ==› das Programm läuft meist in eine *Endlos-Schleife*.

- Wenn das Programm falsche Eingaben korrekt behandeln soll, muss *jeder einzelne Eingabe-Aufruf* in eine ***Schleife*** verpackt werden, die folgende Schritte umfaßt:
  - Den Eingabe-Operator (**>>**, **`getline`**, …) aufrufen.
  - Wenn erfolgreich: Eingabe-Prüf-Schleife verlassen.
  - Wenn **`bad`**: Fehlermeldung, Programm beenden.
  - Wenn **`eof`**: Je nach Programmlogik, wiederholen sinnlos!
  - **`clear`** aufrufen, um den **`fail`**-Status zurückzusetzen.
  - **`ignore`** aufrufen, um die fehlerhaften Eingabedaten zu überlesen (meist: **`inFile.ignore(10000, '\n');`**).
  - Ev. Meldung an den Benutzer und neuen Input-Prompt ausgeben.
  - Nächster Schleifendurchlauf: Das **>>** noch einmal probieren…

  Die Fehlerprüfung eines **>>** ist nicht Prüfungsstoff,
  aber in der Praxis bei allen "ernsthaften" Programmen zu verwenden!

Formatierung

Für die Formatierung der Ein- und Ausgabe ist der Header **`iomanip`** zuständig.

Er enthält u.a. Konstrukte, um folgende Eigenschaften von **<<** und **>>** zu ändern:

- Fixe *Feldbreite*, Ausgabe *links- oder rechtsbündig* im Ausgabe-Feld, Auffüllen mit *führenden Nullen*.
- Anzahl der *Nachkomma-Stellen*, Exponenten-Notation.
- Zahlenbasis (*Hexadezimal*, oktal, …).

Die Formatierungsbefehle werden zwischen die Ein- und Ausgabewerte in **<<** und **>>** eingefügt. Details siehe Doku von **`iomanip`**.

Achtung:
Manche (z.B. links- und rechtsbündig) gelten, bis man sie explizit wieder *zurücksetzt*, andere (z.B. Feldbreite) gelten nur für den *nächsten Wert*!

Die Klasse `stringstream`

Ein **`stringstream`**-Objekt (Header **`sstream`**) verhält sich wie ein **`iostream`**-Objekt: Man kann mit **<<**, **>>** usw. herauslesen oder hineinschreiben, und die Werte werden wie bei normaler Ein- und Ausgabe konvertiert bzw. formatiert.

Hinter einem **`stringstream`**-Objekt steht aber weder das Terminal noch eine Datei, sondern ein ganz normaler, interner *String* (C++ **`string`**, nicht C **`char`**-Array). Dieser String kann jederzeit mit der Methode **`str()`** ausgelesen oder mit der Methode **`str(s)`** auf einen bestimmten Inhalt gesetzt werden (man kann auch gleich in der *Deklaration* des **`stringstream`**-Objektes angeben, welchen Inhalt der String am Anfang haben soll).

Ein **`ostringstream`** schreibt formatierten Output in den String,
ein **`istringstream`** liest und konvertiert Werte aus einem String,
und bei einem **`stringstream`** kann man zuerst Werte in den String
formatiert ausgeben und dann aus demselben String wieder einlesen.

`stringstream`'s werden u.a. in folgenden Fällen verwendet:

- Zur *Umwandlung beliebiger Werte in Texte und umgekehrt*
  (z.B. Verwandlung eines **`double`** oder eines **`int`** in einen String), wenn der Text dann nicht gleich ausgegeben, sondern weiterverarbeitet werden soll.
- Zur *sicheren formatierten Eingabe*: Wenn man mit **`>>`** direkt vom Terminal oder aus einer Datei einliest, ist das Wiederaufsetzen nach einer fehlerhaften Eingabe (oder z.B. einer Eingabe-Zeile mit einem Wert zuviel oder zu wenig) schwierig und unzuverlässig (siehe oben).

  Viel sicherer und einfacher ist es, mit **`getline`** genau *eine Zeile* zu lesen, diese in einen **`istringstream`** zu verwandeln, und aus diesem mit **`>>`** zu lesen: Im *Fehlerfall* liest man einfach die *nächste Zeile* und probiert es *damit nochmals*. Auf diese Weise kann es nicht passieren, dass man plötzlich halbe Zeilen liest oder bei einem Fehler zu viel bzw. zu wenig Daten verwirft.