Ergänzungen zur Ein/Ausgabe

ein bezeichnet einen Eingabestrom (z.B. `cin`) und *aus* einen Ausgabestrom (z.B. `cout`, `cerr`).

- **Stromzustand**

  Ein/Ausgabeströme haben einen Zustand, der durch Zustandsbits beschrieben und mit Komponentenfunktionen abgefragt bzw. verändert werden kann. *(Auszug)*

`ein.good()`	liefert `true`, falls nächste Operation gelingen könnte
`ein.eof()`	liefert `true` bei Stromende
`ein.fail()`	liefert `true` bei Fehlern
`ein.bad()`	liefert `true` bei schweren Fehlern (Datenverlust)
`ein.clear()`	löscht Fehlerflags (danach liefert `ein.good()` den Wert `true`) *Evtl. ist das Löschen unverarbeiteter Zeichen im Eingabestrom notwendig:* `ein.ignore(numeric_limits<streamsize>::max(),'\n')`

  An Stelle der Abfrage `ein.fail()==false` kann die Abfrage `ein!=0` treten. (Bei dem Vergleich wird `ein` in einen Zeiger vom Typ `void*` [$\hat{=}$ Adresse] konvertiert.)

  Bsp.: Interaktive Eingabe in Großbuchstaben ausgeben (Beenden mit Ctrl-D [Unix])

```
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
  char c;
  cin >> noskipws; // Zwischenraum nicht ueberlesen
  while (cin >> c) cout << static_cast<char>(toupper(c));
  return 0;
}
```

- **Unformatierte Ein/Ausgabe**

  Mit den folgenden auszugsweise aufgeführten Komponentenfunktionen, die zum Teil C-Funktionen entsprechen, kann zeichenweise von einem Eingabestrom gelesen bzw. auf einen Ausgabestrom geschrieben werden. Im folgenden bezeichnet *c* ein Zeichen vom Typ `char` und *cs* eine genügend lange C-Zeichenkette (vom Typ `char []`).

`ein.get()`	liest das nächste Zeichen und liefert es als `int` bzw. im Fehlerfall `EOF`
`ein.get(c)`	liest nächstes Zeichen auf *c* ein und liefert Zust. von `ein` als `void*`
`ein.unget()`	stellt das zuletzt gelesene Zeichen in die Eingabe zurück
`ein.peek()`	liefert das nächste Zeichen als `int` bzw. `EOF` ("vorausschauen") (entspricht `get`, gefolgt von `unget`)
`ein.read(cs,n)`	liest höchstens n Zeichen bis zum `EOF` und legt sie in `cs` ab
`ein.getline(cs,n)`	liest höchstens $n-1$ Zeichen bis einschließlich `\n`, legt aber `\n` nicht in *cs* ab; *cs* wird mit `\0` abgeschlossen *Zum Einlesen von Zeilen ist im allg. die Funktion* `getline(ein,s)` *(→ Inf.bl.8, S.4) viel bequemer!*
`ein.gcount()`	liefert die Anzahl der Zeichen, die die zuletzt aufgerufene unformatierte Eingabefunktion gelesen hat
`ein.tellg()`	Liefert aktuelle Eingabestromposition (vom Datentyp `ios::pos_type`)
`ein.seekg(p)`[1]	Positioniert auf Eingabestromposition p (vom Datentyp `ios::pos_type`)
`aus.put(c)`	gibt *c* aus und liefert den Zustand von `cout` als `void*`
`aus.tellp()`	Liefert aktuelle Ausgabestromposition (vom Datentyp `ios::pos_type`)
`aus.seekp(p)`[1]	Positioniert auf Ausgabestromposition p (vom Datentyp `ios::pos_type`)

[1] Nur anwendbar auf Dateiströme und evtl. Stringströme

Ein/Ausgabe auf Dateien (`fstream`)

Das Öffnen von Dateien erfolgt durch Konstruktoraufruf (z.B. implizit in Definitionen) mit dem Dateinamen als Argument (vom Typ `string` oder `char []`) und optionaler Angabe eines Zugriffsmodus (Oder-Ausdruck von Zugriffflags). Nicht alle Kombinationen sind zulässig, die in der C-Funktion `fopen` zulässigen Zugriffsmodi (z.B. `r+b`) besitzen Entsprechungen.

Zugriffflag	*Bedeutung*
`ios::in`	Lesen
`ios::out`	Schreiben
`ios::append`	beim Schreiben anhängen
`ios::ate`	beim Öffnen an das Ende positionieren ("at end")
`ios::trunc`	zu Beginn löschen
`ios::binary`	Binärzugriff

Im folgenden bezeichne *fname* den Dateinamen (vom Typ `string`[1]), *mode* einen Zugriffsmodus, *ein* einen Ein-, *aus* eine Aus-, *einaus* einen Ein/Ausgabedateistrom und *strom* einen Dateistrom.

Operation	*Wirkung*
`ifstream ein(fname)`[2]	Definition und Öffnen zum Lesen (Modus: `ios::in`)
`ifstream ein(fname,mode)`	Definition und Öffnen zum Lesen (Modus: `ios::in|`*mode*)
`ofstream aus(fname)`[2]	Def. und Öffnen zum Schreiben (Modus: `ios::out`[3])
`ofstream aus(fname,mode)`	Def. und Öffnen zum Schreiben (Modus: `ios::out|`*mode*[4])
`fstream einaus(fname)`[2]	Def. und Öffn. zum Les. und Schreib. (Modus: `ios::in|ios::out`)
`fstream einaus(fname,mode)`	Definition und Öffnen im Modus *mode*
`strom.open(fname)`	Öffn. entspr. voreing. Modus für Typ von *strom* (kein Rückg.w.)
`strom.open(fname,mode)`	Öffnen im Modus *mode* (kein Rückgabewert)
`strom.close()`	Schließen (kein Rückgabewert)
`strom.is_open()`	Abfrage Stromzustand

Bsp.: Binärdatei kopieren

```
#include <string>
#include <iostream>
#include <fstream>
#include <cerrno>
#include <cstring>

using namespace std;

int main()
{
  string s; char c;
  cout << "Eingabedatei: "; cin  >> s;
  ifstream ein(s,ios::binary);
  if (!ein) { cerr << strerror(errno) << ": " << s << endl; return 1; }
  cout << "Ausgabedatei: "; cin  >> s;
  ofstream aus(s,ios::binary);
  if (!aus) { cerr << strerror(errno) << ": " << s << endl; return 1; }
  while(ein.get(c)) //  ein >> noskipws;
    aus.put(c);     //  while(ein >> c) aus << c;
  return 0;
}
```

[1] Vor C++11 `char[]` erforderlich

[2] parameterlose Vereinbarung ohne Klammern ebenfalls zulässig

[3] wirkt wie `ios::out|ios::trunc`

[4] wirkt für *mode*=`ios::binary` wie `ios::out|ios::binary|ios::trunc`

Ein/Ausgabe auf Strings (`sstream`)

Es ist möglich, Daten von Strings zu lesen oder auf Strings auszugeben. An die Stelle von Dateistromklassen treten dann Stringstromklassen.

Bsp: Stromklassen für Strings

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main()
{
  int i;
  istringstream ein;    // alternativ fuer beide Zeilen:
  ein.str("123");       // istringstream ein("123");
  ein >> i;             // i hat jetzt den Wert 123

  string s;
  ostringstream aus;
  aus << i;
  s = aus.str();        // s enthaelt den String 123
  return 0;
}
```

Bsp.: Mehrfachaufruf von str() für identische Stringstreamobjekte ist heikel

```
               :
int main()
{
  int i;
  istringstream ein;
  ein.str("123");
  ein >> i;         // i hat jetzt den Wert 123

  ein.clear();      // EOF loeschen (!)
  ein.str("45");
  ein >> i;         // i hat jetzt den Wert 45 (und nicht 123)

  i=123;
  string s;
  ostringstream aus;
  aus << i;
  s = aus.str();    // s enthaelt den String 123

  i=45;
  aus.str("");      // gespeicherten String loeschen (!)
  aus << i;
  s = aus.str();    // s enthaelt jetzt den String 45 (und nicht 12345)
  return 0;
}
```

Pufferung

Mit den STL-Strömen ist immer ein Puffer verbunden, der Zeichen zwischenspeichern kann. Das geschieht aus Gründen der Effizienz und ermöglicht bei der Eingabe ein „Vorausschauen“ mit `peek` und auch das „Zurückstellen“ einzelner Zeichen mit `unget` oder `putback`.

Ein Datenstrom kann vollständig, zeilenweise oder gar nicht gepuffert sein, d.h. dass die Daten blockweise dem Betriebssystem zur Ein/Ausgabe übergeben werden können. Bei vollständiger bzw. zeilenweiser Pufferung soll das erfolgen, wenn ein Puffer gefüllt ist oder ein `'\n'` im Datenstrom vorgefunden wird. Ungepufferte Daten sollen so bald wie möglich übertragen werden.

`cin` und `cout` sind beim Programmstart genau dann vollständig gepuffert, wenn sie *nicht* nicht mit einem interaktiven Gerät verbunden sind. `cerr` ist beim Programmstart nie vollständig gepuffert. (Unix: `cerr` ist immer ungepuffert. `cin`, `cout` sind zeilenweise gepuffert, falls interaktiv verbunden; andernfalls vollständig gepuffert.)

Leeren eines Ausgabepuffers bedeutet, dass die Daten im Puffer dem Betriebssystem zur Ausgabe übergeben werden; Leeren eines Eingabepuffers hingegen, dass die darin enthaltenen Daten weggeworfen werden.

Die Komponentenfunktion oder der Manipulator `flush` bewirken das Leeren des Puffers. Der Manipulator `endl` schreibt das Newline-Zeichen in den Puffer *und* leert ihn auch.

Bsp.: (Ubuntu Linux 20.04, g++-9.4)

```
#include <iostream>
#include <unistd.h>       // wegen sleep

using namespace std;

int main()
{ cout << "Ausgabe";      // Ausgabe erst am Programmende (nach 10 Sek.)
  sleep(10);              // Keine Verzoegerung, falls cout << "..." << flush;
  return 0;               // oder cout << "..." << endl;
}
```