## Ein/Ausgabe in C++ (`iostream`,`iomanip`) [Auszug Programmieren I]

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

- **Formatierte Ausgabe**

  *Syntax:* `aus <<` *Argument1* `<<` *Argument2* `<<` ...

  Die Formatierung der Ausgabe kann durch sog. Manipulatoren beeinflußt werden. *(Auszug)*

| *Manipulator* | *Wirkung* | *Bemerkungen* |
|---|---|---|
| `setw(n)`[1] | Min. Feldbreite (Voreinst.: `0`) | `#include <iomanip>` |
| `setprecision(n)`[2] | Ziffern nach Dezimalpunkt (Voreinst.: `6`) | `#include <iomanip>` |
| `setfill(c)` | Füllzeichen `c` (Voreinst.: `'␣'`) | `#include <iomanip>` |
| `left` | linksbündig | |
| `right` | rechtsbündig (Voreinst.) | |
| `internal` | Vorzeichen linksbündig, Zahl rechtsbündig | |
| `showpos` | pos. Vorzeichen ausgeben | |
| `noshowpos` | pos. Vorzeichen nicht ausgeben (Voreinst.) | |
| `fixed` | Festpunktformat bei Gleitpunktzahlen | |
| `scientific` | Exponentialformat bei Gleitpunktzahlen | |
| `dec` | Dezimalausgabe bei ganzen Zahlen (Voreinst.) | |
| `oct` | Oktalausgabe bei ganzen Zahlen | |
| `hex` | Hexadezimalausgabe ganzen Zahlen | |
| `showbase` | führende `0` (oktal), `0x`, `0X` (hexadezimal) bei ganzen Zahlen ausgeben | |
| `noshowbase` | keine Zahlsystemangabe (Voreinst.) | |
| `boolalpha` | logische Werte als Zeichenkette ausgeben | |
| `noboolalpha` | logische Werte als Zahlen ausgeben(Voreinst.) | |
| `endl` | Zeilenvorschub | |
| `flush` | Ausgabepuffer leeren | |

  Statt durch Manipulatoren können Formatflags auch durch `aus.setf` bzw. durch Komponentenfunktionen gesetzt werden.

| | |
|---|---|
| `aus.width(n)`[1] | Min. Feldbreite |
| `aus.precision(n)`[2] | Ziffern nach Dezimalpunkt |
| `aus.fill(c)` | Füllzeichen `c` (Voreinst.: `'␣'`) |
| `aus.setf(ios::left,ios::adjustfield)` | linksbündig |
| `aus.setf(ios::right,ios::adjustfield)` | rechtsbündig |
| `aus.setf(ios::internal,ios::adjustfield)` | Vorz. linksbündig, Zahl rechtsbündig |
| `aus.setf(ios::showpos)` | pos. Vorzeichen ausgeben |
| `aus.unsetf(ios::showpos)` | pos. Vorzeichen nicht ausgeben |
| `aus.setf(ios::fixed,ios::floatfield)` | Festpunktformat bei Gleitpunktzahlen |
| `aus.setf(ios::scientific,ios::floatfield)` | Exponentialformat bei Gleitpunktzahlen |
| `aus.unsetf(ios::floatfield)` | Allgemeinformat bei Gleitpkt.zahlen (Voreinst.) |
| `aus.setf(ios::dec,ios::basefield)` | Dezimalausgabe bei ganzen Zahlen |
| `aus.setf(ios::oct,ios::basefield)` | Oktalausgabe bei ganzen Zahlen |
| `aus.setf(ios::hex,ios::basefield)` | Hexadezimalausgabe bei ganzen Zahlen |
| `aus.setf(ios::showbase)` | führende `0` (oktal), `0x`, `0X` (hex.) ausg. |
| `aus.unsetf(ios::showbase)` | keine Zahlsystemangabe bei ganzen Zahlen |
| `aus.setf(ios::boolalpha)` | log. Werte als Zeichenkette ausg. |
| `aus.unsetf(ios::boolalpha)` | log. Werte als Zahlen ausg. (Voreinst.) |
| `aus.flush()` | Ausgabepuffer leeren |

[1] Wirkt nur auf die nächste Ausgabeoperation
[2] im Gleitpunktformat `fixed` oder `scientific`
[3] Voreinstellung

- **Formatierte Eingabe**

  *Syntax:* `ein >>` *Argument1* `>>` *Argument2* `>>` ...

  Mit Manipulatoren kann der Eingabevorgang verändert werden.

  | | |
  |---|---|
  | `skipws` | führenden Zwischenraum überlesen (Voreinst.) |
  | `noskipws` | führenden Zwischenraum nicht überlesen |
  | `dec oct hex` | Dezimal/Oktal/Hexadezimaleingabe bei ganzen Zahlen |

  Die entsprechenden Formatflags können auch mit `ein.setf` gesetzt werden.

  | | |
  |---|---|
  | `ein.setf(ios::skipws)` | führenden Zwischenraum überlesen (Voreinst.) |
  | `ein.unsetf(ios::skipws)` | führenden Zwischenraum nicht überlesen |
  | `ein.setf(ios::dec,ios::basefield)` | Dezimaleingabe bei ganzen Zahlen |
  | `ein.setf(ios::oct,ios::basefield)` | Oktaleingabe bei ganzen Zahlen |
  | `ein.setf(ios::hex,ios::basefield)` | Hexadezimaleingabe bei ganzen Zahlen |

  Wird neben der C++-Ein/Ausgabe auch die C-Ein/Ausgabe benutzt, dann sollte der Funktionsaufruf `ios::sync_with_stdio()` vor der ersten C++-Ein/Ausgabeoperation erfolgen.

- **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 Dateiende |
  | `ein.fail()` | liefert `true` bei Fehlern |
  | `ein.bad()` | liefert `true` bei schweren Fehlern (Datenverlust) |
  | `ein.clear()` | löscht Fehlerflags (danach liefert `cin.good()` den Wert `true`) |

  An Stelle der Abfrage `ein.fail()==false` kann die Abfrage `ein!=0` treten. (Bei dem Vergleich wird `ein` in einen Zeiger vom Typ `void *` konvertiert.)

- **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 `ein` 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(cin,s)` *(→ Seite 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 `aus` 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 |

### 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. zu 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
      :
```

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

[2] parameterlose Vereinbarung ohne Klammern ebenfalls zulässig

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

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

## STL-Zeichenketten in C++ (`string`) [Auszug Programmieren I]

Im Unterschied zu C-Zeichenketten werden C++-Zeichenketten nicht mit `\0` abgeschlossen, sondern es wird die aktuelle Länge mit abgespeichert.
Im folgenden sind einige wichtige Operatoren und Funktionen für den Datentyp `string` aufgeführt. Dabei stehen $s$ und $t$ für C++-Zeichenketten, $ct$ für eine `\0`-terminierte C-Zeichenkette, $c$ für ein Zeichen vom Typ `char`, $i$ und $n$ für ganze Zahlen vom Typ `string::size_type` (g++-9.4 unter Ubuntu Linux 20.04 amd64: `unsigned long`), *ein* und *aus* für einen Ein- bzw. Ausgabestrom.

| *Operation* | *Wirkung* |
|---|---|
| `string s` | vereinbart leere Zeichenkette |
| `string s(t)` | vereinbart Zeichenkette, kopiert Zeichenkette $t$ auf $s$ |
| `string s(t,i)` | vereinbart Zeichenkette, kopiert $t$ auf $s$ ab Index $i$ |
| `string s(t,i,n)` | wie `string s(t,i)`, höchstens $n$ Zeichen werden kopiert |
| `string s(ct)` | vereinbart Zeichenkette, kopiert C-Zeichenkette $ct$ auf $s$ |
| `string s(n,c)` | vereinbart Zeichenkette und initialisiert sie mit $n$ Zeichen $c$ |
| `s[i]` | Komponentenwert zum Index $i$ |
| `s.at(i)` | Komponentenwert zum Index $i$ mit Bereichsüberwachung |
| `s=t s=ct s=c` | weist Zeichenkette, C-Zeichenkette bzw. Zeichen zu |
| `s+t s+ct ct+s` | liefert Verkettung zweier Zeichenketten als neue Zeichenkette |
| `s+=t s+=ct s+=c` | hängt Zeichenkette, C-Zeichenkette bzw. Zeichen an $s$ an |
| `s==t s!=t s>t s<t s>=t s<=t` | vergleicht $s$ und $t$ lexikographisch |
| `s==ct s!=ct s>ct s<ct s>=ct s<=ct` | vergleicht $s$ und $ct$ lexikographisch |
| `cs==t cs!=t cs>t cs<t cs>=t cs<=t` | vergleicht $cs$ und $t$ lexikographisch |
| `s.size() s.length()` | liefert Zeichenzahl (als `string::size_type`) |
| `s.max_size()` | liefert Maximalzahl von Zeichen in $s$ |
| `s.substr(i) s.substr(i,n)` | liefert Teilzeichenkette ab Index i bzw. mit max. Länge $n$ als neue Zeichenkette (falls $i \leq s.size()$) |
| `s.c_str()` | liefert $s$ als konstante C-Zeichenkette (mit angehängtem `\0`) |
| `s.data()` | liefert $s$ als konstanten C-Vektor aus `char` (ohne angehängtes `\0`) |
| `s.find(t) s.find(ct) s.find(c)` | sucht erstes $t$,$ct$,$c$ in $s$, liefert Index oder `string::npos` |
| `s.find(t,i) s.find(ct,i)` | sucht erstes $t$,$ct$,$c$ in $s$ ab Index $i$, liefert Index |
| `s.find(c,i)` | oder `string::npos` |
| `s.rfind(t) s.rfind(ct) s.rfind(c)` | wie `s.find`, jedoch Suche nach letztem Auftreten |
| `s.rfind(t,i) s.rfind(ct,i)` | " |
| `s.rfind(c,i)` | " |
| `s.insert(i,t) s.insert(i,ct)` | fügt $t$ bzw. $ct$ in $s$ ab Index $i$ ein (falls $i \leq s.size()$) |
| `s.erase(i) s.erase(i,n)` | entfernt alle bzw. max. $n$ Zeichen von $s$ ab Index $i$ (falls $i \leq s.size()$) |
| `s.replace(i,n,t) s.replace(i,n,ct)` | ersetzt max. $n$ Zeichen in $s$ ab Index $i$ durch $t$ bzw. $ct$ (falls $i \leq s.size()$) |
| `s.copy(ct,n,i)` | kopiert max. $n$ Zeichen ab Index $i$ (Voreinst.: 0) von $s$ nach $ct$ (falls $i \leq s.size()$), `\0` wird *nicht* angehängt |
| `aus << s ein >> s` | Ausgabe, Eingabe (Länge der Zeichenkette $s$ wird angepasst) |
| `getline(ein,s)` | liest Zeile von Eingabe und schreibt diese nach Entfernen von `\n` in $s$, liefert Zustand von `ein` |