Ein-/Ausgabe in C++ (`iostream`, `iomanip`)

Im Folgenden bezeichne *i* einen Eingabestrom (z.B. `cin`) und *o* einen Ausgabestrom (z.B. `cout`, `cerr`).

Formatierte Ausgabe

Die Ausgabe auf einen Strom erfolgt durch geeignet überladene shift-Operatoren <<, wie folgt: o << arg_1 << arg_2 ...

Als Argumente können Werte aller Typen fungieren, für die der shift-Operator geeignet überladen ist. Insb. die eingebauten Typen `int`, `double`, `char`, ... und alle eigenen Klassen/Typen bei denen dies der Fall ist.

Zusätzlich können die folgenden Manipulatoren als Argument verwendet werden um das Verhalten danach folgender Ausgaben zu verändern:

`setw(`n`)`	Setze[1] minimale Feldbreite auf n (Voreinst.: 0)
`setprecision(`n`)`	Wenn `fixed`, setze Ziffern nach Dezimalpunkt; wenn `scientific`, setze signifikante Stellen (Voreinst.: 6)
`setfill(`c`)`	Setze Füllzeichen auf c, vgl. Feldbreite (Voreinst.: Leerzeichen)
`left right internal`	Links- bzw. Rechts- bzw. nur Vorzeichen links- und Zahl rechtsbündig, vgl. Feldbreite (Voreinst.: rechtsbündig)
`showpos noshowpos`	Positives Vorzeichen ausgeben bzw. nicht ausgeben (Voreinst.: nicht ausgeben)
`fixed scientific defaultfloat`	Festpunkt-, Exponentialformat oder automatische Auswahl bei Gleitpunktzahlen Voreinst.: auto. Auswahl)
`dec oct hex`	Dezimal-, Oktal- oder Hexadezimale Ausgabe bei ganzen Zahlen (Voreinst.: dezimal)
`showbase noshowbase`	Präfix für Zahlenbasis (`0` für oktal bzw. `0x`/`0X` für hexadezimal) ausgeben bzw. nicht ausgeben (Voreinst.: nicht asugeben)
`boolalpha noboolalpha`	Ausgabe von `bool`-Werten als Zeichenkette bzw. numerisch (Voreinst: numerisch)
`endl`	Zeilenumbruch
`flush`	Ausgabepuffer leeren

Formatierte Eingabe

Das Lesen von Eingabe aus einem Strom erfolgt ebenfalls durch geeignet überladene shift-Operatoren. In Analogie zur *Richtung, in die die Information fließt* für die Eingabe durch >>, wie folgt: i >> arg_1 >> arg_2 ...

Auch hierfür können Werte aller Typen als Argumente fungieren, für die der shift-Operator geeignet überladen ist. Insb. wieder die eingebauten Typen `int`, `double`, `char`, ... und alle eigenen Klassen/Typen bei denen dies der Fall ist.

[1] Wirkt sich i.d.R. nur auf die nächste Ausgabe aus

Folgende Manipulatoren stehen zur Verfügung:

`skips noskipws`	Nicht-druckbare Zeichen (insb. Leerzeichen) am aktuellen Stand überlesen, bis zur nächsten Eingabe, oder nicht (Voreinst.: überlesen)
`dec oct hex`	Dezimale, Oktale, bzw. Hexadezimale Eingabe bei ganzen Zahlen (Voreinst.: dezimal)

Stromzustand

Ein- und Ausgabeströme haben jeweils einen internen numerischen Zustand. Der Typ des Zustands `ios::iostate` ist nicht näher spezifiert, jedoch sicherlich groß genug um 3 bits für jeweils einen Ausnahmezustand zu enthalten. Es werden die bitmasken `ios::badbit`, `ios::eofbit` und `ios::failbit` bereitgestellt um vermöge geeigneter bitweiser Arithmetik auf die einzelnen Felder zuzugreifen.

Bequemer stehen für einen Strom *s* die folgenden Methoden zur Verfügung:

s`.good()`	Liefert `true`, falls keines der Ausnahme-bits gesetzt ist – die nächste Aktion könnte gelingen
s`.eof()`	Liefert `true`, wenn *s* keine weitere Eingabe gelesen werden kann/keine Ausgabe mehr erfolgen kann, da Ende des Stroms erreicht ist
s`.fail()`	Liefert `true`, falls ein Fehler bzgl. *s* aufgetreten ist
s`.bad()`	Liefert `true`, falls ein hinreichend schwerer Fehler bzgl. *s* aufgetreten ist, dass Datenverlust aufgetreten sein könnte
s`.clear()`	Setzt internen Zustand auf 0; *s*`.good()` liefert dann i.d.R. `true`

Um zu prüfen, ob ein Strom *s* in einem *guten* Zustand ist, empfiehlt sich `if (!`*s*`.fail()) {...}` oder, äquivalent, `if (`*s*`) {...}`.

Unformatierte Ein-/Ausgabe

Mit den folgenden Funktionen kann zeichenweise von einem Eingabestrom *i* gelesen bzw. auf einen Ausgabestrom *o* geschrieben werden. Im folgenden sei *c* eine Variable vom Typ `char` und *cs* eine hinreichend lange C-Zeichenkette vom Typ `char[]`.

i`.get()`	Liest ein einzelnes Zeichen von *i* und liefert es als `int`. Liefert Wert identisch mit Konstante `EOF` im Fehlerfall
i`.get(`*c*`)`	Liest ein einzelnes Zeichen von *i* und speichert es auf der Referenz *c*. Liefert eine Referenz auf *i* als Rückgabewert
i`.unget()`	Setzt cursor-Position im Strom ein Zeichen zurück. Liefert eine Referenz auf *i* als Rückgabewert
i`.peek()`	Liefert analog zu *i*`.get()` nächstes Zeichen, verändert jedoch cursor-Position im Strom nicht

`i.read(cs, n)`	Liest höchstens n Zeichen von i und speichert diese in cs. Liefert eine Referenz auf i als Rückgabewert
`i.gcount()`	Liefert Anzahl von Zeichen, die von letzter unformattierter Eingabefunktion erfolgreich gelesen wurden
`s.tellg()`	Liefert aktuelle cursor-Position im Ein- bzw. Ausgabestrom s als `ios::pos_type`
`s.seekg(p)`	Setzt aktuelle cursor-Position im Ein- bzw. Ausgabestrom s auf Wert von p (ebenfalls `ios::pos_type`)
`o.put(c)`	Gibt c aus auf o. Liefert eine Referenz auf o als Rückgabewert

Ein-/Ausgabe auf Dateien (`fstream`)

Dateien können als Strom geöffnet werden, indem ein Objekt vom Typ `ifstream` (Eingabe) bzw. `ofstream` (Ausgabe) konstruiert wird. Die Konstruktoren akzeptieren jeweils einen Dateipfad (`string` oder `char[]`) als erstes Argument und optional einen *Modus* als zweites Argument.

Stromobjekte können auch ohne Konstruktorargumente mit dem Standardkonstruktor initialisiert werden. In diesem Fall wird nicht sofort eine Datei geöffnet.

Der Modus ist ein numerischer Wert vom Typ `ios::mode`. Es stehen die folgenden bitmasken zur Verfügung um Modi zu konstruieren bzw. zu modifizieren:

`ios::in`	Datei zum Lesen öffnen
`ios::out`	Datei zum Schreiben öffnen
`ios::append`	Beim Schreiben hinten an Datei anhängen
`ios::ate`	Beim Öffnen cursor am Ende der Datei positionieren
`ios::trunc`	Inhalt der Datei beim Öffnen löschen
`ios::binary`	Datei im Binärmodus öffnen; verhindert automatische Übersetzung bestimmter Kontrollzeichen auf bestimmten Betriebssystemen

Im Folgenden bezeichne f einen Dateinamen vom Typ `string`, m einen Zugriffsmodus vom Typ `ios::mode`, i einen Eingabe-, o einen Ausgabe-, b einen Ein-/Ausgabestrom und s einen beliebigen Strom. Es stehen folgende Operationen zur Verfügung:

`ifstream i(f)`	Vereinbare Variable i vom Typ `ifstream`, öffne dabei f zum Lesen mit Modus `ios::in`
`ifstream i(f, m)`	Vereinbare Variable i vom Typ `ifstream`, öffne dabei f zum Lesen mit Modus `ios::in | m`
`ofstream o(f)`	Vereinbare Variable o vom Typ `ofstream`, öffne dabei f zum Schreiben mit Modus `ios::out`
`ofstream o(f, m)`	Vereinbare Variable o vom Typ `ofstream`, öffne dabei f zum Schreiben mit Modus `ios::out | m`

`fstream` b`(`f`)`	Vereinbare Variable b vom Typ `fstream`, öffne dabei f zum Schreiben und Lesen mit Modus `ios::out | ios:in`
`fstream` b`(`f`, `m`)`	Vereinbare Variable b vom Typ `fstream`, öffne dabei f zum Schreiben mit Modus m
s`.open(`f`)`	Öffne f, Modus entsprechend dem Typ von s (vgl. oben)
s`.open(`f`, `m`)`	Öffne f, Modus entsprechend dem Typ von s und m (vgl. oben)
s`.close()`	Schließe Strom
s`.is_open()`	Prüfe ob Strom aktuell geöffnet

Ein-/Ausgabe auf Strings (`sstream`)

Es können auch Ströme erzeugt werden, die auf buffern vom Typ `string` agieren.

Im Folgenden bezeichne s einen `string`, m einen Zugriffsmodus vom Typ `ios::mode`, i ein Eingabe-, o ein Ausgabe-, b einen Ein-/Ausgabetsrom und s einen beliebigen Strom. Es stehen folgende Operationen zur Verfügung:

`istringstream` i	Vereinbare Variable i vom Typ `istringstream`, Buffer von i initial leer, Modus des Stroms `ios::in`
`istringstream` i`(`s`)`	Vereinbare Variable i vom Typ `istringstream`, Buffer von i initial Kopie von s, Modus des Stroms `ios::in`
`istringstream` i`(`s`, `m`)`	Vereinbare Variable i vom Typ `istringstream`, Buffer von i initial Kopie von s, Modus des Stroms `ios::in |` m
`ostringstream` o	Vereinbare Variable i vom Typ `ostringstream`, Buffer von o initial leer, Modus des Stroms `ios::out`
`ostringstream` o`(`s`)`	Vereinbare Variable i vom Typ `ostringstream`, Buffer von o initial Kopie von s, Modus des Stroms `ios::out`
`ostringstream` o`(`s`, `m`)`	Vereinbare Variable i vom Typ `ostringstream`, Buffer von o initial Kopie von s, Modus des Stroms `ios::out |` m
`stringstream` b	Vereinbare Variable i vom Typ `stringstream`, Buffer von b initial leer, Modus des Stroms `ios::in | ios::out`
`stringstream` b`(`s`)`	Vereinbare Variable i vom Typ `stringstream`, Buffer von b initial Kopie von s, Modus des Stroms `ios::in | ios::out`
`stringstream` b`(`s`, `m`)`	Vereinbare Variable i vom Typ `stringstream`, Buffer von b initial Kopie von s, Modus des Stroms m
s`.str()`	Liefert Kopie des aktuellen Bufferinhalts
s`.str(`s`)`	Setze aktuellen Bufferinhalt auf Kopie von s

demo_sstream.cpp

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

using namespace std;

int main() {
  int i;
  istringstream ein{"   123abc "};
  ein >> i;
  cout << "i: " << i << endl;

  string s;
  ostringstream aus;
  aus << hex << i;
  cout << "s: " << aus.str() << endl;

  return 0;
}
```

```
i: 123
s: 7b
```

STL-Zeichenketten (`string`)

C behandelt Zeichenketten als array vom Typ `char[]` unbekannter Länge. Es wird das Null-Byte `\0` als Marker innerhalb des arrays verwendet um das logische Ende der Zeichenkette zu markieren. Fehlende oder inkorrekt gesetzte Null-Bytes in C-Zeichenketten sind eine häufige Quelle von Fehlern und Sicherheitsproblemen in diverser Software.

Die Verwendung von diesen C-Zeichenketten ist auch in C++ an vielen Stellen möglich. Es ist jedoch, insb. wegen der häufigen Problemtik bzgl. Null-Bytes, stark davon abzuraten.

Analog zu STL-Vektoren als Ersatz für C arrays bietet C++ ebenfalls einen `string` Typ, der einen geeigneten Buffer von `char` und die zugehörige Länge (vom Typ `string::size_type`) intern verwaltet. Ein Terminierungszeichen wie `\0` ist dann nicht mehr nötig, was eine häufige Fehlerquelle systematisch behebt.

Im Folgenden bezeichnen s und t Zeichenketten vom Typ `string`, s_C eine korrekt `\0`-terminierte C-Zeichenkette vom Typ `char[]`, c ein Zeichen vom Typ `char`, i und n ganze Zahlen vom Typ `string::size_type` und i und o einen Ein- bzw. Ausgabestrom. Folgende Operationen stehen zur Verfügung:

`string` s	Vereinbart Variable s vom Typ `string` leer (Länge 0)
`string` s{t}	Vereinbart Variable s vom Typ `string` mit Inhalt als Kopie dessen von t
`string` s{t, i}	Vereinbart Variable s vom Typ `string` mit Inhalt als Kopie dessen von t ab Index i bis Ende

`string s{t, i, n}`	Vereinbart Variable s vom Typ `string` mit Inhalt als Kopie dessen von t ab Index i bis $i + n - 1$ (Länge maximal n)
`string s{`s_C`}`	Vereinbart Variable s vom Typ `string` mit Inhalt als Kopie dessen von s_C
`string s{n, c}`	Vereinbart Variable s vom Typ `string` mit Inhalt n viele Kopien des Zeichen c
`s[i]`	Wert vom Typ `char` des Zeichen an Index i von string s
`s.at(i)`	Analog zu `s[i]`, löst jedoch Ausnahme (exception) aus, falls i außerhalb des Bereichs von validen Indizes
`s = t` `s = `s_C `s = c`	Setzt Inhalt von s auf den von Kopie von t bzw. den von s_C bzw. Länge 1 und Inhalt c
`s + t` `s + `s_C `s + c`	Nimmt neuen String als Wert an mit Kopie von s konkateniert mit t bzw. s_C bzw. c, wie oben
`s += t` `s += `s_C `s += c`	Setzt s auf Kopie von s konkateniert mit t bzw. s_C bzw. c, wie oben
`s == t` `s != t` `s > t` `s < t` `s >= t` `s <= t`	Vergleiche s und t lexikographisch, nimmt Wert vom Typ `bool` an
`s == `s_C `s != `s_C `s > `s_C `s < `s_C `s >= `s_C `s <= `s_C	Vergleiche s und s_C lexikographisch, nimmt Wert vom Typ `bool` an
s_C `== t` s_C `!= t` s_C `> t` s_C `< t` s_C `>= t` s_C `<= t`	Vergleiche s_C und t lexikographisch, nimmt Wert vom Typ `bool` an
`s.size()` `s.length()`	Nimmt Länge von s als Wert vom Typ `string::size_type` an
`s.substr(i)` `s.substr(i, n)`	Nimmt neuen string als Wert an mit Inhalt Kopie von s ab Index i, optional mit maximaler Länge n
`s.find(t)` `s.find(`s_C`)` `s.find(c)` `s.find(t, i)` `s.find(`s_C`, i)` `s.find(c, i)`	Sucht erstes von Vorkommen von t, s_C, bzw. c in s, optional erst ab Index i, nimmt index vom Typ `string::size_type` an, wenn gefunden, sonst `string::npos`
`s.find(t)` `s.find(`s_C`)` `s.find(c)` `s.find(t, i)` `s.find(`s_C`, i)` `s.find(c, i)`	Analog zu `.find(...)`, liefert jedoch letztes Vorkommen in s
`s.insert(i, t)` `s.insert(i, `s_C`)` `s.insert(i, n, c)`	Fügt t bzw. s_C bzw. n viele Kopien von c in s ein ab Index i, s wird hierdurch verlängert um Länge von t bzw. Länge von s_C bzw. n
`s.erase()` `s.erase(i)` `s.erase(i, n)`	Entfernt erstes Zeichen bzw. Zeichen mit Index i bzw. n viele Zeichen ab Index i, verkürzt s um 1 bzw. 1 bzw. n
`s.replace(i, n, t)` `s.replace(i, n, `s_C`)`	Entfernt n Zeichen ab Index i in s und fügt an ihrer Stelle den Inhalt von t bzw. s_C ein, verlängert bzw. verkürzt s um bis zu Länge von t bzw. s_C abzüglich n
`o << s` `i >> s`	Aus- bzw. Eingabe, Länge von s wird geeignet angepasst, Eingabe beachtet `skipws` bzw. `noskipws` und liest dann bis zum ersten nicht-druckbaren Zeichen (insb. Leerzeichen)
`getline(i, s)`	Liest Zeile von i, entfernt Zeilenumbruch, speichert verbleibende Zeichenkette in s, nimmt Kopie von i als Wert an

string_io_demo.cpp

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

using namespace std;

int main() {
  istringstream buf{"  wort1 wort2 wort3  "};
  string s;
  buf >> s;

  cout << "»" << s << "«" << endl;

  istringstream buf2{"  wort1 wort2 wort3  "};
  string s2;
  buf2 >> noskipws >> s2;

  cout << "»" << s2 << "«" << endl;

  istringstream buf3{"  wort1 wort2 wort3  "};
  string s3;
  getline(buf3, s3);

  cout << "»" << s3 << "«" << endl;

  return 0;
}
```

```
»wort1«
»«
»  wort1 wort2 wort3  «
```