Technisches Programmieren in C++

Ein-/Ausgabe in C++

Karl Riedling
Institut für Sensor- und Aktuatorsysteme

Ein-/Ausgabe in C++

- Befehlszeilenparameter und Rückgabewerte
- Die C++ *Class Library* `iostream`
- Abspeichern von Klassen-Objekten in Dateien

Ein-/Ausgabe in C++

- **Befehlszeilenparameter und Rückgabewerte**
- Die C++ *Class Library* `iostream`
- Abspeichern von Klassen-Objekten in Dateien

Ein-/Ausgabe in C++

- Befehlszeilenparameter und Rückgabewerte
  - **Befehlszeilenparameter**
  - Rückgabewerte

Befehlszeilenparameter

- Programmaufruf-Parameter können mit Hilfe von zwei optionalen Argumenten der Funktion `main()` eingelesen und verwendet werden:
  - Anzahl der Argumente (`argc`);
  - Feld von Zeigern auf `char` — Argumente (`argv`).
- Der Startup-Code des Programms "bricht" die Eingabe-zeile an bestimmten Trennzeichen in separate Argumente auf, die den Elementen von `argv[1]` an zugewiesen werden:
  - Führender *White Space* wird entfernt.
  - Argumente in Anführungszeichen werden nicht getrennt.

Befehlszeilenparameter

- Das Element `argv[0]` enthält einen Zeiger auf jenen Befehlsstring, der zum Aufruf des Programms verwendet wurde.
- Der Startup-Code übergibt noch ein drittes Argument, `envp`, an `main()`: Die Startadresse eines Feldes von Zeigern auf `char`, die auf je eine Eintragung im Umgebungsbereich des Rechners (*Environment*) zeigen.
- Das Feld `envp` wird von einem Null-Zeiger (einem Zeiger mit dem numerischen Wert `0`) abgeschlossen.

Befehlszeilenparameter

- Der vollständige Prototyp von `main()` ist damit:

```
int main (int argc, char *argv[],
  char *envp[]);
```

- Wegen der Äquivalenz von Feldern und Zeigern kann man auch schreiben:

```
int main (int argc, char **argv,
  char **envp);
```


Ein-/Ausgabe in C++

- Befehlszeilenparameter und Rückgabewerte
  - Befehlszeilenparameter
  - **Rückgabewerte**

Rückgabewerte

- Für die Rückgabe von numerischen Resultaten an das Betriebssystem gibt es die folgenden Möglichkeiten:
  - Verwendung von `return` in `main()` mit einem numerischen Argument;
  - Aufruf der Funktionen `exit()` oder `_exit()` mit einem numerischen Argument; dieser Aufruf kann in jeder beliebigen Funktion erfolgen.
    - `exit()` nimmt noch "Aufräumungsarbeiten" vor, bevor es die Kontrolle ans Betriebssystem übergibt (insbesondere das Ausschreiben von Ausgabe-Puffern);

Rückgabewerte

    - `_exit()` bricht sofort und bedingungslos die Ausführung des Programms ab.
- In beiden Fällen wird das übergebene Argument als Rückgabewert verwendet.

Ein-/Ausgabe in C++

- Befehlszeilenparameter und Rückgabewerte
- **Die C++ *Class Library* `iostream`**
- Abspeichern von Klassen-Objekten in Dateien

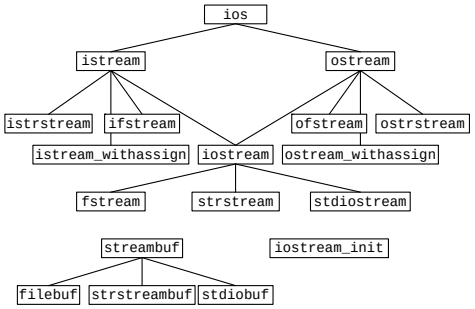
Ein-/Ausgabe in C++

- Die C++ *Class Library* `iostream`
  - **Allgemeines**
  - Konsol-Ausgabe mit C++-*Streams*
  - Konsol-Eingabe mit C++-*Streams*
  - C++-*Stream*-Ein-/Ausgabe von/auf Puffer
  - C++-*Stream*-Ein-/Ausgabe in Dateien
  - C- und C++-Ein-/Ausgabe im Vergleich

Die C++ *Class Library* `iostream`

- Die *Class Library* `iostream` besteht aus einer Reihe standardmäßig definierter Klassen, die zum größten Teil von der Basisklasse `ios` virtuell abgeleitet sind:

Die C++ *Class Library* `iostream`



Ein-/Ausgabe in C++

- Die C++ *Class Library* `iostream`
  - Allgemeines
  - **Konsol-Ausgabe mit C++-*Streams***
  - Konsol-Eingabe mit C++-*Streams*
  - C++-*Stream*-Ein-/Ausgabe von/auf Puffer
  - C++-*Stream*-Ein-/Ausgabe in Dateien
  - C- und C++-Ein-/Ausgabe im Vergleich

Konsol-Ausgabe mit C++-*Streams*

- Eine Ausgabe auf die Konsole unter Verwendung der Funktionen der Klasse `iostream` kann mittels der folgenden *Objekte* erfolgen:

`cout`	Standard-Ausgabe (entspr. `stdout` in ANSI-C).
`cerr`	Standard-Ausgabekanal für Fehlermeldungen (entspr. `stderr` in ANSI-C) mit begrenzter Pufferung.
`clog`	wie `cerr`, aber mit voller Pufferung.

Konsol-Ausgabe mit C++-*Streams*

- Andere *Stream*-Objekte (die beispielsweise einen Puffer oder eine Datei beschreiben) können explizit konstruiert werden.
- Die Ausgabe erfolgt grundsätzlich in formatierter Form.
- Die Klassenelement-Funktionen der Klasse `ostream` sehen eine Default-Formatierung in Abhängigkeit vom Typ eines für die Ausgabe vorgesehenen Objekts vor.
- Die Default-Formatierung kann durch Klassenelement-Funktionen und *Manipulatoren* geändert werden.
- Die wichtigsten dieser Funktionen und Manipulatoren sind im folgenden Programm enthalten:

Konsol-Ausgabe mit C++-*Streams*

```
#include <iostream.h>
#include <iomanip.h>                    // für Manipulatoren

int ai[] = { 3, 17, 123, 4321, 23456 };
double ad[] = { .1234, 43.21, 543.456, 2345.6,
765432.109876 };
char *as[] = { "The", "quick", "brown", "fox", "jumps" };

int main ()
{
  cout << "\nDefault-Formatierung:\n";

  for (int i = 0; i < 5; i++)
    cout << "int: " << ai[i] << ", double: " << ad[i] <<
      ", string: " << as[i] << endl;     // endl == "\n"
```

Konsol-Ausgabe mit C++-*Streams*

```
cout << "\nKonstante Spaltenbreite:\n";
for (i = 0; i < 5; i++)
{
  cout.width (10);                // Spaltenbreite = 10
                        // gilt nur für das folgende Feld!
  cout << di[i] << endl;
}
for (i = 0; i < 5; i++)
  cout << "int: " << setw(7) << ai[i]
    << ", double: " << setw(10) << ad[i]
    << ", string: " << setw(7) << as[i] << endl;
                // setw gilt nur für das folgende Feld!
```


Konsol-Ausgabe mit C++-*Streams*

```
cout << "\nSpalte auffüllen:\n";
cout.fill ('*');                        // Füllzeichen "*"
for (i = 0; i < 5; i++)
  cout << "int: " << setw(7) << ai[i]
    << ", double: " << setw(10) << ad[i]
    << ", string: " << setw(7) << as[i] << endl;
cout.fill (' ');                    // zurück zum Default

cout << "\nLinksbündige Ausgabe:\n";
cout << setiosflags (ios::left);
for (i = 0; i < 5; i++)
  cout << "int: " << setw(7) << ai[i]
    << ", double: " << setw(10) << ad[i]
    << ", string: " << setw(7) << as[i] << endl;
cout << resetiosflags (ios::left);
```


Konsol-Ausgabe mit C++-*Streams*

```
cout << "\nAuflösung: 2 Stellen:\n";
cout << setprecision (2);
for (i = 0; i < 5; i++)
  cout << "int: " << setw(7) << ai[i]
    << ", double: " << setw(10) << ad[i]
    << ", string: " << setw(7) << as[i] << endl;

cout << "\nExponentenschreibweise:\n";
cout << setiosflags (ios::scientific);
for (i = 0; i < 5; i++)
  cout << "int: " << setw(7) << ai[i]
    << ", double: " << setw(10) << ad[i]
    << ", string: " << setw(7) << as[i] << endl;
cout << resetiosflags (ios::scientific);
```


Konsol-Ausgabe mit C++-*Streams*

```
  cout "\nGleitkommaschreibweise:\n";
  cout << setiosflags (ios::fixed);
  for (i = 0; i < 5; i++)
    cout << "int: " << setw(7) << ai[i]
      << ", double: " << setw(10) << ad[i]
      << ", string: " << setw(7) << as[i] << endl;

  cout << "\nRadix:\n" << setprecision(6);
  for (i = 0; i < 5; i++)
    cout << "Dezimal: " << dec << ai[i]
      << ", oktal: " << oct << ai[i]
      << ", hex: " << hex << ai[i] << endl;
}
```


Konsol-Ausgabe mit C++-*Streams*

- Der Manipulator `flush` bewirkt ein sofortiges Ausschreiben des Ausgabepuffers, der bei gepufferter Konsol-Ausgabe ansonsten erst bei einem Eingabebefehl ausgeschrieben würde (entspricht der Standard-C-Funktion `fflush()`):

Konsol-Ausgabe mit C++-*Streams*

```
#include <iostream.h>
#include <time.h>
int main ()
{
  time_t endz = time (NULL) + 5;
  cout << "Ausgabe ohne \"flush\":\n";
  cout << "Bitte 5 Sekunden warten...";
  while (time (NULL) < endz);                  // warten
  cout << "Wartezeit vorbei.\n";
  endz = time (NULL) + 5;
  cout << "Ausgabe mit \"flush\":\n";
  cout << "Bitte 5 Sekunden warten..." << flush;
  while (time (NULL) < endz);     // warten
  cout << "Wartezeit vorbei.\n";
}
```

Konsol-Ausgabe mit C++-*Streams*

- Der Operator "<<" kann auch für eigene Datenobjekte definiert werden. Das folgende Beispiel zeigt eine Definition für eine Klasse `Punkt`:

Konsol-Ausgabe mit C++-*Streams*

```
#include <iostream.h>
class Punkt
{
public:
  Punkt(short x = 0, short y = 0) { _x = x; _y = y; }
private:
  short _x, _y;
friend ostream& operator << (ostream&, Punkt&);
};
ostream& operator<<(ostream& os, Punkt& p)
{
  os << "x = " << p._x << ", y = " << p._y;
  return os;
}
```


Konsol-Ausgabe mit C++-*Streams*

```
int main ()
{
  Punkt pt1 (3, 5);                    // Objekt
  Punkt *ppt2 = new Punkt (11, 23);    // Zeiger
    cout << "Punkt 1: " << pt1 <<
      "; Punkt 2: " << *ppt2 << endl;
}
```


Konsol-Ausgabe mit C++-*Streams*

- Auch Manipulatoren, die keine Argumente erfordern, können einfach selbst definiert werden. Im folgenden Beispiel wird ein Manipulator `sterne` definiert:

Konsol-Ausgabe mit C++-*Streams*

```
#include <iostream.h>

ostream& sterne (ostream& os)
{
  return os << "******";
}

int main ()
{
  cout << "12345" << sterne << "67890\n";
}
```


Ein-/Ausgabe in C++

- Die C++ *Class Library* `iostream`
  - Allgemeines
  - Konsol-Ausgabe mit C++-*Streams*
  - **Konsol-Eingabe mit C++-*Streams***
  - C++-*Stream*-Ein-/Ausgabe von/auf Puffer
  - C++-*Stream*-Ein-/Ausgabe in Dateien
  - C- und C++-Ein-/Ausgabe im Vergleich

Konsol-Eingabe mit C++-*Streams*

- Eine Eingabe von der Konsole unter Verwendung der Funktionen der Klasse `iostream` erfolgt zweckmäßigerweise mittels des Objekts `cin`. Andere *Stream*-Objekte (die beispielsweise einen Puffer oder eine Datei beschreiben) können explizit konstruiert werden.
- Die Standard-Eingabe unter Verwendung von *Stream*-Objekten erfolgt unter Verwendung des (überladenen) Operators ">>".
- Die für die Ausgabeoperationen definierten Manipulatoren gelten zwar im Prinzip auch für die Eingabe; wirksam sind jedoch nur die drei Radix-Manipulatoren `dec`, `oct` und `hex`.

Konsol-Eingabe mit C++-*Streams*

- Bei Verwendung des Manipulators `hex` sind die folgenden Eingabe-Strings zulässig und ergeben denselben Wert:

```
abcd
ABCD
aBcD
0xabcd
...
```

- Bei Eingabe eines fehlerhaften Wertes wird ein internes Fehler-Flag gesetzt und der *Stream* wird unbrauchbar.
- Der Status eines *Streams* kann mit den folgenden Klassenelement-Funktionen geprüft werden:

Konsol-Eingabe mit C++-*Streams*

`bad()`	TRUE bei einem nicht korrigierbaren Fehler.
`fail()`	TRUE bei einem "vorhersehbaren" oder nicht korrigierbaren Fehler.
`good()`	TRUE, wenn kein Fehler vorliegt und das Ende der Eingabedaten noch nicht erreicht wurde.
`eof()`	TRUE am Ende der Eingabedaten.
`clear(0)`	Setzt den internen Fehler-Status; ein Aufruf mit dem Argument `0` setzt alle Fehler-Flags zurück.
`rdstate()`	Gibt den aktuellen Fehler-Status zurück.

Konsol-Eingabe mit C++-*Streams*

- Der Operator `"!"` ist durch *Overloading* so definiert, dass er die selbe Funktion erfüllt wie `fail()`. Die folgenden beiden Ausdrücke sind daher äquivalent ("wenn die Eingabe fehlerhaft war"):

```
if (! cin) ...
if (cin.fail()) ...
```

- Der Operator `void*()` ist als das Gegenteil von `fail()` definiert; die beiden folgenden Ausdrücke bedeuten daher "wenn bei der Eingabe *kein* Fehler aufgetreten ist":

```
if (cin) ...
if (! cin.fail()) ...
```


Konsol-Eingabe mit C++-*Streams*

- Es gilt

```
cin.good() == ! cin.fail() && ! cin.eof()
```

- Das folgende Programm soll den Einsatz und die Wirkungsweise der Status-Klassenelement-Funktionen demonstrieren:

Konsol-Eingabe mit C++-*Streams*

```
#include <iostream.h>
int main ()
{
  int do_clear = 1;    // Flag-Variable - "clear" aktiv
  int do_flush = 1;    // Flag-Variable - Rest einlesen
  char antwort;
  cout << "\"clear\" nach einem Fehler (J/n)? ";
  cin >> antwort;
  if (antwort == 'n' || antwort == 'N')
    do_clear = 0;
  cout << "Eingabezeile nach einem Fehler ausleeren (J/n)? ";
  cin >> antwort;
  if (antwort == 'n' || antwort == 'N')
    do_flush = 0;
```

Konsol-Eingabe mit C++-*Streams*

```
for (int i = 0; i < 10; i++)
{
  int x;
  cout << "Bitte eine ganze Zahl eingeben: ";
  cin >> x;
  if (cin.good())
    cout << "Status ist \"good\"";
  if (cin.bad())
    cout << "Status ist \"bad\"";
```


Konsol-Eingabe mit C++-*Streams*

```
  if (cin.fail())
  {
    cout << "Status ist \"failed\"";
    if (do_clear)
      cin.clear ();
    if (do_flush)
    {
      char buffer[128];
      cin >> buffer;
    }
  }
  cout << " - Eingabe war: " << x << endl;
 }
}
```


Konsol-Eingabe mit C++-*Streams*

- Es ist bei Eingabedaten, die nicht reine Text-Daten sind, zweckmäßiger, immer eine *ganze* Zeile einzulesen und anschließend zu konvertieren. Dafür stehen in der Familie der *Stream*-Ein-/Ausgabefunktionen die Funktionen

  `istream& get (char* ptr, int len, char delim='\n');`

  und

  `istream& getline (char* ptr, int len, char delim='\n');`

  zur Verfügung. `len` ist die maximale Anzahl von Zeichen, die in den Puffer gelesen werden sollen, auf den `ptr` zeigt.

Konsol-Eingabe mit C++-*Streams*

- Beide Funktionen lesen Zeichen bis zum ersten Zeichen, das gleich `delim` ist; `getline()` entfernt den *Delimiter* aus dem Eingabepuffer, `get()` nicht.
- Keine der beiden Funktionen kopiert den *Delimiter* nach `ptr`.
- Das folgende Programm zeigt eine derartige "stabilere" Eingaberoutine:

Konsol-Eingabe mit C++-*Streams*

```
#include <iostream.h>
#include <strstream.h>     // für istrstream
int main ()
{
  char buffer[128];
  for (int i = 0; i < 10; i++)
  {
    cout << "Ganze Zahl: ";
    cin.getline (buffer, 128);  // komplette Zeile einlesen
    istrstream bStream (buffer);
      // Objekt der Klasse istrstream erzeugen
    int x;
    bStream >> x;               // Zuweisung an x
    if (bStream.good())         // OK
      cout << "OK:     ";
```


Konsol-Eingabe mit C++-*Streams*

```
    if (bStream.fail())         // Fehler
    {
      bStream.clear();
      cout << "Fehler: ";
    }
    cout << x << endl;
  }
}
```

Ein-/Ausgabe in C++

- Die C++ *Class Library* `iostream`
  - Allgemeines
  - Konsol-Ausgabe mit C++-*Streams*
  - Konsol-Eingabe mit C++-*Streams*
  - **C++-*Stream*-Ein-/Ausgabe von/auf Puffer**
  - C++-*Stream*-Ein-/Ausgabe in Dateien
  - C- und C++-Ein-/Ausgabe im Vergleich

C++-*Stream*-Ein-/Ausgabe von/auf Puffer

- Für eine Eingabe von oder Ausgabe auf einen Puffer muss ein Objekt der geeigneten Klasse (`istrstream`, `ostrstream` oder `strstream`) konstruiert werden.
- Der Puffer (ein Feld von `chars`) muss zu diesem Zeitpunkt existieren und dem Konstruktor als Argument übergeben werden.
- Die solcherart erzeugten Objekte können genauso verwendet werden wie etwa `cin` oder `cout`:

C++-*Stream*-Ein-/Ausgabe von/auf Puffer

Eingabe von Puffer:	Eingabe von Konsole:

`char b[128];  // Puffer`
`istrstream str(b);`
`// Konstruktor`	

`double x;` `str >> x;`	`double x;` `cin >> x;`

Ausgabe auf Puffer:	Ausgabe auf Konsole:

`char b[128];  // Puffer`
`ostrstream str(b);`
`// Konstruktor`	

`double x = 123.456;` `str << "x = " << x << endl;`	`double x = 123.456;` `cout << "x = " << x << endl;`

Ein-/Ausgabe in C++

- Die C++ *Class Library* `iostream`
  - Allgemeines
  - Konsol-Ausgabe mit C++-*Streams*
  - Konsol-Eingabe mit C++-*Streams*
  - C++-*Stream*-Ein-/Ausgabe von/auf Puffer
  - **C++-*Stream*-Ein-/Ausgabe in Dateien**
  - C- und C++-Ein-/Ausgabe im Vergleich

C++-*Stream*-Ein-/Ausgabe in Dateien

- Ein *Stream*-Objekt der geeigneten Klassen-Type muss zuerst konstruiert werden. Die Type richtet sich danach, ob ausschließlich Eingabe- oder ausschließlich Ausgabeoperationen vorgenommen werden sollen, oder eine Mischung von beiden:

Type der Operation	***Stream*-Klasse**
Eingabe	`ifstream`
Ausgabe	`ofstream`
Ein- und Ausgabe	`fstream`

C++-*Stream*-Ein-/Ausgabe in Dateien

- Ein *Stream*-Objekt für die Datei-Ein-/Ausgabe muss generell in zwei Stufen vorbereitet werden:
  - Erstellung des Objekts selbst unter Verwendung eines geeigneten Konstruktors, und
  - Verknüpfung dieses Objekts (das im wesentlichen Kontrollstrukturen enthält) mit der gewünschten Datei.
- Dabei sind die folgenden Vorgangsweisen möglich:
  - Konstruktion eines Objekts unter Verwendung des Default-Konstruktors, und anschließender Aufruf der Klassenelement-Funktion `open()`:

```
ifstream myfile;              // Stream-Objekt
myfile.open ("Datei", iosmode);
```

C++-*Stream*-Ein-/Ausgabe in Dateien

- Die Konstruktion des *Stream*-Objekts und das Öffnen der zugehörigen Datei kann in einem gemeinsamen Konstruktor-Aufruf geschehen:

```
ifstream myfile ("Datei", iosmode);
```

- Einem *Stream*-Objekt kann ein Datei-Deskriptor zugeordnet werden, der mit einer der Standard-C-Funktionen erhalten wurde:

```
int fd = open ("Datei", mode);
ifstream myfile1 (fd);
  // gepufferter Modus (Default)
ifstream myfile2 (fd, NULL, 0);
  // ungepufferter Modus
ifstream myfile3 (fd, buffer, size);
  // gepufferter Modus mit Benutzer-Puffer
```


C++-*Stream*-Ein-/Ausgabe in Dateien

- Ausgabe-Dateien können analog geöffnet werden.
- "`Datei`" ist der Dateiname und evtl. -Pfad als Null-terminierter ASCII-String; `iosmode` ist einer der in `iostream.h` definierten Enumeratoren, die den Zugriffsmodus auf die Datei angeben:

C++-*Stream*-Ein-/Ausgabe in Dateien

`ios::in`	Öffnen für Eingabe, oder Öffnen für Ausgabe, ohne dass die Datei verkürzt wird.
`ios::out`	Öffnen für Ausgabe.
`ios::nocreate`	Öffnen, ohne dass eine nicht existierende Datei neu erstellt wird.
`ios::app`	Öffnen für Anhängen von Daten an das Ende der Datei.
`ios::ate`	Ähnlich wie `ios::app`.
`ios::noreplace`	Falls die Ausgabedatei bereits existiert, misslingt das Öffnen.
`ios::binary`	Datei wird im Binär-Mode (statt im Text-Mode) geöffnet.

C++-*Stream*-Ein-/Ausgabe in Dateien

- Andere *Flags* steuern die gemeinsame Benützung von Dateien (*File Sharing*).
- Alternativ kann ein *Stream*-Objekt in Speicher vom *Free Store* erstellt werden:

```
ifstream *pmyfile = new ifstream;
    // Zeiger auf ein Stream-Objekt
pmyfile->open ("Datei", iosmode);
```

- Ein explizites Schließen einer geöffneten Datei (mit der Klassenelement-Funktion `close()`) ist in der Regel nicht notwendig, weil die Datei automatisch geschlossen wird, wenn die Gültigkeit des *Stream*-Objekts endet.
- Die C++-Ein-/Ausgabe-Funktionen sollen anhand eines Telefonregister-Programms illustriert werden:

C++-*Stream*-Ein-/Ausgabe in Dateien

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <string.h>

class tellist
{
public:
  static int zahl;          // Zahl der Eintragungen
  tellist () { *_name = 0; *_nummer = 0; }
  tellist (char *Name, char *Nummer)
  {
    strncpy (_name, Name, 40);
    _name[39] = 0;
    strncpy (_nummer, Nummer, 20);
    _nummer[19] = 0;
  }
```


C++-*Stream*-Ein-/Ausgabe in Dateien

```
  void disp ()
  {
    cout << setiosflags (ios::left) << "Name: "
      << setw (40) << _name << ": Nummer: "
      << setw (20) << _nummer << endl
      << resetiosflags (ios::left);
  }
private:
  char _name[40], _nummer[20];
};
```

C++-*Stream*-Ein-/Ausgabe in Dateien

```
void lesen (tellist *liste, int max_zahl)
{
  int bytes_rd = 0;          // Anzahl der gelesenen Bytes
  ifstream is ("TELLIST.DAT", ios::in | ios::binary
    | ios::nocreate);
  if (is)                    // erfolgreich geöffnet
  {
    is.read ((char*) liste, max_zahl * sizeof(tellist));
    bytes_rd = is.gcount (); // Anzahl der gelesenen Bytes
  }
  tellist::zahl = bytes_rd / sizeof (tellist);
  cout << tellist::zahl << " Eintragungen gelesen.\n";
}
```


C++-*Stream*-Ein-/Ausgabe in Dateien

```
void schreiben (tellist *liste)
{
  int bytes_wr = 0;          // geschriebene Bytes
  ofstream os ("TELLIST.DAT", ios::out | ios :: trunc |
    ios::binary);
  if (os)           // erfolgreich geöffnet
  {
    os.write ((char*) liste, tellist::zahl*sizeof(tellist));
    bytes_wr = os.tellp();
      // Position des Ausgabezeigers = geschriebene Bytes
  }
  cout << bytes_wr / sizeof (tellist) <<
    " Eintragungen geschrieben.\n";
}
```


C++-*Stream*-Ein-/Ausgabe in Dateien

```
int tellist::zahl = 0;     // statisches Datenelement
const int max_zahl = 20;   // maximale Zahl der Einträge
tellist liste[max_zahl];   // Telefonliste
int main ()
{
  lesen (liste, max_zahl); // Liste einlesen
  for (int i = 0; i < tellist::zahl; i++)
    liste[i].disp();       // Liste ausschreiben
  cout << "\nNeue Eintragungen eingeben; Ende mit "
            "leerem Namensfeld:\n";
```


C++-*Stream*-Ein-/Ausgabe in Dateien

```
  while (tellist::zahl < max_zahl)
  {
    char buffer1[128], buffer2[128];
    cout << "Name:    ";
    cin.getline (buffer1, 128);
    if (! *buffer1)        // leerer String
      break;
    cout << "Nummer:  ";
    cin.getline (buffer2, 128);
    if (! *buffer2)        // leerer String
      break;
    liste[tellist::zahl++] = tellist (buffer1, buffer2);
  }
  for (i = 0; i < tellist::zahl; i++)
    liste[i].disp();       // Liste ausschreiben
  schreiben (liste);       // Liste in Datei schreiben
}
```


Ein-/Ausgabe in C++

- Die C++ *Class Library* `iostream`
  - Allgemeines
  - Konsol-Ausgabe mit C++-*Streams*
  - Konsol-Eingabe mit C++-*Streams*
  - C++-*Stream*-Ein-/Ausgabe von/auf Puffer
  - C++-*Stream*-Ein-/Ausgabe in Dateien
  - **C- und C++-Ein-/Ausgabe im Vergleich**

Ein-/Ausgabe in C++

- Die C++ *Class Library* `iostream`
  - C- und C++-Ein-/Ausgabe im Vergleich
    - **Vorteile der C++-*Stream*-Ein-/Ausgabe-Funktionen**
    - Vorteile der Standard-C-Ein-/Ausgabe-Funktionen

Vorteile der C++-*Stream*-Ein-/Ausgabe

- Vollständige Typenprüfung:
  - Objekte werden immer korrekt eingelesen oder ausgegeben; Fehlinterpretationen wie bei den Standard-C-Funktionen sind unmöglich.
- Einfache formatierte Ausgabe:
  - Formatierte Ausgabe ist einfacher als mit den Standard-C-Funktionen.
- Weniger kritische Eingabe:
  - Eingabefunktionen von C++ sind weniger kritisch in ihren Anforderungen an die Formatierung der Eingabedaten.

Vorteile der C++-*Stream*-Ein-/Ausgabe

- Vereinheitlichte Handhabung:
  - Formatierte Ausgabe auf die Konsole, in einen Puffer oder in eine Datei ist konzeptuell identisch.
- Erweiterungsmöglichkeiten für benutzerdefinierte Datentypen:
  - *Overloading* der Operatoren "<<" und ">>" erlaubt auch benutzerdefinierte Klassen-Typen in Ein-/Ausgabe-Operationen.

Ein-/Ausgabe in C++

- Die C++ *Class Library* `iostream`
  - C- und C++-Ein-/Ausgabe im Vergleich
    - Vorteile der C++-*Stream*-Ein-/Ausgabe-Funktionen
    - **Vorteile der Standard-C-Ein-/Ausgabe-Funktionen**

Vorteile der Standard-C-Ein-/Ausgabe

- Geringerer Overhead:
  - Standard-C-Funktionen sind in Programmgröße und Geschwindigkeit der Ausführung deutlich effizienter.
- Einfachere Realisierung komplizierter Formate:
  - Aufwendige Formatierung von Ausgabedaten ist mit den Standard-C-Funktionen `printf()`, `sprintf()` und `fprintf()` wesentlich einfacher als mit den C++-*Stream*-Funktionen.

Vorteile der Standard-C-Ein-/Ausgabe

- Bequemere Fehlererkennung:
  - Die Standard-C-Ein-/Ausgabefunktionen geben ein Ergebnis zurück, das eine Prüfung auf Fehler erlaubt. Diese Information muss bei den C++-*Stream*-Funktionen durch separate Funktionsaufrufe ermittelt werden.
- Dynamischer Aufbau von Formaten:
  - Format-Information ist bei Standard-C-Funktionen in einem String enthalten, der dynamisch je nach den Erfordernissen des Programms erstellt oder geändert werden kann.

Ein-/Ausgabe in C++

- Befehlszeilenparameter und Rückgabewerte
- Die C++ *Class Library* `iostream`
- **Abspeichern von Klassen-Objekten in Dateien**

Abspeichern von Objekten in Dateien

- Zwei Wege für die Abspeicherung der in Klassen-Objekten enthaltenen Daten in einer Datei:
  - Umsetzung der Daten in Text; oder
  - binäre Abspeicherung des oder der Klassen-Objekte.
- Umsetzung in (ASCII-)Text erleichtert den Austausch von Daten mit anderen Applikationen und erlaubt die Modifikation mit einem Text-Editor.
- Binäre Abspeicherung ist kompakter, schneller und einfacher als eine Umsetzung in ASCII-Text; das Wiedereinlesen der Daten erfordert nicht eine Interpretation einer Text-Datei.

Abspeichern von Objekten in Dateien

- Binäre Abspeicherung von Gleitkommawerten vermeidet Genauigkeitsverluste durch Rundungsfehler.
- Bei binärer Abspeicherung von Klassenobjekten werden *alle* Elemente der Klasse abgespeichert (auch unsichtbare, z.B. zur Handhabung virtueller Funktionen).
- Das folgende Beispiel erweitert das "Bibliotheks-verwaltungsprogramm" aus dem Abschnitt "Einfache Vererbung" um eine Ein-/Ausgabe auf eine Plattendatei:

Abspeichern von Objekten in Dateien

```
#include <iostream.h>
#include <iomanip.h>
#include <strstrea.h>
#include <stdio.h>
#include <string.h>
class Dokument
{
public:
  Dokument() { anzahl++; }            // Default-Konstruktor
  virtual ~Dokument() { anzahl--; }   // Destruktor
  virtual void zeige() { }
  static int anzahl;      // Anzahl der definierten Objekte
```


Abspeichern von Objekten in Dateien

```
protected:
  int getval()  // Eingabefunktion für numerische Daten
  {
    char buf[80];
    cin.getline (buf, 80);
    istreamstr str (buf);             // Stream-Konstruktor
    str >> _i;                        // Wert einlesen
  }
  char _s1[40], _s2[20];              // Datenelemente
  int _i;
};
int Dokument::anzahl = 0;             // Definiere anzahl
```


Abspeichern von Objekten in Dateien

```
class Buch : public Dokument
{
public:
  Buch ()                             // Default-Konstruktor
  {
    cout << "Buch-Autor:  ";
    cin.getline (_s2, 20);
    cout << "Buch-Titel:  ";
    cin.getline (_s1, 40);
    cout << "Buch-Seiten: ";
    getval ();
  }
  void zeige ()
  {
    cout << setiosflags (ios::left) << "Autor:  "
       << setw (20) << _s2 << " Titel: "
       << setw (40) << _s1 << "\n\t" << _i << " Seiten\n";
  }
};
```


Abspeichern von Objekten in Dateien

```
class Manual : public Dokument
{
public:
  Manual ()                           // Default-Konstruktor
  {
    cout << "Manual-Titel:  ";
    cin.getline (_s1, 40);
    cout << "Manual-Seiten: ";
    getval ();
   }
  void zeige ()
  {
    cout << setiosflags (ios::left) << "Manual: "
      << setw (40) << _s1 << " - " << _i << " Seiten\n";
  }
};
```

Abspeichern von Objekten in Dateien

```
class Datei : public Dokument
{
public:
  Datei ()                              // Default-Konstruktor
  {
    cout << "Datei-Inhalt: ";
    cin.getline (_s1, 40);
    cout << "Datei-Pfad:   ";
    cin.getline (_s2, 20);
    cout << "Datei-Bytes:  ";
    getval ();
  }
  void zeige ()
  {
    cout << setiosflags (ios::left) << "Datei:  "
      << setw (40) << _s1 << " Pfad: "
      << setw (20) << _s2 << "\n\t" << _i << " Bytes\n";
  }
};
```


Abspeichern von Objekten in Dateien

```
void lesen (Dokument **ppD, int max_zahl)
{
  FILE *file;                           // File Pointer
  int i = 0;
  if (file = fopen ("LITLIST.DAT", "rb"))
  {                                     // erfolgreich geöffnet
    for ( ; i < max_zahl; i++)
    {
      ppD[i] = new Dokument;
      if (! fread (ppD[i], sizeof(Dokument), 1, file))
      {                                 // am Ende der Datei
        delete ppD[i];
        break;
      }
    }
    fclose (file);
  }
  cout << i << " Eintragungen gelesen.\n";
}
```


Abspeichern von Objekten in Dateien

```
void schreiben (Dokument **ppD)
{
  FILE *file;                           // File Pointer
  int i = 0;
  if (file = fopen ("LITLIST.DAT", "wb"))
  {                                     // erfolgreich geöffnet
    for ( ; i < Dokument::anzahl; i++)
      if (! fwrite (ppD[i], sizeof(Dokument), 1, file))
        break;                          // Fehler beim Schreiben
    fclose (file);
  }
  cout << i << " Eintragungen geschrieben.\n";
}
const int max_Eintr = 20;
Dokument *doc[max_Eintr];               // Feld von Zeigern
```


Abspeichern von Objekten in Dateien

```
int main ()
{
  lesen (doc, max_Eintr);               // Liste einlesen
  for (int i = 0; i < Dokument::anzahl; i++)
    doc[i]->zeige();                    // Liste ausschreiben
  cout << Dokument::anzahl << " Eintragungen.\n";
  while (Dokument::anzahl < max_Eintr)
  {
    char buf[10];
    if (! cin)                          // Eingabe-Fehler
    {
      cout << "\nFehlerhafte Eingabe!\n";
      cin.clear();                      // Fehler-Flags löschen
    }
    cout << "'B'uch, 'M'anual, 'D'atei, oder 'E'nde ?  ";
    cin.getline (buf, 10);              // Schaltzeichen
```


Abspeichern von Objekten in Dateien

```
    switch (*buf)
    {
    case 'B':
    case 'b':
      doc[Dokument::anzahl - 1] = new Buch;
      break;
    case 'M':
    case 'm':
      doc[Dokument::anzahl - 1] = new Manual;
      break;
    case 'D':
    case 'd':
      doc[Dokument::anzahl - 1] = new Datei;
      break;
```


Abspeichern von Objekten in Dateien

```
    case 'E':
    case 'e':
      goto weiter;
    default:
      cout << "\nFalscher Befehl!\n";
    }
  }
weiter:                     // Einsprung bei Programm-Abbruch
  for (i = 0; i < Dokument::anzahl; i++)
    doc[i]->zeige();        // Liste ausschreiben
  cout << Dokument::anzahl << " Eintragungen.\n";
  schreiben (doc);          // Liste in Datei schreiben
}
```

Abspeichern von Objekten in Dateien

- Aufruf der korrekten virtuellen Funktionen ist auch nach dem Abspeichern und Wiederladen der Daten möglich, weil *alle* Datenelemente in der Klasse `Dokument` konzentriert sind, und weil die Information über virtuelle Funktionen in `Dokument` (unsichtbar) als Datenelement abgelegt wird.
- Diese Methode ist nur dann praktikabel, wenn sicher-gestellt werden kann, dass das gleiche Programm die Daten wieder liest, das es auch geschrieben hat (beispielsweise bei temporär angelegten Dateien).
- Wenn die gespeicherten Daten portabel sein sollen, muss ein alternativer Weg beschritten werden:

Abspeichern von Objekten in Dateien

```
// #includes wie zuvor
// Aufzählungs-Type zur Charakterisierung des Objekts
enum dtype { dokument, buch, manual, datei };
class Dokument
{
public:
  Dokument()
  {
    anzahl++;
    _dt = dokument;
  }
  virtual ~Dokument() { anzahl--; }
  virtual void zeige() { }
  static int anzahl;
```


Abspeichern von Objekten in Dateien

```
protected:
  int getval()                          // Eingabefunktion
  {
    // Definition von getval() wie zuvor
  }
  char _s1[40], _s2[20];
  int _i;
  dtype _dt;
  friend void lesen (Dokument **, int);
};
int Dokument::anzahl = 0;
```


Abspeichern von Objekten in Dateien

```
class Buch : public Dokument
{
public:
  Buch ()                           // Default-Konstruktor
  {
    cout << "Buch-Autor:  ";
    cin.getline (_s2, 20);
    cout << "Buch-Titel:  ";
    cin.getline (_s1, 40);
    cout << "Buch-Seiten: ";
    getval ();
    _dt = buch;                     // Daten-Type
  }
```


Abspeichern von Objekten in Dateien

```
  Buch (Dokument &d)              // alternativer Konstruktor
  {
    strcpy (_s1, d._s1);
    strcpy (_s2, d._s2);
    _i = d._i;
    _dt = buch;
  }
  void zeige ()
  {
    // Definition von zeige() wie zuvor
  }
};
```


Abspeichern von Objekten in Dateien

```
class Manual : public Dokument
{
  // analoge Definition wie zuvor, mit "_dt = manual;" im
  // Default-Konstruktor; alternativer Konstruktor wie bei
  // class Buch, jedoch mit "_dt = manual;"
};
class Datei : public Dokument
{
  // analoge Definition wie zuvor, mit "_dt = datei;" im
  // Default-Konstruktor; alternativer Konstruktor wie bei
  // class Buch, jedoch mit "_dt = datei;"
};
```

Abspeichern von Objekten in Dateien

```
void lesen (Dokument **ppD, int max_zahl)
{
  FILE *file;
  int i = 0;
  Dokument Temp;           // Hilfs-Objekt
  if (file = fopen ("LITLIST1.DAT", "rb"))
  {
    for ( ; i < max_zahl; i++)
    {
      if (! fread (&Temp, sizeof (Dokument), 1, file))
        break;
      switch (Temp._dt)      // verzweige je nach Datentype
      {
      case dokument:
        cout << "Falsche Objekt-Type!\n";
        i--;                 // kein Feldelement definiert
        break;
```


Abspeichern von Objekten in Dateien

```
      case buch:
        ppD[i] = new Buch (Temp);
        break;
      case manual:
        ppD[i] = new Manual (Temp);
        break;
      case datei:
        ppD[i] = new Datei (Temp);
        break;
      }
    }
    fclose (file);
  }
  cout << i << " Eintragungen gelesen.\n";
}
```


Abspeichern von Objekten in Dateien

```
void schreiben (Dokument **ppD)
{
  // gleiche Definition wie zuvor für schreiben()
}
const int max_Eintr = 20;
Dokument *doc[max_Eintr];        // Feld von Zeigern
int main ()
{
  // gleiche Definition wie zuvor für main()
}
```