Ein- und Ausgabe

Die elementare Ausgabe von Daten auf externe Medien, wie Dateien und das Einlesen davon wird demonstriert.

Komplexe E/A-Operationen werden erst diskutiert, nachdem das Klassenkonzept erarbeitet ist.

Inhalt

1. Standardein- und Ausgabe 2

   1.1. Eingabe 3

2. Ein- und Ausgabe mit Dateien 7

3. Binärdateien 12

1. Standardein- und Ausgabe

Die Ein- und Ausgabe unter Verwendung von Ein/Ausgabekanälen haben wird bereits gesehen.

Folgende Ein- und Ausgabeströme sind vordefiniert:

- `cin` – Standardeingabe (i.A. die Tastatur)
- `cout` – Standardausgabe (i.A. der Bildschirm)
- `cerr` – Standardfehlerausgabe (i.A. der Bildschirm)

1.1.Eingabe

Durch die Funktion `get()` kann man **einzelne Zeichen** des Eingabestroms lesen, mit `put()` ein Zeichen auf den Ausgabestrom schreiben:

Code	Beschreibung

`char c;`
`while (cin.get(c))`
`  cout.put(c);`
	Zeichenweise Kopieren

`char buf[10];` `cin.get(buf,6);`	Lesen von maximal 6 Zeichen und Ablage im Array buf.
`char buf[10];` `cin.get(buf,6,'\n');`	Übernahme von maximal 6 Zeichen, wobei maximal bis zu einem Terminatorzeichen ‚\n' gelesen wird. Das Terminatorzeichen verbleibt im Eingabestrom

`int  i;`
`while ( (i=cin.get()) != EOF`
`    cout.put(char(i));`
	Get holt nächstes Zeichen und erzeugt int-Wert gemäß ASCII Tabelle.

Eine **ganze Zeile** kann mit der Funktion `getline()` gelesen werden.

Eine **Vorschau** (also ohne ein Zeichen zu konsumieren) auf den Eingabestrom ist durch `peek()` möglich.

Ein bereits konsumiertes Zeichen `c` kann durch `putback(c)` **zurückgelegt** werden;

also ist „`c=cin.get(); cin.putback(c));`“ gleichbedeutend mit „`c=peek();`“.

Durch **Ein/Ausgabeumlenkung** auf Ebene des Betriebssystems kann man so genannte Filterprogramme schreiben:

> Ein Filter liest von Standardeingabe und schreibt das Ergebnis auf die Standardausgabe ohne Benutzerinteraktion.

Beispiel (cat von Unix, type von DOS):

```
$ cat simpelCat.cpp
#include <iostream.h>
int main(int argc, char* argv[]) {
        char c;
        while (cin.get(c))
                cout.put(c);
        cout << endl;
}
$
```

Auf Betriebssystemebene kann man in Unix (auch unter DOS) die Eingabe umlenken, indem man das Zeichen „<“ verwendet. Damit lässt sich dann das o.a. Programm verwenden, um eine beliebige Datei zu lesen:

```
$ simpleCat < simpleCat.cpp
#include <iostream.h>
int main(int argc, char* argv[]) {
        char c;
        while (cin.get(c))
                cout.put(c);
        cout << endl;
}
$
```

Die Umlenkung der Ausgabe ist auch möglich, damit ist das Programm auch zum Kopieren von Dateien zu gebrauchen:

```
$ simpleCat < simpleCat.cpp > kopieVonSinpmleCat.cpp
$ simpleCat < kopieVonSinpmleCat.cpp
#include <iostream.h>
int main(int argc, char* argv[]) {
        char c;
        while (cin.get(c))
                cout.put(c);
        cout << endl;
}
$
```

2. Ein- und Ausgabe mit Dateien

Das Lesen und Schreiben von Dateien über die Ein-/Ausgabeumlenkung ist eigentlich nur für die Systemprogrammierung gedacht (neue BS-Kommandos, ...).

In Anwendungsprogrammen muss häufig auf externe Medien zugegriffen werden. Dazu können die Ein- und Ausgabeoperatoren >> und << zusammen mit dem Header <fstream> verwendet werden.

Die zu verwendeten Funktionen put(), get(), open(), close() benutzen dabei Systemaufrufe, die das zugrunde liegende Betriebssystem bereitstellt.

Der Zugriff auf eine Datei geschieht etwa wie folgt:

1. Definition eins Datei-Objektes, über das die Datei angesprochen wird:
   Datentyp des Dateiobjektes ist ifstream zum Lesen, ofstream zu Schreiben.
2. Die Verbindung zu einer Datei wird durch die Funktion open() hergestellt; dabei übergibt man den Namen der Datei als Parameter.
3. Nachdem kein Zugriff mehr auf die Datei erforderlich ist, kann die Verbindung durch close() gelöst werden.

Das Schreiben auf eine Datei erfolgt i.A. gepuffert, d.h. ein spezieller Datenbereich (Puffer) wird reserviert. Erst wenn der Puffer voll ist, wird der Pufferinhalt physisch auf die Platte geschrieben. Ein `close()` sorgt dafür, dass stets geschrieben wird, auch wenn der Puffer noch nicht voll ist.

Das Lesen erfolgt i.A. auch gepuffert.

Beispiel (simpleCp):

```
$ cat simpleCp.cpp
#include <iostream.h>                              // wg. cerr
#include <cstdlib>                                 // wg. exit
#include <fstream.h>                               // wg. Dateioperationen
int main(int argc, char *argv[]) {                 // copy argv[1] argv[2]
        if (argc != 3) {                           // falsche Verwendung
                cerr << "Verwendung: " << argv[0] << " Quelle Ziel" << endl;
                exit (-1);
        }
```

```
$ simpleCp
Verwendung: simpleCp.exe Quelle
Ziel
$ simpleCp  /etc/passwda xx
/etc/passwda kann nicht geöffnet
werden!
$ simpleCp  /etc/passwd xx
$
```

```
        ifstream quelle;                        // Quelldatei (Lesen)
        quelle.open(argv[1], ios::in);          // zum Lesen öffnen
        if (!quelle) {                          // Datei kann nicht geoeffnet werden
                cerr << argv[1] << " kann nicht geöffnet werden!\n";
                exit(-2);
        }

        ofstream ziel;                          // Zieldatei (Schreiben)
        ziel.open(argv[2], ios::out);           // zum Schreiben öffnen
        if (!ziel) {                            // Datei kann nicht geoeffnet werden
                cerr << argv[2] << " kann nicht geöffnet werden!\n";
                exit(-3);
        }

        char c;
        while (quelle.get(c))                   // zeichenweise kopieren
                ziel.put(c);

        quelle.close();                         // Schliessen der Quelldatei
        ziel.close();                           // Schliessen der Zieldatei
}
$
```

Sollen nicht nut Textdateien, sondern auch Binärdateien bearbeitet werden, so ist beim Öffnen der Schalter `ios::binary` anzugeben. Dabei wird verhindert, dass die (in DOS) sonst automatisch Umwandlung von Zeilenendekennung '\n' in CR/LF erfolgt:

```
ziel.open(argv[2], ios::binary|ios::out);
```

ODER Verknüpfung von Schaltern

Weitere Schalter sind:

`ios::app`	append: beim Schreiben anm Ende der Datei anhängen
`ios::ate`	nach dem Öffnen an das Dateiende springen
`trunc`	Voheriger Inhalt der Datei Löschen

Weiterhin sind gewisse Statusinformationen mittels Funktionen abfragbar, etwa ob das Ende der Datei schon erreicht ist durch `eof()`.

Beispiel (lineCount)

```
char c;
int zaehler = 0;
while (!datei.eof()) {                    // Zeichenweise lesen
        datei.get(c);
        if (c=='\n') zaehler++;
}
```

Neben dem Arbeiten mit Zeichen, können auch Werte (im ASCII Format) von einer Datei gelesen bzw. auf sie geschrieben werden (im ASCII Format) werden.

```
double x;
ifstream dateiIn;
ofstream dateiOut;
…
dateiIn >> x; // Lesen von Datei
…
dateiOut << x+1;
```

Hörsaalübung:

In einer Datei stehen double Messwerte. Schreiben Sie ein Programm, dass die als Parameter angegebene Messwertdatei liest und

- Anzahl,
- Summe,
- Mittelwert
- und kleinsten und größten Messwert

ausgibt und zusätzlich in einer Datei speichert.

3. Binärdateien

Bisher sind die Lese- und Schreioperationen immer auf **ASCII** Dateien ausgeübt worden. Dabei wird ein double immer in eine Folge von Ziffern (und Dezimalpunkt) umgewandelt. Wird z.B. der Wert -1.234567890 in eine ASCI Datei geschrieben, so werden insgesamt 12 Byte in die Datei geschrieben.

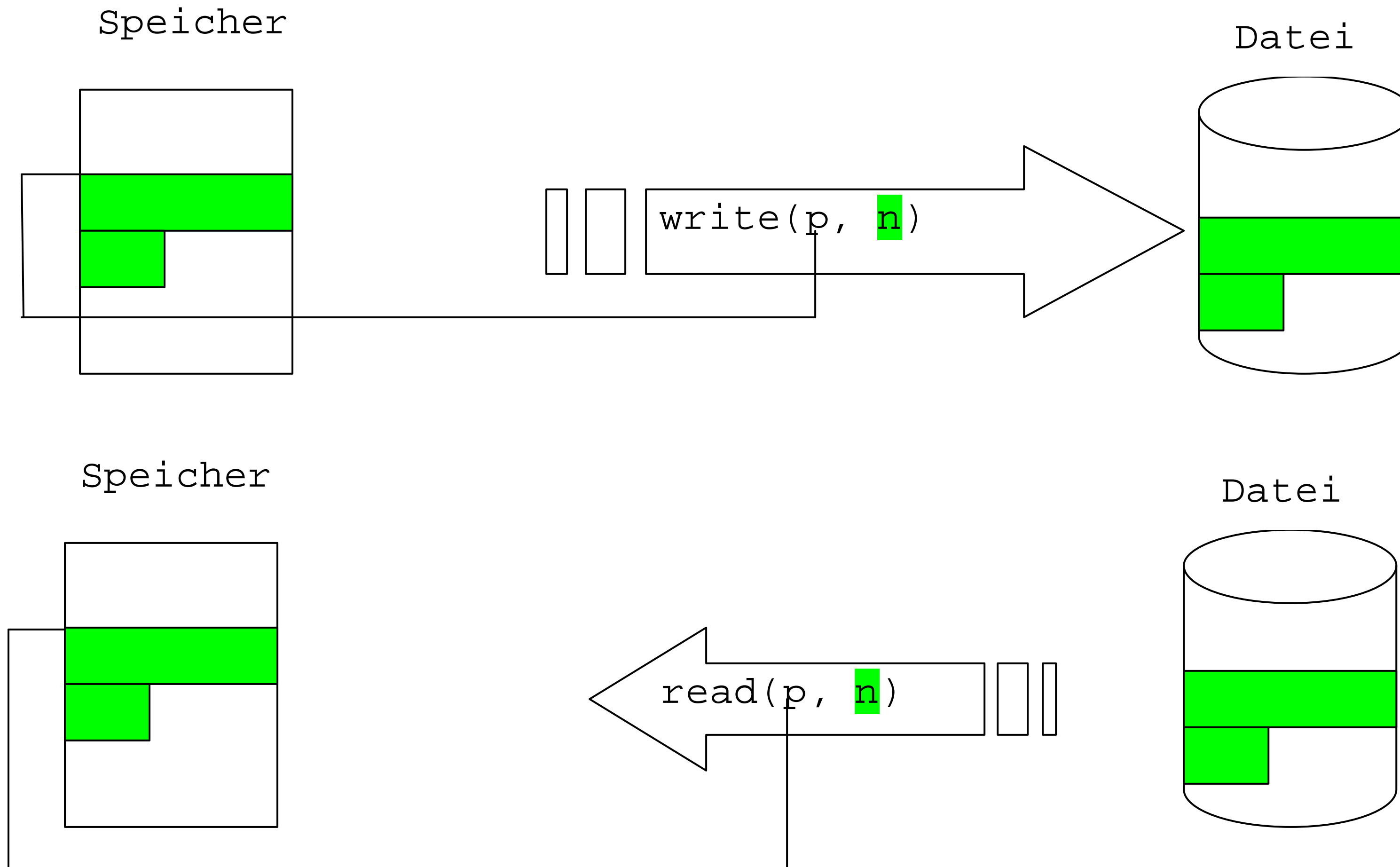
Wird derselbe Wert in eine Binärdatei geschrieben, so werden so viele Bytes belegt, wie ein double ausmacht: in vielen Rechnern 8 Byte.

Während die **ASCII Datei** mit einem normalen **Editor** lesbar ist, kann eine **Binärdatei** nur mit **Hex-Editoren** gelesen werden oder mit **speziellen Programmen**, die „wissen“, wie das Binärbild zu lesen ist.

Binäre Ein-/ und Ausgabe ist also unformatiert: beim Lesen und Schreiben wird keine Konvertierung, etwa von ‚\n' vorgenommen.

Zum Schreiben und Lesen existieren die Funktionen `write(adresse,bytes)` und `read(adresse, bytes)`.

Die Wirkung ist:



Soll also ein double-Wert in eine Datei geschrieben und anschließend gelesen werden, ist folgendes Programm erforderlich:

```
$ cat doubeBinary.cpp
...
        double d;

        ofstream ausgabe;
        ausgabe.open(dateiname.c_str(), ios::out|ios::binary);
        if (!ausgabe) {                          // Datei kann nicht geoeffnet werden
                cerr << dateiname << " kann nicht geöffnet werden!\n";
                exit(-2);
        }
        // Anzahl Bytes von double ab Adresse von d in Datei schreiben
        ausgabe.write((char *)&d, sizeof(d));
        ausgabe.close();

        ifstream eingabe;
        eingabe.open(dateiname.c_str(), ios::in|ios::binary);
        if (!eingabe) {                          // Datei kann nicht geoeffnet werden
                cerr << dateiname << " kann nicht geöffnet werden!\n";
                exit(-2);
        }
        // Anzahl Bytes von double aus Datei lesen und ab Adresse d speichern
        eingabe.read((char *)&d, sizeof(d));
        eingabe.close();
...
```

Hörsaalübung:

Schreiben Sie ein Programm, das eine Folge von double einliest, in einem Array speichert und das gesamte Array mit einem `write()` in eine Binärdatei speichert.

In einem zweiten Programm soll die obige Binardatei so gelesen werden, dass jeweils ein double-Wert mittels `read ()` in einer Arraykomponentre abgelegt wird.