Rechtlicher Hinweis

Legal notice

Informatik I (B.Sc. Physik)

Ein- und Ausgaben

Dr. Paul Bodesheim
(Paul.Bodesheim@uni-jena.de)

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Fakultät für Mathematik und Informatik
Lehrstuhl für Digitale Bildverarbeitung

SoSe 2020

Inhalt

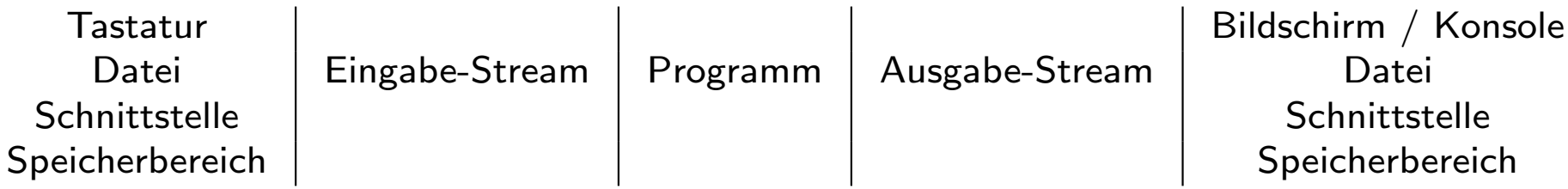
Inhalt

Ein- und Ausgabe allgemein

- Ein- und Ausgabe ist jeglicher Datenaustausch des Programms mit seiner "Umwelt":
  - Kommunikation mit dem Nutzer
  - Speicherung von Dateien auf einem Speichermedium
  - Kommunikation mit anderen Programmen
    - lokal oder
    - über das Netzwerk / Internet
- Die meisten dieser Ein- und Ausgaben laufen in C++ über **streams**
- Gleichartige Behandlung aller Ein- und Ausgaben
- Mit Einschränkungen, z.B. interaktive Eingabe teilweise anders behandeln als Datei einlesen

Was sind streams?

- Ein *Stream* ist ein abstrakter Datenkanal
- An einem Ende liest oder schreibt das Programm Daten
- Am anderen Ende steht:
  - Die Tastatur oder der Bildschirm / Konsole
  - Eine Datei
  - Eine Schnittstelle
  - Ein Speicherbereich



Arbeiten mit streams

3 Typische Schritte beim Verwenden von streams:

1. Öffnen
   - Arbeit mit dem *stream* vorbereiten
   - Verbinden der Ein- oder Ausgabe mit einer Datei oder einem Gerät
   - Bei Notwendigkeit werden Sperren gesetzt, um konkurrierende Bearbeitung zu verhindern
2. Lesen oder Schreiben
   - Lesen oder Schreiben der eigentlichen Daten
3. Schließen
   - Beenden der Arbeit mit dem *stream*
   - Phyisches Schreiben der Daten, die noch nicht abgelegt worden sind / noch im Puffer (Buffer) sind
   - Ggf. Freigabe von Sperren

Formen der Ein- und Ausgabe

2 grundsätzliche Formen der Ein- und Ausgabe:

1. Formatierte Ein- und Ausgabe
   - Alle Darstellungen in Textform
   - Wie bei der normalen und bereits bekannten Ein- und Ausgabe am Bildschirm
   - Auch für Dateien sinnvoll, lassen sich dann vom Anwender lesen
2. Unformatierte Ein- und Ausgabe
   - Ablegen der Daten in ihrer internen Repräsentation ("binär")
   - Für den Anwender nicht lesbar, daher nicht für Konsole geeignet
   - In der Regel kompakter
   - Anwendungsbeispiel: Bilder

Streams in Klassen (mit Header-Dateien)

- <*istream*>: input stream (Eingabe-Stream)
- <*ostream*>: output stream (Ausgabe-Stream)
- <**iostream**>: input-output stream (bindet die beiden vorherigen ein)

- <**fstream**>: file stream (für Dateien), definiert Klassen für:
- *ifstream*: input file stream (Eingabe-Stream für Dateien)
- *ofstream*: output file stream (Ausgabe-Stream für Dateien)

- <**sstream**>: string stream (für Zeichenketten / strings), definiert folgende Klassen: *stringstream*, *istringstream*, *ostringstream*:

Inhalt

Die Konsole

- Die Konsole ist Standard für die Text-Eingabe und die Bildschirm-Ausgabe
- Daten als Datenstrom: fortlaufende Abfolge von Zeichen
- Keine (direkte) Steuerung der Position der Zeichen auf dem Bildschirm bei der Ausgabe

  $\Rightarrow$ Konsole wie Datei behandeln
  (Datei kennt auch keine "Position auf dem Bildschirm")

- Synonyme für Konsole / ähnliche Begriffe:
  - Terminal: Schnittstelle für Nutzerinteraktion
  - Kommandozeile (command line): Schnittstelle für Befehle
  - Shell: Interpreter für Kommandozeile (command-line shell)

Spezielle streams für die Konsole

- **cout**: Ausgabe auf Konsole
- **cin**: Eingabe von der Konsole
- **cerr**: "Fehlerkanal" für Fehlermeldungen gedacht
  - Ungepufferter *stream*, Ausgabe wird typischerweise nicht umgeleitet
  - Muss nicht ausschließlich für Fehlermeldungen verwendet werden
  - Für Ausgaben, die den Anwender unbedingt erreichen müssen, z.B. Fehlermeldungen
- Konsolen-Streams **cout, cin, cerr** sind nach Import von *iostream* vorhanden (**#include** <iostream>)

  $\Rightarrow$ Sie müssen weder definiert, noch geöffnet oder geschlossen werden
- Wichtigstes Mittel zum Lesen von und zur Ausgabe auf der Konsole: Operatoren << und >>

Inhalt

Die binären Operatoren << und >>

- Beide Operatoren sind überladen für entsprechende *streams* (Konsolen-Streams, Datei-Streams, ...)
- Operator << für Ausgaben, >> für Eingaben

- Linker Operand ist immer der *stream*
- Rechter Operand:
  - Variable zum Speichern der Eingabe
  - Daten (über Variablen, Konstanten, Literale) zum Ausgeben

- Bei der Ausgabe: Umwandlung aller Argumente in Textform, bevor sie ausgegeben werden
- Kein automatisches Einfügen Trennzeichen $\Rightarrow$ Verantwortung des Programmierers

- Ein- und Ausgaben erfolgen **gepuffert**

Eingabe-Puffer

- Verarbeitung der (Nutzer-) Eingaben über >> erst nach Drücken der Enter-Taste, erst dann wird ausgewertet
- Mehrere Eingabebefehle (z.B. cin):
  - Eingabebefehle werden solange abgearbeitet, wie weitere Zeichen im Puffer vorhanden sind
  - Erst wenn bei Eingabebefehl kein Zeichen mehr im Puffer ist, wird auf eine (Nutzer-) Eingabe gewartet
  - Mehrfacheingabe durch Verkettung mit >> möglich, z.B.
    cin >> x >> y >> z;
- Einlesen eines Wertes mit dem Operator >> wie folgt:
  - Überlesen aller "white space"-Zeichen (Leerzeichen, Tabulator, Zeilenvorschub/neue Zeile)
  - Einlesen des Wertes:
    - bis zum nächsten "white space"-Zeichen
    - bis zum Auftreten eines Fehlers
- Weitere Zeichen bleiben im Puffer und können durch nächsten Eingabebefehl eingelesen werden

Eingabe-Puffer: Beispiel 1

```
#include <iostream>

using namespace std;

int main()
{
    int x,y,z;

    cout << "x eingeben: ";
    cin >> x;

    cout << "y eingeben: ";
    cin >> y;

    cout << "z eingeben: ";
    cin >> z;

    cout << "x=" << x << " , y=" << y << " , z=" << z << endl;

    return 0;
}
```

Eingabe-Puffer: Beispiel 2 (Verkettung)

```
#include <iostream>

using namespace std;

int main()
{
    int x,y,z;

    cout << "3 Zahlen eingeben: ";

    cin >> x >> y >> z;

    cout << "x=" << x << " , y=" << y << " , z=" << z << endl;

    return 0;
}
```

Ausgabe-Puffer

- Ausgaben mittels << erfolgen nicht sofort / erscheinen nicht sofort auf dem Bildschirm
- Sie landen zunächst im Puffer
- Leerung des Puffers (und dadurch tatsächliche Ausgabe der Daten), wenn:
  - Der Puffer voll ist
  - Eine Zeile mit endl abgeschlossen wird
  - Eine Eingabe erwartet wird
  - Die Methode flush aufgerufen wird, z.B. cout.flush()

Beispiel: Ausgabe-Puffer

```
#include <iostream>
#include <chrono>
#include <thread>
using namespace std;

void sleep(int s) { this_thread::sleep_for(chrono::seconds(s)); }
```

Variante 1

```
int main()
{
    cout << "1 ";


    sleep(5);
    cout << "2 ";


    sleep(5);
    cout << "3 ";

    return 0;
}
```

Variante 2

```
int main()
{
    cout << "1 " << endl;


    sleep(5);
    cout << "2 " << endl;


    sleep(5);
    cout << "3 " << endl;

    return 0;
}
```

Variante 3

```
int main()
{
    cout << "1 ";
    cout.flush();

    sleep(5);
    cout << "2 ";
    cout.flush();

    sleep(5);
    cout << "3 ";

    return 0;
}
```

Inhalt

Fehlerhafte Eingaben

- cin überprüft keine Datentypen
- Was passiert, wenn anstelle einer Zahl ein Buchstabe oder eine Zeichenkette eingegeben wird?

```
#include <iostream>

using namespace std;

int main()
{
    int i;

    cout << "Zahl eingeben: " << endl;;
    cin >> i;

    cout << "i=" << i << endl;

    return 0;
}
```

Status eines streams

- Methoden zum Abfragen des Status eines *streams*
- Liefern Wahrheitswert

- **cin.good()**: true, falls kein Fehler bei der Eingabe aufgetreten ist
- **cin.fail()**: true, falls Eingabe wegen eines Fehlers nicht ausgeführt werden konnte (z.B. falscher Datentyp)

- Damit kann überprüft werden, ob die Eingabe erfolgreich war
- Diese Methoden existieren auch für andere *streams*, z.B. Datei-Streams

- Weitere Methoden für einen *stream* s:
  - **s.eof()**: true, falls Eingabe wegen Dateiende nicht ausgeführt werden konnte (end-of-file) $\Rightarrow$ eof() auf Konsole via STRG+D
  - **s.bad()**: true, falls Fehler im stream nicht behebbar ist

Beispiel: Falsche Eingaben abfangen

```
#include <iostream>

using namespace std;

int main()
{
    int i;
    cout << "Zahl eingeben: " << endl;
    cin >> i;

    while (cin.fail())
    {
        cout << "Falsche Eingabe! Zahl eingeben: " << endl;
        cin >> i;
    }
    cout << "i=" << i << endl;

    return 0;
}
```

- Falsche Eingabe (Fehler) führt zu Endlos-Schleife!
- Puffer wird bei Fehler nicht geleert, fehlerhaftes Zeichen bleibt im Puffer
- Nachfolgende Eingabe-Befehle scheitern ebenfalls

Fehlerhafte Eingaben ignorieren

- cin . ignore (streamsize n=1, **int** delim = EOF)

  ⇒ entnimmt Zeichen aus dem Puffer und ignoriert diese, entweder bis n Zeichen ignoriert wurden oder bis ein Zeichen gleich dem Trennzeichen delim ist (EOF ist hier vordefinierte Konstante)
- Kann auch genutzt werden, um Mehrfacheingaben (mit "white space" dazwischen) zu unterbinden

- cin . clear ()

  ⇒ Fehlerzustand löschen (Zustand auf *good* setzen)

  ⇒ Mit clear lassen sich explizit auch andere Zustände setzen (*eof*, *fail*, *bad*)

  ⇒ Diese sind zusammen mit *good* als Schalter (flags) implementiert

- Beide Funktionen existieren auch für andere *input streams*

Beispiel: Falsche Eingaben abfangen und ignorieren

```
#include <iostream>
#include <limits>

using namespace std;

int input(const string & prompt);

int main()
{
    int x,y,z;
    x = input("x: ");
    y = input("y: ");
    z = input("z: ");

    cout << "x=" << x << " , y=" << y << " , z=" << z << endl;
    return 0;
}
```

Beispiel: Falsche Eingaben abfangen und ignorieren (2)

```
int input(const string & prompt)
{
    int i;
    cout << prompt;
    cin >> i;

    while (cin.fail())
    {
        cin.clear(); // Fehlerzustand loeschen

        // Puffer loeschen (bis maximale Groesse bzw. Zeilenende)
        cin.ignore(numeric_limits<streamsize>::max(), '\n');

        cout << endl << "Falsche Eingabe!" << endl;
        cout << prompt;
        cin >> i;
    }

    // Puffer loeschen damit Zahlen einzeln eingegeben werden
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    return i;
}
```

Zeichenketten einlesen

- Für die Eingabe von Zeichenketten eignet sich der Operator >> nur bedingt
- Grund: Leerzeichen ist "white space" und "unterbricht" die Eingabe von Wortgruppen und Sätzen
- Es würden sich nur einzelne Wörter einlesen lassen

- Funktion **getline** liest eine ganze Zeile (also bis zum nächsten Zeilenvorschub) ein:

```
string s;
getline(cin,s); // Eingabe in s speichern
```

- Zeichenketten einlesen, die Leerzeichen enthalten
- Auch für andere *input streams* verwendbar

Zeichenweise einlesen oder ausgeben

- Zeichenweise lesen mit überladener Funktion **get(char c)** bzw. **get()**
- Auch für andere *input streams* als cin

```
char c;
cin.get(c); // Eingabe in c speichern
c = cin.get(); // Alternative
```

- Zeichenweise schreiben / ausgeben mit Funktion **put(char c)**

```
string s = "Test";
for (int i=0; i<s.size(); i++)
{
    cout.put(s[i]);
    cout.put('\n');
}
```

Inhalt

Methoden zur Formatierung

- Manipulatoren zur Steuerung der Ausgabeform
- bereits bekannt: *endl*

  $\Rightarrow$ Schließt eine Zeile ab, gibt Zeilenvorschub aus, leert Puffer

- Weitere Funktionalitäten einbinden über **#include** <iomanip>
  - *fixed*: Festpunktdarstellung für Gleitkommazahlen
  - *scientific*: Exponentialdarstellung für Gleitkommazahlen
  - *setw(int n)*: Breite der Darstellung der nächsten Ausgabe angeben
  - *setprecision(int n)*: Genauigkeit (Anzahl Nachkommastellen)
  - *boolalpha*: true/false statt 0/1 für Boolesche Werte
  - *noboolalpha*: 0/1 statt true/false für Boolesche Werte

Beispiel: Formatierung

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    double pi = 3.14159265;

    cout << setprecision(5);
    cout << scientific << pi << endl;
    cout << fixed << pi << endl;

    cout << setprecision(10);
    cout << setw(20) << pi << endl;
    cout << setw(5) << pi << endl;

    cout << boolalpha << (3>5) << " " << (3<5) << endl;
    cout << noboolalpha << (3>5) << " " << (3<5) << endl;
}
```

Inhalt

Arbeiten mit Dateien

- Klassen *ifstream* (input file stream) und *ofstream* (output file stream)
- Definiert in Header-Datei **fstream** $\Rightarrow$ **#include** <fstream>
- Eingabe / aus Datei lesen bzw. Ausgabe / in Datei schreiben

- 1.Schritt: Öffnen
  - *Stream* mit Datei verbinden
  - Lese- bzw. Schreiboperationen vorbereiten
  - Methode für *streams*: **open**

Beispiel: Datei-Inhalt kopieren

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream ifs;
    ifs.open("eingabe.txt");
    // Alternativ: ifstream ifs("eingabe.txt");

    string outfile="ausgabe.txt";
    ofstream ofs;
    // Alternativ: ofstream ofs("ausgabe.txt");

    ofs.open(outfile); // C++11
    // vor C++11: ofs.open(outfile.c_str());

    ... // weiter auf folgender Folie
```

Beispiel: Datei-Inhalt kopieren (Fortsetzung)

```
    ... // Fortsetzung von letzter Folie

    string str;

    while (ifs.good())
    {
        ifs >> str;

        if (ifs.good())
            ofs << str << endl;
    }

    ifs.close(); // muss nicht
    ofs.close();

    return 0;
}
```

- Übersetzung mit **-std=c++11**
- Beachte: Operator >> liest keine "white space"-Zeichen

Anpassung: Datei zeilenweise kopieren

```
    ... // Fortsetzung von vorletzter Folie

    string str;

    while (ifs.good())
    {
        getline(ifs,str); // Ersetzte Zeile

        if (ifs.good())
            ofs << str << endl;
    }

    ifs.close(); // muss nicht
    ofs.close();

    return 0;
}
```

Vereinfachen der while-Schleife

- Eingaben mit Operator >> lassen sich als logische Ausdrücke verwenden
- true, falls alles gut gegangen ist (keine Fehler bei der Eingabe)

```
while (ifs.good())
{
    ifs >> str;

    if (ifs.good())
        ofs << str << endl;
}
```

```
while (ifs >> str)
{
    ofs << str << endl;
}
```

- Gleiches gilt für **getline**

```
while (ifs.good())
{
    getline(ifs,str);

    if (ifs.good())
        ofs << str << endl;
}
```

```
while (getline(ifs,str))
{
    ofs << str << endl;
}
```

Fehlermeldungen und Status

- Geeignete Fehlerabfragen und -meldungen zeichnen ein gutes Programm aus

  $\Rightarrow$ Später nochmal genauer bei Ausnahmen

- Statusmeldungen wie bereits definiert:
  - **ifs.good()**: kein Fehler
  - **ifs.fail()**: Fehler aufgetreten
  - **ifs.eof()**: end-of-file (Dateiende), Fehler
  - **ifs.bad()**: nicht behebbarer Fehler

Überschreiben vs. Anhängen

- Existiert Ausgabedatei nicht, wird sie neu angelegt
- Was passiert, wenn die Ausgabedatei bereits existiert?

- Beobachtung: existierende Ausgabedatei wird überschrieben!
- $\Rightarrow$ Standard-Verhalten für Datei-Ausgabe

- Soll Ausgabe an bisherigen Inhalt angehangen werden, muss dies dem *stream* (*ofstream*) explizit mitgeteilt werden

- Modifikation beim Öffnen der Ausgabe-Datei:
  ofs.open(outfile , ios::app); // app fuer append

Inhalt

Was sind Binärdateien?

- Binärdateien enthalten Daten in binärer Form, entsprechend der internen Darstellung
- Wie die Daten in einer Binärdatei abgelegt werden, muss Programmierer steuern
- Problem: keine einheitliche, interne Darstellung von Werten im Rechner

- Darstellung in einer Datei ablegen

  $\Rightarrow$ Datei wäre dann nur auf gleichem System wieder lesbar

- Programmierer muss Format selber festlegen und die Speicherung entsprechend organisieren
- Gemeinsame Basis: das Byte bzw. in C++ der Datentyp **char**

- Binärdatei: beim Öffnen des *streams* angeben $\Rightarrow$ ios :: binary

Beispiel: Binärdateien

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream ifs;
    ifs.open("eingabe.dat", ios::binary);
    // Alternativ: ifstream ifs("eingabe.dat", ios::binary);

    string outfile="ausgabe.dat";
    ofstream ofs;
    // Alternativ: ofstream ofs("ausgabe.txt", ios::binary);

    ofs.open(outfile, ios::binary); // C++11
    // vor C++11: ofs.open(outfile.c_str(), ios::binary);

    ... // weiter auf folgender Folie
```

Beispiel: Binärdateien (Fortsetzung)

```
    ... // Fortsetzung von letzter Folie

    char c;

    while (!ifs.eof())
    {
        ifs.get(c);

        if (c!=EOF)
            ofs.put(c);
    }

    ifs.close(); // muss nicht
    ofs.close();

    return 0;
}
```

- Zudem: Methoden **read** und **write**, um mehrere Bytes lesen / schreiben zu können
- Daten in einem Feld vom Typ char (siehe C-Arrays)

Inhalt

Textformatierung

- Mit *streams* lassen sich Texte formatiert ausgeben: auf dem Bildschirm oder in eine Datei
- Gleiche Möglichkeit zur Formatierung von Texten im Programm?
- Beispiel: Dateiname bestehend aus Zeichenkette und laufender Nummer

- Ausgabe eines solchen Dateinamen: `cout << "Datei" << i << ".txt"; // int i`
- Wie erhält man eine Zeichenkette / string mit diesem Inhalt?
- Antwort: *stream* für strings

- Klassen *istringstream* und *ostringstream*
- Definiert in Header **sstream** ⇒ **#include** <sstream>
- Funktion `.str()` zum Auslesen und Setzen des Inhalts eines streams
- Funktion `.clear()` zum Löschen eines Fehlerstatus

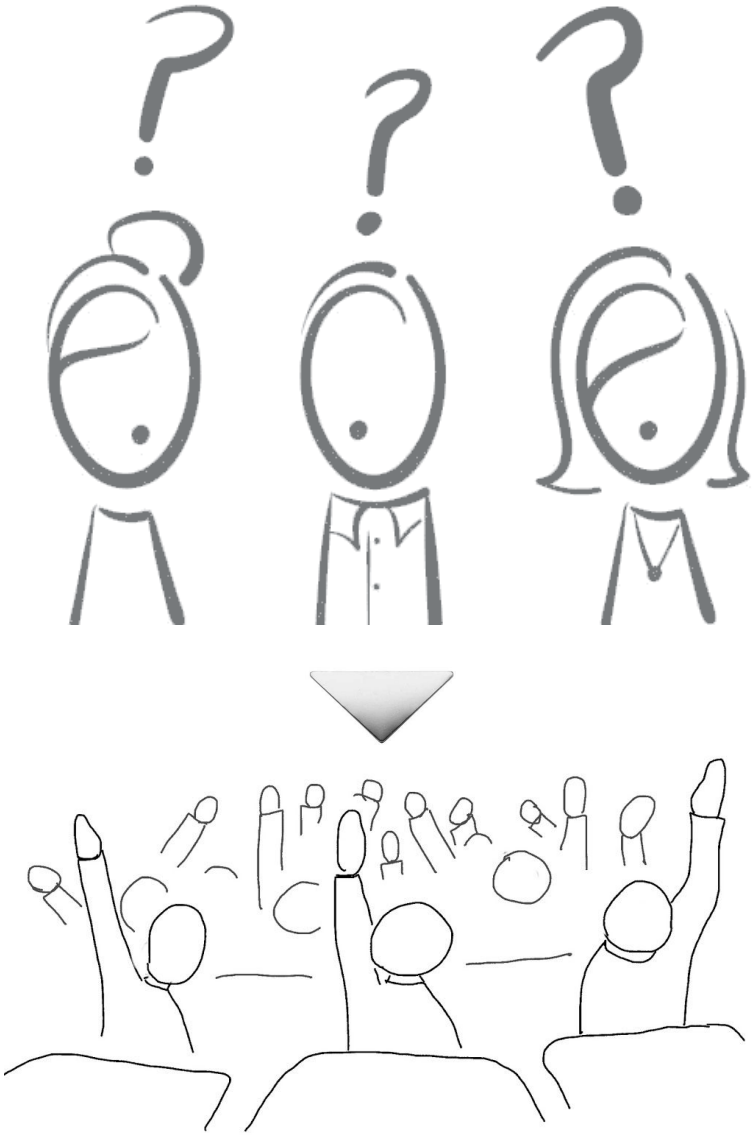
Beispiel: ostringstream (string erzeugen)

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

using namespace std;

int main()
{
    int i = 123;

    ostringstream oss;
    oss << "Datei" << i << ".txt"; // Formatierung

    // Funktion .str() liefert Inhalt des streams als Zeichenkette
    string fn = oss.str();
    cout << fn << endl; // Datei123.txt

    oss << " noch da!";
    fn = oss.str();
    cout << fn << endl; // Datei123.txt noch da!

    // Funktion .str() kann Inhalt des streams setzen
    oss.str(""); // Inhalt des streams auf "leer" setzen
    oss << "nicht mehr da!"; // neuer Inhalt
    fn = oss.str();
    cout << fn << endl; // nicht mehr da!

    return 0;
}
```

Beispiel: istringstream (Zahl von string zu int umwandeln)

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

using namespace std;

int main()
{
    int n = 0;

    string zs = "4711";
    istringstream iss(zs); // Inhalt des streams beim Anlegen setzen
    iss >> n; // stream auslesen
    cout << n << endl; // 4711

    // Danach ist Fehlerstatus "end-of-file" gesetzt!
    cout << iss.good() << " " << iss.fail() << " " << iss.eof() << endl; // 0 0 1

    iss.clear(); // Fehlerstatus (eof) loeschen
    iss.str("1987"); // Funktion .str() kann Inhalt des streams setzen
    iss >> n; // stream auslesen
    cout << n << endl; // 1987

    iss.clear();
    iss.str("123 456"); // mehrere Zahlen im stream
    iss >> n;
    cout << n << endl; // 123
    iss >> n;
    cout << n << endl; // 456

    return 0;
}
```

Gibt es Fragen?



(Es gibt keine dummen Fragen!)