

Informatik II

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Sommersemester 2010

Inhalt

Lernziele dieser Vorlesung

Kommandozeilen

- Parameter von main
- Kommandozeilenbearbeitung

Lesen und Schreiben von Dateien

- Ein- und Ausgaben
- Dateien
- Dateien: Beispiele

Zusammenfassung

Vorlesung 3. Dateibearbeitung und Kommandozeilen

Lernziele dieser Vorlesung

Kommandozeilen

- Parameter von main
- Kommandozeilenbearbeitung

Lesen und Schreiben von Dateien

- Ein- und Ausgaben
- Dateien
- Dateien: Beispiele

Zusammenfassung

Lernziele

- ▶ Übergabe von Parametern an Programme
- ▶ Standardschema zur Bearbeitung von Programmparametern
- ▶ Ein- Und Ausgaben
- ▶ Lesen und Schreiben von Dateien

Parameter von `int main()`

- ▶ Beim **Programmaufruf** können auf der Kommandozeile **Argumente** übergeben werden.
- ▶ Argumente der Kommandozeile werden als **Parameter** an die Funktion `int main()` übergeben.
- ▶ Es stehen für `int main()` **zwei Parameter** zur Verfügung.
- ▶ Die Parameter enthalten
 1. **Anzahl** der Kommandozeilen-Argumente (`int argc`),
 2. **Argumente** als Feld von Zeichenketten (`char *argv[]`, bzw. `char **argv`)

Parameter von `int main()` (Forts.)

```
int main(int argc, char *argv[])  
{  
    // ...  
}
```

- ▶ Die Parameter müssen nicht `argc` und `argv` heißen, Konvention:
`argc = argument count`, `argv = argument value`
- ▶ Es werden **alle Zeichenketten** der Kommandozeile **gezählt** (`argc`)

Parameter von `int main()` (Forts.)

- ▶ Aufbau der Kommandozeile (Programmname ist `args`):

```
args dies ist ein test
```

- ▶ Parameter von `main()`

```
argc = 5  
argv[0] = "args" // Programmname  
argv[1] = "dies"  
argv[2] = "ist"  
argv[3] = "ein"  
argv[4] = "test"
```

UNIX-Konvention

Schema von Kommandozeilen unter UNIX/Linux (gilt auch für viele andere Bereiche)

kommando [*option*]* [*dateinamen*]*

- ▶ *option* beginnt mit einem **Strich** (-) gewöhnlich gefolgt von einem Zeichen, z. B. -v
- ▶ *option* kann zu ihrem Namen ein oder mehrere Argumente besitzen, z. B. -oausgabedatei
- ▶ Beispiel Kommandozeilenformat (Programmdokumentation)

```
print_file [-v] [-l<length>] [-o<name>] [file1] [file2] ..
```


Kommandozeilenbearbeitung

Optionen

- ▶ Aufgabe: alle Options-Argumente eines Programmaufrufs abarbeiten
- ▶ Voraussetzung: Optionen stehen vor den Datei-Argumenten

Schleifenkonstruktion für Optionen

```
// Argument ist Option
while (argc > 1 && argv[1][0] == '-') {
    // Optionsbearbeitung
    ++argv; // naechstes Argument
    --argc; // ein Argument abgearbeitet
}
```

Kommandozeilenbearbeitung (Forts.)

Dateien

- ▶ Aufgabe: alle Datei-Argumente eines Programmaufrufs abarbeiten
- ▶ Voraussetzung: alle Optionen sind abgearbeitet

Schleifenkonstruktion für Dateinamen

```
if (argc == 1) { // keine Datei angegeben
  do_file("file.in"); // Standard-Datei
} else {
  while (argc > 1) {
    do_file (argv[1]);
    ++argv; // naechste Datei
    --argc; // eine Datei bearbeitet
  }
}
```

Kommandozeilenbearbeitung (Forts.)

Optionsbearbeitung

```
switch (argv[1][1]) {  
    case 'v': verbose = 1; break;  
    case 'o': out_file = &argv[1][2]; break;  
    case 'l': line_max = atoi(&argv[1][2]); break;  
    default:  
        cerr << "Bad option" << argv[1] << endl;  
        usage (); // Drucke Kommandozeilenformat  
}
```

- ▶ `-o<name>`: `argv[1][0] == '-'`, `argv[1][1] == 'o'`,
`argv[1][2] ==` Beginn von `<name>`
- ▶ `-l<length>`: `stdlib.h`-Bibliotheksfunktion `atoi ()` konvertiert eine Zeichenkette in eine ganze Zahl (ascii to int)

Beispiel Kommandozeilenbearbeitung

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int verbose = 0;    // verbose mode (default = false)
char *out_file = "print.out"; // output file name
char *program_name; // name of the program (for errors)
int line_max = 66; // number of lines per page

void do_file(char *name) // dummy routine to handle file
{
    cout << "Verbose=" << verbose << "┘Lines="
         << line_max << "┘Input=" << name
         << "┘Output=" << out_file << endl;
}
```

Beispiel Kommandozeilenbearbeitung (Forts.)

```
void usage(void)
{
    cerr << "Usage_␣is_␣" << program_name
          << "␣[options]␣[file-list]" << endl;
    cerr << "Options" << endl;
    cerr << "␣-v␣verbose" << endl;
    cerr << "␣-l<number>␣Number␣of␣lines" << endl;
    cerr << "␣-o<name>␣␣Set␣output␣file␣name" << endl;
    exit (8);
}
```

Beispiel Kommandozeilenbearbeitung (Forts.)

```
int main(int argc, char *argv[])
{
    program_name = argv[0];
    while (argc > 1 && argv[1][0] == '-') {
        switch (argv[1][1]) {
            case 'v': verbose = 1; break;
            case 'o': out_file = &argv[1][2]; break;
            case 'l': line_max = atoi (&argv[1][2]); break;
            default:
                cerr << "Bad option" << argv[1] << endl;
                usage();
        }
        ++argv;
        --argc;
    }
}
```

Beispiel Kommandozeilenbearbeitung (Forts.)

```
if (argc == 1) {
    do_file("print.in");
} else {
    while (argc > 1) {
        do_file(argv[1]);
        ++argv;
        --argc;
    }
}
return (EXIT_SUCCESS);
}
```

Beispiel Kommandozeilenbearbeitung (Forts.)

Programmausgabe

C:\ DOSe

```
M:\>Kommandzeile -x
Bad option -x
Usage is Kommandzeile [options] [file-list]
Options
  -v verbose
  -l<number> Number of lines
  -o<name> Set output file name

M:\>Kommandzeile -l80 -v -oAusgabe.txt test.txt
Verbose=1 Lines=80 Input=test.txt Output=Ausgabe.txt

M:\>Kommandzeile -l80 test.txt
Verbose=0 Lines=80 Input=test.txt Output=print.out

M:\>
```


Ein- und Ausgaben

Stream of Bytes

C++ betrachtet Dateien (auch Tastatur und Bildschirm) als Strom von Zeichen (**Streams of bytes**).

Klassen zur Ein- und Ausgabe

In C++ gibt es drei Klassen zur Ein-Ausgabe-Bearbeitung. Deklaration in `<iostream>`.

- ▶ `istream` für Eingabe
- ▶ `ostream` für Ausgabe
- ▶ `iostream` für Ein-Ausgabe

Ein- und Ausgaben (Forts.)

- ▶ Jeder Stream (istream oder ostream) besitzt einen Zustand.
- ▶ Fehler und unnormale Bedingungen werden durch Setzen und Testen dieses Zustands behandelt.

Funktionen zum Stream-Zustand

```
bool good () const; // naechste Operation kann klappen
bool eof () const; // Ende der Eingabe gefunden
bool fail () const; // naechste Operation schaeft fehl
bool bad () const; // Stream nicht mehr verwendbar
iostate rdstate () const; // Ein-/Ausgabe-Zustand abfragen
// Ein-/Ausgabe-Zustand setzen
void clear (iostate f = goodbit);
// Ein-/Ausgabe-Zustand zusaetzlich setzen
void setstate (iostate f)
operator void * () const; // == 0, falls fail ()
bool operator ! () const; {return fail ();}
```

Ein- und Ausgaben (Forts.)

Variablen

Bei Programmstart werden automatisch Variablen angelegt:

Variable	Benutzung
cin	Standard Eingabe
cout	Standard Ausgabe
cerr	Standard Fehler
clog	Konsole Log

Ein- und Ausgaben (Forts.)

Weitere Hilfsmittel

- ▶ Formatierung: z. B. Genauigkeit bei Fließkommazahlen
- ▶ Locales: z. B. sprach- und länderspezifische Zahlen und Datumsformate
- ▶ Stream-Puffer: effiziente Ein- und Ausgabe, Strategie zur Datenpufferung

Dateien

Die Arbeit mit **Dateien** in C++-Programmen erfolgt stets in **vier Schritten**:

1. **Erzeugen** eines **Datenstromobjekts** zur Klasse `ifstream` oder `ofstream`.
2. **Öffnen** einer Datei durch Aufruf der Funktion `open()`.
3. **Bearbeiten** der Daten in der Datei.
4. **Schließen** der Datei durch Aufruf der Funktion `close()`.

Wichtig

Fertig bearbeitete Dateien immer schließen. Erst dadurch werden die vom Betriebssystem belegten Ressourcen für die Verbindung zwischen Programm und Datei wieder freigeben. Die Anzahl gleichzeitig geöffneter Dateien ist in jedem Betriebssystem beschränkt.

Datenstromobjekte

- ▶ Für Dateien gibt es in C++ **abgeleitete Klassen** von `istream`, `ostream` und `iostream` (`#include <fstream>`):
 - ▶ `ifstream` für Eingabedateien
 - ▶ `ofstream` für Ausgabedateien
 - ▶ `fstream` für Ein-Ausgabedateien
- ▶ Benutzung

```
// Deklaration einer Eingabedatei  
ifstream eingabedatei;  
// Deklaration einer Ausgabedatei  
ofstream ausgabedatei;  
// Oeffnen zum Lesen  
eingabedatei.open("zahlen.dat");  
// Oeffnen zum Schreiben  
ausgabedatei.open("ergebnis");  
eingabedatei.close(); // Schliessen  
ausgabedatei.close(); // Schliessen
```

Datei-Modi

- ▶ Der **Dateimodus** gibt an, auf welche Weise eine Datei geöffnet werden soll.
- ▶ Um mehrere **Modi** zu **kombinieren**, werden sie **bitweise mit ODER verknüpft**, d. h. es wird ein einfacher senkrechter Strich (|) zwischen die Moduskonstanten gesetzt.

Konstante	Bedeutung
<code>ios::in</code>	Datei zum Lesen öffnen
<code>ios::out</code>	Datei zum Schreiben öffnen
<code>ios::ate</code>	Positionieren ans Ende der Datei
<code>ios::app</code>	Ausgaben werden ans Ende der Datei geschrieben
<code>ios::trunc</code>	Wenn die Datei existiert, wird der Inhalt vor dem Beginn der Ausgabe gelöscht
<code>ios::binary</code>	Datei soll als Binärdatei behandelt werden

Dateien öffnen und schließen

```
open(const char* filename, openmode mode) ;
```

Diese Funktion öffnet die Datei mit dem Bezeichner filename und liefert bei Erfolg die Adresse des Objekts zurück, an dem die Funktion ausgeführt wurde. Bei Fehlern wird NULL gemeldet.

Beispiel:

```
ofstream out ;  
out.open("example.bin", ios::out | ios::app | ios::binary) ;
```

Öffne die Datei example.bin für binäre Ausgabe, wobei alles am Dateiende angehängt werden soll.

```
out.close() ;
```

Diese Funktion schließt den Datenstrom out.

Datenstromobjekte: Konstruktor

- ▶ Wie bei anderen Klassen, wird bei der Deklaration von Ein-/Ausgabedateien ein Konstruktor der betreffenden Klasse aufgerufen.
- ▶ Für `ifstream`

```
ifstream::ifstream(const char *name);
```

- ▶ Für `ofstream`

```
ofstream::ofstream(const char *name);
```

Dateien: Benutzung

- ▶ Deklaration und öffnen einer Datei

```
ifstream eingabedatei("zahlen.dat");
```

Dies ist äquivalent zu

```
ifstream eingabedatei;  
eingabedatei.open("zahlen.dat");
```

- ▶ Entsprechend ist die Deklaration von Ausgabedateien.
- ▶ Benutzung geschieht wie bei den Variablen `cin` und `cout`

```
int var;  
// ...  
eingabedatei >> var;  
ausgabedatei << var;
```

Beispiel Eingabedatei

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
using namespace std;

/** Lese 100 Zahlen aus der Datei "zahlen.dat"
 *  * Warnung: Es gibt keine Ueberpruefung,
 *  * ob die Datei auch 100 Zahlen enthaelt!
 *  */
int main()
{
    const int DATA_SIZE = 100; // Anzahl der Daten
    int data_array[DATA_SIZE]; // Daten
    ifstream data_file("zahlen.dat"); // Eingabedatei
```

Beispiel Eingabedatei (Forts.)

```
if (!data_file.good ()) {  
    // Problem beim Oeffnen oder Lesen  
    cerr << "Fehler: _Kann_zahlen.dat_nicht_oeffnen"  
         << endl;  
    exit (8);  
}
```

Beispiel Eingabedatei (Forts.)

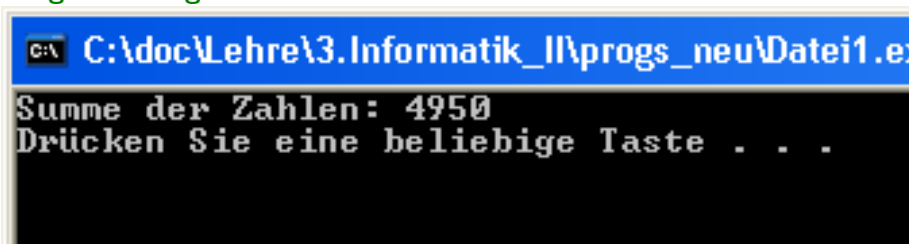
```
for (int i = 0; i < DATA_SIZE; ++i)
    data_file >> data_array[i];

int total(0); // Summe der Zahlen
for (int i = 0; i < DATA_SIZE; ++i)
    total += data_array[i];

cout << "Summe der Zahlen: " << total << endl;
return (EXIT_SUCCESS);
}
```

Beispiel Eingabedatei (Forts.)

Programmausgabe



```
C:\doc\Lehre\3.Informatik_II\progs_neu\Datei1.exe
Summe der Zahlen: 4950
Drücken Sie eine beliebige Taste . . .
```

Textzeile lesen

- ▶ Der Eingabeoperator `>>` arbeitet mit `fstream`-Objekten in der gleichen Weise wie mit `cin`.
- ▶ Die Datei wird so gelesen, als käme ihr Inhalt von der Tastatur.
- ▶ Dazu gehört, dass **Leerzeichen**, **Tabulatoren** und **Zeilenvorschübe** als **Eingabetrenner** interpretiert werden.

Textzeile lesen (Forts.)

- ▶ Das gilt auch, wenn der Datenstrom in Zeichenketten-Variablen fließt.
- ▶ Um aus den Dateien Textzeilen mit Leerzeichen auszulesen, wird die **Member-Funktion** `getline(char *, int)` verwendet.
- ▶ Als erster Parameter wird ein C-String, d. h. ein Zeiger auf `char`, übergeben.
- ▶ Die Funktion arbeitet mit den **klassischen C-Strings**.
- ▶ Der zweite Parameter ist die maximale Anzahl von Zeichen, die in den C-String, d. h. den Einlesepuffer, passt.

Beispiel Textzeile lesen

```
#include <fstream>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    fstream datei;
    char cstring[256]; //Einlesepuffer
    datei.open (argv[1], ios::in);
    while(!datei.eof()) // eof() == Dateiende
    {
        datei.getline(cstring, sizeof(cstring));
        cout << cstring << endl;
    }
    datei.close();
    return (EXIT_SUCCESS);
}
```

Beispiel Textzeile lesen (Forts.)

Programmausgabe

```
C:\WINDOWS\system32\cmd.exe - getline.exe getline.cpp
```

```
> getline.exe getline.cpp
#include <fstream>
#include <iostream>
using namespace std;

int main (int argc, char *argv[])
{
    fstream datei;
    char cstring[256];
    datei.open (argv[1], ios::in);
    while (!datei.eof ()) // eof () = Dateiende
    {
        datei.getline (cstring, sizeof (cstring));
        cout << cstring << endl;
    }
    datei.close ();
    system("PAUSE");
    return (EXIT_SUCCESS);
}
```

Textzeile lesen mit C++-String

- ▶ Zum Lesen von Textzeilen in einen C++-String wird die globale Funktion `getline` (`ifstream`, `string`) verwendet.
- ▶ Erster Parameter ist ein Objekt vom Typ `ifstream`.
- ▶ Zweiter Parameter ist ein C++-String.

Beispiel Textzeile lesen mit C++-String

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main (int argc, char *argv[])
{
    ifstream inFile; // Datei-Handle
    string line;     // Zeile in Datei
    inFile.open(argv[1], ios::in);
    while (!inFile.eof())
    {
        getline(inFile, line); // Lese eine Zeile
        cout << line << endl;
    }
    inFile.close();
    return (EXIT_SUCCESS);
}
```

Datendateien

- ▶ Zum Lesen und Schreiben von **binären Daten** sind die Operatoren `>>` und `<<` nicht gut geeignet.
- ▶ Das Lesen und Schreiben von binären Daten erfolgt meist in **festen Datenblöcken**.
- ▶ Dafür gibt es die Member-Funktionen `read()` und `write()`.
- ▶ Für die **Datenblöcke** werden meistens Objekte als **Hauptspeicherpuffer** verwendet.

Datendateien (Forts.)

Lesen und Schreiben mit `read()` und `write()`

```
tDaten Daten;  
fstream f("Datei.bin", ios::out|ios::binary|ios::in);  
f.write(&Daten, sizeof(Daten));  
//...  
f.read(&Daten, sizeof(Daten));
```

- ▶ Mit dem Aufruf von `f.write()` wird das Objekt `Daten` in eine Datei ab der aktuellen Position geschrieben.
- ▶ Mit dem Aufruf von `f.read()` wird aus der Datei in ein Objekt `Daten` gelesen.

Datenpuffer

- ▶ **Datenpuffer** enthalten die Daten eines Datenblocks, der in eine **binäre Datei** geschrieben oder aus dieser gelesen wird.
- ▶ Datenpuffer werden sinnvollerweise als **Klassen** implementiert.
- ▶ Datenpuffer sollen in jedem Fall die **Daten selbst** enthalten und **nicht Zeiger** darauf.

Datenpuffer (Forts.)

- ▶ Falls Zeiger im Datenpuffer enthalten sind, werden die **Hauptspeicheradressen** auf die Daten und nicht die Daten selbst in die Datei **geschrieben**.
- ▶ Zeiger sind nicht immer offen erkennbar: z. B. sind C++-Strings Zeiger.
- ▶ **Geeignet** für Dateipuffer ist ein klassischer **C-String**, d. h. ein festes Array von **char**.

Datenpuffer mit C-Strings (daten.h)

```
/** Abbildung der binären Daten in einen C-String
 * d. h. festes char-Feld
 */
class tDaten
{
private:
    char data[256];
public:
    void Set(char *para)
    {
        strcpy (data, para, sizeof(data));
        data[sizeof(data)-1] = 0;
    }
    void Show()
    {
        cout << data << endl;
    }
};
```

Beispiel Datenpuffer mit C-Strings

Testprogramm

- ▶ Wird das Programm mit einem Parameter aufgerufen, dann wird der erste Parameter in der Datei `testdatei` gesichert.
- ▶ Wird das Programm ohne Parameter aufgerufen, dann wird der zuletzt abgelegte Name wieder aus der Datei gelesen und auf dem Bildschirm ausgegeben.

Beispiel Datenpuffer mit C-Strings (Forts.)

```
#include <fstream>
#include <daten.h>
using namespace std;

int main (int argc, char**argv)
{
    tDaten Daten;
    fstream f("testdatei", ios::out|ios::binary|ios::in);
    if (argc>=2) // Parameter beim Aufruf uebergeben?
    {
        Daten.Set(argv[1]); // In Daten ablegen
        // Objekt in Datei speichern
        f.write((char *)&Daten, sizeof(Daten));
    }
}
```

Beispiel Datenpuffer mit C-Strings (Forts.)

```
// Kein Argument? Dann Datei auslesen.  
if (argc==1)  
{  
    // Dateiinhalt in Objekt lesen  
    f.read((char *)&Daten, sizeof(Daten));  
    Daten.Show(); // ... und anzeigen  
}  
return (EXIT_SUCCSS);  
}
```

Beispiel Datenpuffer mit C-Strings (Forts.)

Programmausgabe

```
C:\WINDOWS\system32\cmd.exe
```

```
> daten_c_string abcde.bin
```

```
> daten_c_string  
abcde.bin
```

```
>
```

Datenpuffer mit C++-Strings (funktioniert nicht!)

```
/** Abbildung der binären Daten in einen C++-String
 * string enthält nicht die eigentlichen Daten.
class tStrDaten
{
private:
    string data;
public:
    void Set(char *para)
    {
        data = para;
    }
    void Show()
    {
        cout << data << endl;
    }
};
```

Beispiel Datenpuffer mit C-Strings (Forts.)

Programmausgabe

```
C:\WINDOWS\system32\cmd.exe - daten_cpp_string

> daten_cpp_string
r 8.0\Reader\plug_ins\test
.WSF; .WSH PROCESSOR_ARCHITECTURE=AMD64
tepping 5, GenuineIntel Processor Family 17
:\Programme PROMPT=$g SE
TEMP=C:\DOKUME~1\OJ5FC5~1\Temp USERDOMAIN=OLIVER
oj.OLIVER windir=C:\WINDOWS\system32\cmd.exe

jklnopqrstuvwxyzaBC
e i ä
= p/= 00= X0= z0= L0= b0=
@* ALLUSERSPROFILE=C:\Programme\Com
monProgramFiles

S* APPDATA=C:\Dokumente und Einstellungen\oj.OLIVER
CommonProgramFiles=C:\Programme\Com
monProgramFiles
R * I* ComSpec=C:\WINDOWS\system32\cmd.exe
* FP_NO_HOST_CHECK=NO
d Einstellungen\oj.OLIVER
SSORS=1 * h*
```

Zusammenfassung

- ▶ **Parameter** an C++-Programme werden als Zeichenketten auf der Kommandozeile übergeben.
- ▶ **Optionen** eines Programms werden per Konvention mit einem '-'-Zeichen begonnen.
- ▶ Ein- und Ausgaben werden in C++ als **Strom von Daten** betrachtet.
- ▶ Dateien und die Konsole werden gleich behandelt, es gibt **vordefinierte Objekte für die Konsole**: `cin`, `cout`, `cerr`, `clog`.
- ▶ Grundsätzlich stehen drei **Klassen für Ein- und Ausgaben** zur Verfügung: `istream`, `ostream`, `iostream`.