Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

25. Oktober 2019

Input/Output

Inhalt

- Übersicht Input/Output
- Beispiele: Konsolen-Output
- Vor- und Nachteile
- C++ I/O-Streams
- I/O-Arten
- Formatierte/Unformatierte Ein-/Ausgabe
- Stream-Manipulatoren
- Zustände von Streams
- Fehlerbehandlung
- File-Streams
- Stream-Operatoren

Übersicht Input/Output

- ermöglicht unterschiedlichste Schnittstellen
  - Pipes unter Linux (In/Out)
  - Logging (Out)
  - Files (In/Out)
  - Netzwerk (In/Out)
- unterschiedliche Varianten
  - C-Style: `printf` und `scanf`
  - C++-Style: `<iostream>`
  - andere Libraries: z.B: libfmt (Out) oder ZeroMQ (Netzwerk)

Online Dokus/Kurse

- C: printf
- C++: Input/output library
- Stream IO and File IO

Beispiele: Konsolen-Output

```
1  struct DataPoint {
2    std::string name;
3    uint8_t flags;
4    double value;
5  } dp;
6
7  // C-Style
8  printf("DataPoint: name=%s, flags=%hhu, value=%f \n",
9         dp.name.c_str(), dp.flags, dp.value);
```

Beispiele: Konsolen-Output

```cpp
// C++-Style
std::cout << "DataPoint: name=" << dp.name
          << ", flags=" << static_cast<unsigned>(dp.flags)
          << ", value=" << dp.value
          << std::endl;

// oder mit überladenem operator<<
std::cout << "DataPoint: " << dp << std::endl;

// libfmt
fmt::print("DataPoint: name={2}, flags={0:d}, value={1}",
           dp.flags, dp.value, dp.name);
```

Vor- und Nachteile

C-Style:

- schnell

C++ Streams:

- Typ-Sicherheit
- überladbarer `operator<<`

libfmt:

- Typ-Sicherheit
- schnell
- Python- oder `printf`-Syntax
- überladbarer `fmt::formatter`

Nachteile:

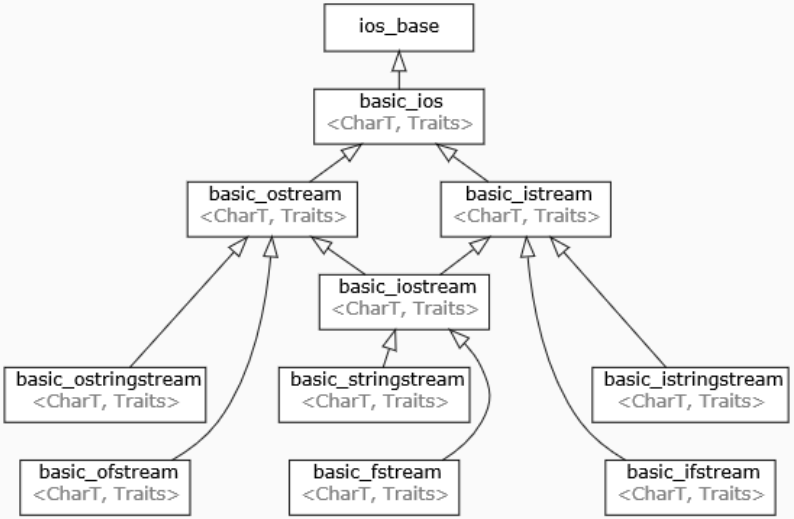
- keine Typ-Sicherheit
- langsam
- externe Library

Verwendung

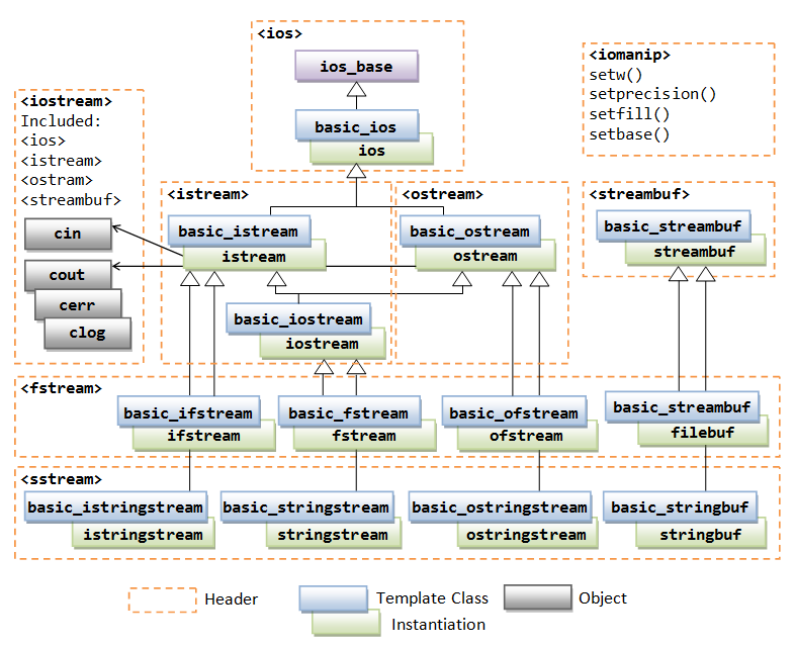
- C++ Streams: wenn keine besonderen Anforderungen
- Library wie `libfmt`: wenn sehr viel oder schnelles I/O

C++ I/O-Streams

- Header: `<iostream>` etc.
- Input: `operator>>` / Output: `operator<<`



C++ I/O-Streams


<ios>
ios_base
basic_ios
ios
<iomanip>
setw()
setprecision()
setfill()
setbase()
<iostream>
Included:
<ios>
<istream>
<ostram>
<streambuf>
cin
cout
cerr
clog
<istream>
basic_istream
istream
<ostream>
basic_ostream
ostream
basic_iostream
iostream
<streambuf>
basic_streambuf
streambuf
<fstream>
basic_ifstream
ifstream
basic_fstream
fstream
basic_ofstream
ofstream
basic_streambuf
filebuf
<sstream>
basic_istringstream
istringstream
basic_stringstream
stringstream
basic_ostringstream
ostringstream
basic_stringbuf
stringbuf
Header
Template Class
Instantiation
Object

I/O-Arten

String-I/O:

- Klasse: `std::stringstream`
- Funktion: `std::to_string`

Stream-I/O:

- Objekte: `std::cout`, `std::cerr`, `std::clog`, `std::cin`

File-I/O:

- Klassen: `std::ofstream`, `std::ifstream`

Formatierte Ausgabe

```cpp
static constexpr size_t kColumnCount = 4;
std::cout << "ASCII-Tabelle" << std::endl;

for (size_t i = 32; i < 128; ++i) {
  std::cout.width(3);                                  // Zahlenbreite: 3
  std::cout.fill('0');                                 // mit führenden Nullen auffüllen
  std::cout << i << " = 0x";
  std::cout.setf(std::ios::hex, std::ios::basefield);  // Basis 16
  std::cout.setf(std::ios::uppercase);                 // Hex mit Grossbuchstaben
  std::cout << i << ": ";
  std::cout.unsetf(std::ios::hex);                     // wieder auf Dezimal umstellen
  std::cout << static_cast<char>(i) << '\t';

  if (i % kColumnCount == kColumnCount - 1) {
    std::cout << std::endl;                            // Zeilenumbruch nach 4 Spalten
  }
}
```

Formatierte Ausgabe

```
ASCII-Tabelle
032 = 0x20:         033 = 0x21: !       034 = 0x22: "       035 = 0x23: #
036 = 0x24: $       037 = 0x25: %       038 = 0x26: &       039 = 0x27: '
040 = 0x28: (       041 = 0x29: )       042 = 0x2A: *       043 = 0x2B: +
044 = 0x2C: ,       045 = 0x2D: -       046 = 0x2E: .       047 = 0x2F: /
048 = 0x30: 0       049 = 0x31: 1       050 = 0x32: 2       051 = 0x33: 3
052 = 0x34: 4       053 = 0x35: 5       054 = 0x36: 6       055 = 0x37: 7
056 = 0x38: 8       057 = 0x39: 9       058 = 0x3A: :       059 = 0x3B: ;
060 = 0x3C: <       061 = 0x3D: =       062 = 0x3E: >       063 = 0x3F: ?
064 = 0x40: @       065 = 0x41: A       066 = 0x42: B       067 = 0x43: C
068 = 0x44: D       069 = 0x45: E       070 = 0x46: F       071 = 0x47: G
072 = 0x48: H       073 = 0x49: I       074 = 0x4A: J       075 = 0x4B: K
...
112 = 0x70: p       113 = 0x71: q       114 = 0x72: r       115 = 0x73: s
116 = 0x74: t       117 = 0x75: u       118 = 0x76: v       119 = 0x77: w
120 = 0x78: x       121 = 0x79: y       122 = 0x7A: z       123 = 0x7B: {
...
```

Formatierte Eingabe

```
1  std::string name;
2  int age;
3  bool male;
4  float fahrzeit;
5
6  std::cout << "Name: ";
7  std::cin >> name;
8
9  std::cout << "Alter: ";
10 std::cin >> age;
11
12 std::cout << "Fahrzeit: ";
13 std::cin >> fahrzeit;
14
15 std::cout << "Maennlich: ";
16 std::cin.setf(std::ios::boolalpha);
17 std::cin >> male;
18 std::cin.unsetf(std::ios::boolalpha);
```

Unformatierte Eingabe

```
1  char c;
2  char initial;
3  char plz[5];                    // mit Platz für Stringende
4
5  std::cout << "Initiale: ";
6  std::cin.get(initial);
7  std::cin.get(c);                // Return lesen
8
9  std::cout << "Postleitzahl: ";
10 std::cin.getline(plz, 5);       // liest und entfernt delimiter (\n) automatisch
```

Stream-Manipulatoren

- Flags können nicht nur über `setf()` gesetzt werden, sondern auch mittels Stream-Manipulatoren
- Manipulatoren können Parameter haben

```
1  // vorher
2  std::cout.setf(std::ios::hex, std::ios::basefield); // Basis 16
3  std::cout.setf(std::ios::uppercase);                // Hex mit Grossbuchstaben
4  std::cout << i << std::endl;
5
6  std::cout.width(3);                                 // Zahlenbreite: 3
7  std::cout.fill('0');                                // mit füllenden Nullen auffüllen
8  std::cout << i << std::endl;
9
10 // nachher
11 std::cout << std::hex << std::uppercase << i << std::endl;
12 std::cout << std::setw(3) << std::setfill('0') << i << std::endl;
```

Zustände von Streams

- Der Zustand eines Streams ist in der Variable `iostate` gespeichert, welche mit `rdstate()` ausgelesen werden kann:
  - `0` bedeutet, dass alles in Ordnung ist.
  - Alle anderen Zahlen bedeuten, dass sich der Stream in einem Fehlerzustand befindet, wobei eines oder mehrere Fehler-Bits gesetzt sein können
- Die Funktionen `good()`, `eof()`, `fail()` und `bad()` dienen jeweils dazu, herauszufinden, ob bestimmte Bits in `iostate` gesetzt sind.

Zustände von Streams

iostate	Bedeutung	good()	eof()	fail()	bad()
goodbit	Keine Fehler (iostate = 0)	true	false	false	false
eofbit	Ende der Datei erreicht (bei Input)	false	true	false	false
failbit	logischer Fehler bei I/O Operation	false	false	true	false
badbit	Lese/Schreib-Fehler auf Stream-Buffer	false	false	true	true

Fehlerbehandlung

```
1  int i;
2  std::cout << "Bitte ganze Zahl eingeben: ";
3  std::cin >> i;
4
5  if (std::cin.good()) {                    // Kurzform: if (std::cin) {
6    std::cout << "Die eingegebene Zahl ist: " << i << std::endl;
7  } else {
8    std::cin.clear();
9    std::cout << "Fehler! Falscher Input: ";
10
11   do {
12     std::string s;
13     std::cin >> s;
14     std::cout << '[' << s << "] ";
15   } while (std::cin.get() != '\n');      // Return lesen
16 }
```

File-Streams

- Unterschiedliche Arten von Streams
- File-Streams mit In/Out: `ofstream`, `ifstream`, `fstream`
- Andere Flags wie:
  - Read/Write: `std::ios::in | std::ios::out`
  - Read-Only: `std::ios::in`
  - Binär: `std::ios::binary`
  - Überschreiben: `std::ios::trunc`
  - Anfügen: `std::ios::app`

Stream-Operatoren

Mittels `operator<<` die Ausgabe oder `operator>>` die Eingabe für eigenen Typ definieren:

```
1  struct DataPoint {
2    std::string name;
3    uint8_t flags;
4    double value;
5  };
6
7  friend std::ostream&
8  operator<<(std::ostream& os, const DataPoint& dp) {
9    os << "name=" << dp.name
10      << ", flags=" << static_cast<unsigned>(dp.flags)
11      << ", value=" << dp.value;
12   return os;
13 }
```