Ein/Ausgabe in C++ (`iostream`,`iomanip`)

- **Formatierte Ausgabe über `cout`**

Syntax: `cout <<` *Argument1* `<<` *Argument2* `<<` ...

Die Formatierung der Ausgabe kann durch sog. Manipulatoren beeinflußt werden. *(Auszug)*

Manipulator	*Wirkung*	*Bemerkungen*
`setw(n)`[1]	Min. Feldbreite (Voreinst.: `0`)	`#include <iomanip>`
`setprecision(n)`[2]	Ziffern nach Dezimalpunkt (Voreinst.: `6`)	`#include <iomanip>`
`setfill(c)`	Füllzeichen `c` (Voreinst.: `'␣'`)	`#include <iomanip>`
`left`	linksbündig	
`right`	rechtsbündig (Voreinst.)	
`internal`	Vorzeichen linksbündig, Zahl rechtsbündig	
`showpos`	pos. Vorzeichen ausgeben	
`noshowpos`	pos. Vorzeichen nicht ausgeben (Voreinst.)	
`fixed`	Festpunktformat bei Gleitpunktzahlen	
`scientific`	Exponentialformat bei Gleitpunktzahlen	
`defaultfloat`[3]	Allgemeinformat bei Gleitpunktzahlen (Voreinst.)	
`dec`	Dezimalausgabe bei ganzen Zahlen (Voreinst.)	
`oct`	Oktalausgabe bei ganzen Zahlen	
`hex`	Hexadezimalausgabe ganzen Zahlen	
`showbase`	führende `0` (oktal), `0x`, `0X` (hexadezimal) bei ganzen Zahlen ausgeben	
`noshowbase`	keine Zahlsystemangabe (Voreinst.)	
`uppercase`	Großbuchstaben in der Zahlausgable	
`nouppercase`	Kleinbuchstaben in der Zahlausgabe (Voreinst.)	
`boolalpha`	logische Werte als Zeichenkette ausgeben	
`noboolalpha`	logische Werte als Zahlen ausgeben(Voreinst.)	
`endl`	Zeilenvorschub	
`flush`	Ausgabepuffer leeren	

Statt durch Manipulatoren können Formatflags auch durch `cout.setf` bzw. durch Komponentenfunktionen gesetzt werden.

`cout.width(n)`[1]	Min. Feldbreite
`cout.precision(n)`[2]	Ziffern nach Dezimalpunkt
`cout.fill(c)`	Füllzeichen `c` (Voreinst.: `'␣'`)
`cout.setf(ios::left,ios::adjustfield)`	linksbündig
`cout.setf(ios::right,ios::adjustfield)`	rechtsbündig
`cout.setf(ios::internal,ios::adjustfield)`	Vorz. linksbündig, Zahl rechtsbündig
`cout.setf(ios::showpos)`	pos. Vorzeichen ausgeben
`cout.unsetf(ios::showpos)`	pos. Vorzeichen nicht ausgeben
`cout.setf(ios::fixed,ios::floatfield)`	Festpunktformat bei Gleitpunktzahlen
`cout.setf(ios::scientific,ios::floatfield)`	Exponentialformat bei Gleitpunktzahlen
`cout.unsetf(ios::floatfield)`	Allgemeinformat bei Gleitpkt.zahlen (Voreinst.)
`cout.setf(ios::dec,ios::basefield)`	Dezimalausgabe bei ganzen Zahlen
`cout.setf(ios::oct,ios::basefield)`	Oktalausgabe bei ganzen Zahlen
`cout.setf(ios::hex,ios::basefield)`	Hexadezimalausgabe bei ganzen Zahlen
`cout.setf(ios::showbase)`	führende `0` (oktal), `0x`, `0X` (hex.) ausg.
`cout.unsetf(ios::showbase)`	keine Zahlsystemangabe bei ganzen Zahlen
`cout.setf(ios::uppercase)`	Großbuchstaben in der Zahlausgabe
`cout.unsetf(ios::uppercase)`	Kleinbuchstaben in der Zahlausgabe
`cout.setf(ios::boolalpha)`	log. Werte als Zeichenkette ausg.
`cout.unsetf(ios::boolalpha)`	log. Werte als Zahlen ausg. (Voreinst.)
`cout.flush()`	Ausgabepuffer leeren

[1] Wirkt nur auf die nächste Ausgabeoperation
[2] im Gleitpunktformat `fixed` oder `scientific`
[3] C++11

- **Formatierte Eingabe über `cin`**

  Syntax: `cin >>` *Argument1* `>>` *Argument2* `>>` ...

  Mit Manipulatoren kann der Eingabevorgang verändert werden.

`skipws`	führenden Zwischenraum überlesen (Voreinst.)
`noskipws`	führenden Zwischenraum nicht überlesen
`dec oct hex`	Dezimal/Oktal/Hexadezimaleingabe bei ganzen Zahlen

  Die entsprechenden Formatflags können auch mit `cin.setf` gesetzt werden.

`cin.setf(ios::skipws)`	führenden Zwischenraum überlesen (Voreinst.)
`cin.unsetf(ios::skipws)`	führenden Zwischenraum nicht überlesen
`cin.setf(ios::dec,ios::basefield)`	Dezimaleingabe bei ganzen Zahlen
`cin.setf(ios::oct,ios::basefield)`	Oktaleingabe bei ganzen Zahlen
`cin.setf(ios::hex,ios::basefield)`	Hexadezimaleingabe bei ganzen Zahlen

  Wird neben der C++-Ein/Ausgabe auch die C-Ein/Ausgabe benutzt, dann sollte der Funktionsaufruf `ios::sync_with_stdio()` vor der ersten C++-Ein/Ausgabeoperation erfolgen.

- **Beispiele für die formatierte Ausgabe in C++**

  Programm: *Ausgabe:*

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

int main()
{ int i=123; double x=12.345; string s="Ausgabe";

  cout << "|" << setw(6) << i << "|" << endl;                |   123|
  cout << "|" << setw(6) << left  << i << "|" << endl;       |123   |
  cout << "|" << setw(2) << right << i << "|" << endl;       |123|

  cout << "|" << fixed << x << "|" << endl;                  |12.345000|
  cout << "|" << setw(8) << setprecision(4)
       << x << "|" << endl;                                  | 12.3450|
  cout << "|" << setw(8) << showpos << setprecision(2)
       << x << "|" << endl;                                  |  +12.35|
  cout << "|" << noshowpos << right << setw(1)
       << setprecision(1) << x << "|" << endl;               |12.3|
  cout << "|" << scientific << setprecision(6)
       << x << "|" << endl;                                  |1.234500e+01|
  cout << "|" << setw(14) << x << "|" << endl;               |  1.234500e+01|
  cout << "|" << setw(14) << setprecision(7)
       << uppercase << x << "|" <<endl;                      | 1.2345000E+01|

  cout << "|" << setw(10) << s << "|" << endl;               |   Ausgabe|
  cout << "|" << setw(10) << left << s << "|" << endl;       |Ausgabe   |
  return 0;
}
```

Übersetzen von C–Programmen *(Ubuntu Linux 20.04, g++-9.4)*

c++ [*option(s)*] *file(s)* [*librar(ies)*]

Das Kommando **c++** oder **g++** startet im Normalfall nicht nur den Übersetzungsvorgang, sondern ruft davor den Präprozessor und danach den Linker auf. Die Verarbeitung der einzelnen Dateien hängt von der Endung des Dateinamens ab. Während der einzelnen Stufen werden Dateien erzeugt, die normalerweise nach Abschluß der nächsten Stufe gelöscht werden.

Vorgang	*bearbeitete Dateien*	*erzeugte Dateien*	*Bemerkungen*
Präprozessorlauf und Übersetzen	`.cpp`	`.o`	Erzeugung von Objektcodedateien aus C++-Quelldateien
Binden	`.o`, `.a`	`a.out` (Voreinst.)	Binden von Objektcodedateien und Bibliotheksroutinen zu einer ausführbaren Objektcodedatei

Option	*Bedeutung*
-o *file*	ausführbare Objektcodedatei (Voreinstellung: `a.out`)
-l*name*	benötigte Unterprogramme aus Objektbibliothek `lib`*name*`.a` einbinden
-L*directory*	Verzeichnis zu best. Standardverzeichnissen (z.B. `/usr/lib`) hinzufügen, in denen nach Objektbibliotheken gesucht wird
-I*directory*	zusätzliches Verzeichnis für die Suche nach `#include`-Dateien im Präprozessorlauf
-D*macro*[*=def*]	Präprozessormakro definierten
-O	optimierten Code erzeugen
-g	debugfähigen Objektcode erzeugen bzw. binden
-E	nur Präprozessorlauf durchführen
-Wall	(viele sinnvolle) Warnungen anzeigen *[sehr empfehlenswert!]*
-w	alle Warnungen unterdrücken *[nicht empfehlenswert!]*
-c	nur übersetzen
-std=c++17	C++17
--help	Kurzinformation über die Optionen

Eine große Anzahl weiterer Optionen erhält man mit den Befehlen `man c++` bzw. `info c++`.

- Programmbibliotheken können über die Option **-l** (vor allem Systembibliotheken) oder über den Dateipfad angegeben werden. Die mit **-l** angegebenen Programmbibliotheken werden zunächst in den durch die Umgebungsvariable `LD_LIBRARY_PATH` angegebenen Verzeichnissen, dann in den durch **-L** spezifizierten und zum Schluß in den Standardverzeichnissen (s.o) gesucht. Auf die Reihenfolge von Objektcodedateien und -bibliotheken ist zu achten, weil aus jeder Bibliothek die zu diesem Zeitpunkt fehlenden Routinen geladen werden.

  Bsp.: `c++ -Wall gamma.cpp -lgsl -lgslcblas` Übersetzen von `gamma.cpp`, Laden benötigter Routinen aus den Bibliotheken `libgsl` und `libgslcblas` (Suche der Bibliotheken in den durch `LD_LIBRARY_PATH` spezifizierten Verzeichnissen und in Standardverzeichnissen).

- `export GCC_COLORS=1` Farbige Fehleranzeige

- Die Voreinstellung für den Compiler ist C++14 mit Gnu-Erweiterungen.

  Bsp.: `c++ -std=c++17 prog.cpp` Übersetzen eines C++17-Programms

Debugging *(Ubuntu Linux 20.04, g++-9.4)*

gdb [*objectfile*] [*corefile*]

Ruft den Quellcodedebugger auf. Die zu bearbeitende Objektcodedatei muß mit der Option **-g** übersetzt und gebunden worden sein. Ist das Programm mit einem Speicherabzug (core dump) abgestürzt oder abgebrochen worden (z.B. durch CTRL \ bzw. `kill -3`), dann kann über den Parameter *core file* (Voreinstellung: `core`) der Zustand im Abbruchzeitpunkt untersucht werden.

Subkommandos (Auszug)	*[können abgekürzt werden]*
help [*cmd*]	Information zu einem Kommando bzw. alle Kommandos anzeigen
run [*programparameter*]	Objektcodedatei starten
where	aktuelle Position und Aufrufverschachtelung anzeigen
backtrace	"
print [*expression*]	Wert eines Ausdrucks (z.B. Variablenwert) anzeigen
list [*startline*,*endline*]	Quellcode anzeigen
break *line*	Haltepunkt setzen
continue	Programmausführung ab Haltepunkt fortsetzen
clear [*line*]	Haltepunkt löschen
quit	Debugger beenden

Bsp.:

`c++ -Wall -g prog.cpp`	*debugfähigen Objektcode erzeugen*
`gdb a.out`	*Debugger starten*
`run`	*Programm innerhalb Debugger starten, ggf. Werte eingeben*
`where`	*Funktionsverschachtelung anzeigen, falls Programm abstürzt*
`quit`	

Unter Linux wird oft per Voreinstellung *kein* core dump erzeugt. Mit geeigneten Shell-Befehlen läßt sich das ändern, z.B. in der Bash: `ulimit -c unlimited` .

Starten von Programmen

Programme werden durch Angabe des Dateipfads des übersetzten Programms und evtl. Programmparameter gestartet.

oft: `./a.out` *arg1 arg2* ...

Ein-/Ausgabeumleitungen können auch über die Shellmetazeichen `<` und `>` vorgenommen werden.

Weitere Compiler und Debugger *(Ubuntu Linux 20.04, clang++-10.0)*

Alternativ zur GNU-Compiler-Suite, die außer für C und C++ auch Compiler für weitere bekannte Programmiersprachen enthält, kann die LLVM-Compiler-Suite zum Übersetzen von C++-,C- und Objective-C-Programmen verwendet werden. Im wesentlichen werden auch hier die bereits genannten Compileroptionen unterstützt. Fehlermeldungen wirken übersichtlicher. Der zugehörige Debugger `lldb` ist jedoch in der aktuellen Betriebssystemumgebung schwieriger zu bedienen.

clang++ [*option(s)*] *file(s)* [*librar(ies)*]

lldb *objectfile*

Bsp.:

`clang++ -g prog.cpp`	*debugfähigen Objektcode erzeugen*
`lldb a.out`	*Debugger starten*
`run`	*Programm innerhalb Debugger starten*
`bt`	*Funktionsverschachtelung anzeigen, falls Programm abstürzt*
`quit`	