Eingabe und Ausgabe

Unsere 2 Mustersprachen unterscheiden sich deutlich bei der Ausgabe bzw. Eingabe von Daten. Ich erkläre hier kurz und ohne Details, wie man die Python-Funktionen `print()` und `input()` in C++ ersetzt.

`print()` <-> `std::cout` << (Bildschirm)

`std::cout` (sprich SI-AUT) ist der Standard-Ausgabestream, der alles **normalerweise am Bildschirm** ausgibt. Man gibt *`irgendetwas`* aus, indem man

```
std::cout << irgendetwas
```

schreibt. *`irgendetwas`* kann eine Konstante, eine Variable, ein Rechenergebnis usw. sein. Man kann auch mehrere Ausgaben mittels weiterer `<<` verketten:

```
std::cout << "a = " << a << '\n';
```

Obiges gibt ein C-String-Literal `"a = "`, den Inhalt der Variablen `a` und den `char '\n'` aus.

Den Scope `std::` darf man weglassen, wenn man am Programmanfang die Anweisung

```
using namespace std;
```

geschrieben hat. Dann schreibt man dafür kurz:

```
cout << "a = " << a << '\n';
```

`print(…, …, …)` wird in C++ gleichwertig ersetzt durch

`std::cout << … << ' ' << … << ' ' << … << '\n';`

Man muss also in C++ die Trennzeichen wie Leerzeichen und das Newline explizit ausgeben.

`input` <-> `std::cin` >> (Tastatur)

`std::cin` (sprich SI-IN) ist der Standard-Eingabestream, der alles **normalerweise von der Tastatur** einliest. Man kann Werte nur in Variablen einlesen (nicht in Konstante). Ist z.B. `a` eine Variable (von irgendeinem Standardtyp), so liest man einen Wert dafür mittels

```
std::cin >> a;
```

ein (die spitzen Klammern >> zeigen immer in die Richtung, wohin die Daten fließen). Verkettungen wie

```
std::cin >> a >> b >> c;
```

sind ebenso möglich. Wie bei der Ausgabe darf man den Scope `std::` weglassen, wenn man die `using namespace std;` Anweisung verwendet hat.

Eine Eingabeaufforderung, die man als Argument von Python-`input()` angeben kann, muss man hier mittels einer eigenen Ausgabeanweisung vorher realisieren:

Python:
```
x = int(input("Geben Sie den Integer x ein: "))
y = int(input("Geben Sie den Integer y ein: "))
```
C++:
```
cout <<"Geben Sie die Integer x, y ein: ";
int x, y;
cin >> x >> y;
```

Pythons `input()` Funktion liest immer eine **ganze Zeile** ein und liefert einen **String** zurück. Wird ein anderer Datentyp gebraucht, muss man in Python eine Umwandlungsfunktion aufrufen (hier: `int()`). Stehen falsche Zeichen im Input, stürzt die Funktion ab. Will man mehrere Zahlen auf einmal einlesen, muss man in Python den Inputstring splitten.

C++ Eingabe-Operator >> liest direkt in eine Variable ein und verbraucht dabei nur so viele Zeichen vom Input, wie dazu nötig sind. Bei falschen Zeichen wird entweder gar nichts eingelesen (Fehler schon am Anfang) oder bis zum ersten unpassenden Zeichen gelesen. Nichtverarbeitete Zeichen verbleiben im Eingabepuffer und werden bei der nächsten Einlese-Operation verarbeitet.

`format()` – Formatierte Ausgabe

Sowohl Python als auch C++ besitzen mehrere Möglichkeiten, die Form der Ausgabe zu beeinflussen. Ich habe in Python heuer sowohl die f-Strings als auch die `format()`-Stringmethode erklärt. Seit C++20 besitzt C++ eine sehr ähnliche (das war gewollt) Funktion `format()`, die dieselben Formatieranweisungen versteht. In Python ist es eine Methode der String-Klasse `str`, in C++ ist es eine normale Funktion.

Python: *Formatstring*`.format(…, …, …)`
C++: `format(`*Formatstring* `, …, …, …);`
Beide Funktionen erzeugen einen formatierten String, den man dann noch mittels `print()` in Python bzw. << in C++ ausgeben muss (oder man verwendet gleich `print()` in C++)

Installation und Verwendung der C++ Library `fmt`

Eigentlich gehört die Format-Bibliothek zum Sprachumfang von C++20, allerdings ist sie 2020 bei keinem Compiler im Lieferumfang enthalten gewesen☹ Deshalb musste man sie letztes Jahr noch händisch dazu installieren. 2021 wurde die Situation kaum besser und nur Microsofts neuester Visual C++ Compiler beinhaltet diese Funktion (eigentlich eine ganze Bibliothek). In der Theorie sollte das folgende funktionieren:

```
#define FMT_HEADER_ONLY          // MUSS als erstes stehen
#include <format>
using std::format;               // optional, aber sinnvoll!
using std::print;                // optional, wenn man gleich drucken will
```

Ohne die erste Zeile (das `#define`) würde das Programm deutlich schneller kompiliert werden, allerdings müsste man dann Bibliotheken dazu linken und alles wird etwas komplizierter (weil auch abhängig vom Betriebssystem).

Python: `print( "sin({:6.2f}) = {:12.6f}\n".format(x, sin(x)) )`
C++: `print("sin({:6.2f}) = {:12.6f}\n", x, sin(x));`

Wenn man eine ältere Compilerversion verwendet (verwenden muss), ist die Einbindung dieser Bibliothek wesentlich komplizierter und hängt obendrein noch vom Betriebssystem und Compiler ab. Dabei gäbe es eine perfekte Implementierung, die die Compiler nur noch übernehmen müssten. Für Leute mit Experimentierdrang lässt sich das leicht ausprobieren:

1) Besuch der Projektseite: https://github.com/fmtlib/fmt
2) Download des Codes als ZIP-Datei
3) Kopieren des Ordners `include/fmt` der Zip-Datei in das `include` Verzeichnis des C++ Compilers, also z.B.
   unter Unix: `/usr/include/c++/`*Version.des.Compilers*
   unter MSYS2: `/msys64/mingw64/include/c++/` *Version.des.Compilers*

Da es sich um eine externe Library handelt (die nicht `std::` benutzen darf), muss man obige Zeilen abändern zu:

```
#define FMT_HEADER_ONLY          // MUSS als erstes stehen
#include <fmt/format.h>
using fmt::format;               // optional, aber sinnvoll!
using fmt::print;                // optional, aber sinnvoll!
```

Ich habe diese Änderungen in der Datei `format-fmt.cpp` gemacht und es funktioniert.

Wer nicht in Systemverzeichnisse kopieren darf, muss einen anderen Weg gehen, der aber ohnehin sinnvoller ist (sonst muss man nämlich bei jedem Compilerupdate die Files erneut kopieren):

Man legt in seinem Arbeitsbereich ein C++-Verzeichnis an, z.B
`i:\C++` oder unter MSYS2 `/home/`*Username*`/C++` und darin 2 Unterverzeichnisse `lib` und `include`. In `include` kopiert man alle Headerdateien (eigene und `fmt`) und in `lib` kopiert man alle Programmbibliotheken. Jetzt muss man nur noch dafür sorgen, dass der Compiler sie dort auch findet. Folgende Compileroptionen (g++ und clang) brauchen wir:

`-I` *Directory*: sucht Includefiles auch in *Directory*

`-L` *Directory*: (großes L) sucht Libraries auch in *Directory*

Diese Optionen hängen wir an unsere Optionen im Alias `g++` bzw. `c++` an:

`-I /i/C++/include -L /i/C++/lib` (`/i/` statt i:) bzw.

`-I /home/`*Username*`/C++/include -L /home/`*Username*`/C++/lib`

Falls dann irgendwann die fmt-Library im C++ Compiler enthalten ist, kann man ab dann diese verwenden, ohne dass es einen Konflikt gibt.