Datentypen - Ströme

Anmerkung: Ströme dienen dazu, Eingaben aus verschiedenen Quellen (z.B. Konsole, Strings, Dateien) zu holen.

`std::stringstream`	Datentyp für **String-Streams**

Erfordert: `#include<sstream>`

Dient dazu, um Eingaben aus Strings zu holen.

Beachte: "*String*-Stream" heisst, dass im Objekt ein String *enthalten* ist. Es heisst aber **nicht**, dass nur Strings (oder `chars`) daraus *extrahiert* werden können. Darin können beispielsweise auch Zahlen als String vorliegen. Diese kann man als String (oder `char`) oder als Zahl extrahieren.

Objekte des Typs `std::stringstream` können nicht direkt kopiert werden. Deshalb sollte man sie immer via Call-by-Reference an Funktionen übergeben.

```
std::stringstream s ("b345e"); // stringstream with value "b345e"
char c;
s >> c; // c gets value 'b' and s is now "345e"
s >> c; // c gets value '3' (!as char! since type of c is char)
int n;
s >> n; // n gets value 45 (!as int! since type of n is int)
        // (This works since the computer sees that the next
        // 2 characters in the string "45e", namely '4' and '5',
        // can be used as the int 45. So after this operation
        // s is "e".)
```

`std::ifstream`	Datentyp für das **Auslesen einer Datei**

Erfordert: `#include<fstream>`

Dient dazu, um Eingaben aus Dateien zu holen.

Objekte des Typs `std::ifstream` können nicht direkt kopiert werden. Deshalb sollte man sie immer via Call-by-Reference an Funktionen übergeben.

(...)

(...)

```
// Count appearances of 'u' in my_file.txt
std::ifstream reader ("my_file.txt");
// rest of usage is same as for std::cin
char c;
int ctr = 0;
while(reader >> c)
    if(c == 'u')
        ++ctr;
```

std::istream	Datentyp für **Input-Streams**

Erfordert: `#include<istream>` oder `#include <iostream>`

Allgemeiner Datentyp, um Input-Ströme zu beschreiben. Man kann ihn sehr gut verwenden, um Funktionen, welche Input-Ströme als Argumente nehmen, unabhängig vom genauen zugrunde liegenden Typ (`std::stringstream`, `std::ifstream`, ...) zu gestalten.

Beispielsweise `std::cin` hat den Typ `std::istream`. Objekte der Typen `std::stringstream` und `std::ifstream` können auch als `std::istream` verwendet werden.

Objekte des Typs `std::istream` können nicht direkt kopiert werden. Deshalb sollte man sie immer via Call-by-Reference an Funktionen übergeben.

```
// POST: Two characters are removed from is. If is contains less
//       characters it is emptied.
void remove_two (std::istream& is) {
    char a;
    is >> a >> a; // remove two chars
}

int main () {
    // Assume that the user enters "Informatics".
    remove_two(std::cin); // istream
    char out;
    while (std::cin >> out)
        std::cout << out; // Output: formatics
    std::ifstream from_file ("my_file.txt");
    remove_two(from_file); // ifstream
    return 0;
}
```

`std::ostream`	Datentyp für **Output-Streams**

Erfordert: `#include<ostream>` oder `#include <iostream>`

Beispielsweise `std::cout` hat den Typ `std::ostream`. Objekte des Typs `std::stringstream` können auch als `std::ostream` verwendet werden.

Objekte des Typs `std::ostream` können nicht direkt kopiert werden. Deshalb sollte man sie immer via Call-by-Reference an Funktionen übergeben.

```
// POST: wrote the highscore of a given player to out.
void print (std::ostream& out, std::string name, int score) {
    out << "Player: " << name << "  Score: " << score << "\n";
}

int main () {
    print(std::cout, "Pete", 335);
    print(std::cout, "Paula", 410);
    return 0;
}
```

`struct`	**Container** für Datentypen

(...)

(...)

Wichtige Befehle:

Definition:	`struct str_name { int mem1; bool mem2; int mem3; };`
Objekt erstellen:	`str_name obj1;`
mit Startwerten:	`str_name obj2 = {3, true, 4};`
aus anderem Objekt:	`str_name obj3 = obj2;`
Zugriff auf Member:	`obj1.mem1`

Die *Definition* eines Structs hat ein `;` am Schluss.

Nur der Zuweisungsoperator (`=`) wird automatisch erstellt (und kopiert dann die Member einzeln). Die anderen Operatoren (z.B. `==`, `!=`, ...) muss man selbst passend überladen (siehe Eintrag `operator...`).

Bei der Default-Initialisierung eines Objekts des Typs `str_name` werden alle Member einzeln default-initialisiert. Für fundamentale Typen (`int`, `float`, usw.) bedeutet das, dass sie *uninitialisiert* sind, bis man ihnen nachträglich einen Wert zuweist. Das führt zu Problemen, falls man ihren Wert vorher schon ausliest.

```
struct candidate {
    std::string name;     // Name of the participant
    int age;              // Her/his age
};

int main () {
    // Initialization
    candidate mary;              // default-initialisation
    std::cout << mary.age;       // Undefined behavior
    mary.name = "Mary"; mary.age = 43;
    std::cout << mary.age;       // Problem gone: mary.age is 43
    candidate bob = {"Bob", 28}; // using starting values
    candidate fred = bob;        // using other object
    fred.name = "Fred";
    return 0;
}
```

Input/Output

`std::ws`	**Entfernt Whitespaces** am Anfang eines Input-Streams.

Erfordert: `#include<istream>` oder `#include <iostream>`

```
char c;
std::stringstream text ("d  a\n\nb");
text >> std::noskipws; // Do not ignore whitespaces.

// Output text without whitespaces
text >> c; std::cout << c; // Output: 'd'
text >> std::ws; // Remove: "  "
text >> c; std::cout << c; // Output: 'a'
text >> std::ws; // Remove: "\n\n"
text >> c; std::cout << c; // Output: 'b'
// Output in total: dab
```

`my_stream.eof()`	Prüfe, ob das **Ende des Streams** erreicht worden ist.

Erfordert: `#include<iostream>`

Ein Stream kann sich in verschiedenen Zuständen befinden. Und es gibt Funktionen, um den aktuellen Zustand abzufragen. `eof()` ist eine solche Funktion. Mit ihr prüft man, ob sich der Stream im Zustand "Ende der Eingabe erreicht" befindet.

EoF bedeutet "*End-of-File*".

```
std::stringstream b ("33");
int z;
b >> z; std::cout << b.eof(); // true (>> read 33 and reached
                              //      end of b)
```

`my_stream.fail()`	Prüfe, ob im Stream ein **ungültiges Zeichen** hätte gelesen werden sollen.

(...)

(...)

Erfordert: `#include<iostream>`

Ein Stream kann sich in verschiedenen Zuständen befinden. Und es gibt Funktionen, um den aktuellen Zustand abzufragen. `fail()` ist eine solche Funktion. Mit ihr prüft man, ob sich der Stream im Zustand "gescheiterte Eingabe" befindet.

```
std::stringstream b ("33a");
int z;
b >> z; std::cout << b.fail(); // false (>> read 33)
b >> z; std::cout << b.fail(); // true (>> ought to read 'a'
                               //      which is not an int.)
```

`my_stream.peek()`	Im Stream nächstes **Zeichen anschauen, ohne es zu entfernen**.

Erfordert: `#include<istream>` oder `#include <iostream>`

Der Rückgabewert ist die `int`-Repräsentierung des nächsten Zeichens (als `char`) im Stream. **Der Datentyp des Rückgabewerts ist also `int`.**

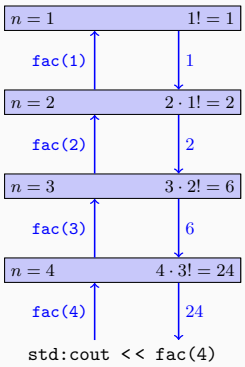
Diese Funktion ignoriert Whitespaces nie (unabhängig davon, ob der Stream zuerst an `std::noskipws` übergeben wurde oder nicht).

```
std::stringstream str ("my subst");
str >> std::noskipws; // Do not ignore whitespaces.
char c;

// remove everything before the first 's' (but leave 's' in str)
while (str.peek() != 's')
    str >> c;
// str is now: "subst"

// miscellaneous
std::stringstream num ("3 a");
std::cout << num.peek() << "\n"; // Output: 51 and NOT '3'
num >> c;
std::cout << c << "\n"; // Output: '3'
```

Funktionen

Rekursion	**Selbstaufruf** einer Funktion

Jeder rekursive Funktionsaufruf hat seine eigenen, unabhängigen Variablen und Argumente. Dies kann man sich sehr gut anhand des in der Vorlesung gezeigten Stacks vorstellen (`fac` ist im Beispiel unten definiert):




```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{
    if (n <= 1) return 1;
    return n * fac(n-1); // n > 1
}
```