SimpleStat als Objekt

Als Ausgangspunkt nehmen wir folgende Definition (Version 1) von `SimpleStat`:

```
struct SimpleStat {
     std::string title;        // Überschrift bei der Ausgabe
     int n = 0;                // Anzahl der Stichproben
     double sum = 0.;          // Summe der  Stichproben
     double sum2 = 0.;         // Summe der  Stichproben-Quadrate
};
```

Dieser Datentyp ist ein noch kein echtes Objekt (mit Konstruktoren etc.) aber auch kein POD (Plain Old Data), da er einen String enthält, der selbst ein echtes Objekt ist. Ein solcher Typ wird gerne als Aggregat bezeichnet und kann mit der Aggregat-Initialisierung (`{…}`) initialisiert werden (das sind noch keine Konstruktoren!). Alle Attribute sind oben **public**. Warum ist das ein Problem? In einem Anwendungsprogramm steht dann z.B. :

```
SimpleStat stat;
…
if (stat.samples = 10)    // Fehler! Es sollte == statt = heißen
```

Hier wird der Stichprobenzähler versehentlich umgesetzt und die Statistik liefert völlig falsche Werte. Verhindern lässt sich das nur dadurch, dass man die Datenattribute **private:** macht (Version 2):

```
struct SimpleStat {
private:
     std::string title;        // Überschrift bei der Ausgabe
     int n = 0;                // Anzahl der Stichproben
     double sum = 0.;          // Summe der  Stichproben
     double sum2 = 0.;         // Summe der  Stichproben-Quadrate
};
```

Genau denselben Effekt hätte man erzielt, wenn man das Schlüsselwort **`struct`** durch **`class`** ersetzt. Durch diese Änderung wird `SimpleStat` zu einem echten Objekttyp, der KEINE Aggregatinitialisierung zulässt sondern nur mehr durch Konstruktoren initialisiert werden darf. Leider kann jetzt niemand mehr mit der Version 2 arbeiten, weil NICHTS darauf zugreifen darf, außer den von C++ generierten speziellen Objekt-Methoden. Mit diesen kann man solche Objekte nur kreieren, kopieren, zuweisen, aber nicht bearbeiten. Man muss deshalb zusätzliche `public` Objektmethoden oder befreundete Funktionen definieren, da diese (und nur diese) auf die `private:` Attribute zugreifen dürfen. Uns

genügt es, eine Stichprobe zu verarbeiten und die statistische Endauswertung auszugeben. Erstere kann man gut als Objektmethode programmieren, ich verwende dazu den Operator +=

```
void operator+=(double x)
{    ++n; sum += x; sum2 += x*x; }
```

Die Streamausgabe muss sicher auf die Interna des Objekts zugreifen können, hat aber als erstes Argument den Ausgabestream, weshalb sie keine `SimpleStat`-Methode sein kann. Also muss man sie entweder als `friend` realisieren oder die Ausgabe durch eine Hilfsmethode erledigen lassen (ich habe dazu die konstante Methode `print()` geschrieben, weil ich später den Runtime-Polymorphismus demonstrieren möchte, was nur mit Methoden aber nicht mit Funktionen geht!):

```
struct SimpleStat {
     void operator+=(double);
     void print(std::ostream&) const;
private:
     std::string title;        // Überschrift bei der Ausgabe
     int n = 0;                // Anzahl der Stichproben
     double sum = 0.;          // Summe der  Stichproben
     double sum2 = 0.;         // Summe der  Stichproben-Quadrate
};


std::ostream& operator<<(std::ostream& os,const SimpleStat& s)
{    s.print(os); return os; }
```

C++ beschwert sich jetzt beim Beispielprogramm noch über die fehlerhafte Instanzierung:

```
     SimpleStat daten{"double Zahlenreihe"};
```

Echte Objekte kann man nicht mehr ohne Konstruktoren instanzieren und initialisieren. Eine Liste in geschwungenen Klammern mit einigen oder allen Attributwerten (Aggregat-Initialisierung) reicht dann nicht mehr. Wir schreiben also einen Konstruktor, der den Titel setzt. Alle übrigen Werte sind ohnehin 0 oder 0.0. Der Konstruktor muss, damit man ihn überhaupt verwenden darf, **public** sein. Hier eine einfache Version:

```
struct SimpleStat {
public:
     SimpleStat(const std::string&);
     void operator+=(double);
     void print(std::ostream&) const;
```

```
private:
     std::string title;
     int n = 0;                  // Anzahl der Stichproben
     double sum = 0.;            // Summe der  Stichproben
     double sum2 = 0.;           // Summe der  Stichproben-Quadrate
};
```

Es ist üblich (es gibt aber keinen zwingenden Grund dafür!), Objekte mit privaten Teilen eher als `class` statt als `struct` zu definieren.

```
class SimpleStat {
public:
     SimpleStat(const std::string&);
     void operator+=(double);
     void print(std::ostream&) const;
private:
     std::string title;          // Überschrift bei der Ausgabe
     int n = 0;                  // Anzahl der Stichproben
     double sum = 0.;            // Summe der  Stichproben
     double sum2 = 0.;           // Summe der  Stichproben-Quadrate
};
```

Hält man die Objektdefinition codefrei (das soll besonders schön sein?), muss man die Methoden anschließend oder in einem eigenen .cpp File implementieren. Dabei muss man den Scope der Methoden mit angeben!

```
SimpleStat::SimpleStat(std::string text) : title{text}
{}

void SimpleStat::operator+=(double x)
{    ++n; sum += x; sum2 += x*x; }

void SimpleStat::print(std::ostream& os) const
{    os << … }
```

Damit wäre man eigentlich schon fertig. Aber: Obiger Konstruktor (da er genau ein Argument besitzt, nämlich einen `std::string`) würde von C++ auch als automatische Umwandlung von `std::string` in `SimpleStat` eingesetzt werden. Da dies sicher nicht erwünscht ist, sollte man das verhindern, indem man den Konstruktor `explicit` macht.

```
class SimpleStat {
public:
     explicit SimpleStat(const std::string& text);
```

```
    …
};
```

`SimpleStat` ist bereits jetzt ein nützliches Tool zur statistischen Messwert-Analyse. Um es leicht in C++-Programmen einzusetzen, kann man eine von beiden Vorgehensweisen verwenden:

1) Man extrahiert den Objektcode in 2 separate Dateien: Eine Headerdatei `SimpleStat.h` (nur die Objektdefinition und die **Deklarationen** der Utility-Funktionen) sowie eine Codedatei `SimpleStat.cpp`. Das Anwendungsprogramm inkludiert den Header und beim Kompilieren gibt man die Codedatei mit an. Die Anwendung heiße `anwendung.cpp`:
2) Man belässt den gesamten Code in einer Headerdatei (das ist Pflicht, wenn man aus `SimpleStat` ein Objekt-Template macht: In diesem Fall **MUSS MAN ALLE METHODEN UND UTILITYFUNKTIONEN im HEADERFILE programmieren)**. Manche verwenden dafür die Extension `.hpp` , genauso oft wird aber nur `.h` verwendet. Zur Unterscheidung der beiden Varianten in diesem Ordner verwende ich die Headerdatei `StatisticObjects.h`:

Variante 1: `SimpleStat.h + SimpleStat.cpp`

Die Anwendung **`anwendung.cpp`**:

```
#include "SimpleStat.h"
…
SimpleStat daten{"double Zahlenreihe"};

for (double x; cin >> x;)
     daten += x;
cout << daten;
```

Die Headerdatei **`SimpleStat.h`**:

```
#ifndef SIMPLESTAT_H
#define SIMPLESTAT_H    // protect against multiple Inclusions

#include <string>
#include <iostream>

class SimpleStat {
public:
     SimpleStat(std::string);
     void operator+=(double);
     void print(std::ostream&) const;
private:
     std::string title;
```

```
    int n = 0;                  // Anzahl der Stichproben
    double sum = 0.;            // Summe der  Stichproben
    double sum2 = 0.;           // Summe der  Stichproben-Quadrate
};

std::ostream& operator<<(std::ostream&, const SimpleStat&);
#endif              // #ifndef SIMPLESTAT_H
```

Die Headerdatei muss alles inkludieren, was sie selbst benötigt `(iostream, string)`. **Auch ist es schlechter Stil, in Headerdateien die `using namespace std;` Anweisung zu verwenden**, weshalb man hier die benötigten `std::` schreiben muss. Erklärung: Ein Anwender von `SimpleStat` möchte gerne selbst entscheiden, ob er diese `using`-Anweisung verwenden will oder nicht. Es kommt nicht gut an, wenn eine Headerdatei, die man inkludieren muss, diese Entscheidung trifft. Auch sollte man einen Schutz gegen mehrmaliges Inkludieren des Headers vorsehen (s.o.).

Den C++-Code der Methoden und der Streamausgabe packt man in `SimpleStat.cpp`. Diese inkludiert **als erste Anweisung** `SimpleStat.h` und implementiert nur mehr den fehlenden Code der Methoden und der Streamausgabe. Eine kleine Unterlassung in der Ausgabe habe ich auch noch behoben. Selbstverständlich darf man hier die `using namespace std;` Anweisung wieder verwenden, wenn man möchte, da diese Datei nirgends inkludiert wird:

```
#include "SimpleStat.h"         // als erste Anweisung
#include <cmath>

using namespace std;

SimpleStat::SimpleStat(string text): title{text}
{}

void SimpleStat::operator+=(double x)
{    ++n; sum += x; sum2 += x*x; }

void SimpleStat::print(ostream& os) const
{
    os << "Statistische Auswertung: " << title << '\n';
    os << "Anzahl der Stichproben: " << n << '\n';
    if (n == 0)
        return;
    auto mw = sum/n;
    os << "Mittelwert der Daten:   " << mw << '\n';
    if (n == 1)
```

```
          return;
     auto sa = (sum2 - n*mw*mw)/(n-1);
     os << "Standardabweichung:     " << sa << '\n';
     os << "Streuung:               " << sqrt(sa) << '\n';
}

ostream& operator<<(ostream& os,  const SimpleStat& stat)
{    stat.print(os); return os; }
```

Variante 2: `StatisticObjects.h`

Um die Realisierung als Template zu rechtfertigen, machen wir den Typ unserer Summenvariablen und auch den Stichprobenzähler-Typ variabel. Das kann sicher nützlich sein, wenn man sehr viele (mehr als 2 Milliarden) Stichproben verarbeiten möchte. Man erspart sich jede Menge Ärger, wenn man alle Methoden und vor allem Freunde eines Objekttemplates innerhalb der Objektdefinition programmiert (und darauf pfeift, dass das hässlich sein soll). Ich nenne den Objekttyp `SimpleStatisticObject` statt `SimplStat` und definiere danach `SimplStat` kompatibel als Spezialfall.

Die Anwendung **`anwendung.cpp`** bleibt fast genau gleich:

```
#include "StatisticObjects.h"
…
SimpleStat daten{"double Zahlenreihe"};

for (double x; cin >> x;)
     daten += x;
cout << daten;
```

Die Headerdatei **`StatisticObjects.h`**:

```
#ifndef STATISTICOBJECTS_H
#define STATISTICOBJECTS_H // protect against multiple Inclusions

#include <string>
#include <iostream>

template<typename DataType, typename CounterType = int>
class SimpleStatisticObject {
public:
     SimpleStatisticObject(const std::string& t) : title{t}
```

```
    {}

    void operator+=(DataType x)
    { ++samples; sum += x; sum2 += x*x; }

    void print(std::ostream& os) const
    {
        os << "Statistische Auswertung: " << title << '\n';
        os << "Anzahl der Stichproben: " << n << '\n';
        if (n == CounterType{})
            return;
        auto mw = sum/samples;
        os << "Mittelwert der Daten:   " << mw << '\n';
        if (n == CounterType{1})
            return;
        auto sa = (sum2 - n*mw*mw)/(n -1);
        os << "Standardabweichung:     " << sa << '\n';
        os << "Streuung:               " << sqrt(sa) << '\n';
    }
private:
    std::string title;
    CounterType n{};          // Anzahl der Stichproben, = 0
    DataType sum{};           // Summe der  Stichproben, = 0.
    DataType sum2{};          // Summe der  Stichproben-Quadrate, = 0.
};

template<typename DataType, typename CounterType = int>
    // Auch diese Funktion wird zu einem Template
std::ostream& operator<<(std::ostream& os,
    const SimpleStatisticObject<DataType, CounterType>& s)
{   s.print(os);
    return os;
}

using SimpleStat = SimpleStatisticObject<double, int>;
// definiert SimpleStat als SimpleStatisticObject<double, int>
// so bleibt man zur alten Version kompatibel
// benutze z.B.  SimpleStatisticObject<double, long>  für viele Stichproben

#endif
```

In der modernen Zeit würde man die Streamausgabe eher als „hidden friend“ implementieren, da das ganze einfacher wird, schneller compiliert und Fehlermeldungen einsparen kann, also nur Vorteile hat. Dazu verschiebt man die Definition der Streamausgabe

in die Klassendefinition. Da es aber eine Funktion sein muss (keine Methode), muss man sie noch als `friend` kennzeichnen. Damit dürfte diese Funktion selbst schon auf die Klasseninterna zugreifen und bräuchte die `print()`−Methode dazu gar nicht. Diese braucht man nur, wenn man Polymorphismus haben will (das geht nur mit Objektmethoden). Man ersetzt die externe Definition von `operator<<` durch folgendes im `public` oder `private` Teil:

```
friend inline std::ostream& operator<<(std::ostream& os,
     const SimpleStatisticObject& s)     // ohne <…>
{    s.print(os); return os; }
```