StoringStat Objekte

Wir haben bereits das Statistik-Objekt `SimpleStat` entwickelt und in Headerdatei und Codedatei aufgespalten. Da `SimpleStat` schon die einfachen Standardkennzahlen ausgeben kann, wollen wir nun eine nichttriviale Kennzahl ermitteln und ausgeben: den Median!

Dazu muss man alle Daten der Größe nach sortieren und bei einer ungeraden Stichprobenzahl den mittleren Wert, bei einer geraden Stichprobenzahl den **Mittelwert der beiden mittleren Werte** nehmen. Also muss man für diesen Zweck alle Werte speichern, was `SimpleStat` nicht kann. Wir entwickeln dazu eine neues Statistik-Tool `StoringStat`, das alles kann, was `SimpleStat` schon konnte und das zusätzlich den Median ermittelt. Es sollte wie ein `SimpleStat`-Objekt verwendet werden können, d.h. am besten ein `SimpleStat`-Objekt sein. Wir implementieren `StoringStat` daher **als Erweiterung** von `SimpleStat`. Also wird `StoringStat` zu einer Klasse, die von `SimpleStat` **abgeleitet ist** (von `SimpleStat` erbt). Umgekehrt wird `SimpleStat` zu einer **Basisklasse** von `StoringStat`:

```
class StoringStat : public SimpleStat {
…
};
```

Jetzt ist `StoringStat` ein `SimpleStat` mit Benefits, alle Teile (Attribute und Methoden außer Konstruktoren, Destruktor, Zuweisungen) von `SimpleStat` sind nun auch in `StoringStat` vorhanden. Innerhalb der Klammern `{}` definiert man nur jene Teile von `StoringStat`, die man **zusätzlich** benötigt. Man braucht sicherlich einen Container zum Speichern der Stichproben (etwa `std::vector<double> data`). Ein neuer Konstruktor wird auf jeden Fall benötigt, denn den erbt man nicht. Es muss aber auch die Verarbeitung einer Stichprobe geändert werden (diese muss jetzt zusätzlich gespeichert werden). Und die neue Ausgabe sollte natürlich auch den Median drucken. Man kann diesen neuen erweiterten Methoden andere Namen geben (z.B. `push_back_and_store()`), oder dieselben Namen verwenden (`operator+=()`). Man muss die zweite Variante wählen, wenn man von der Polymorphie profitieren will. C++ erkennt am Referenzobjekt, welche Variante es nehmen muss:

```
SimpleStat s1;  s1 += 1.0     // SimpleStat Methode
StoringStat s2;  s2 += 1.0;   // StoringStat Methode
```

Da `s2` aber **auch eine `SimpleStat`-Instanz ist**, dürfte man damit auch die `SimpleStat` Methoden aufrufen. Man muss dann aber den Scope explizit angeben:

```
s2.SimpleStat::operator+=(1.); // SimpleStat Methode von s2 verwenden
```

Die vollständige Definition der neuen Klasse `StoringStat` ist daher:

```
class StoringStat : public SimpleStat {
public:
     explicit StoringStat(std::string); // neuer Konstruktor
     void operator+=(double);           // neues  +=  mit Speichern
     void print(std::ostream&) const;   // neue  Ausgabe mit Median
     const std::vector<double>& get_data() const noexcept;
          // Getter-Methode: erlaube Lesezugriff auf die Stichproben
private:
     vector<double> data;               // der Stichprobenspeicher
};
```

Die Attribute `title, n, …` übernehmen wir durch das Erben von `SimpleStat`. Im Code der neuen Methoden verwenden wir die alten mit, indem wir deren Scope angeben:

```
void StoringStat::operator+=(double x)
{
     data.push_back(x);                 // wir speichern den Wert zuerst
     SimpleStat::operator+=(x);         // als SimpleStat verarbeiten
}
```

Analog schreibt man auch die Funktion `print()`. Diese erzeugt zusätzlich zum alten Output die Ausgabe des Medians. Beim Sortieren würde der Stichproben-Speicher verändert werden, was eine konstante Methode nicht machen darf. Wir kopieren daher die Stichproben und sortieren die Kopie mit dem `std::sort()` Utility. Um das auch von außerhalb abfragen zu können, implementieren wir die Medianberechnung in einer eigenen Objektmethode

```
double median() const
{    if (n > 0) {              // Fehler!!!! s.u.
         vector<double> tmp{data};      // wir kopieren (Copy-Konstruktor)
         sort(begin(tmp), end(tmp));    // sortieren die Kopie
         return (samples % 2 != 0 ?     // gerade ungerade?
             tmp[samples /2] :                          // ungerade!
             (tmp[samples /2-1] + tmp[samples /2])/2);// gerade
    } else
          return 0.;
}

void print(std::ostream& os) const
{    SimpleStat::print(os);             // alter Ausdruck
     os << "\tMedian:      " <<  median() << '\n';
}
```

Der neue Konstruktor ist auch sehr einfach, wenn man die Prinzipien kennt: C++ initialisiert **VOR** der Abarbeitung des Konstruktor-Codes in `{ … }` immer alle Basisklassen (hier: `SimpleStat`), alle Klassenattribute mit eigenem Konstruktor (`vector data`) und alle Klassenattribute mit Initialisierungen in der Klassendefinition (hier keine). Ist keine Initialisierungsliste vorhanden, werden die Default-Konstruktoren bzw. die vordefinierten Initialisierer aufgerufen. Eine vorhandene Initialisierungsliste kann diese Automatik abändern (aber nicht verhindern!), da sie Vorrang hat.

```
StoringStat::StoringStat(const string& t) : SimpleStat{t}
{}
```

Diese Initialisierungsliste ruft den `SimpleStat`-Konstruktor mit dem Titel auf. Und das ist gut so, denn einen anderen besitzt `SimpleStat` nicht. Ohne Initialisierungsliste wäre hier der Default-Konstruktor von `SimpleStat` aufgerufen worden, und das wäre natürlich ein Fehler gewesen, da der nicht existiert! Das einzige neue Attribut `data` fehlt in der Initialisierungsliste. Deshalb wird es (da es ein Objekt ist) mit dessen `Default`-Konstruktor als leerer `vector<double>` initialisiert. Man könnte es alternativ auch explizit angeben (was von vielen empfohlen wird):

```
StoringStat::StoringStat(string text)
: SimpleStat{text}, data{}
{}
```

Leider findet C++ einen Fehler in diesem Code, denn die neue `median()`−Methode von `StoringStat` greift auf den **privaten** Stichprobenzähler `n` von `SimpleStat` zu, und **das ist NICHT erlaubt.**

Methoden der abgeleiteten Klassen (Kinder) dürfen nicht auf die privaten Teile der Basisklassen (Eltern) zugreifen (wie schön wäre das im wirklichen Leben! ☺).

Einfacher Ausweg: Man ändert **in der Basisklasse** das Schlüsselwort `private:` in `protected:` um, das genau zu diesem Zweck erfunden wurde: Dann dürfen Methoden der abgeleiteten Klassen aller Generationen (Kinder, Enkel, Großenkel …) darauf zugreifen. Alternativ hätte man natürlich die Stichprobenzahl aus der Größe des Stichproben-vectors `data` ermitteln können, wozu kein Zugriff auf `SimpleStat`s Privatsphäre nötig ist.

Virtuelle Methoden

Unsere Implementierung von `StoringStat` ist noch nicht polymorph, da keine virtuellen Funktionen definiert wurden!

```
StoringStat daten{"double Zahlenreihe"}; …
daten.print(cout);        // Ausgabe mit Median
cout << daten;            // Ausgabe OHNE Median
```

Die erste Ausgabe ruft natürlich die `StoringStat::print()`−Methode auf, die den Median ausgibt, da `daten` den Typ `StoringStat` hat. Was passiert bei der Streamausgabe (und warum funktioniert diese sogar, wenn auch nicht völlig richtig)? Wir haben nämlich bisher **keine Streamausgabe für die Klasse `StoringStat`** definiert, es gibt nur jene für `SimpleStat`:

```
ostream& operator<<(ostream& os,  const SimpleStat& s)
{     s.print(os);        // gib die Statistik mit print() aus
      return os;          // gib den Stream zurück für die Verkettung von <<
}
```

1) **Warum kann man diese auch auf `StoringStat`-Instanzen anwenden?**
2) Warum ruft sie die `SimpleStat::print()` Methode auf?

Frage 1 ist leicht zu beantworten: Jede `StoringStat`-Instanz **ist (auch)** eine `SimpleStat`-Instanz (wegen der Vererbung) und kann daher auch als `SimpleStat`-Referenz oder `SimpleStat`-Pointer übergeben werden (**Polymorphismus**!).

Antwort auf Frage 2 ist noch einfacher: What else (Copyright George Clooney + Nespresso)

Mögliche Lösungen:

1) Wir schreiben eine eigene Streamausgabe für `StoringStat`. Diese würde dann für `StoringStat`-Instanzen genau passen und daher vom Compiler in diesem Fall bevorzugt werden.
2) Wir sorgen dafür, dass die existierende Streamausgabe für `SimpleStat` die „richtige“ `print()`−Methode aufruft. Nur dadurch werden wir echt polymorph.

Variante 2 ist die bevorzugte und sie ist relativ leicht zu implementieren:

Man definiert **in der Basisklasse** `SimpleStat` (!!!**)** die `print()`−Methode als **`virtual`**. Dasselbe wird man sinnvollerweise auch für die `operator+=()`−Methode machen, die ebenfalls in `StoringStat` umdefiniert wurde. Danach stellt man fest, dass alles korrekt funktioniert, d.h. beide Ausgabe-Anweisungen von oben drucken den Median aus. Die neue Definition von `SimpleStat` ist somit (Änderungen fett):

```
class SimpleStat {        // geht genauso mit struct statt class
public:
     explicit SimpleStat(std::string);  // Konstruktor
```

```
    virtual void operator+(double);     // Stichprobe  verarbeiten
    virtual void print(std::ostream&) const;// Ausgabe auf Stream
protected:
    std::string title;
    int n = 0;                  // Anzahl der Stichproben
    double sum = 0.;            // Summe der  Stichproben
    double sum2 = 0.;           // Summe der  Stichproben-Quadrate
};
```

Virtueller Destruktor ???

In allen C++ Lehrbüchern findet man die Empfehlung, dass Basisklassen mit virtuellen Methoden auch einen virtuellen Destruktor haben sollten, weil ansonsten die Instanzen der abgeleiteten Klasse „falsch“ zerstört werden können (???). Diese Regel ist durchaus ernst zu nehmen: Betrachten wir folgendes (in dieser Einfachheit zweifelhaftes ?) Codefragment:

```
SimpleStat* stat = new StoringStat{"double Zahlenreihe"};
delete stat;
```

Diese komplizierte erste Zeile generiert eine `StoringStat`-Instanz am Heap und speichert ihre Adresse in einem `SingleStat`-Pointer (warum auch immer). Das ist erlaubt, da `StoringStat`-Instanzen auch `SimpleStat`-Instanzen sind. In der 2. Zeile wird diese Instanz wieder zerstört. Dabei wird der Destruktor von `SimpleStat` aufgerufen! Und dieser zerstört natürlich nicht den Datenvektor von `stat`, da er von diesem keine Ahnung hat. So entsteht ein Speicherleck: Der Datenvektor `stat.data` bleibt erhalten, obwohl niemand ihn mehr verwenden kann.

Deklariert man jedoch den Destruktor der Basisklasse als `virtual`, dann ist er das auch in allen abgeleiteten Klassen. Obiger Fehler kann nicht mehr auftreten, da in der 2. Zeile der Destruktor über die Sprungtabelle aufgerufen wird und deshalb der `StoringStat` Destruktor zum Einsatz kommt und dieser den Datenvektor korrekt zerstört! Das gilt selbstverständlich auch dann, wenn der Destruktor von C++ erzeugt wird!

Hat man selbst in der Basisklasse einen Destruktor geschrieben, ergänzt man `virtual` davor. Hat man keinen geschrieben (und der Destruktor ist C++-erzeugt), schreibt man am besten in der Basisklasse:

```
virtual  ~Klassenname() = default;
```

Dadurch wird der C++-erzeugte Destruktor `virtual.` Es gibt auch Alternativen zu einem virtuellen Destruktor.

Noch mehr Ideen?

Multi-Threading:

Obige Implementierung ist nicht threadsafe, d.h. greifen mehrere Threads auf DASSELBE Statistikobjekt zu, gibt es wieder sog. „race-conditions“ und der Ausgang ist zufallsabhängig. Nicht schön! **Das gilt nicht**, wenn jeder Thread ein EIGENES Statistik-Objekt benutzt, d.h. trotz Multithreadings KANN es funktionieren. Für den allgemeinen Fall könnte man neue threadsafe Versionen `SimpleStatMT` und `StoringStatMT` programmieren (MT für Multi-Threading), die jeden kritischen Zugriff mit Schloss und Riegel (`std::mutex`) schützen.

Statistik-Templates?

Manchmal braucht ein Anwender spezielle Statistiken über z.B. `long double` Messwerte mit der größtmöglichen Genauigkeit, oder aber über sehr viele Messwerte (der `int`-Stichprobenzähler läuft bei ca. 2 Milliarden über!). `StoringStat` benötigt viel Speicherplatz, sodass man nicht ohne Not immer `long double` speichern möchte. Also wären Varianten mit anderem Datentyp und/oder anderem Zählertyp „nice“. Genau dafür gibt es die Objekt-Templates in C++: Der Anwender kann sich den verwendeten Typ des Statistik-Objekts selbst aussuchen. So wurde schon aus `SimpleStat` schon ein `SimpleStatisticObject<…>`.

Macht man das analog für `StoringStatisticObject<…>`, das natürlich wieder von `SimpleStatisticObject<…>` abgeleitet wird (ein Template erbt von einem Template), so gibt es einige weitere Fallstricke zu beachten, auf die ich hier nicht eingehen kann.

Tipps für die Template-Programmierung

1) Templates (Funktions-Templates und Objekt-Templates) müssen **IMMER komplett im Headerfile** programmiert werden (d.h. eine Auftrennung in Headerdatei mit Deklarationen und Codedatei **IST NICHT MÖGLICH!!).**
2) Widerstehen Sie der Versuchung, „schön zu programmieren“ und die Objektdefinition codefrei zu halten. Sie ersparen sich jede Menge Ärger, wenn Sie **ALLE Objektmethoden und ganz besonders ALLE `friend`-Funktionen** komplett innnerhalb der `class`-Definition **PROGRAMMIEREN (inclusive Code!).** Da Sie den Code ohnehin nicht in eine Code-Datei auslagern können, wäre der Nettogewinn dieser Mehrarbeit ohnehin 0.

3) Korrekte Auftrennung von `friend`-Funktionsdeklaration und deren Programmierung außerhalb der Klassendefinition kann einen Programmierer in den Wahnsinn treiben! Ich habe mein erstes (und einziges) Beispiel erst nach Studium eines langen Youtube-Videos erfolgreich geschafft. Jetzt weiß ich immerhin, dass und wie es im Prinzip geht, aber auch, dass ich es definitiv nicht so machen will!