Bitte beachten Sie, dass Sie für eine Klausurzulassung mindestens einmal in Ihrer Übungsgruppe vorgerechnet haben müssen. Sie können *nicht* davon ausgehen, dass das auf den letzten Drücker klappt.

Übung 9.1 Lesen und Schreiben von Dateien

Wir haben uns bisher auf die Bearbeitung kurzer Zeichenfolgen beschränkt, die interaktiv vom Programm eingelesen oder ihm auf der Kommandozeile übergeben wurden. Häufig möchte man jedoch größere Datenmengen verarbeiten, zum Beispiel längere Texte oder in der Numerik oder Physik lange Tabellen aus gemessenen oder von einem Programm berechneten Werten.

Solche Zeichenfolgen werden in der Regel in Dateien gespeichert. Für den Umgang mit Dateien werden über den Header `fstream` die beiden Klassen `std::ifstream` (Input Filestream) und `std::ofstream` (Output Filestream) zur Verfügung gestellt. Diese sind von den bereits bekannten, allgemeineren Streamklassen `std::istream` bzw. `std::ostream` abgeleitet, und man kann sie im Wesentlichen wie `std::cin` und `std::cout` benutzen. Man öffnet einen solchen Filestream mit `stream.open(<Datei>)`, prüft mit `stream.good()`, ob Lesen bzw. Schreiben möglich ist, und schließt den Filestream über `stream.close()`.

Schreiben Sie eine Funktion, die für einen gegebenen Bezeichner `<DATEI>` auf die beiden Dateien `<DATEI>.txt` und `<DATEI>-a.txt` zugreift. Lesen Sie die erste Datei über die Funktion

```
std::istream& getline ( std::istream& is, std::string& str )
```

zeilenweise ein, und schreiben Sie diesen String (wie mit `std::cout`) in die zweite Datei, allerdings mit vorangestellter Zeilennummer. Nach der Nummer sollen ein Doppelpunkt und ein Leerzeichen folgen, die sie von der ursprünglichen Zeile trennen.

Anmerkungen: Das Ende der Datei können Sie daran erkennen, dass die Methode `good() false` liefert. Für die String-Klasse sind die Operatoren so überladen, dass sich die Verknüpfung von Strings einfach als Addition schreiben lässt. Benutzen Sie dies bei der Erzeugung der Dateinamen.

5 Punkte

Übung 9.2 Zweidimensionales Array

Für viele Anwendungen benötigt man Datenstrukturen, welche einen indizierten Zugriff mit mehreren Indizes ermöglichen. Siehe beispielsweise das folgende Programmfragment, das für eine Matrixklasse `Matrix` eine Einheitsmatrix initialisiert:

```
Matrix A(5,5,0.0); // Initialisiere 5x5 Matrix mit Nullen
for (int i=0; i<A.rows(); i=i+1)
  A[i][i] = 1.0;
```

Wir wollen uns in dieser Übung damit beschäftigen, wie man in C++ die bequeme Syntax mit zweifachem indizierten Zugriff realisieren kann. In Vorbereitung einer zukünftigen Aufgabe wollen wir einen Container für `bool`-Werte schreiben. Es gibt prinzipiell mehrere Möglichkeiten, den doppelten Zugriff zu implementieren:

- Man wählt als interne Datenstruktur ein Array von Arrays (Typ `bool**`). Der indizierte Zugriff kann dann direkt durch einen indizierten Zugriff in die interne Datenstruktur implementiert werden. Das korrekte Allozieren von Speicher ist in diesem Fall jedoch aufwändig und fehleranfällig. Diese Variante wird daher im Allgemeinen vermieden!
- Man wählt als interne Datenstruktur ein einzelnes großes Array, welches alle Daten der Matrix, zeilenweise hintereinandergeschrieben, enthält. Der `operator[]` des Containers kann einen Pointer `bool*` auf den entsprehchenden Zeilenanfang zurückgeben. Durch die Äquivalenz von

Pointern und Arrays kann auf diesem Objekt wiederum ein `operator[]` aufgerufen werden. Diese Variante hat wie obige den Nachteil, daß der Nutzer wenn er nur einen einfachen indizierten Zugriff ausführt ein Objekt erhält, dessen Semantik unklar ist.

- Wiederum ausgehend von einem einzelnen großen Array, implementiert man den `operator[]` so, dass dieser ein temporäres Objekt zurück, dessen Lebensdauer nur ausreicht, um eine weitere Methode auf dem Objekt aufzurufen - in diesem Fall dessen `operator[]`. Dieser "innere" Operator hat beide Indizes zur Verfügung und muss den entsprechenden `bool` aus dem Datenarray heraussuchen. Es ist hier wichtig, eine Referenz anstatt einer Kopie zurückzugeben, denn sonst wäre ein Schreiben in den Container durch indizierten Zugriff (wie im Matrixbeispiel) nicht möglich.

Da es sich um die technisch eleganteste Möglichkeit handelt, wollen wir Variante Nummer 3 implementieren. Ihr Container soll das folgende Interface erfüllen:

```
class TwoDBoolArray
{
  public:
  // Initialisiere ein nxm Array
  TwoDBoolArray(int n, int m);
  // Copy-Konstruktor
  TwoDBoolArray(const TwoDBoolArray& other);
  // Destruktor
  ~TwoDBoolArray();
  // Zuweisungsoperator
  TwoDBoolArray& operator=(const TwoDBoolArray& other);
  // Gebe Zeilenzahl zurück
  int rows();
  // Gebe Spaltenzahl zurück
  int cols();
  // ein Objekt das vom operator[] zurückgegeben wird
  class RowProxy
  {
    public:
    // Konstruktor
    RowProxy(bool* daten, int spalte, int spaltenzahl);
    // der "innere" Klammerzugriffsoperator
    bool& operator[](int j);
    private:
    bool* _daten; int spalte; int spaltenzahl;
  };
  // der "äußere" Klammerzugriffsoperator
  RowProxy operator[](int i);
  private:
  bool* daten;
  int m,n;
};
```

Beachten Sie bei Ihrer Implementierung die auf Blatt 8 besprochene Rule of Three. [*5 Punkte*]

In der Vorlesung haben Sie die Streamoperatoren `operator<<` und `operator>>` kennengelernt. Wir wollen diese nun für unsere Klasse `TwoDBoolArray` überladen. Im Gegensatz zu den bisher überladenen Operatoren lassen sich diese nicht als Klassenmethoden implementieren. Dies ist bei binären Operatoren (also solchen, die zwei Argumente haben) immer nur dann möglich wenn die Klasse als linker Operand auftritt, nicht jedoch wenn sie als rechter Operand auftritt. Dies wird besonders deutlich, wenn man die Multiplikation mit einem Skalar für eine Vektorklasse betrachtet:

```
Vektor x(5,1.0); // Initialisiere Vektor mit 5 Einträgen
Vektor y; double scalar;
y = x * scalar; // rufe Vektor::operator*(double scalar) auf.
y = scalar * x; // Wir können double::operator* nicht überladen!!!
```

Auch die Streamoperatoren sind binäre Operatoren: Das linke Argument ist der Stream (auf den wir keinen Einfluss haben), das rechte Argument die Daten. Man schafft sich hier Abhilfe durch das Implementieren von "freien" Operatoren. Diese sind Funktionen, die nicht an einen Klassennamensraum gebunden sind. Der Compiler verwendet die Regeln des Function Overloadings um die

richtige Implementierung auszuwählen. Sie können das folgdende Gerüst für Ihre Implementierung verwenden:

```
std::ostream& operator<<(std::ostream& stream, TwoDBoolArray& array)
{
  // ...
  return stream;
}

std::istream& operator>>(std::istream& stream, TwoDBoolArray& array)
{
  // ...
  return stream;
}
```

Es ist wichtig, dass diese Funktionen den modifizierten Stream zurückgeben. Beim `operator>>` müssen Sie Annahmen darüber treffen, in welcher Form die Daten vorliegen. Gehen Sie davon aus, dass es sich um Textdateien bzw. Konsoleneingabe handelt, und dass die ersten beiden Zeichenketten der Eingabe die Zeilen- bzw. Spaltenzahl sind. Danach folgen zeilenweise die Einträge des Arrays.

[*5 Punkte*]

10 Punkte

ÜBUNG 9.3 VERERBUNG

Finden Sie die Fehler im folgenden Programmfragment. Begründen Sie kurz, warum der jeweilige Ausdruck falsch ist.

```
3  class A {
4  public:
5      int ap;
6      void X();
7  private:
8      int aq;
9      void aX();
10 };
11
12 class B : public A {
13 public:
14     int bp;
15     void Y();
16 private:
17     int bq;
18     void bY();
19 };
20
21 class C : public B {
22 public:
23     int cp;
24     void Z();
25 private:
26     int cq;
27     void cZ();
28 };

   ...

38 void B::Y()
39 {
40     bq = bp;
41     aq = ap;
42     bY();
43 }
44
45 void C::cZ()
46 {
47     ap = 1;
48     bp = 2;
49     cq = 3;
50     X();
51     Y();
52     aX();
53 }
54
55 int main()
56 {
57     A a;  B b;  C c;
58
59     a.X();
60     b.bY();
61     c.cp = 4;
62     c.bp = 1;
63     c.ap = 2;
64     c.aq = 5;
65
66     b.ap = c.ap;
67
68     return 0;
69 }
```

Den Code können Sie über

`http://conan.iwr.uni-heidelberg.de/teaching/info1_ws2014/c++/vererbung.cc`

herunterladen.

5 Punkte