Übung 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.

(4 Punkte)

Übung 2 *Vererbung*

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

```
3  class A
4  {
5  public:
6    int ap;
7    void X();
8  private:
9    int aq;
10   void aX();
11 };
12
13 class B : public A
14 {
15 public:
16   int bp;
17   void Y();
18 private:
19   int bq;
20   void bY();
21 };
22
23 class C : public B
24 {
25 public:
26   int cp;
27   void Z();
28 private:
29   int cq;
30   void cZ();
31 };
32
33 void A::X() { };
34 void A::aX() { };
35 void B::bY() { };
36 void C::Z() { };
37
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

`https://conan.iwr.uni-heidelberg.de/data/teaching/info1_ws2019/vererbung.cc`

herunterladen.

(8 Punkte)

Übung 3 *Virtuelle Methoden*

Betrachten Sie folgende Hierarchie von drei Klassen.

```
class A
{
public :
  void a();
  virtual void va() = 0;
  virtual void a(int n);
private :
  void c();
};
```

```
class B : public A
{
public :
  void b();
  virtual void vb();
  void a(double d);
  void a(int i);
  virtual void va();
};
```

```
class C : public B
{
public :
  virtual void c();
  void a(float);
  virtual void a();
  virtual void va();
};
```

Eine Implementierung aller deklarierten Methoden existiert. Was passiert, wenn Sie den folgenden Programmteil kompilieren wollen? Welche Ausdrücke sind nicht korrekt, und warum?

```
1    A a;  B b;   C c;  A* pa=&b;  B* pb=&c; float x = 1.2;
2    pa->a();  pa->va();  pa->a(1);  pa->c();  pa->b();  pa->vb();  pa->a(x);
3    pb->a();  pb->va();  pb->a(1);  pb->c();  pb->b();  pb->vb();  pb->a(x);
4    pa = &c;
5    pa->a();  pa->va();  pa->a(1);  pa->c();  pa->b();  pa->vb();  pa->a(x);
```

Wenn Sie die falschen Ausdrücke entfernt haben und das Programm ausführen, von welcher Klasse werden dann die Methoden aufgerufen? Geben Sie für jeden Methodenaufruf an, ob die Methodenimplementierung von Klasse `A`, `B` oder `C` verwendet wird. **(8 Punkte)**

Übung 3

→ Alle rot markierten Felder sind nicht korrekt weil:

A * pa = &b;

pa -> c();: c() ist privat, kein Zugriff auf private Member der Base-Klasse möglich.

pa -> b();: Methoden aus B können nur aufgerufen werden, wenn sie in A sind. Werden Methoden aus A nicht in B überschrieben, gilt die "Implementierung" aus A. vererbte Vererbung, durch Polymorphie.

pa -> vb();: Vb() ist nicht in A, egal ob virtual. Es gilt auch hier die vererbte Vererbung.

pa -> a(x);: funktioniert. Float wird implizit in int gewandelt. Aber Nicht-Zahlentypen gehen nicht.

B * pb = &c;

pb -> c();: c() ist privat, kein Zugriff auf private Member der Base-Klasse möglich.

pb -> a(x);: funktioniert, ist aber double.

pa = &c

pa -> c();: c() ist wieder privat.

pa -> b();: Auch wenn C von B erbt geht es nicht, weil A eine Referenz von C und deswegen nur Methoden von A gelten können oder überschrieben werden können. Vererbung mit Verlusten.

pa -> vb();: geht nicht, Methode nicht in A.

pa -> a(x);: funktioniert, wird implizit in int umgewandelt.

→ Jetzt, alle Ausdrücke, die funktionieren:

A * pa = &b;

pa -> a();: wird in A aufgerufen, keine Überschreibung in B

pa -> va();: wird in B aufgerufen, wegen virtual in A und weil in B implementiert bzw überschrieben.

pa -> a(1);: wird in B aufgerufen, Methode in A wird in B überschrieben, egals ob in **B!** virtual davor steht oder nicht.

B * pb = &c;

pb -> a();: wird in A aufgerufen weil void A() in A nicht virtual.

pb -> va();: wird in C aufgerufen und überschrieben, virtual in Base-Klasse.

pb -> a(1);: wird in B aufgerufen, zuerst wird A initialisiert, dann aber B dann C.

pb -> b();: wird in B aufgerufen, da nur in B existiert.

pb -> vb();: wird in B aufgerufen, da nur in B existiert.

pa = &c

pa -> a();: wird in A aufgerufen, existiert nicht in C und nicht virtual in A.

pa -> va();: wird in C aufgerufen, weil virtual in A deklariert und in C überschrieben.

pa -> a(1);: wird in A aufgerufen, virtuelle int-Methode von A verwendet.