Abend 4
Übung : Erweitern von Klassen durch Vererbung

Ziel, Inhalt

- Wir erweitern die Klassen, die wir zum Zeichnen mit TsuZeichnen verwenden. Dabei wenden wir die Vererbung an um die Klassen zu spezialisieren

Erweitern von Klassen durch Vererbung

Übung 1: Eine rote Zeichenfläche

Die erste Übung dient dazu selber eine Klasse zu erstellen, die von einer anderen abgeleitet ist.
Erstelle ein neues Projekt und hole die das TsuZeichnen rein! Das heisst kopiere Dir das exe und die Klassen in das neue Verzeichnis. Füge die Klassendateien in das Projekt ein.
Erzeuge eine neue Klasse z.B. *RoteZeichenflaeche* und leite diese von der Klasse *Zeichenflaeche* ab.
Schreibe einen passenden Konstruktor, so dass die Zeichenfläche rot wird. Verhindere auch, dass die Farbe nachträglich noch geändert wird, indem du die Methode zum Farbe ändern redefinierst.

Tipp:

Du kannst eine Methode auch als *private* redefinieren! Wenn du die Methode zum Setzen der Farbe redefinierst kannst du darin die Methode der Basisklasse aufrufen. Hier ein kleines Beispiel:

```
class A
{
public:
    void Function();
};

// Klasse B von Klasse A
// ableiten
class B : public A
{
public:
    // Konstruktor
    B();

private:
    // Methode redefinieren
    // aber private machen!
    void Function();
};
```

```
// Methode der Klasse A
void A::Function()
{
    // tut nichts
}

// Konstruktor der Klasse B
B::B()
{
    // Funktion der Basisklasse aufrufen
    A::Function();
}

// redefinierte Methode der Klasse B
void B::Function()
{
    // Wir rufen nur die
    // Methode der Basisklasse auf
    A::Function();
}

int main()
{
    A einA;
    // kein Problem die Methode
    // für ein Objekt der Klasse
    // A aufzurufen
    einA.Function();

    B einB;
    // das lässt der Compiler nicht zu
    einB.Function();

    return 0;
}
```

Übung 2: Ein <<-operator für die Klasse Text

Leite jetzt von der Klasse Text ab ein neue Klasse ab. Überschreibe hier den <<-operator für verschiedene Datentypen, so dass man ein Objekt deiner neuen Klasse ähnlich wie das cout-Objekt verwenden kann.

Tipp

Als erstes brauchen wir eine Möglichkeit den string aus einem Text-Objekt abzufragen! Ergänze also in der Klasse Text im public-Teil folgenden Code:

```
// den Text abfragen
string holeText() const
{
    return m_text;
}
```

Gewünscht ist hier eigentlich, dass wir ein Text-Objekt verwenden können als wäre es ein ostream-Objekt. Wenn wir gross sind und besser C++ können werden wir in der Lage sein dieses Problem mit Ableitung von ostream zu lösen. Jetzt werden wir einfach eine neue Klasse *TextStream* von *Text* ableiten.

```
class TextStream : public Text
{
    // Rest kommt noch
};
```

Ein solches *TextStream*-Objekt soll aber genau gleich funktionieren wie ein Text-Objekt. Da das *Text*-Objekt einen Konstruktor mit Parametern hat, müssen wir in unserer abgeleiteten Klasse diesen Konstruktor in der Initialisierungsliste speziell aufrufen. Am besten ist es wir definieren in unserer Klasse *TextStream* einen Konstruktor wie in der Klasse *Text*. Dieser Konstruktor hat zwei Argumente nämlich die x- und die y-Position.

```
class TextStream : public Text
{
    public:
        // Konstruktor wie Klasse Text
        TextStream(long x, long y);

        // den Rest lasse ich jetzt noch
        // weg
};


// Hier der Code für den Konstruktor
TextStream::TextStream(long x, long y)
           :Text(x, y) // Hier Basisklassenkonstruktor aufrufen
{
}
```

Man muss als den Konstruktor der Basisklasse explizit aufrufen und zwar in der Initialisierungsliste. Denn der Konstruktor der Basisklasse (hier *Text)* muss immer vor dem Konstruktor der abgeleiteten Klasse (hier *TextStream*) aufgerufen werden.
Nun gilt es noch den operator-<< für verschiedene Datentypen zu überschreiben! Das sieht etwa so aus:

```
class TextStream : public Text
{
    public:
        // Konstruktor wie Klasse Text
        TextStream(long x, long y);

        TextStream& operator<<(long zahl);
};
```

Um die gewünschte Funktionalität zu erreichen brauchen wir eine Klasse, die sich ähnlich verhält wie das cout-Objekt. Es soll nämlich ein long in einen Text umwandeln.

```
int zahl = 55;
cout << zahl;
```

Ausgegeben wird nicht die interne Repräsentation eines int, der ja 32 bit braucht, sondern es werden die ASCII-Zeichen ‚5' ‚5' ausgegeben, was dem Wert 53 entspricht. Es findet also eine Umwandlung in ASCII-Zeichen um. Wir brauchen also ein Objekt, das sich verhält wie das cout-Objekt, aber die Umwandlung nicht ausgibt, sondern wieder in einen string umwandeln kann. Eine solche Klasse gibt es, sie ist wie das cout-Objekt ein *std::ostream*, indem es von der Klasse *std::ostream* ableitet. Die Klasse heisst *std::stringstream*. Hier also der operator für den long:

```
#include <sstream> // für die Klasse stringstream

using namespace std;

// und hier die Implementation
TextStream& TextStream::operator<<(long zahl)
{
    // Objekt der Klasse stringstream erstellen
    stringstream stream;
    // den bereits bestehenden Text übernehmen
    string alterText = holeText();
    stream << alterText;
    // die Zahl hinzufügen
    stream << zahl;
    // jetzt diesen Text wieder setzen,
    // so dass er auf dem Bildschirm erscheint
    string neuerText = stream.str();
    // Methode der Basisklasse aufrufen
    setzeText(neuerText);

    // eine Referenz auf uns selber
    // zurückgeben
    return *this;
}
```

Wichtige Änderung an der Klasse Text

Damit der Code oben kompiliert ist es wichtig, dass du in der Klasse *Text* die Methode *holeText* ergänzt!
Definiere nun den operator<< für andere Datentypen wie *char, const char*, const string&, double, float,* etc.