Weitere nützliche Dinge im Zusammenhang mit Klassen

Nur Teile explizit in der Vorlesung behandelt
-- zum Selbststudium --

Konstante Methoden und Objekte

Elementfunktionen, die den Zustand eines Objektes nicht verändern, sollten **unbedingt** mit dem Attribut **const** versehen werden, wie im folgenden Code die Methode **f:**

```
class X {
    . . .
    void f(...) const;
    . . .
};
```

Das Attribut **const** gibt dem Benutzer eine Zusicherung, deren Einhaltung -- ganz im Sinne der statischen Typisierung -- zur Compilezeit überprüft wird.
Dieses sollte genutzt werden.

Das Schlüsselwort mutable

Für ein konstantes Objekt können dann nur konstante Elementfunktionen aufgerufen werden, das wird vom Compiler überprüft!
Nun sind Fälle denkbar, in denen das Objekt zwar im *Wesentlichen* konstant ist, wobei es dann aber trotzdem möglich sein soll, z.B. eine interne *Hilfsvariable* zu verändern.

```
class X {
    double table[max];
    mutable int pos;
  public:
    X() { "table mit Zufallszahlen fuellen"; pos = 0; }
    double get() const { return table[pos++]; }
    void init() { "table mit Zufallszahlen fuellen"; pos = 0; }
};
```

Das Schlüsselwort mutable (2)

```
class X {
      double table[max];
      mutable int pos;
   public:
      X() {
         "table ... fuellen"; pos = 0;
      }
      double get() const {
          if (pos < max)
                return table[pos++];
          else {pos=0;
                 return table[max-1];}
      }
    void init() {
       "table ... fuellen"; pos = 0;
    }
};
```

```
const X x;     // konstantes Objekt
. . .
cout << x.get();  // richtig
. . .
x.init();         // Fehler
. . .
X y;              // nicht konstantes
                  // Objekt
. . .
cout << y.get();  // richtig
. . .
y.init();         // richtig
```

Statische Klassenelemente

Statische Klassenelemente

Statische Daten in Klassen entsprechen im Prinzip globalen Variablen unter Verwendung der **Kapselungseigenschaft.**

Sie existieren unabhängig davon, wie oft eine Klasse instanziiert wurde, genau einmal.

Auf statische Elemente einer Klasse kann jederzeit zugegriffen werden, unabhängig davon, ob schon Instanzen der Klasse existieren oder nicht.

Daneben gibt es auch Memberfunktionen (Methoden), die statisch definiert werden können. Diese kennen aber kein aktuelles Objekt. Daher kann in diesen nur auf statische Klassenelemente zugegriffen werden. Damit können sie auch nicht const sein.

Aufgabe zu static-Elementen für Matrix-Klasse

Erweitern Sie die Matrix-Klasse, sodass die Anzahl der vorhandenen Matrix-Objekte gezählt wird.

Im Konstruktor wird bei Erzeugung einer Matrix jeweils die Anzahl der nun vorhandenen Matrix-Objekte inkrementiert, im Destruktor dekrementiert.

Außerdem soll man eine Methode *GetAnzMatrizen* aufrufen können, die die Anzahl der aktuell vorhandenen Matrizen liefert.
Beim Programmstart und -ende sollte diese Methode 0 zurückliefern, da andernfalls Speicherlecks vorhanden wären.

Aufgabe zu static-Elementen

Erweitern Sie die Vorgangs-Klasse, sodass die Anzahl der vorhandenen Vorgangs-Objekte gezählt wird.

Im Konstruktor wird bei Erzeugung eines Vorgangs jeweils die Anzahl der nun vorhandenen Vorgangs-Objekte inkrementiert, im Destruktor dekrementiert.

Außerdem soll man eine Methode *GetAnzVorgaenge* aufrufen können, die die Anzahl der aktuell vorhandenen Vorgänge liefert.
Beim Programmstart sollte diese Methode idealerweise 0 zurückliefern, da andernfalls Speicherlecks vorhanden sind.

Lösung der Aufgabe zu static-Elementen (1)

Datei: Vorgang.h

```
class Vorgang {
 public:
  Vorgang(double d);   ~Vorgang();
  double getDauer() const;
  double getFruehAnf() const;   double getSpaetEnd() const;
  static int getAnzVorgaenge();
 private:
  void setFruehAnf(double fa);   void setSpaetEnd(double se);
  int getId() const;    void setId(int id);
  double dauer,fruehanf,spaetend; int id;
//! Anzahl der aktuell aktiven Vorgangs-Objekte
   static int anzVorgaenge;

 friend class Netz;
};
```

Lösung der Aufgabe zu Vorgangs-Elementen (2)

Datei: Vorgang.cpp

```
#include "Vorgang.h"

// Definition und Initialisierung der Vorgangsanzahl
int Vorgang::anzVorgaenge = 0;

Vorgang::Vorgang(double d) {
   dauer = d; id = -1;  ++anzVorgaenge;}
}
Vorgang::~Vorgang() {--anzVorgaenge;  }

int Vorgang::getAnzVorgaenge() {   return anzVorgaenge; }
```

Anwendung:

```
. . .
Vorgang v1(10), v2(4), v3(9.6), v4(21);
cout << "Vorgänge: " << Vorgang::getAnzVorgaenge() << endl;
cout << "Vorgaenge: " << v4. getAnzVorgaenge() << endl;
```

Übung zum Kopierkonstruktor

```
class A {

  private:
    char* name;
    int len1;

  public:
    A(const char* nam="");
    ~A();
    A(const A& a);
    A& operator= (const A& a);
    void Print(ostream& str) const;
    friend bool operator==(const A& a1, const A& a2);
};
```

Implementieren Sie zur Schnittstelle der folgenden Klasse die Source-Datei, d.h.

- den Konstruktor A::A
- den Destruktor A::~A
- den Kopierkonstruktor A::A(const A&)
- den Zuweisungsoperator A::operator=
- den Vergleichoperator ==

Ein Vorschlag für Namenskonventionen

1. Präfix:

Globale Variablen beginnen mit einem "g":

```
int g_zahl;
```

Funktionsargumente beginnen mit "a"

```
funk(double ad_durch,  const int ai_wert)
```

Member (Elemente) in Klassen mit "m"

```
class Mitarbeiter {...
double m_umsatz;
}
```

Lokale Variablen erhalten keinen ersten Präfix.

Ein Vorschlag für Namenskonventionen (2)

Der zweite Präfix gibt an, ob es sich um eine Referenz (r) oder um einen Zeiger (p) handelt.

```
int * p_anzahl;
funk( double & ard_durch, int * apn_zahlen)
```

Der dritte Präfix legt den Typ fest:

- i für int, s für short
- n für einen ganzzahligen Typ
- d für double, f für float
- c für eine Klasseninstanz
- t für einen Typ, der durch typedef definiert wurde
- ...

```
funk (Stack & arc_stack,  float *apf_summe)
int i_anzahl;
```

C++-11

constexpr / const

const und constexpr legen beides Konstanten fest, die nie rechts vom Gleichheitszeichen auftauchen dürfen.
constexpr können allerdings schon zur Compilezeit ausgewertet werden
constexpr wird erst ab Visual Studio 2015 unterstützt.

```
class  CVec {
public:
   CVec() : si(0) {}
   constexpr CVec(int s) : si(s) {
      if (si < 0) {
        cout << "si (" << si << ") < 0 ";
        exit(-1);}}
private:
   int si;
};
```

```
int main() {
/* mit constexpr sollte es zur
Compilezeit-Fehlermeldung geben */
constexpr CVec v(-1);
constexpr CVec v(1); // geht
}
```

Im Visual Studio Debugger können Sie beim Debuggen eines nicht optimierten Debugbuilds feststellen, ob eine `constexpr` Funktion zur Kompilierungszeit ausgewertet wird, indem Sie einen Haltepunkt darin einfügen. Wenn der Haltepunkt erreicht wird, wurde die Funktion zur Laufzeit aufgerufen. Wenn dies nicht der Fall ist, wurde die Funktion zum Zeitpunkt der Kompilierung aufgerufen.

C++-11

constexpr / const

const und constexpr legen beides Konstanten fest, die nie rechts vom Gleichheitszeichen auftauchen dürfen.
constexpr können allerdings schon zur Compilezeit ausgewertet werden
constexpr wird erst ab Visual Studio 2015 unterstützt.

```
#if defined(_defined_MSC_VER) > 1800
constexpr
#endif
int factorial(int n) {
    return n <= 1 ? 1 :
           (n * factorial(n - 1));
}
```

```
int main() {
constexpr int fact3 = factorial(3);
cout << fact3 << endl;
}
```

fact3 wird ab Visual Studio 2015 also bereits zur Compile-Zeit mit 6 belegt, während die Berechnung in früheren Versionen erst zur Laufzeit erfolgt.
Allerdings muss vor fact3 auch constexpr stehen