Kurseinheit 4: Klassen 3

1. Beziehungen zwischen Klassen
2. Assoziation / Aggregation
3. Komposition
4. Speichermanagement

1. Beziehungen zwischen Klassen

Möglichkeiten der Kommunikation zwischen Objekten

Klassen, bzw. deren Objekte, müssen teilweise miteinander in einem Softwaresystem kommunizieren. Dabei gibt es grundsätzlich zwei Möglichkeiten einer Kommunikation:

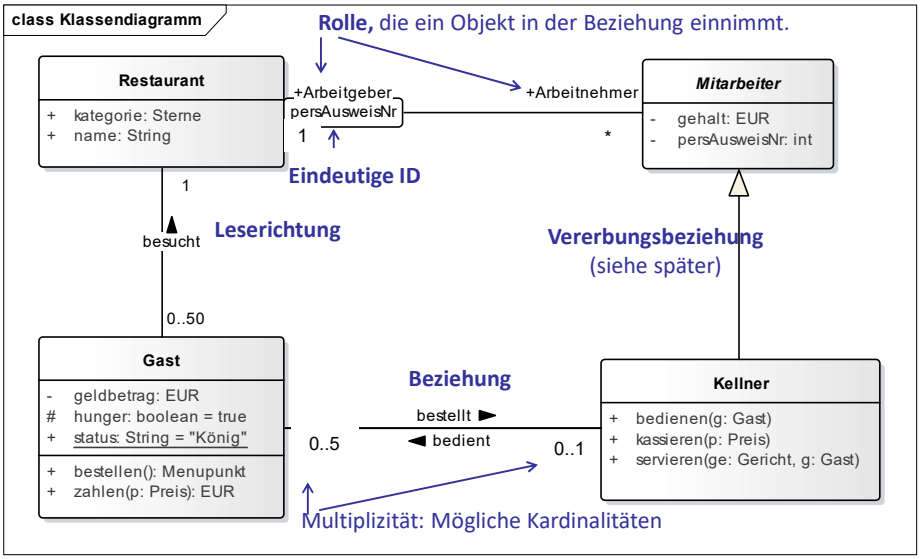
- **Synchrone Kommunikation**
- **Asynchrone Kommunikation**

Bei der **asynchronen Kommunikation** werden Nachrichten verschickt. Der Sender arbeitet nach dem Versenden der Nachrichten im Programmcode weiter. “Irgendwann” erhält der Empfänger die Nachricht und bearbeitet diese. Dies geschieht oft in einem Multitasking- oder Multithreading-Umfeld. Als hinterlegte Infrastruktur sind Messaging Systeme zu verstehen. Beispiele: Queues bei Echtzeitsystemen, Signal/Slot-Prinzip in Qt, Message Queues im Windows-Betriebssystem. Diese Art der Kommunikation wird im Rahmen dieser Lehrveranstaltung nicht behandelt.

Die **synchrone Kommunikation** geschieht über Methodenaufrufe (wie gewohnt aus C). Der Sender bleibt solange “stehen”, bis die aufgerufene Methode abgearbeitet ist und ein return erfolgt. Damit eine synchrone Kommunikation über Objektgrenzen hinweg erfolgen kann, müssen sich die Objekte kennen. Dies wird auch im Klassendiagramm abgebildet. Im Rahmen dieser LV wird nur die synchrone Kommunikation über Methodenaufrufe vertieft.

1. Beziehungen zwischen Klassen

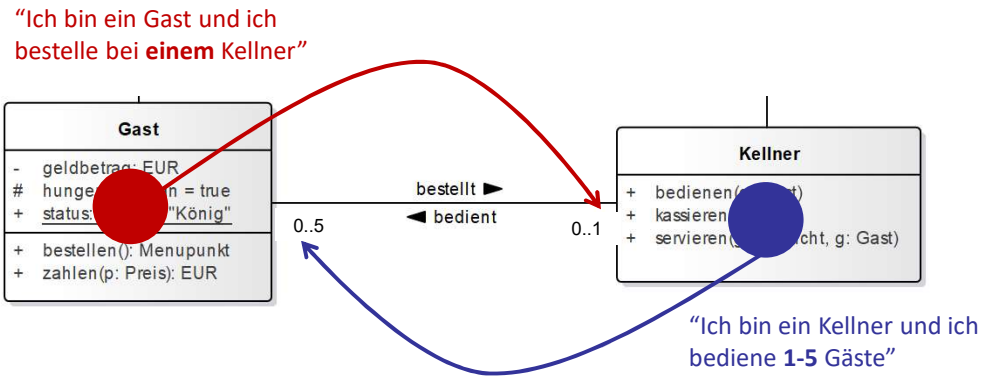
Beispiel UML-Klassendiagramm



Quelle: Kecher, Hoffmann-Elbern et. al: UML 2.5: Das umfassende Handbuch, Rheinwerk, 7. Auflage, 2021

1. Beziehungen zwischen Klassen

Bestimmung der Multiplizitäten



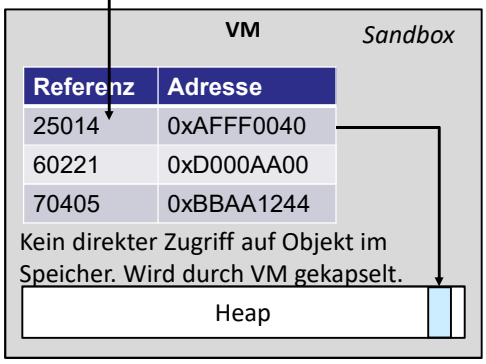
Die Multiplizitäten müssen für die gesamte Laufzeit des Programms gültig sein. Selbst bei einfachen Beispielen gibt es schnell Fragestellungen/Inkonsistenzen:

- Gast wird noch nicht bedient (kein Kellner zu geordnet) oder Gaststätte leer
- Gast bestellt Getränke bei Kellner A und Essen bei Kellner B
- Restaurant ist voll: Kellner bedient mehr als 5 Gäste

Unterschiede C#/Java versus C++: Zugriff auf ein Objekt

In C#/Java wird **nur** über Referenzen auf Objekte zugegriffen.

```
Car car = new Car();
car.SetSpeed(100U);
```



In C++ gibt es **drei** Möglichkeiten:

- Objekt (wie Attribut) – Car2
- Referenz (wie in Java/C#) – rCar2
- Zeiger – pCar2

```
Car Car2{};
Car& rCar2 = Car2;
Car* pCar2 = &Car2;
Car& rCar2b = *pCar2;
```

Der Zugriff auf das Objekt geschieht wie folgt:

```
Car2.vSetMaxSpeed(111U);
rCar2.vSetMaxSpeed(112U);
pCar2->vSetMaxSpeed(300U);
```

Wichtige Gleichung ;-)

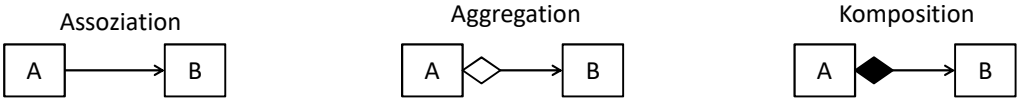
$$\frac{\text{Schwierigkeitsgrad C++}}{\text{Schwierigkeitsgrad Java}} = 3$$

Klassifizierung von Beziehungen

Art der Beziehung

- ***Assoziation***: Allgemeine Beziehung
- ***Aggregation***: Teile-Ganze-Beziehung
- ***Komposition:*** Strengere Teile-Ganzes-Beziehung

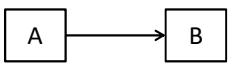
} Wird in C++ auf die gleiche Art und Weise umgesetzt



Teile-Ganze-Beziehung (Rautesymbol)

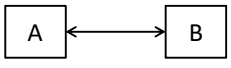
Richtung der Beziehung

- ***Unidirektional***

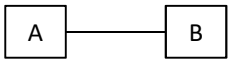


Ein A kennt ein B, aber das B kennt sein A nicht.

- ***Bidirektional***



äquivalent



Anzahl der Klassen in einer Beziehung: rekursiv (unär), binär, tertiär, …

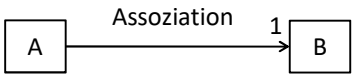
1. Beziehungen zwischen Klassen

Beispiel: Unidirektionale binäre Assoziation

Ein A kennt genau ein B.

↓

Frage: Wie lässt sich dies in C++ abbilden?




```
class A
{
public:
  A(); // To Do
  void vCallMe(C*);
private:
   B* pMyB_;
   //…
};
```

```
class B
{
   //…
};
```

```
class B
{
   //…
};
```

Nur wenn die Beziehung dauerhaft über ein Attribut besteht (hier pMyB_), wird eine Assoziation gezeichnet (A->B und nicht von A->C).

Eine **„zu eins"-Abbildung** lässt sich in C++ mittels Zeiger oder Referenz abbilden. Da Referenzen aber nie Null sein können, müsste diese in der Initialisierungsliste initialisiert werden (nicht im Rumpf des Konstruktors).
Eine **„zu many"-Abbildung** lässt sich in C++ bei kleiner Anzahl durch ein Array abbilden. Dies ist allerdings nicht zu empfehlen (Operationen: Einfügen, Löschen, …). Hierzu werden später die Collection-Klassen aus der Standard-Template-Library verwendet.

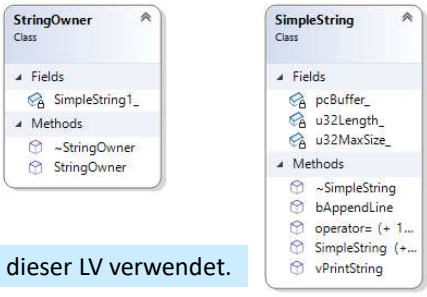
1. Beziehungen zwischen Klassen

Class Designer in Microsoft Visual Studio

In MSVS kann der Class Designer mitinstalliert werden. Auch eine Nachinstallation über den Installer ist selbstverständlich möglich. Allerdings unterscheidet der Class Designer nicht zwischen Assoziation, Aggregation und Komposition.
Projekt -> Add -> New Item -> Utility -> Class Diagramm .. h-Dateien mit Drag&Drop einfügen

Beispiel:
Klassische UML-Darstellung einer Komposition

Darstellung im Class Designer:
Der Class Designer kann **keine** Komposition darstellen. Aggregationen und Assoziationen haben darin die gleiche Darstellung.



Im vorliegenden Fall lässt sich durch geschickte Namenswahl (SimpleString1_) erahnen, dass es sich um Objekt SimpleString in StringOwner handelt (Komposition).

Trotz dieser **Einschränkung** wird der Class Designer in dieser LV verwendet.

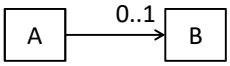
2. Assoziation / Aggregation

Implementierung einer „zu eins“-Assoziation

```
#include "B.h"

class A
{
public:
   A(B* pB);

private:
   B* pMyB_;
};
```

```
pMyB_->DoIt(); //in A
```

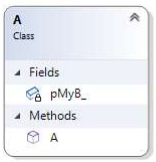
```
class B
{
public:
   void DoIt(void);
};
```



Pointer pMyB_ könnte auch nullptr sein.

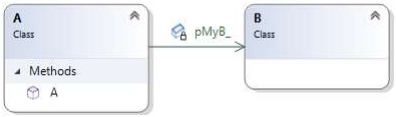
```
B B1;
A A1(&B1);
// A A2; //Error: no default constructor

// B1 does not know A1
```

Assoziation wird als Attribut (Field) angezeigt



Rechte Maus auf pMyB_, Show as Association

Assoziation wird als Pfeil angezeigt



2. Assoziation / Aggregation

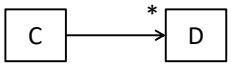
Implementierung einer „zu many“-Assoziation

```
#include "D.h"
#include <list>
class C
{
public:
   C(std::list<D*> DList);
   void vCallAllD(void);

private:
   std::list<D*> DList_;
};
```

```
void C::vCallAllD(void)
{
   for(auto ait : DList_)
   {
      ait->DoIt();
   }
}
```

auto ranged loop mit Collectionklasse list!

```
class D
{
public:
   void DoIt();
};
```



Multiplizität * bedeutet 0 bis many

```
std::list<D*> localDList;
D D1;
D D2;
D D3;
localDList.push_back(&D1);
localDList.push_back(&D2);
localDList.push_back(&D3);
C C1(localDList);
C1.vCallAllD();
```

„zu many“-Assoziation lässt sich einfach mit einer Collectionklasse abbilden. Details über die Unterschiede list, set, vector, … später.

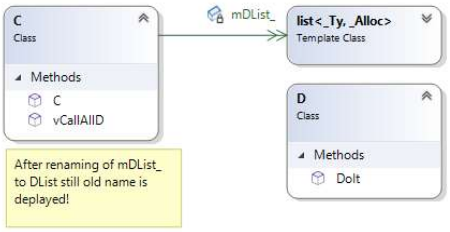
2. Assoziation / Aggregation

Implementierung einer „zu many“-Assoziation

Assoziation wird als Attribut (Field) angezeigt

Geschickte Wahl des Attributs (Field) lässt eine Liste von D erahnen.

Assoziation wird als Pfeil angezeigt



Im Gegensatz zur „zu eins“-Assoziation wird mit dem Doppelpfeil oben die „zu many“-Assoziation angedeutet.
Der Class Designer kann die Beziehung zu D nicht auflösen. Es ist nur eine Liste (Templateklasse). In den Abbildungen sollen daher immer die entsprechende Klasse darunter angezeigt werden.

3. Komposition

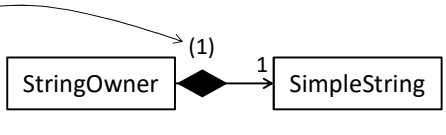
Definition und Beispiel

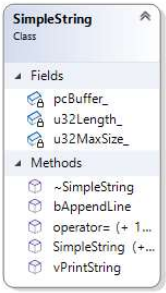
Bei einer Komposition handelt es sich um ein Teile-Ganzes-Beziehung. Eine Komposition stellt eine strengere Art der Teile-Ganzes-Beziehung dar (im Gegensatz zur Aggregation):

Der Unterschied zur Aggregation ist, dass ein Objekt, das als Ganzes Teile enthält, für die Existenz der Teile verantwortlich ist. https://de.wikipedia.org/wiki/Assoziation_(UML)#Komposition

Ein Teil kann nur einem Ganzen zugeordnet werden.

Die (1) kann somit auch weggelassen werden.





Wird das Ganze gelöscht, müssen auch die Teile gelöscht werden.
Dies geschieht im Beispiel automatisch, da SimpleString1_ als Variable innerhalb von StringOwner enthalten ist.

Beispiel StringOwner

```
class StringOwner
{
public:
   StringOwner(const char* cpc);
   ~StringOwner();

private:
   SimpleString SimpleString1_;
};
```

Object SimpleString1_ ist Attribut von StringOwner.
Dies ist keine Referenz!

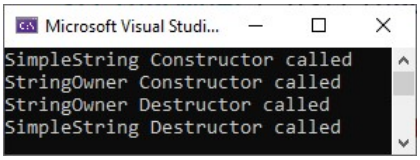
```
StringOwner::StringOwner(const char* cpc) : SimpleString1_{ 20U }
{
   if (SimpleString1_.bAppendLine(cpc) == false)
   {
      throw std::runtime_error("Not enough Memory!");
   }
   std::cout << "StringOwner Constructor called" << std::endl;
}
StringOwner::~StringOwner()
{
   std::cout << "StringOwner Destructor called" << std::endl;
}
```

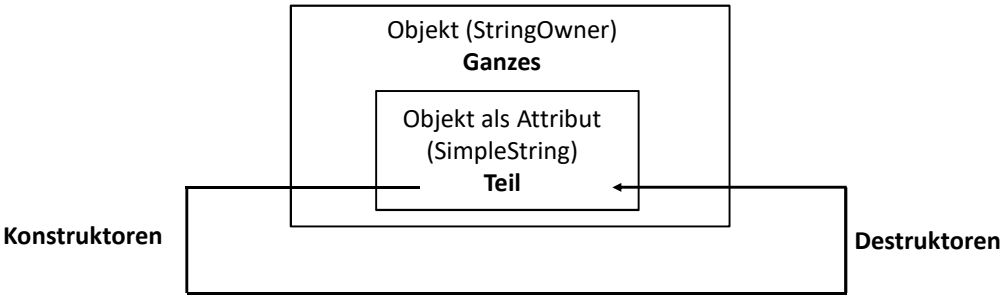
Brace yourself!

Reihenfolge Konstruktor / Destruktor

```
StringOwner StringOwner1{ "Hi Students" };
```

StringOwner enthält SimpleString Objekt (Ganzes-Teil).





Zuerst werden die Objekte als Attribute (Teil) erstellt (Konstruktor) – danach das Ganzes.
Beim Freigeben (Destruktor) ergibt sich umgekehrte Reihenfolge.

Allokation von Objekten

Statische Allokation

```
Car Car1{};
```

Möglichkeiten statischer Allokation
- Global (DS)
- Modulglobal (DS)
- Statisch in Methode (DS)
- Lokal (SS)
- Übergabeparameter (SS)

Sonderfall: Attribut einer Klasse

Allokation dann vor dem Ganzen. Speicherort abhängig vom Ganzen.

```
Car& rCar1 = Car1;
Car& rCar2 = *pCar2;
```

Mit Referenzen kann auf Objekte zugegriffen werden.

Dynamische Allokation

```
Car* pCar2 = new Car{};
```

Möglichkeiten dynamischer Allokation
- Mit new-Operator (HS)

Statische Allokation

Statische Allokation

```
Car Car1{};
```

Konstruktor: Globale, modulglobale und statische Objekte innerhalb einer Funktion werden von main.cpp aus initialisiert (Konstruktoraufrufe). Lokale Objekte und übergebene Objekte werden am Funktionsaufruf oder vor dem Funktionsaufruf initialisiert (Konstruktor):

Sobald ein Objekt seine Gültigkeit verliert, wird **automatisch** der **Destruktor** aufgerufen.

- Global (DS)
- Modulglobal (DS)
- Statisch in Methode/Funktion (DS)

} Ende des Programms

- Lokal (SS)
- Übergabeparameter (SS)

} Ende der Funktion

Hier ist nichts Wichtiges zu beachten. Die Klassen müssen sich nur an **RAII** halten. Dynamische allokierte Objekte innerhalb dieser Objekte müssen dort selbst freigegeben werden. Destruktoren von Objekten als Attribute einer Klassen werden automatisch aufgerufen.

Statische Allokation - Beispiel: Einfache Car-Klasse

```
class Car
{
public:
Car(std::string);
   Car(Car& rCar);
   Car(Car&&) = delete;
   ~Car();

   Car& operator=(const Car&) = delete;
   Car& operator=(Car&&)      = delete;

private:
   std::string strName_;
};
```

Move-Konstruktor deleted

Assign-Operator deleted
Move-Assign-Operator deleted

Attribut strName_ soll ein Objekt eindeutig identifizieren.

Hier müsste im folgenden Beispiel z.B. „OG-MM-42“ oder „WOL-FI-73“ stehen (Objekte). Es soll aber vereinfachend der Herstellertyp verwendet werden (nicht wirklich eindeutig…).

Statische Allokation - Beispiel: Einfache Car-Klasse

```
Car::Car(std::string str ) : strName_{ str }
{
   std::cout << "Contstructor: Car " << strName_ << " contructed" << std::endl;
}


Car::Car(Car& rCar)
{
   strName_ = rCar.strName_;
   std::cout << "Copy-Constructor: Car " << strName_ << " contructed" << std::endl;
}


Car::~Car()
{
   std::cout << "Car object with name " << strName_ << " destructed" << std::endl;
}
```

Brace yourself!

Statische Allokation - Beispiel: Einfache Car-Klasse

```
Car Car1{ "Audi" };
static Car Car2{ "Opel" };

int main(void)
{
   std::cout << "---main entered-----
   static Car Car3{ "Seat" };
   Car Car4{ "Renault" };

   std::cout << "---Before foo call---
   foo(Car4);
   std::cout << "---After foo call----

   return 0;
}
void foo(Car Car5)
{
   Car Car6{ "Tesla" };
   std::cout << "---Before foo end-----------------" << std::endl;
}
```

Copy-Constructor: Zwei Objekte mit Namen „Renault“

```
Microsoft Visual Studio Debug Co...
Constructor: Car Audi contructed
Constructor: Car Opel contructed
---main entered-----------------
Constructor: Car Seat contructed
Constructor: Car Renault contructed
---Before foo call---------------
Copy-Constructor: Car Renault contructed
Constructor: Car Tesla contructed
---Before foo end-----------------
Car object with name Tesla destructed
Car object with name Renault destructed
---After foo call----------------
Car object with name Renault destructed
Car object with name Seat destructed
Car object with name Opel destructed
Car object with name Audi destructed
```


Dynamische Allokation

Bei der dynamischen Allokation ist zu berücksichtigen:

Es gibt (noch) **keinen** Garbage Collector (dt. Müll) in C++, welcher nicht benötigte Objekte löscht! Der C++-Programmierer muss sich darum selbst kümmern! Sonst: Memory Leaks wie bei malloc.

Java/C# enthält einen Garbage Collector in seiner VM:

VM

	referenziert auf				
Ref	R1	R2	R3	R4	R5
R1		x			
R2			x		
R3					
R4					
R5					x

Der Garbage Collector (GC) ist ein Thread innerhalb der VM, der nicht referenzierte Objekte (interne Tabelle) findet und diese dann löscht.

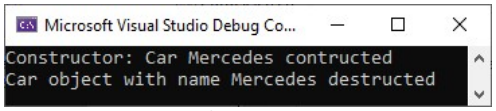
Sonderrolle: R1 sei main-Funktion!

- Nichtreferenzierte Objekte ☐
- Zyklen (Rx->Ry->Rx)
- Selbstreferenzierung (Rx->Rx) ☐

GC wird irgendwann aufgerufen und hat eine gewisse Laufzeit: Daher sind Java/C# nicht echtzeitfähig.

Dynamische Allokation – Beispiel: Einfache Car-Klasse

```
void goo(void)
{
   Car* pCar7 = new Car("Mercedes");
   // Do some work!
   delete pCar7;
}
```

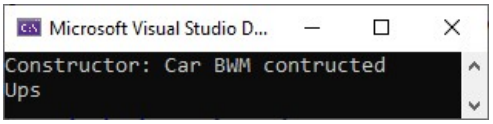


Empfehlung: new / delete im gleichen „Scope“!

```
void hoo(void)
{
   try
   {
      Car* pCar8 = new Car("BWM");
      // Do some work!
      ioo();
      delete pCar8;
   }
   catch (std::exception ex)
   {
      std::cout << ex.what() << std::endl;
   }
}
```

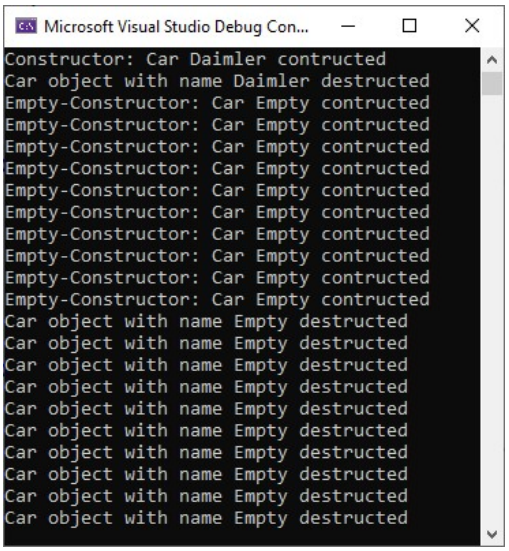
Bei Exceptions wird oft vergessen, Objekte wieder zu löschen! Hier verliert man schnell den Überblick. **Memory Leaks entstehen.**

```
void ioo(void)
{
   throw std::exception("Ups");
}
```



Dynamische Allokation – Ein Objekt und mehrere Objekte

```
void joo(void)
{
   Car* pCar9 = new Car("Daimler");
   delete pCar9;

   Car* pCar10X = new Car[10];
   // delete pCar10X[]; // Error
   // delete pCar10X;   // Assertion
   delete[] pCar10X;
}
```



Beim zweiten new wird ein leerer Konstruktor benötigt.
Nicht ganz intuitiv ist die Freigabe eines Arrays von Objekten.

Dynamische Allokation – Zweidimensionales int32_t Array

```
void koo(void)
{
   // Allocation
   int32_t** pps32 = new int32_t* [100];
   for (int32_t s32C = 0; s32C < 100; s32C++)
   {
      pps32[s32C] = new int32_t[100];
   }

   // Use
   pps32[50][49] = 73;

   // Deallocation - inversed order!!!
   for (int32_t s32C = 0; s32C < 100; s32C++)
   {
      delete[] pps32[s32C];
   }
   delete[] pps32;
}
```

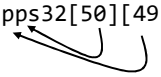
Ähnlich wie in C mit malloc/free.

Coding Styleguide hier wieder nützlich:

pps32[s32C]

Klammer(Index-Operator) entfernt ein p. Es handelt sich somit um einen int32_t-Zeiger.

pps32[50][49]



Zwei Index-Operatoren entfernen zwei p. Es handelt sich somit um einen int32_t-Wert.

Auffinden von Memory Leaks

MS Visual Studio bietet eine Möglichkeit Memory Leaks zu finden:

```
#include <crtdbg.h>
```

```
_CrtDumpMemoryLeaks();
```

z.B. am Ende von main: Zeigt zu diesem Zeitpunkt Memory Leaks an. Besser: vTestCreateObjects (Übung 3) verwenden.

```
void hoo(void)
{
   try
   {
     Car* pCar8 = new Car("BWM");
      // Do some work!
      ioo();
   }
   catch (std::exception ex)
   {
      std::cout << ex.what() << std::endl;
   }
}
void ioo(void)
{
   throw std::exception("Ups");
}
```

Zwei Objekte!
8-Bytes und 28 Bytes!
Car-Objekt und enthaltenes String-Objekt.

Output-Fenster von MSVS:

```
Detected memory leaks!
Dumping objects ->
{157} normal block at 0x011DE228, 8 bytes long.
 Data: <        > D8 9B 1D 01 00 00 00 00
{155} normal block at 0x011D9BD8, 28 bytes long.
 Data: <(   BWM         > 28 E2 1D 01 42 57 4D 00 1C 00 00 00 D8 9B 1D 01
Object dump complete.
```