Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 8

8. Einige Klassen für spezielle Design-Zwecke

Smart-Pointer in C++ (1)

- **Eigenschaften** :

  ◇ Smart-Pointer sind **Objekte**, die **wie Pointer verwendet** werden können (→ "intelligente" Pointer).
  D. h. sie **referieren** ein **anderes Objekt**.

  ◇ Gegenüber "normalen" Pointern besitzen sie **zusätzliche Fähigkeiten**.
  Beispiele :

  ▻ **Durchführung besonderer Aktionen beim Zugriff** zu dem von Ihnen referierten Objekt
  (z.B. Laden eines Objekts vom Hintergrundspeicher, falls es sich bei einem Zugriff zu ihm noch nicht im Arbeitsspeicher befindet, automatisches Abspeichern des Objekts beim Zerstören des referierenden Auto-Pointer-Objekts)

  ▻ **Automatische Zerstörung** eines **dynamisch allozierten Objekts**, wenn der referierende Pointer vernichtet wird.
  Die explizite Zerstörung des Objekts mittels `delete` ist nicht erforderlich (und auch nicht mehr sinnvoll).
  Es wird auch zerstört, wenn der Gültigkeitsbereich des Pointers anormal verlassen wird (Werfen einer Exception).
  → Realisierung einer einfachen *Garbage Collection* (**Auto-Pointer**)
  (z.B. Klassen-Template **`auto_ptr`** der ANSI-C++-Standardbibliothek)

- **Implementierung** :

  ◇ Smart-Pointer enthalten einen **Pointer** auf das von ihnen **referierte Objekt** als **Datenkomponente**.
  Diese Datenkomponente wird vom Konstruktor initialisiert – in vielen Fällen mit einem diesem hierfür übergebenen Parameter. Dies erlaubt die gleiche Syntax bei der initialisierenden Erzeugung eines Smart-Pointer-Objekts wie bei einer gewöhnlichen Pointer-Variablen.

  ◇ Die **automatische Zerstörung** des **referierten Objekts** wird durch einen entsprechenden **`delete`**-Ausdruck im **Destruktor des Smart-Pointer**-Objekts realisiert.
  Analog lässt sich das **automatische Abspeichern** des **referierten Objekts** ebenfalls im Destruktor des Auto-Pointer-Objekts realisieren.

  ◇ Um für unterschiedliche Objekt-Typen verwendbar zu sein, sind Auto-Pointer-Klassen häufig als **Klassen-Templates** realisiert.

  ◇ **Beispiel** für einen einfachen Smart-Pointer mit Auto-Pointer-Funktionalität :

```
template<class T>
class AutoPtr
{ public :
    AutoPtr(T* ptr=NULL);
    ~AutoPtr();
    // ...
  private :
    T* m_pT;
};


template <class T> inline AutoPtr<T>::AutoPtr(T* ptr)
{ m_pT = ptr; }

template <class T> inline AutoPtr<T>::~AutoPtr()
{ delete m_pT; m_pT = NULL;}


void func(void)
{
  AutoPtr<double> apd = new double; // Auto-Pointer
  double* pd = new double;          // normaler Pointer
  // ...

  delete pd; // explizite Zerstörung der von pd referierten double-Variablen

}            // automat. Zerstörung der von apd referierten double-Variablen
```

Smart-Pointer in C++ (2)

- **Implementierung, Forts.** :

  ◇ Um mit Smart-Pointern weitgehend ähnlich wie mit normalen Pointern umgehen zu können, müssen i.a. **weitere Funktionen** für diese implementiert werden. Üblicherweise handelt es sich hierbei mindestens um die folgenden Funktionen :

  - ▻ Copy-Konstruktor
  - ▻ Operatorfunktion für Zuweisungs-Operator `(=)`
  - ▻ Operatorfunktion für dereferenzierenden Komponenten-Operator `(->)`
  - ▻ Operatorfunktion für Dereferenzierungs-Operator `(*)`

  ◇ Durch den Destruktor eines Smart-Pointers wird i.a. das referierte Objekt in irgendeiner Weise manipuliert (Bei einem Auto-Pointer wird es z. B. zerstört). Diese Manipulation darf für ein bestimmtes referiertes Objekt nur einmal erfolgen. Daraus folgt, dass es immer nur von einem einzigen Smart-Pointer "besessen" werden darf, der allein für die jeweilige Manipulation zuständig ist.
  Das bedeutet, dass durch den **Copy-Konstruktor** und den **Zuweisungsoperator**, der "Besitz" des referierten Objekts vom alten Smart-Pointer (rechte Seite) auf den neuen Smart-Pointer (linke Seite) übertragen werden muß.

  Achtung : Der **Parameter** des Copy-Konstruktors bzw. der Zuweisungs-Operatorfunktion darf hier nicht – wie sonst üblich – eine Referenz auf ein konstantes Objekt sondern muß eine **Referenz auf ein veränderliches** Objekt sein.

  ◇ Der **dereferenzierende Komponenten-Operator** muß als **Delegations-Operator** wirken und den als Datenkomponente gespeicherten "gewöhnlichen Pointer" auf das referierte Objekt zurückgeben.

  ◇ Der **Dereferenzierungs-Operator** muß das referierte Objekt (als Referenz) zurückliefern

  ◇ Wenn die für gewöhnliche Pointer `p` geltende Beziehung **`p->m == (*p).m == p[0].m`** auch für Smart-Pointer gelten soll, muß zusätzlich der **Index-Operator** `([])` in geeigneter Weise überladen werden.

  ◇ Soll auch **Pointer-Arithmetik** wie für gewöhnliche Pointer möglich sein, müssen auch die **hierfür verwendeten Operatoren** geeignet überladen werden.

- **Probleme** :

  ◇ Auch bei Implementierung aller o.a. benötigten Memberfunktionen existieren Einschränkungen und Probleme bezüglich der Verwendung von Smart-Pointern.
  → Smart-Pointer lassen sich nicht 100%ig genauso wie gewöhnliche Pointer verwenden.

  ◇ Smart-Pointer-Parameter sollten grundsätzlich per **Referenz übergeben** werden.
  Andernfalls würde durch den bei Wertübergabe aufgerufenen Copy-Konstruktor der Besitz des ursprünglichen Smart-Pointers (=aktuellen Parameter) an dem von ihm referierten Objekt verloren gehen.

  ◇ I.a. kann eine zur Referierung von Einzelobjekten geeignete Smart-Pointer-Klasse nicht auch zur Referierung von **Objekt-Arrays** eingesetzt werden.
  Zur Referierung von Objekt-Arrays wird vielmehr eine eigene Smart-Pointer-Klasse benötigt.
  Im Fall eines Auto-Pointers z.B. muß dessen Destruktor bei der Referierung eines Einzelobjekts den Operator `delete` und bei der Referierung eines Arrays den Operator `delete[]` aufrufen.

  ◇ I.a. darf ein Auto-Pointer nicht zur Referierung von **statisch** (lokal oder global) **allozierten Objekten** verwendet werden. Sein Destruktor würde versuchen, ein derartiges Objekt mittels `delete` freizugeben, was unzulässig ist.

  ◇ Zur Behebung der verschiedenen Anwendungsprobleme sind diverse Lösungen realisiert worden, die aber zu teilweise sehr komplexen und relativ aufwendigen Smart-Pointer-Implementierungen führen.

Einfaches Beispiel eines Auto-Pointer-Klassen-Templates (1)

```
// C++-Headerdatei autoptr.h
// Definition des Klassen-Templates AutoPtr
// Beispiel für eine Smart-Pointer-Klasse

#ifndef AUTO_PTR_H
#define AUTO_PTR_H

#include <cstdlib>

template<class T>
class AutoPtr
{
  public :
    AutoPtr(T* ptr=NULL) { m_pT = ptr; }
    ~AutoPtr()           { delete m_pT; m_pT = NULL;}
    AutoPtr(AutoPtr& ap);
    AutoPtr<T>& operator=(AutoPtr& ap);
    T* operator->();
    T& operator*();
  private :
    T* m_pT;
};


template <class T> AutoPtr<T>::AutoPtr(AutoPtr& ap)
{
  m_pT=ap.m_pT;     // neue Klasse wird "Besitzer" von *(ap.m_pT)
  ap.m_pT=NULL;
}


template <class T> AutoPtr<T>& AutoPtr<T>::operator=(AutoPtr& ap)
{
  if (this!=&ap)
  {
    delete m_pT;
    m_pT=ap.m_pT;   // Klasse "auf linker Seite" wird "Besitzer" von *(ap.m_pT)
    ap.m_pT=NULL;
  }
}


template <class T> T* AutoPtr<T>::operator->()
{
  if (m_pT!=NULL)
    return m_pT;
  else
    throw("Kein Objekt gebunden");
}


template <class T> T& AutoPtr<T>::operator*()
{
  if (m_pT!=NULL)
    return *m_pT;
  else
    throw("Kein Objekt gebunden");
}


#endif
```

Einfaches Beispiel eines Auto-Pointer-Klassen-Templates (2)

```
// C++-Headerdatei icks.h
// Definition einer einfachen Beispielklasse Icks
// zur Verwendung mit dem Klassen-Template AutoPtr<T>

#ifndef _ICKS_H
#define _ICKS_H

class Icks
{
  public :
    Icks(int x=0)      { m_iX=x; }
    int getVal()       { return m_iX; }
    void setVal(int x) { m_iX=x; }
  private :
    int m_iX;
};

#endif
```

```
// C++-Quelldatei autoptr_m.cpp
// einfaches Demonstrations-Beispiel zur Anwendung des Klassen-Templates AutoPtr<T>
// Implementierung des Moduls mit der Funktion main()

#include "autoptr.h"
#include "icks.h"

#include <iostream>
using namespace std;


void func(AutoPtr<Icks>& par)
{ cout << endl << "alter Wert Parameter par : " << par->getVal();
  par->setVal(9);
  cout << endl << "neuer Wert Parameter par : " << (*par).getVal() << endl;
  AutoPtr<Icks> apxf = par;
  cout << endl << "Wert lokale Var apxf in func : " << apxf->getVal();
}


int main(void)
{ AutoPtr<Icks> apxm = new Icks(5);
  try
  { func(apxm);
    cout << endl << "Wert lokale Var apxm in main : ";
    cout << apxm->getVal() << endl;
  }
  catch (char* ex)
  { cout << ex << endl;
  }
  return 0;
}
```

Ausgabe des Programms :

```
alter Wert Parameter par : 5
neuer Wert Parameter par : 9

Wert lokale Var apxf in func : 9
Wert lokale Var apxm in main : Kein Objekt gebunden
```

Interface-Klassen (1)

- **Aufgabe und Eigenschaften** :

  ◇ Eine Interface-Klasse (**Adapterklasse**) wird benötigt, wenn die **Schnittstelle** zur Verwendung einer Dienstleistung, die von einer anderen Klasse angeboten wird, an besondere Bedürfnisse **angepasst** werden soll.

  ◇ I.a. ist die Funktionalität einer Interface-Klasse relativ gering.
  Typischerweise stellt sie eine oder mehrere **Memberfunktionen** bereit, über die jeweils eine der **Memberfunktionen des eigentlichen Dienstleistungs-Objekts aufgerufen** werden kann (***forwarding functions***).
  Die Memberfunktionen der Interface-Klasse **kapseln** somit den Aufruf der eigentlichen Dienstleistungsfunktionen, wobei sie i.a. eine gegenüber diesen **modifizierte Aufruf-Schnittstelle** besitzen.

  ◇ Die **Modifikation der Aufruf-Schnittstelle** kann z.B. bestehen
  - ▻ in einer **Änderung** des **Funktionsnamens** (z.B. zur Vermeidung von Namenskonflikten)
  - ▻ in einer **Änderung** der Anzahl / Typen der **Funktionsparameter** und/oder des **Rückgabetyps**
  - ▻ in einer **Umsetzung** von **Parameterwerten**
  - ▻ in der **Anpassung** an ein durch eine **andere Ableitungshierarchie** festgelegtes **Interface**

  ◇ Durch eine Interface-Klasse kann eine – spezielle – Benutzung einer Dienstleistungsklasse vereinfacht werden. Notwendige Anpassungen werden aus dem Anwendungscode herausgenommen und in der Interface-Klasse implementiert. Gegebenenfalls wird die Benutzung der Dienstleistungs-Klasse durch die Interface-Klasse überhaupt erst ermöglicht.

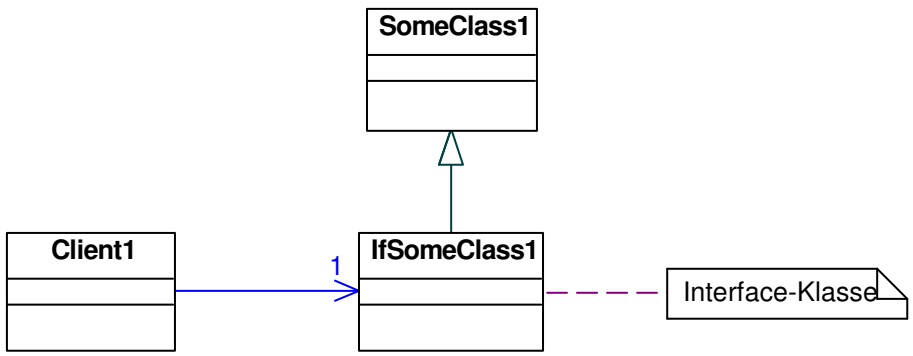
  ◇ Sehr häufig ist eine Interface-Klasse von der Klasse, deren Schnittstelle sie anpassen soll, **abgeleitet**.

  ◇ In anderen Fällen enthält ein Objekt der Interface-Klasse ein Objekt der anzupassenden Klasse oder einen Pointer auf ein derartiges Objekt als Datenkomponente (**Aggregation** bzw. **Assoziation**).

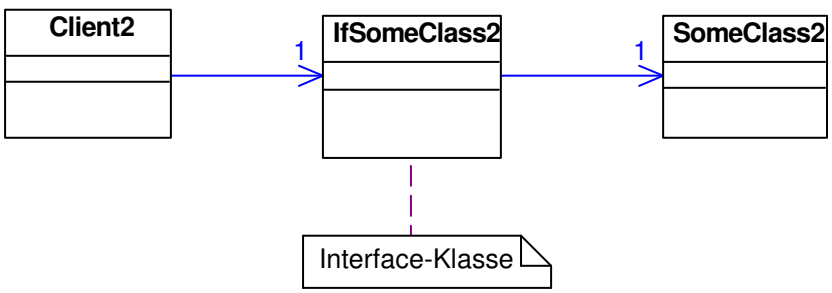
  ◇ In **allgemeinerer Form** wird die Anpassung eines Klassen-Interfaces durch das **Design-Pattern Adapter** realisiert.

- **Klassendiagramme**

  ◇ **Interface-Klasse von der anzupassenden Klasse abgeleitet**



  ◇ **Interface-Klasse assoziiert die anzupassende Klasse**



Interface-Klassen (2)

- **Beispiel 1:** Modifikation des Indexbereichs einer Klasse **`IntVector`** → **Interface-Klasse `IntModVec`**

  ◇ Objekte der Klasse `IntVector` (Vektor von Integerzahlen) besitzen `size` Komponenten.
  Diese sind über den Indexbereich `0 .. (size-1)` zugänglich (Index-Operatorfunktion `operator[]()`)

  ◇ Benötigt werden aber Vektor-Objekte, deren **unterer und oberer Index beliebig festgelegt** werden kann.
  → abgeleitete Klasse **`IntModVec.`**
  Objekte dieser Klasse speichern zusätzlich den unteren Index.
  Ihre Index-Operatorfunktion `operator[]()` setzt den ihr übergebenen Parameter (Auswahl-Index) vor dem Aufruf der Index-Operatorfunktion der Klasse `IntVec` entsprechend um.

  ◇ **Definition der Klasse `IntVector`**

```
#define DEF_SIZE 10

class IntVector
{ public :
    IntVector(unsigned size=DEF_SIZE);
    ~IntVector();
    int& operator[](int ind);
    unsigned size() { return m_uSize; };
    // ...
  private :
    int* m_pData;
    unsigned m_uSize;
};
```

  ◇ **Definition der Interface-Klasse `IntModVec`**

```
class IntModVec : public IntVector
{ public :
    IntModVec(int low, int high) : IntVector(high-low+1) { m_iLow=low;}
    ~IntModVec() {}
    int& operator[](int ind) { return IntVector::operator [](ind-m_iLow); }
    int low()  { return m_iLow; }
    int high() { return m_iLow+size()-1; }
  private :
    int m_iLow;
};
```

  ◇ **Anwendung der Interface-Klasse `IntModVec`**

```
int main(void)
{
  IntModVec imvec(-5, 5);
  // ...
  for (int i=imvec.low(); i<=imvec.high(); i++)
    imvec[i]=i;
  // ...
  return 0;
}
```

Interface-Klassen (3 - 1)

- **Beispiel 2: Interface-Klassen zur Vermeidung von Namenskonflikten**

  ◇ Es existieren **2 verschiedenene Klassenhierarchien** (hier : abstrakte Basisklassen **`Fahrzeug`** und **`Gebaeude`**). Beide definieren jeweils eine **virtuelle Funktion gleichen Namens** mit ganz unterschiedlicher – auch nicht ähnlicher – Funktionalität (hier **`steuern()`**).

  ◇ Beide Klassenhierarchien werden durch **Mehrfachvererbung** in einer gemeinsamen Klasse **zusammengefasst** (hier : Ableitung von **`Hausboot`** von **`Haus`** (abgel. von `Gebaeude`) und **`Boot`** (abgel. von `Fahrzeug`)).

  ◇ In der mehrfach abgeleiteten Klasse (hier : `Hausboot`) sollen **beide virtuellen Funktionen gleichen Namens** (hier : `steuern()`) **getrennt überschrieben werden**.
  Falls beide Funktionen gleichen Namens eine **unterschiedliche Parameterliste** besitzen, ist dies **problemlos möglich**. (die getrennte Namensauflösung richtet sich nach den Regeln für überschriebene Funktionen).
  Falls beide Funktionen die **gleiche Parameterliste** und den **gleichen Rückgabetyp** aufweisen ist in der abgeleiteten Klasse nur **eine Funktion** möglich, die **beide virtuellen Funktionen überschreibt**. Damit bekommen diese die gleiche Funktionalität, was aber nicht der Fall sein soll.
  Falls beide Funktionen die **gleiche Parameterliste** aber einen **unterschiedlichen Rückgabetyp** besitzen, kann überhaupt **keine überschreibende Funktion definiert** werden

  ◇ Lösung : **zwei Interface-Klassen** (hier : **`IfHaus`** und **`IfBoot`**), die die beiden virtuellen Funktionen unter **unterschiedlichen Namen** zugänglich und damit **getrennt überschreibbar** machen.

  ◇ **Klassenhierarchie mit abstrakter Basisklasse `Fahrzeug`**

```
class Fahrzeug
{ public :
    // ...
    virtual void steuern()=0;   // Ziel ansteuern
  // ...
};

// ---------------------------------------------------------------------------

class Boot : public Fahrzeug
{ public :
    // ...
    void steuern();
  // ...
};
```

  ◇ **Klassenhierarchie mit abstrakter Basisklasse `Gebauede`**

```
class Gebaeude
{ public :
    // ...
    virtual float steuern()=0;   // Ermittlung der Gebaeude-Steuer
  // ...
};

// ---------------------------------------------------------------------------

class Haus : public Gebaeude
{ public :
    // ...
    float steuern();
  // ...
};
```

Interface-Klassen (3 - 2)

- **Beispiel 2: Interface-Klassen zur Vermeidung von Namenskonflikten**, Forts.

  ◇ **Definition der Interface-Klassen `IfBoot` und `IfHaus`**

```
class IfBoot : public Boot
{
  public :
    // ...
    void steuern()  { ansteuernZiel(); };
    virtual void ansteuernZiel() = 0;
  // ...
};

// ---------------------------------------------------------------------------

class IfHaus : public Haus
{
  public :
    // ...
    float steuern() { return ermittelnSteuer(); };
    virtual float ermittelnSteuer() = 0;
  // ...
};
```

  ◇ **Definition der mehrfach abgeleiteten Klasse `Hausboot`**

```
class Hausboot : public IfHaus, public IfBoot
{ public :
    // ...
    float ermittelnSteuer();
    void ansteuernZiel();
  // ...
};
```

  ◇ **Verwendung der mehrfach abgeleiteten Klasse `Hausboot`**

```
// Referierung von Hausboot-Objekten über Basisklassen-Pointer


int main(void)
{ Fahrzeug * pf2 = new Hausboot;
  Gebaeude * pg2 = new Hausboot;

  pf2->steuern();     // Aufruf von Hausboot::ansteuernZiel()
  pg2->steuern();     // Aufruf von Hausboot::ermittelnSteuer()
  // ...
}
```

Handle-Klassen in C++ (1)

- **Prinzip und Eigenschaften von Handle-Klassen** :

  ◇ In vielen Fällen ist es sinnvoll und effizient, ein **Objekt nicht direkt** zu verwenden, sondern jeglichen Zugriff zu ihm über ein **spezielles Zugriffs-Objekt** vorzunehmen.
  Das Zugriffs-Objekt wird als **Handle**(-Objekt) bezeichnet → **Handle-Klasse**.
  Das Objekt, zu dem das Handle-Objekt ein Zugriffs-Interface bereitstellt, enthält die eigentliche Funktionalität, die einem Client-Objekt zur Verfügung gestellt wird ("**Repräsentations**"**-Objekt**, **Nutz-Objekt**)
  → Handle-Objekt und Nutz-Objekt werden wie ein einheitliches Objekt verwendet.
  Man kann auch sagen : Ein einheitlich erscheinendes Objekt besteht aus zwei Teilen : dem Interface und der Repräsentation.

  ◇ Eine Handle-Klasse stellt im Prinzip auch eine Interface-Klasse dar, bei der aber **nicht** die **Modifikation der Zugriffs-Schnittstelle** zu einem Dienstleistungs-Objekt im Vordergrund steht, sondern der **grundsätzliche Zugriff** zu diesem.
  Die **Zugriffsschnittstelle** soll dabei i.a. gerade **nicht verändert** werden.
  Häufig ist es sinnvoll oder erforderlich, bei einem derartigen Zugriff bestimmte **Pre- und/oder Post-Aktionen** auszuführen. Diese können in einer Handle-Klasse relativ einfach implementiert werden.

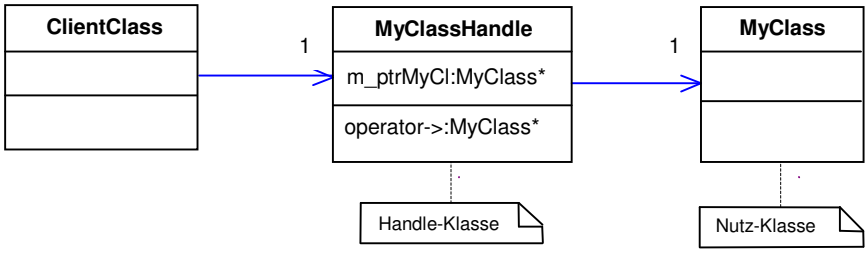
  ◇ Die **Verbindung** zwischen einem Handle-Objekt und seinem Nutz-Objekt wird typischerweise durch einen **Pointer** im Handle-Objekt realisiert.
  Häufig – aber nicht immer – ist dieser Pointer die einzige Datenkomponente eines Handle-Objekts.
  Er wird i.a. dem Konstruktor der Handle-Klasse als Parameter übergeben.

  ◇ Zum **einfachen Zugriff** zum Nutz-Objekt wird in der Handle-Klasse i.a. die **Operatorfunktion** `operator->()` überladen.
  Damit lassen sich die Memberfunktionen des Nutz-Objekts direkt über das Handle-Objekt aufrufen.

  ◇ Sollen beim Aufruf der verschiedenen Memberfunktionen des Nutz-Objekts **Pre- und/oder Post-Aktionen** durchgeführt werden, kann eine **komplexere Implementierung der Zugriffsmöglichkeit** erforderlich sein.
  Sollen nur Pre-Aktionen ausgeführt werden, die für jeden Zugriff (für alle Memberfunktionen) gleich sind, reicht das Überladen des ->-Operators i.a. aus.
  Sind die auszuführenden Pre-Aktionen dagegen für die verschiedenen Memberfunktionen unterschiedlich und/oder sind auch Post-Aktionen auszuführen, muß die Klassenschnittstelle des Nutz-Objekts in der Handle-Klasse durch entsprechende Memberfunktionen nachgebildet werden ("Spiegelung der Nutz-Objekt-Schnittstelle").

  ◇ Eine Handle-Klasse soll i.a. nicht nur für eine einzige Klasse von Nutz-Objekten zuständig sein, sondern für **mehrere unterschiedliche** Klassen.
  Wenn diese Klassen von einer einzigen – häufig abstrakten – **Basisklasse abgeleitet** sind, lässt sich das durch einen Verbindungspointer vom Typ "Pointer auf Basisklasse" realisieren.
  Andernfalls bietet es sich an, die Handle-Klasse als **Klassen-Template** zu definieren.
  Allerdings ist dann die Nachbildung der Klassenschnittstelle der Nutz-Objekte in der Handle-Klasse kaum möglich.

- **Klassendiagramm**



Handle-Klassen in C++ (2)

- **Einfaches Demonstrations-Beispiel**

  ◇ Handle-Klassen-Template **`SetHandle<T>`** zum Zugriff zu Mengen-Objekten (Klassen-Template `Set<T>`)

  ◇ `SetHandle<T>`-Objekte besitzen einen **Zugriffszähle**r, der die Zugriffe zu dem referierten Nutzobjekt zählt.

  ◇ **Definition der Nutzklasse (Klassen-Template `Set<T>`) und der Handle-Klasse (Klassen-Template `HandleSet<T>`)**

```
template <class T>              // Nutzklasse
class Set
{ public :
    Set();
    ~Set();
    void insert(const T&);
    void remove(const T&);
    int isMember(const T&);
    T* first();
    T* next();
  private :
    T* members;
    unsigned max; // ohne Neuallokation mögliche max. Anzahl von Elementen
    unsigned act; // aktuelle Anzahl von Mengenelementen
    unsigned nxt; // Index des nächsten Mengenelements (fuer Mengeniteration)
};

// ---------------------------------------------------------------------------

template <class T>              // Handle-Klasse
class SetHandle
{ public :
    SetHandle(Set<T>* ps) : pSet(ps) { access=0; }
    Set<T>* operator->() { access++; return pSet;}
    unsigned getAcc()    { return access; }
  private :
    Set<T>* pSet;          // Pointer auf referiertes Nutz-Objekt
    unsigned access;       // Zugriffszaehler zu Nutzobjekt
};
```

  ◇ **Anwendungscode (Beispiel)**

```
void fillSet(SetHandle<int>& sh)
{ sh->insert(5);
  sh->insert(7);
  sh->insert(3);
}

void useSet(SetHandle<int>& sh)
{ for (int* ip=sh->first(); ip; ip=sh->next())
  {  // do something
  }
}

int main()
{ SetHandle<int> mySH = new Set<int>;
  fillSet(mySH);
  useSet(mySH);
  cout << "\nAnzahl der Zugriffe : " << mySH.getAcc() << endl;
  return 0;
}
```

Referenzzählung in C++

- **Prinzip**

  ◇ In vielen Fällen soll **ein und dasselbe Nutzobjekt** mehrfach verwendet werden (z.B. gemeinsam von **mehreren Client-Objekten**).
  Hierfür lässt sich sehr sinnvoll das Entwurfsmuster **Referenzzählung** (*reference counting*) einsetzen.

  ◇ Dieses Muster bewirkt, dass bei der **Mehrfachverwendung eines Objekts** keine Kopien von diesem angelegt werden. (→ Reduzierung von Speicherplatz und Rechenzeit)
  Stattdessen werden **Referenzen** auf das **Objekt gezählt**.
  Bei jeder erneuten Verwendung des Objekts wird der Referenzzähler inkrementiert, bei jeder Freigabe des Objekts wird der Referenzzähler dekrementiert.
  Das Objekt wird erst dann **gelöscht**, wenn es von keinem Client-Objekt mehr benutzt wird, der **Referenzzähler** also `==0` geworden ist.

  ◇ Referenzzählung lässt sich nur dann verwenden, wenn bei den verschiedenen Verwendungen das Nutzobjekt nicht individuell verändert wird.

- **Implementierung (1)**

  ◇ Der **Referenzzähler** befindet sich **in dem Nutzobjekt**, das mehrfach verwendet werden soll.

  ◇ Nutzer (Client-Objekte) **referieren** das Nutzobjekt durch **Handle-Objekte**

  ◇ Bei jeder – erneuten – Referierung (Verwendung) wird ein **neues Handle-Objekt** erzeugt, dessen **Konstruktor** den Verbindungspointer auf das Nutzobjekt setzt und den **Referenzzähler** desselben **incrementiert**.

  ◇ Beim **Löschen** eines **Handle-Objekts** bewirkt dessen **Destruktor** eine **Dekrementierung** des **Referenzzählers**. Erreicht dieser den **Wert** `0,` wird das **Nutzobjekt gelöscht**.

  ◇ Beim **Kopieren** zwischen zwei **Handle-Objekten** (durch Copy-Konstruktor oder Zuweisungsoperator) gibt das Ziel-Handle-Objekt zunächst das bisher von ihm referierte Nutzobjekt frei (durch Dekrementieren dessen Referenzzählers), setzt dann seinen Verbindungspointer auf das Nutz-Objekt des Quell-Handle-Objekts und inkrementiert dessen Referenzzähler.

  ◇ Das **Nutzobjekt** muss ein **dynamisch** (mit `new`) **erzeugtes** Objekt sein, da es mit `delete` gelöscht wird.
  Das **Löschen** erfolgt **automatisch** innerhalb des Referenzzählungs-Mechanismus und darf deshalb **nie explizit** durch einen Benutzer des Nutz-Objekts erfolgen

  ◇ Damit das **Handle-Objekt** den **Referenzzähler** des Nutz-Objekts **beeinflussen** kann, muß
  - ▻ die Nutzklasse entsprechende **Zugriffsfunktionen** zur Verfügung stellen, oder
  - ▻ die Handle-Klasse **Freundklasse** der Nutz-Klasse sein, oder
  - ▻ der Referenzzähler eine **`public`-Komponente** des Nutz-Objekts sein.

- **Implementierung (2)**

  ◇ In einer **modifizierten Realisierung** befindet sich der **Referenzzähler nicht im Nutzobjekt**, sondern wird vom ersten **Handle-Objekt**, das das Nutzobjekt referiert, als **dynamisch allozierte Variable angelegt**.
  → Das Nutzobjekt weiß nichts von einer Referenzzählung.

  ◇ Dies ermöglicht eine **Referenzzählung für Objekte beliebiger** – und nicht nur speziell dafür vorgesehener – **Klassen** Diese müssen also weder eine Zugriffsmöglichkeit zu einem Referenzzähler zur Verfügung stellen, noch eine Handle-Klasse zum Freund besitzen.→ vollkommene **Entkopplung** zwischen Nutzobjekt und Handle-Objekt

  ◇ Es gilt allerdings die **Einschränkung**, dass ein **neues Handle-Objekt** für dasselbe Nutzobjekt **immer** mit einem **bereits existierenden Handle-Objekt** (Copy-Konstruktor!) und nicht mit dem Nutzobjekt direkt (normaler Konstruktor) **initialisiert** werden muß.
  Nur der Copy-Konstruktor (und der Zuweisungsoperator) erhöhen den Referenzzähler.

  ◇ Der Destruktor eines Handle-Objekts löscht auch den Referenzzähler, wenn dieser den Wert `0` erreicht.

Beispiel 1 zur Referenzzählung in C++ (1)

- **Beispiel zur Implementierung (1)** (Referenzzähler im Nutzobjekt) **:**

  ◇ Definition eines Handle-Klassen-Templates **`RefCntHdl<T>.`**
  Dies ermöglicht eine weitgehend universelle Verwendung der Handle-Klasse.
  Den Zugriff zu dem jeweiligen Nutzobjekt ermöglicht der geeignet überladene Operator **`->`**.

  ◇ Zusammenfassung der spezifischen Funktionalität von Nutz-Objekten mit Referenzzählung in einer Basisklasse **`RefCntUse.`**
  Diese Klasse enthält den Referenzzähler und stellt Funktionen zum Inkrementieren und Dekrementieren desselben zur Verfügung.
  Wird beim Dekrementieren der Referenzzähler `==0,` vernichtet sich ein `RefCntUse`-Objekt selbst (→ **Suizid-Objekt**).
  Zur Verhinderung des expliziten Anlegens von `RefCntUse`-Objekten sind seine Konstruktoren `protected` bzw `private` (Copy-Konstruktor).

  ◇ Als Beispiel einer explizit verwendbaren Nutzklasse mit Referenzzählung wird die Klasse **`CntString`** von `RefCntUse` abgeleitet. Sie implementiert eine sehr einfache String-Funktionalität.

- **Handle-Klassen-Template `RefCntHdl`** (enthalten in Headerdatei `refcnthdl.h`)

```
template <class T>
class RefCntHdl
{ public :
    RefCntHdl(T* pObj)                  : m_ptr(pObj)      { m_ptr->Inc(); }
    RefCntHdl(const RefCntHdl<T>& obj) : m_ptr(obj.m_ptr) { m_ptr->Inc(); }
    ~RefCntHdl()                        { m_ptr->Dec(); }
    T* operator->()                     { return m_ptr; }
    operator const T&() const           { return *m_ptr; }
    RefCntHdl<T>& operator=(const RefCntHdl<T>& obj);
  private :
    T* m_ptr;
};


template <class T>
RefCntHdl<T>& RefCntHdl<T>::operator=(const RefCntHdl<T>& obj)
{ if (m_ptr!=obj.m_ptr)
  { m_ptr->Dec();
    m_ptr=obj.m_ptr;
    m_ptr->Inc();
  }
  return *this;
}
```

- **Basisklasse für Objekte mit Referenzzählung `RefCntUse`** (enthalten in Headerdatei `refcntuse.h`)

```
class RefCntUse
{ public :
    void Inc()              { ++m_refCnt; }
    void Dec()              { if (--m_refCnt==0) delete this; }   // Selbstmord !
    int getRefCnt() const   { return m_refCnt; }
    virtual ~RefCntUse()    { cout << "\nDestruktor von RefCntUse\n"; }
  private :
    int m_refCnt;
    RefCntUse(const RefCntUse& uObj) { }
  protected :
    RefCntUse()             : m_refCnt(0) { cout << "\nKonstruktor von RefCntUse";}
};
```

Beispiel 1 zur Referenzzählung in C++ (2)

- **Definition der abgeleiteten Klasse für Objekte mit Referenzzählung `CntString`**
  (enthalten in Headerdatei `cntstring.h`)

```
#include "refcntuse.h"

class CntString : public RefCntUse
{ public :
    CntString(const char* str);
    CntString(const CntString&);
    ~CntString();
    void setVal(const char* str);
    const char* getVal() const;
  private :
    char* m_pcStr;
    void newVal(const char* str);               // interne Hilfsfunktion
    CntString& operator=(const CntString&);
};
```

- **Implementierung der abgeleiteten Klasse für Objekte mit Referenzzählung `CntString`**
  (enthalten in C++-Quelldatei `CntString.cpp`)

```
#include "cntstring.h"
#include <cstring>
#include <iostream>
using namespace std;

CntString::CntString(const char* str)
{ newVal(str);         cout << "\nKonstruktor von CntString " << m_pcStr << endl;}

CntString::CntString(const CntString& cs)
{ newVal(cs.m_pcStr);  cout <<"\nCopy-Konstruktor von CntString "<<m_pcStr<<endl;}

CntString::~CntString()
{ cout << "\nDestruktor von CntString " << m_pcStr;
  delete[] m_pcStr;  m_pcStr=NULL;
}

void CntString::setVal(const char* str)
{ delete[] m_pcStr;  newVal(str); }

const char* CntString::getVal() const
{ return m_pcStr; }

// ------------------------------------ private ----------------------------------

void CntString::newVal(const char* str)
{ if (str!=NULL)
  { m_pcStr=new char[strlen(str)+1];
    strcpy(m_pcStr, str);
  }
  else
    m_pcStr=NULL;
}
```

Beispiel 1 zur Referenzzählung in C++ (3)

- **`main()`–Funktion eines einfachen Demonstrationsprogramms** (C++-Quelldatei `refcounttempl_m.cpp`)

```
#include "refcnthdl.h"
#include "cntstring.h"

#include <iostream>
using namespace std;

void infoOut(RefCntHdl<CntString>& rch)
{ cout <<  rch->getVal() << "  (RefCount : " << rch->getRefCnt() << ')'; }

int main(void)
{ RefCntHdl<CntString> rch1 = new CntString("Frage ?");
  RefCntHdl<CntString> rch2 = new CntString("Antwort !");
  RefCntHdl<CntString> rch3 = new CntString(rch2);
  cout << endl << "rch1 : ";  infoOut(rch1);
  cout << endl << "rch2 : ";  infoOut(rch2);
  cout << endl << "rch3 : ";  infoOut(rch3);  cout << endl;
  rch2=rch1;
  cout << endl << "rch1 : ";  infoOut(rch1);
  cout << endl << "rch2 : ";  infoOut(rch2);
  cout << endl << "rch3 : ";  infoOut(rch3);  cout << endl;
  rch1=new CntString("Schweigen");
  rch2=rch3;
  cout << endl << "rch1 : ";  infoOut(rch1);
  cout << endl << "rch2 : ";  infoOut(rch2);
  cout << endl << "rch3 : ";  infoOut(rch3);  cout << endl;
  return 0;
}
```

Beispiel 1 zur Referenzzählung in C++ (4)

- **Ausgabe des Demonstrationsprogramms `RefCountTempl`**

```
Konstruktor von RefCntUse
Konstruktor von CntString Frage ?

Konstruktor von RefCntUse
Konstruktor von CntString Antwort !

Konstruktor von RefCntUse
Copy-Konstruktor von CntString Antwort !

rch1 : Frage ?  (RefCount : 1)
rch2 : Antwort !  (RefCount : 1)
rch3 : Antwort !  (RefCount : 1)

Destruktor von CntString Antwort !
Destruktor von RefCntUse

rch1 : Frage ?  (RefCount : 2)
rch2 : Frage ?  (RefCount : 2)
rch3 : Antwort !  (RefCount : 1)

Konstruktor von RefCntUse
Konstruktor von CntString Schweigen

Destruktor von CntString Frage ?
Destruktor von RefCntUse

rch1 : Schweigen  (RefCount : 1)
rch2 : Antwort !  (RefCount : 2)
rch3 : Antwort !  (RefCount : 2)

Destruktor von CntString Antwort !
Destruktor von RefCntUse

Destruktor von CntString Schweigen
Destruktor von RefCntUse
```

Beispiel 2 zur Referenzzählung in C++ (1)

- **Beispiel zur Implementierung (2)** (Referenzzähler vom ersten Handle-Objekt angelegt) **:**
  - ◇ Auch in diesem Beispiel wird die Handle-Klasse als Template definiert. → **`RefCntHdl2<T>`**
    → universelle Einsetzbarkeit.
  - ◇ Als Klasse für Nutzobjekte kann jede beliebige Klasse dienen.
    Hier wird die Klasse `SimplString`, eine einfache String-Klasse, verwendet.

- **Handle-Klassen-Template `RefCntHdl2`** (enthalten in Headerdatei `refcnthdl2.h`)

```
template <class T>
class RefCntHdl2
{ public :
    RefCntHdl2(T* pObj)                 : m_ptr(pObj) { m_pRefCnt=new int(1); }
    RefCntHdl2(const RefCntHdl2<T>& obj);
    ~RefCntHdl2();
    RefCntHdl2<T>& operator=(const RefCntHdl2<T>& obj);
    T* operator->()                     { return m_ptr; }
    operator const T&() const           { return *m_ptr; }
    int getRefCnt() const               { return *m_pRefCnt; }
  private :
    T* m_ptr;
    int* m_pRefCnt;
};


template <class T>
RefCntHdl2<T>::RefCntHdl2(const RefCntHdl2<T>& obj) : m_ptr(obj.m_ptr)
{ m_pRefCnt=obj.m_pRefCnt;
  (*m_pRefCnt)++;
}


template <class T>
RefCntHdl2<T>::~RefCntHdl2()
{ if (--(*m_pRefCnt)==0)
  { delete m_pRefCnt;
    delete m_ptr;
  }
  m_pRefCnt=0;       // == NULL-Pointer
  m_ptr=0;           // == NULL-Pointer
}


template <class T>
RefCntHdl2<T>& RefCntHdl2<T>::operator=(const RefCntHdl2<T>& obj)
{ if (m_ptr!=obj.m_ptr)
  { if (--(*m_pRefCnt)==0)
    { delete m_pRefCnt;
      delete m_ptr;
    }
    m_ptr=obj.m_ptr;
    m_pRefCnt=obj.m_pRefCnt;
    ( *m_pRefCnt)++;
  }
  return *this;
}
```

Beispiel 2 zur Referenzzählung in C++ (2)

- **Definition der Klasse `SimplString`** (enthalten in Headerdatei `simplstring.h`)

```
class SimplString
{
  public :
    SimplString(const char* str);
    SimplString(const SimplString&);
    ~SimplString();
    void setVal(const char* str);
    const char* getVal() const;
  private :
    char* m_pcStr;
    void newVal(const char* str);                // Hilfsfunktion
    SimplString& operator=(const SimplString&);
};
```

- **Modul mit `main()`-Funktion eines einfachen Demonstrationsprogramms** (`refcounttempl2_m.cpp`)

```
#include "refcnthdl2.h"
#include "simplstring.h"

#include <iostream>
using namespace std;


void infoOut(RefCntHdl2<SimplString>& rch)
{ cout << rch.getRefCnt() << "  Value : " << rch->getVal(); }


void useString(RefCntHdl2<SimplString> rchx)
{ cout << endl << "rchx -> RefCount : ";  infoOut(rchx);  cout << endl;
  // ...
}


int main(void)
{ RefCntHdl2<SimplString> rch1 = new SimplString("Frage ?");
  RefCntHdl2<SimplString> rch2 = new SimplString("Antwort !");
  RefCntHdl2<SimplString> rch3 = rch2;

  cout << endl << "rch1 -> RefCount : ";  infoOut(rch1);
  cout << endl << "rch2 -> RefCount : ";  infoOut(rch2);
  cout << endl << "rch3 -> RefCount : ";  infoOut(rch3);  cout << endl;

  rch1=rch3;
  cout << endl << "rch1 -> RefCount : ";  infoOut(rch1);
  cout << endl << "rch3 -> RefCount : ";  infoOut(rch3);  cout << endl;

  useString(rch1);
  cout << endl << "rch1 -> RefCount : ";  infoOut(rch1);
  cout << endl << "rch3 -> RefCount : ";  infoOut(rch3);  cout << endl;

  rch1=new SimplString("Schweigen");
  cout << endl << "rch1 -> RefCount : ";  infoOut(rch1);
  cout << endl << "rch3 -> RefCount : ";  infoOut(rch3);  cout<< endl;

  rch2->setVal("ist Gold");
  cout << endl << "rch2 -> RefCount : ";  infoOut(rch2);
  cout << endl << "rch3 -> RefCount : ";  infoOut(rch3);  cout << endl;
  return 0;
}
```

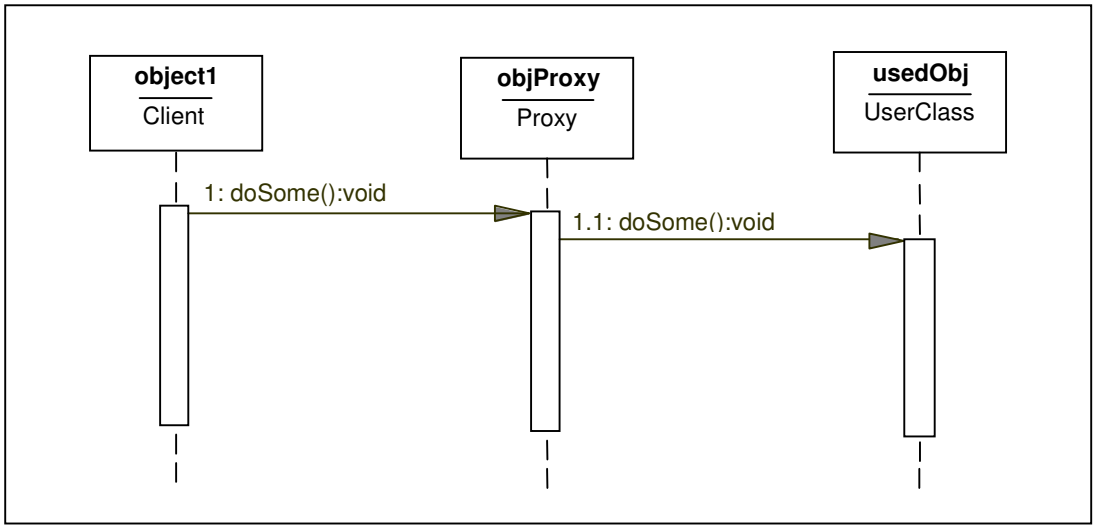
Beispiel 2 zur Referenzzählung in C++ (3)

- **Ausgabe des Demonstrationsprogramms `RefCountTempl2`**

```
rch1 -> RefCount : 1  Value : Frage ?
rch2 -> RefCount : 2  Value : Antwort !
rch3 -> RefCount : 2  Value : Antwort !

rch1 -> RefCount : 3  Value : Antwort !
rch3 -> RefCount : 3  Value : Antwort !

rchx -> RefCount : 4  Value : Antwort !

rch1 -> RefCount : 3  Value : Antwort !
rch3 -> RefCount : 3  Value : Antwort !

rch1 -> RefCount : 1  Value : Schweigen
rch3 -> RefCount : 2  Value : Antwort !

rch2 -> RefCount : 2  Value : ist Gold
rch3 -> RefCount : 2  Value : ist Gold
```

Proxy-Klassen in C++

- **Prinzip und Eigenschaften von Proxy-Klassen** :
  - ◇ Eine **Proxy-Klasse** ist eine Klasse, deren Objekte als **Stellvertreter** für Objekte einer anderen Klasse verwendet werden.
  - ◇ Eine Proxy-Klasse stellt das **gleiche Interface** wie die durch sie vertretene Klasse zur Verfügung.
  - ◇ **Typische Anwendungen** :
    - ▻ Verbergen der privaten Komponenten der eigentlichen Nutzklasse
    - ▻ Nutzung von entfernten – auf anderen Rechnern befindlichen – Objekten (z.B. bei CORBA)
  - ◇ **Sequenzdiagramm** (prinzipielles Beispiel)



- **Proxy-Klassen zum Verbergen der privaten Komponenten einer Nutzklasse**
  - ◇ Die – üblicherweise in einer Headerdatei abgelegte – **Definition einer Klasse** enthält auch die **privaten Komponenten** der Klasse. Diese stellen aber bereits ein **Implementierungsdetail** dar, das dem Nutzer einer Klasse nicht unbedingt immer bekannt gemacht werden soll. Vielmehr sollen sie ihm gegenüber **verborgen** bleiben
  - ◇ Zur **Verwendung einer Klasse** muss aber die **Headerdatei**, die ihre Definition enthält, in den Anwendungscode **eingebunden** werden. Damit sind dann aber auch die **privaten Klassenkomponenten sichtbar**.
  - ◇ Zur Lösung des Problems wird eine **Proxy-Klasse** definiert, die das **gleiche Interface** (`public`-Komponenten) wie die eigentlich zu verwendende Klasse (Nutzklasse) besitzt :
    - ▻ Ein Objekt der Proxy-Klasse enthält als – i. a. einzige – private Datenkomponente einen Pointer auf ein Objekt der Nutzklasse.
    - ▻ Jede Memberfunktion der Proxyklasse ruft – mit den ihr übergebenen Parametern – die gleichnamige Funktion der Nutzklasse auf
    - ▻ Die Headerdatei mit der **Definition der Proxy-Klasse** enthält als **einzigen Verweis** auf die eigentliche **Nutzklasse** eine **Vorwärtsdeklaration** derselben.
      Ein Einbinden der Headerdatei mit der Definition der Nutzklasse ist nicht notwendig, da nur ein Pointer auf diese verwendet wird.
    - ▻ In den **Anwendercode** ist **lediglich** die **Headerdatei** der **Proxy-Klasse einzubinden**.
      Damit enthält der Anwendercode selbst keinerlei direkten Hinweis auf die eigentliche Nutzklasse.
    - ▻ Die Headerdatei der Nutzklasse wird lediglich in der Implementierungsdatei der Proxyklasse benötigt
      Da die Implementierung der Proxy-Klasse aber üblicherweise als Objektcode zur Verfügung gestellt wird, ist ihre Interaktion mit der Nutzklasse ebenfalls nicht sichtbar

Beispiel zu Proxy-Klassen in C++ (1)

- **Definition einer Nutzklasse `UserClass`** (enthalten in Headerdatei `userclass.h`):

```
class UserClass
{ public :
    UserClass(int val)   { m_iVal=val; }
    void setVal(int val) { m_iVal=val; }
    int getVal()         { return m_iVal; }
    void doSome()        { /* ... */ }
  private :
    int m_iVal;
};
```

- **Definition einer Proxyklasse `Proxy`** (enthalten in Headerdatei `proxy.h`):

```
class UserClass;            // Klassen-Vorwärts-Deklaration

class Proxy
{ public :
    Proxy(int val);
    ~Proxy();
    void setVal(int val);   // gleiches Interface wie Klasse UserClass
    int getVal();
    void doSome();
  private :
    UserClass* m_pUC;
};
```

- **Implementierung der Proxyklasse `Proxy`** (enthalten in C++-Quelldatei `proxy.cpp`):

```
#include "proxy.h"
#include "userclass.h"

#include <cstdlib>


Proxy::Proxy(int val)
{ m_pUC = new UserClass(val); }


Proxy::~Proxy()
{ delete m_pUC;
  m_pUC=NULL;
}


void Proxy::setVal(int val)
{ m_pUC->setVal(val); }


int Proxy::getVal()
{ return m_pUC->getVal(); }


void Proxy::doSome()
{ m_pUC->doSome(); }
```

Beispiel zu Proxy-Klassen in C++ (2)

- **Modul mit `main()`-Funktion eines einfachen Demonstrationsprogramms** (`proxybsp_m.cpp`):

```
#include "proxy.h"

#include <iostream>
using namespace std;


int main(void)
{
  Proxy p(3);

  cout << "\nEnthaltener Wert vor  Aenderung : " << p.getVal();
  p.setVal(7);
  cout << "\nEnthaltener Wert nach Aenderung : " << p.getVal() << endl;

  return 0;
}
```

- **Ausgabe des Demonstrationsprogramms `proxybsp`**

```
Enthaltener Wert vor  Aenderung : 3
Enthaltener Wert nach Aenderung : 7
```

Iteratoren in C++ (1)

- **Eigenschaften** :

  ◇ Ein **Iterator** ist ein **Hilfsobjekt**, das einen **Mechanismus** zum **Durchlaufen von Objekten**, die Ansammlungen von Daten (i.a. andere Objekte) enthalten und verwalten (Container-Objekte), zur Verfügung stellt.

  ◇ Im wesentlichen implementieren Iteratoren die folgenden **fundamentalen Funktionalitäten** :

  - ▻ **Zugriff** zu dem gerade **aktuellen Element** (durch aktuelle Position festgelegt)
  - ▻ **Fortschreiten** zum **nächsten Element** (Setzen der aktuellen Position auf das nächste Element)
  - ▻ Erkennen des Erreichens des **letzten Elements**

  ◇ Diese Funktionalitäten implizieren, dass die einzelnen Elemente in einer **Reihenfolge** angeordnet sind.
  Damit unterstützen Iteratoren ein **abstraktes Datenmodell für Container**, bei dem deren Inhalt als **Sequenz von Elementen** (Objekten) aufgefasst wird.
  Eine **Sequenz** kann definiert werden, als eine **Abstraktion** von **"etwas"**, das man mit der Operation **"nächstes_Element"** von Anfang bis Ende durchlaufen kann.

  ◇ Prinzipiell kann der Mechanismus zum Navigieren in Container-Objekten auch in diesen selbst angesiedelt sein. Dann existiert pro Container-Objekt aber immer nur eine aktuelle Position. In vielen Anwendungen ist es aber notwendig zu mehreren Elementen (und damit zu verschiedenen aktuellen Positionen) gleichzeitig zuzugreifen (z.B. Durchlaufen einer Menge in zwei geschachtelten Schleifen zum Vergleich eines Elements mit allen anderen Elementen der **Menge**). Dies lässt sich mittels **mehrerer Iterator-Objekte** für das **gleiche Container-Objekt** relativ leicht realisieren.
  → **Auslagerung** des Navigations- und Zugriffsmechanismus in eine **eigene Iterator-Klasse** ist **wesentlich flexibler.**

  ◇ In einer **allgemeineren Betrachtungsweise** lassen sich **Iteratoren** als **abstrahierende Verallgemeinerung** von **Pointern** (die jeweils auf ein **Element** eines **Arrays zeigen**), auffassen :
  So wie man mittels eines Pointers ein Array durchlaufen und zu den einzelnen Elementen schreibend und lesend zugreifen kann, kann man mit einem Iterator einen Container durchlaufen und zu den einzelnen in ihm gespeicherten Elementen zugreifen.
  → Ein **Iterator** kann als eine **Abstraktion** eines **Zeigers** auf ein **Element** einer **Sequenz** betrachtet werden.

  ◇ **Sequenzen** können **unterschiedlich** strukturiert und organisiert sein. → **unterschiedliche Container-Arten**.
  Zu jeder Container-Art gehört sinnvollerweise eine eigene **passende Iterator-Art**.
  In einer konsistenten Implementierung (z.B. in einer Bibliothek) lassen sich Iteratoren – unabhängig von ihrer jeweiligen Art – aber immer **gleichartig verwenden**.
  Sie stellen damit ein **einheitliches Konzept** zum Positionieren und Durchlaufen von Container-Objekten zur Verfügung.

- **Implementierung** (1) :

  ◇ Zur Ermittlung der jeweiligen aktuellen Position und zum Zugriff zu dem dadurch festgelegten Element benötigt ein Iterator-Objekt **Zugang** zu den **Interna** des **Container-Objekts**.
  Dieser kann realisiert werden

  - ▻ durch eine Deklaration der **Iteratorklasse** als **Freundklasse** der **Containerklasse** oder
  - ▻ durch die Bereitstellung einer geeigneten **öffentlichen Zugriffsschnittstelle** durch die **Containerklasse**

  ◇ Häufig wird eine Iteratorklasse als **innere Klasse** der von ihr bearbeiteten Containerklasse definiert.
  Dadurch wird die Zugehörigkeit der Iteratorklasse zur Containerklasse betont.

  ◇ Ein Iterator-Objekt muß wenigsten **zwei Datenkomponenten** besitzen :

  - ▻ Pointer oder Referenz auf das **Container-Objekt**
  - ▻ **aktuelle Position**

  ◇ Im einfachsten Fall ist ein Iterator-Objekt ein **Funktionsobjekt**, d. h. als wesentliche Memberfunktion ist der Funktionsaufruf-Operator überladen.
  Bei der Erzeugung eines Iterator-Objekts wird die aktuelle Position auf den Anfang (erstes Element) gesetzt. Jede Verwendung des Objekts in Form eines Funktionsaufrufs liefert einen Pointer auf das aktuelle Element und setzt die aktuelle Position auf das nächste Element. Liegt die aktuelle Position nach dem letzten Element wird der `NULL`-Pointer zurückgeliefert (Ende der Element-Sequenz ist erreicht!)

Iteratoren in C++ (2)

- **Beispiel : Realisierung einer einfachen Iteratorklasse zu einem assoziativen Array**

```
#define MAXLEN  10

class Assar                        // Assoziatives Array, Implementierung s. Prog.2
{ public :
    Assar(int =MAXLEN);            // Konstruktor
    int& operator[](const char*);  // Indizierungs-Operator
    friend class AssarIter;        // Iteratorklasse
  private :
    struct paar { char *wort; int zahl; } *vec;
    int max;
    int free;
    Assar(const Assar&);            // verhindert Kopieren bei Initialis.
    Assar& operator=(const Assar&); // verhindert Kopieren bei Zuweisung
};

// ---------------------------------------------------------------------------

class AssarIter                     // Iteratorklasse
{ public :
    AssarIter(const Assar&);
    const char* operator()(void);
  private :
    const Assar* m_pclFeld;         // Array, auf dem Iterator arbeitet
    int m_iAct;                     // aktueller Array-Index
    AssarIter(const AssarIter&);            // verhindert Kopieren bei Initialis.
    AssarIter& operator=(const AssarIter&); // verhindert Kopieren bei Zuweisung
};

// ---------------------------------------------------------------------------

#include <cstdlib>

AssarIter::AssarIter(const Assar& fld) : m_pclFeld(&fld), m_iAct(0) {}

const  char* AssarIter::operator()(void)
{ char* hp;
  if (m_iAct<m_pclFeld->free)
  { hp=m_pclFeld->vec[m_iAct].wort;
    m_iAct++;
  }
  else
  { hp=NULL;
    m_iAct=0;
  }
  return hp;
}

// ---------------------------------------------------------------------------

#include <iostream>
using namespace std;

void einlesen(Assar& feld)
{ const int MAXL=256;
  char buffer[MAXL];
  while (cin >> buffer) feld[buffer]++;        // Indizierung über String
}

int main(void)
{ Assar feld;
  einlesen(feld);
  AssarIter next(feld);                        // Iteratorobjekt
  const char* p;
  while(p=next())
    cout << p << " : " << feld[p] << '\n';
  return 0;
}
```

Iteratoren in C++ (3)

- **Implementierung** (2) :

  ◇ Allgemeiner und flexibler einsetzbare Iteratoren **trennen**
  - den **Zugriff** zum jeweils **aktuellen Element**
  - vom **Fortschalten** der **aktuellen Position** auf das nächste Element

  durch die Bereitstellung getrennter geeigneter Memberfunktionen.
  Zusätzlich stellen sie häufig auch eine Memberfunktion zum Vergleich zweier Iteratoren bzw. der von ihnen aktuell referierten Elemente zur Verfügung.

  ◇ Noch universeller und eleganter lassen sich Iteratoren einsetzen, die die **Analogie zum Pointer implementieren**. Hierfür verfügen sie über **geeignete Operatorfunktionen**, i.a. wenigstens über :

  ▻ `operator++(),`

  ▻ `operator--()` (falls auch Rückwärts-Iteration möglich sein soll) `,`

  ▻ `operator*(),`

  ▻ `operator==().`

  Direktzugriffs-Iteratoren überladen darüber hinaus auch

  ▻ den Additionsoperator

  ▻ und den Subtraktionsoperator,

  ▻ gegebenenfalls auch den Indexoperator

  ▻ sowie die übrigen Vergleichsoperatoren.

  Derartige Iteratoren lassen sich wie Pointer, gegebenenfalls einschließlich "Pointer"-Arithmetik, verwenden.

  ◇ **Containerklassen** sind häufig als **Klassen-Templates** definiert.
  Der Typ (die Klasse) der im Container abgelegten Elemente ist in diesem Fall Template-Parameter.
  Eine für die Container-Klasse zuständige **Iteratorklasse** ist dann ebenfalls von dem **Template-Parameter abhängig**. Das bedeutet, dass sie bei einer **Definition außerhalb** der **Containerklasse ebenfalls als Klassen-Template** definiert werden muß.
  Erfolgt ihre Definition dagegen als **innere Klasse** der **Containerklasse** ist **kein Klassen-Template erforderlich**.
  Die Iteratorklasse kann alle Namen – also auch die formalen Typ-Parameter-Namen – des umschließenden Container-klassen-Templates direkt verwenden.

  ◇ Bei einer **pointer-analogen Implementierung** einer Iteratorklasse, existieren in der **zugehörigen Containerklasse** häufig **zwei Memberfunktionen**, die jeweils ein **Iterator-Objekt** als Funktionswert **erzeugen** :

  ▻ Die eine dieser Funktionen (häufig **`begin()`** genannt) liefert ein Iteratorobjekt, das auf das **erste** enthaltene **Element** zeigt.

  ▻ Die andere Funktion (häufig **`end()`** genannt) liefert ein Iteratorobjekt, das auf ein Element zeigt, das **formal unmittelbar hinter dem letzten** enthaltenen **Element** angeordnet ist, tatsächlich aber gar nicht existiert oder ein Dummy-Element ist. Es dient damit zur Kennzeichnung des **Sequenzendes**.

- **Anmerkung zur ANSI-C++-Standardbibliothek**

  ◇ Die **Standard Template Library (STL)**, ein wesentlicher Bestandteil der Standardbibliothek, definiert mehrere **Iteratorarten** (Iteratorkategorien) und **Iteratorklassen** für verschiedene Container-Klassen sowie zur Anwendung auf Stream- und Streambuffer-Klassen

Iteratoren in C++ (4 - 1)

- **Beispiel : Realisierung einer Iteratorklasse mit pointer-analoger Implementierung**
  - ◇ Iteratorklasse zu einem Klassen-Template für Listen. Der Typ der Listenelemente ist Template-Parameter.
  - ◇ Iteratorklasse ist als innere Klasse des Listen-Klassen-Templates definiert
  - ◇ **Definition des Listen-Klassen-Templates `List<T>` sowie der Iteratorklasse `ListIter`** (enthalten in Headerdatei `list.h`)

```
template <class T>
class List
{
  private :
    struct ListEl { ListEl* next; ListEl* prev; T data; };
  public :
    List();
    ~List();
    void insert(const T&);

    // ------------------------------------------------------------------------

    class ListIter                    // eingebettete Iteratorklasse
    {
      public :
        ListIter(List& lst)           { pLst=&lst; pAct=pLst->first; }
        T& operator*()                { return pAct->data; }
        ListIter& operator++()        { pAct=pAct->next; return *this; }
        bool operator==(ListIter& it) { return pAct==it.pAct; }
        friend class List<T>;
      private :
        ListIter& setPos(ListEl* pa)  { pAct=pa; return *this; }
        List*    pLst;
        ListEl*  pAct;
    };

    // ------------------------------------------------------------------------

    friend class ListIter;
    ListIter begin() { return ListIter(*this); }
    ListIter end()   { return ListIter(*this).setPos(last); }
  private :
    ListEl* first;
    ListEl* last;
};


template <class T>               // Konstruktor von List<T>
List<T>::List()
{ ListEl* dummy=new ListEl;      // formales Dummy-Element
  dummy->next=dummy->prev=NULL;  // gehört nicht zur eigentlichen Liste
  first=last=dummy;              // dient zur Kennzeichnung des Listenendes
}

template <class T>
List<T>::~List()
{ /* Zerstörung aller Listenelemente */ }

template <class T>
void List<T>::insert (const T& dat)
{ /* Einfügen eines neuen Listenelementes am Ende (vor das Dummy-Element) */ }
```

Iteratoren in C++ (4 - 2)

- **Beispiel : Realisierung einer Iteratorklasse mit pointer-analoger Implementierung**, Forts.

  ◇ **Beispiel für Anwendercode** (enthalten in Datei `listiter_m.cpp`)
  → Demonstrationsprogramm **`listiter`**

```
#include <iostream>
using namespace std;

#include "list.h"

int main(void)
{ List<int> ilst;

  for (int i=0; i<10; i++)
    ilst.insert(i);

  List<int>::ListIter it1=ilst.begin();
  List<int>::ListIter ite=ilst.end();

  cout << "\nInhalt int-Liste :\n";
  while (!(it1==ite))
  {
    cout << *it1 << "   ";
    ++it1;
  }
  cout << endl;

  List<double> dlst;

  for (i=1; i<=5; i++)
    dlst.insert(i*1.95583);

  List<double>::ListIter dit1=dlst.begin();
  List<double>::ListIter dite=dlst.end();

  cout << "\nInhalt double-Liste :\n";
  while (!(dit1==dite))
  {
    cout << *dit1 << endl;
    ++dit1;
  }
  cout << endl;

  return 0;
}
```

  ◇ **Ausgabe des Demonstrationsprogramms `listiter`**

```
Inhalt int-Liste :
0   1   2   3   4   5   6   7   8   9

Inhalt double-Liste :
1.95583
3.91166
5.86749
7.82332
9.77915
```

Persistenz in C++ (1)

- **Prinzip** :

  ◇ Üblicherweise sind **Objekte flüchtige Entitäten**. Sie werden während des Programmablaufs angelegt und spätestens bei Beendigung des Programms wieder zerstört.

  ◇ Unter **Persistenz** versteht man die Fähigkeit eines Objekts mit seinem zuletzt eingenommenen **Zustand** über das Ende seines verwendenden Programms hinaus **erhalten** zu bleiben.

  ◇ Um dies zu realisieren muss das Objekt auf einem **Hintergrundspeicher** (z.B. in einer Datei oder auch in einer Datenbank) **abgespeichert** werden. Von dort kann es dann bei Bedarf mit seinem alten Zustand wieder **geladen** (eingelesen) werden.

  ◇ Damit lassen sich

  ▻ **Programm-(Teil-)Zustände abspeichern** und zu einem **späteren Zeitpunkt** – z.B. bei einem Programmneustart – wieder **herstellen**,

  ▻ **Objekte** von einem **Programm** (Prozess) **zu einem anderen** – später laufenden – **übertragen.**

- **Probleme** :

  ◇ Die Abspeicherung und das Wiedereinlesen von **Objekten** einer ganz bestimmten Klasse, die nur **Datenkomponenten einfacher Typen** besitzen, ist relativ einfach und **unproblematisch**.

  ◇ **Enthalten** die abzuspeichernden Objekte dagegen als Datenkomponenten **andere Objekte** oder **Verweise** (Pointer bzw. Referenzen) auf andere Objekte, treten – zum Teil nichttriviale – **Probleme** auf :

  ▻ In diesem Fall müssen **auch** die **Komponentenobjekte** und die **referierten Objekte abgespeichert** werden. Dabei muss die gesamte von dem abzuspeichernden Objekt ausgehende **Objektstruktur so abgespeichert** werden, dass sie beim Wiedereinlesen **fehlerfrei wiederhergestellt** werden kann.

  ▻ Das **Abspeichern von Adressen** ist **sinnlos**. Stattdessen muss bei einem Verweis auf ein anderes Objekt eine **logische Information**, die das **referierte Objekt** eindeutig **kennzeichnet**, abgespeichert werden. Dies kann zum Beispiel durch eine **Objekt-ID**, die jedes im Programm erzeugte Objekt bekommt, erfolgen.

  ▻ Ein Objekt, auf das **mehrfach verwiesen** wird, darf sinnvollerweise **nur einmal** – beim ersten Verweis – **abgespeichert** werden. Bei allen weiteren Verweisen auf dasselbe Objekt wird nur die Tatsache, dass auf das Objekt verwiesen wird, abgespeichert.

  ▻ Gegebenenfalls müssen **Verweiszyklen** berücksichtigt werden.

  ▻ Die **Reihenfolge**, in der das ursprüngliche Objekt und die enthaltenen bzw. referierten Objekte abzulegen sind, muss gut durchdacht werden. Sollen die Objekte rein sequentiell getrennt oder gegebenenfalls geschachtelt abgelegt werden ?
  Der beim Einlesen angewendete Algorithmus zur Initialisierungs-Reihenfolge der Objekte muss bereits beim Abspeichern Berücksichtigung finden.

  ▻ Wie wird beim Wiedereinlesen das Auftreten eines **Verweises** auf ein bis dahin **noch nicht eingelesenes Objekt** gelöst ?

  ◇ Sollen Objekte aus einer **polymorphen Klassenhierarchie** abgespeichert werden, tritt ein **zusätzliches Problem** auf : Die – ja durch Basisklassenzeiger (bzw. -referenzen) referierbaren – Objekte müssen beim Wiedereinlesen exakt rekonstruiert werden können, d.h. ihre **tatsächliche Klasse muß erhalten** bleiben. Dies erfordert die zusätzliche Ablage von **Typ-Informationen** (z.B. Typ-ID oder Typ-Name) zu jedem Objekt.

  ◇ Bei der Implementierung von Persistenz muss auch die **Organisationsform des Hintergrundspeichers**, in dem die Ablage erfolgt, berücksichtigt werden. Drei Organisationsformen finden hauptsächlich Anwendung : rein sequentielle Streams, wahlfrei positionierbare Dateien, Datenbanken.
  Meist erfolgt die Ablage in **sequentiellen Streams**, durch die ja auch Dateien beschrieben werden.

Persistenz in C++ (2)

- **Anmerkungen zur Realisierung** :

  ◇ **Persistenz** lässt sich **nicht** einfach "**von aussen**" auf beliebige Objekte anwenden.
  Vielmehr muss Persistenz als **besondere Eigenschaft innerhalb** der jeweiligen **Klasse implementiert** werden.

  ◇ Sinnvollerweise sollte die Implementierung so erfolgen, dass **jedes Objekt** für das Abspeichern bzw. Wiedereinlesen seiner **eigenen Komponenten selbst zuständig** ist.
  Das bedeutet insbesondere, dass die Komponenten enthaltener bzw. referierter Objekte von diesen – und nicht vom umfassenden Objekt – gespeichert bzw. wiedereingelesen werden sollten.

  ◇ Wegen der o.a. Probleme ist es **sehr schwierig**, einen **universellen** alle möglichen Anwendungsfälle abdeckenden **Persistenz-**Speicher bzw. Einlese-**Algorithmus** zu realisieren.
  In der Praxis wird man daher immer eine "spezielle" Persistenz implementieren, die stark von der zu bearbeitenden Objektstruktur beeinflusst ist.

  ◇ Das **Abspeichern** erfolgt i.a. mittels geeigneter **Memberfunktionen**
  Sollen **Objekte unterschiedlicher Klassen** persistent verwendbar sein, ist es sinnvoll, die für die Realisierung des jeweiligen Abspeicher-Algorithmus benötigten **generellen Funktionen**, in einer – gegebenenfalls abstrakten – **Klasse zusammenzufassen**. Diese stellt damit ein allgemeines **Persistenz-Interface** zur Verfügung und kann dann als **Basisklasse** für alle Klassen mit der gewünschten Persistenz-Eigenschaft dienen. **Klassenspezifische Details** bzw. Besonderheiten werden durch jeweilige **klassenspezifische** – häufig virtuelle – **Funktionen** implementiert.
  Häufig findet sich eine derartige Implementierung in Klassenbibliotheken, die **Persistenz als Konzept** unterstützen.

  ◇ Die Implementierung des **Wiedereinlesens** (Ladens) **von Objekten** ist **komplexer** und damit i.a. **schwieriger** als die Implementierung des Abspeicherns.
  Grundsätzlich muss beim Laden ein **neues Objekt angelegt** und mit den abgespeicherten **Komponentenwerten initialisiert** werden.

  Prinzipiell bieten sich hierfür **zwei Wege** an :

  - **Erzeugung** eines "**leeren**" **Objekts**, das dann die **Initialisierungswerte** mittels einer geeigneten **Memberfunktion** vom Speichermedium **einliest**.
    Hierfür sollte sinnvollerweise ein Default-Konstruktor existieren, was nicht immer der Fall bzw. sinnvoll ist.
  - **Erzeugung** eines gleich "**richtig**" **initialisierten Objekts** mittels eines geeigneten **Konstruktors**, dem eine Referenz auf das Speichermedium als Parameter übergeben wird.
    Bei sequentiellen Streams als – häufigste – Organisationsform des Hintergrundspeichers wird dieser Parameter i.a. eine Referenz auf ein `istream`-Objekt sein.
    Diese Methode wird meist bevorzugt verwendet.

  ◇ **Objektkomponenten**, die von einem **einfachen Datentyp** sind, werden **im Konstruktorrumpf eingelesen**.

  ◇ Bei **Objekten von abgeleiteten Klassen** und **Objekten mit anderen Objekten als Komponenten** erfolgt ein **verketteter** Konstruktoraufruf. Sinnvollerweise werden in derartigen Fällen die **Basisklassen-Teilobjekte** und die **Komponentenobjekte** durch **geeignete Konstruktoraufrufe** initialisiert, die in **Initialisierungslisten** angegeben werden.
  Die **Reihenfolge,** in der die Initialisierungen mittels einer Initialisierungsliste erfolgen, ist allerdings nicht frei wählbar, sondern durch die Abstammungsliste sowie die Komponentendeklaration in der Klassendefinition festgelegt.
  Das bedeutet, dass die entsprechenden Komponentenwerte in dieser Reihenfolge auch abgespeichert werden müssen.

  ◇ Das Laden von Objekten, die **Verweise auf andere Objekte** enthalten, ist **komplizierter**. Insbesondere dann, wenn es sich bei den referierten Objekten um Objekte einer **polymorphen Klassenhierarchie** handelt und die Referierung über Basisklassenpointer (bzw –referenzen) erfolgt.
  Direkt werden in derartigen Fällen nur Objekt-IDs und Klassenkennungen (z.B. Klassennamen) abgespeichert. Die eigentlichen Objektwerte werden i.a. später (oder gegebenenfalls auch früher) außerhalb des referierenden Objekts abgelegt.
  Eine Erzeugung der referierten Objekte und die Ablage ihrer Adressen im referierenden Objekt erfordert **zwei globale Tabellen** :

  - Eine Tabelle, die allen **Klassenkennungen** (z.B. Klassennamen) eine **statische Objekterzeugungsfunktion** zuordnet, die ihrerseits den jeweils richtigen Konstruktor aufruft, mit dem das referierte Objekt eingelesen werden kann.
  - Eine Tabelle, in der zu jeder eingelesenen **Objekt-ID** die **Adresse des zugehörigen Objekts** enthalten ist.

Realisierungsbeispiel zur Persistenz in C++ (1)

• Implementierungseigenschaften und -prinzipien

◇ Realisierung einer Persistenz mit folgenden Eigenschaften :

- ▻ Abspeichern von Objekten in sequentiellen Streams
- ▻ Abspeichern einer Objekt-ID und des Klassennamens (als Klassenkennung) für jedes zu speichernde Objekt
- ▻ Einfache Datenkomponenten werden unmittelbar nach Objekt-ID und Klassenname abgespeichert.
- ▻ Datenkomponenten, die selbst Objekte sind, werden direkt geschlossen (mit Objekt-ID und Klassennamen sowie den einzelnen Komponentenwerten) innerhalb des umschliessenden Objekts abgespeichert.
- ▻ Für Datenkomponenten, die Verweise auf andere Objekte sind, wird lediglich die Objekt-ID und der Klassenname als Referenz abgespeichert.
  Die Ausgabe des referierten Objekts erfolgt erst nach der Abspeicherung des referierenden Objekts.
- ▻ Mehrfach vom gleichen Objekt referierte Objekte werden nur einmal abgespeichert.
  Objekte, die von verschiedenen Objekten referiert werden, würden dagegen auch mehrfach abgespeichert werden.

◇ Zusammenfassung der Rahmenfunktionalität zur Realisierung dieser Persistenz in einer Klasse **`Streamable.`**
Diese Klasse dient als **Basisklasse** für alle Klassen, die diese Persistenz implementieren sollen.

- ▻ Sie definiert die **Datenkomponenten** zur Aufnahme der **Objekt-ID** und des **Klassennamens**.
- ▻ Weiterhin enthält sie mehrere **statische Datenkomponenten**, die zur Realisierung des implementierten Persistenz-Algorithmus benötigt werden. U.a. sind dies je eine **Liste** von Verweisen auf die jeweils zu **speichernden** bzw die bereits **geladenen Objekte**, sowie eine **Klassen-Liste** mit Pointern auf die für das Laden von **polymorphen Objekten** benötigten statischen **Objekterzeugungsfunktionen**.
- ▻ Zum **Abspeichern** stellt die Klasse die folgenden **generellen Funktionen** zur Verfügung :
  - **`void streamOut(ostream&)`**
    zentrale Verwaltungs- Funktion, die zum Abspeichern eines Objekts aufzurufen ist.
    Die Funktion sorgt dafür, dass bei Datenkomponenten, die Verweise auf andere Objekte sind, zunächst nur die Objekt-ID und der Klassenname des referierten Objekts abgespeichert wird und das später abzuspeichernde Objekt in die Liste der noch zu speichernden Objekte (Speicherliste) eingetragen wird. Zum Abspeichern jedes in der Speicherliste eingetragenen Objekts ruft sie ihrerseits die Funktion `outstreamObject()` auf.
  - **`void outstreamObject(ostream&)`**
    Diese Funktion dient zum eigentlichen Abspeichern eines Objekts.
    Sie gibt zunächst die Objekt-ID und den Klassennamen des Objekts aus (mit **`outstreamID(...)`**) und ruft dann die virtuelle Funktion `doStream()` auf, die für das Abspeichern der Datenkomponenten zuständig ist.
- ▻ Die jeweilige **klassenspezifische Besonderheit**, d. h. das **Abspeichern** der eigentlichen **Datenkomponenten** eines abgeleiteten Objekts wird durch die **virtuelle Funktion `void doStream(ostream&)`** implementiert.
  Für die Klasse `Streamable` besitzt diese Funktion eine leere Funktionalität.
  Sie ist die **einzige** Funktion, die zur Implementierung der **Ausgabefunktionalität** der Persistenz in einer **abgeleiteten Klasse definiert** werden muss.

◇ Zum **Laden von Objekten** müssen für die verschiedenen persistenten Klassen jeweils definiert werden :

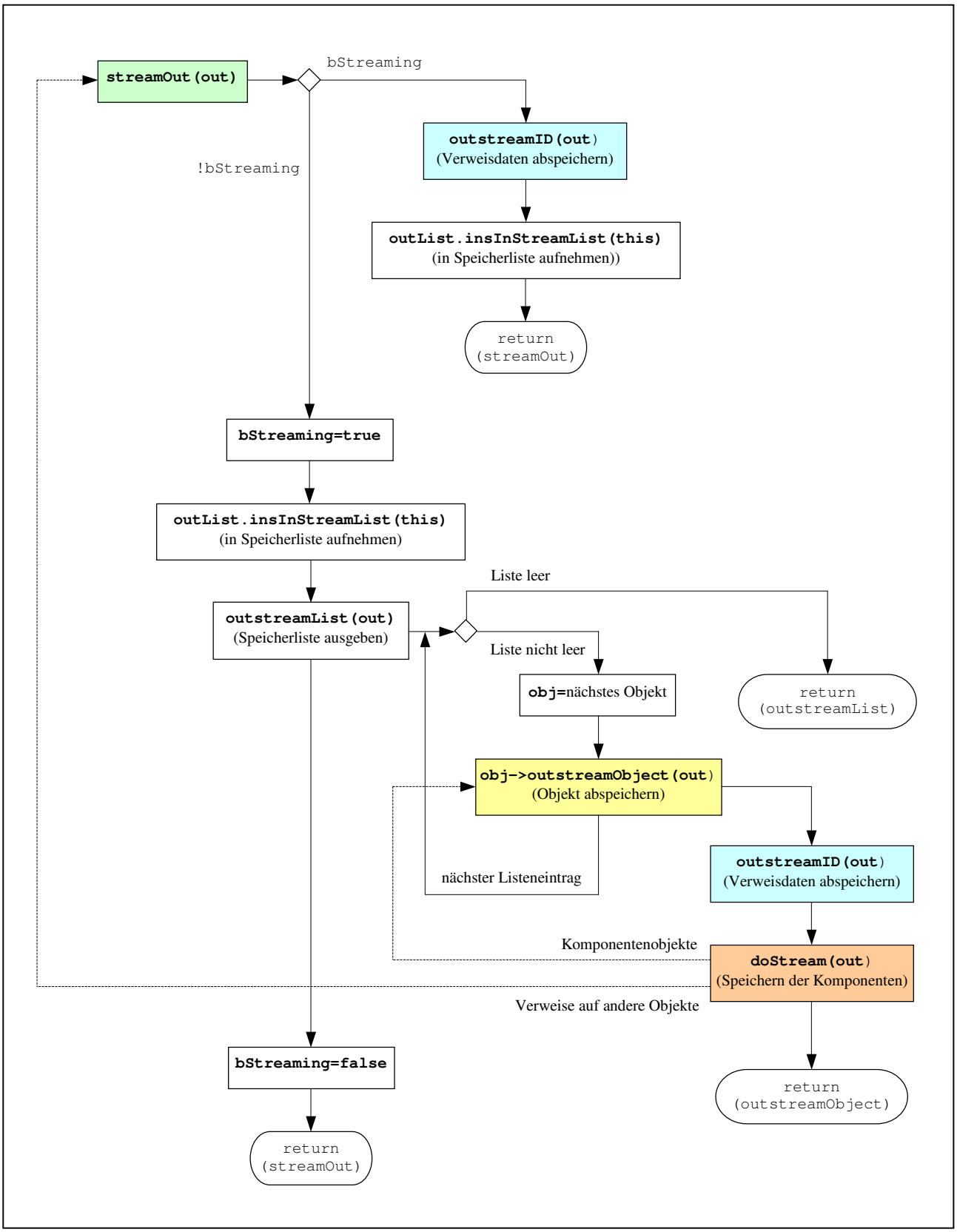
- ▻ geeigneter **Konstruktor** mit einem Parameter vom Typ `istream&` ("Lade"-Konstruktor).
  Dieser muss in einer Initialisierungsliste die "Lade"-Konstruktoren seiner persistenten Basisklassen und Komponentenklassen aufrufen, bei direkter Ableitung von `Streamable` also auch den "Lade"-Konstruktor dieser Klasse.
- ▻ statische **Objekterzeugungsfunktion**, wenn auch Verweise auf Objekte dieser Klasse abgespeichert werden sollen

◇ Wesentliche Funktionen der Klasse `Streamable` zur Implementierung der Lade-Funktionalität sind :

- ▻ **"Lade"-Konstruktor** (Parameter vom Typ `istream&`)
- ▻ **`static Streamable* refObject(istream& in, unsigned long id)`**
  Diese Funktion wird in den "Lade"-Konstruktoren von Objekten, die Referenzen auf andere – üblicherweise polymorphe – Objekte enthalten, benötigt. Sie ermittelt die **Adresse** des durch die **Objekt-ID `id` bestimmten Objekts**. Hierfür durchsucht sie die Liste der geladenen Objekte. Ist das Objekt noch nicht geladen, liest sie solange weitere Objekte ein, bis das gesuchte Objekt gefunden ist.
  Das Laden eines – polymorphen – Objekts erfolgt mit dem "Lade-"Konstruktor von `Streamable,` der ja auch den Klassennamen einliest, und der klassenspezifischen Objekterzeugungsfunktion, die aus der Klassen-Liste über den Klassennamen ermittelt wird.

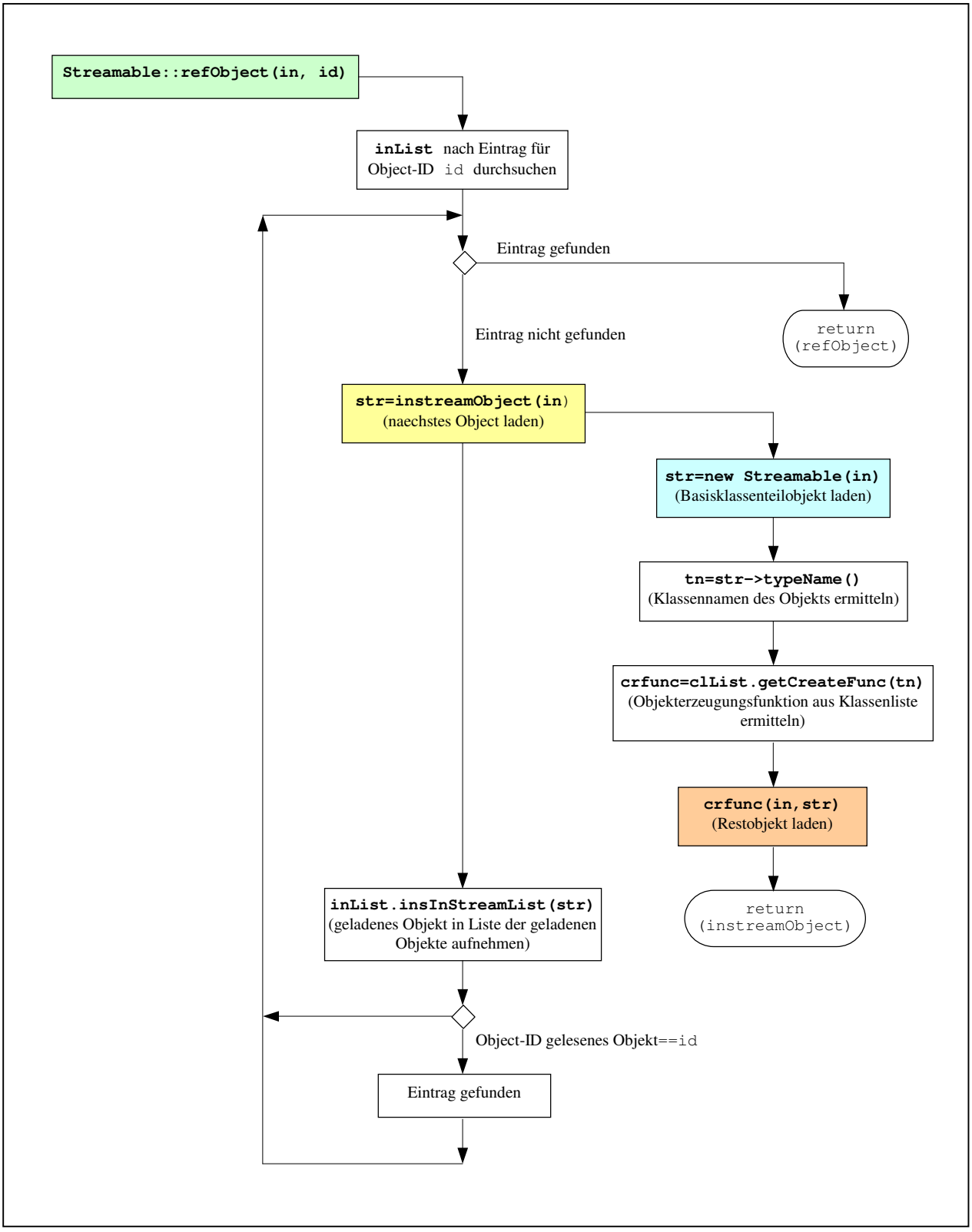
Realisierungsbeispiel zur Persistenz in C++ (2)

- **Abspeicher-Algorithmus**



Realisierungsbeispiel zur Persistenz in C++ (3)

- **Algorithmus zum Laden von Objekten, die durch andere Objekte referiert werden**



Realisierungsbeispiel zur Persistenz in C++ (4)

- **Definition der Klasse `Streamable`**

```
#include <iostream>
using namespace std;

#define START_ID 1001
#define LIST_SIZE 10
#define MAX_NAME_LEN 256

class Streamable
{ public :
    virtual ~Streamable() { };
    void streamOut(ostream&);
    virtual void doStream(ostream&){};
    void outstreamObject(ostream&);
    unsigned long getID() { return uObjId; }
    const char* typeName();
    static void emptyAllStreamLists();
  protected :
    class ClassList;
    class StreamObjectList;
    Streamable();
    Streamable(istream&, Streamable* =NULL);
    void chkName();
    static unsigned long instreamID(istream&);
    static const char* instreamName(istream&);
    static Streamable* refObject(istream&, unsigned long);
    static ClassList clList;                    // Liste aller streamable-Klassen
    static StreamObjectList inList;             // Liste der geladenen Objekte
    static StreamObjectList outList;            // Liste der auszugebenden Objekte
  private :
    unsigned long uObjId;                       // Objekt-ID
    const char* cpName;                         // Ablage des Typ-Namens
    static unsigned long uMaxId;                // aktuelle max. Objekt-ID +1
    static bool bStreaming;                     // Speichervorgang findet statt
    static int iNumListOut;                     // Anzahl der bereits gespeicherten
                                                // Objekte aus Speicher-Liste
    void outstreamID(ostream&);
    static void outstreamList(ostream&);
    static Streamable* instreamObject(istream&);
};

class Streamable::ClassList                     // Liste aller streamable-Klassen
{ public :
    typedef Streamable* (*CreateFunc)(istream&, Streamable*);
    ClassList(unsigned = LIST_SIZE);
    // Destruktor und Memberfunktion insert(...)
    CreateFunc getCreateFunc(const char*);    // Ermittlung der create-Funktion
  private :
    const char** names;                       // Klassennamen
    CreateFunc* createfuncs;                  // statische create-Funktionen
    // Datenkomponenten uSize und uAnz
};

class Streamable::StreamObjectList
{ public :
    StreamObjectList(unsigned = LIST_SIZE);
    // Destruktor und Memberfunktionen contains(...) und emptyStreamList()
    void insInStreamList(Streamable*);              // Objekt in Liste einfügen
    Streamable* entry(unsigned i) const;            // i-ter Listeneintrag
    unsigned getAnz() const { return iNumInList; }  // Anzahl der Listeneinträge
  private :
    Streamable** objects;                           // Objekte in der Liste
    // Datenkomponenten uListSize und uNumInList und Memberfunktion resizeList(...)
};
```

Realisierungsbeispiel zur Persistenz in C++ (5)

- **Implementierung der Klasse `Streamable`, 1. Teil (Funktionalität zum Abspeichern)**

```
#include "streamable.h"
#include "myexception.h"
#include <cstring>

// ---------------------------------------------------------------------------

unsigned long Streamable::uMaxId = START_ID;

bool Streamable::bStreaming = false;

int Streamable::iNumListOut = 0;

Streamable::ClassList Streamable::clList;

Streamable::StreamObjectList Streamable::inList;

Streamable::StreamObjectList Streamable::outList;

// ---------------------------------------------------------------------------

Streamable::Streamable()                              // protected
{ uObjId=uMaxId++;
  cpName=NULL;
}

void Streamable::streamOut(ostream& out)
{ if (!bStreaming)
  { bStreaming=true;
    outList.emptyStreamList();
    iNumListOut=0;
    outList.insInStreamList(this);
    outstreamList(out);
    bStreaming=false;
  }
  else
  { outstreamID(out);
    outList.insInStreamList(this);
  }
}

void Streamable::emptyAllStreamLists()                // static
{ inList.emptyStreamList();
  outList.emptyStreamList();
}

void Streamable::outstreamList(ostream& out)          // private und static
{ Streamable* obj;
  while (iNumListOut != outList.getAnz())
  { obj=outList.entry(iNumListOut++);
    out << endl;
    obj->outstreamObject(out);
  }
}

void Streamable::outstreamObject(ostream& out)
{ outstreamID(out);
  doStream(out);
}

const char* Streamable::typeName()
{ if (cpName==NULL)
    cpName=typeid(*this).name();
  return cpName;
}

void Streamable::outstreamID(ostream& out)            // private
{  out << uObjId << ' ' << typeName() << endl;
}
```

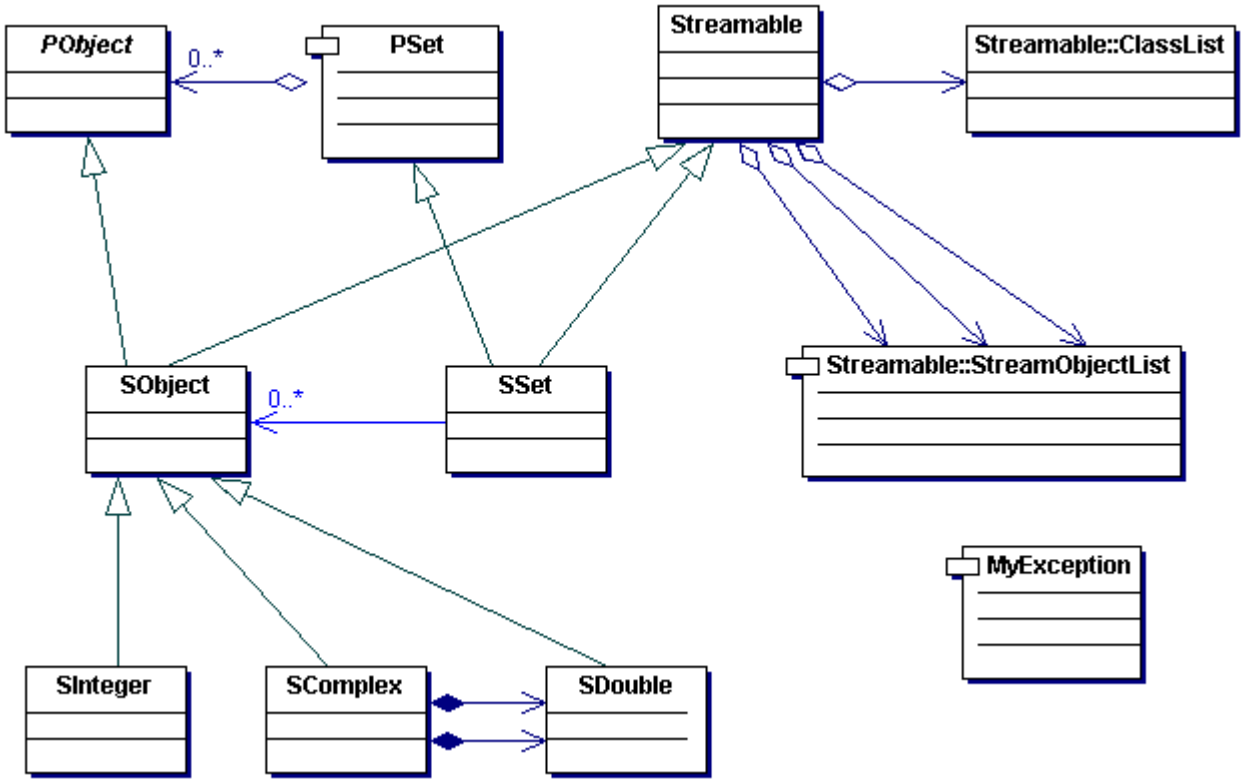
Realisierungsbeispiel zur Persistenz in C++ (6)

- **Implementierung der Klasse `Streamable`, 2. Teil (Funktionalität zum Laden)**

```
Streamable::Streamable(istream& in, Streamable* st)   // protected
{ if (st==NULL)
  { uObjId=instreamID(in);
    cpName=instreamName(in);
  }
  else
  { uObjId=st->uObjId;
    cpName=st->cpName;
    delete st;
  }
}

unsigned long Streamable::instreamID(istream& in)     // protected und static
{ unsigned long id;
  in >> id;
  if (uMaxId<=id)
    uMaxId=id+1;
  return id;
}

const char* Streamable::instreamName(istream& in)     // protected und static
{ char buff[MAX_NAME_LEN];
  in.getline(buff, MAX_NAME_LEN);
  char* nam=buff;
  while (*nam==' ') nam++;
  char* cpHilf=new char[strlen(nam)+1];
  strcpy(cpHilf, nam);
  return cpHilf;
}

void Streamable::chkName()                            // protected
{ if (strcmp(cpName, typeid(*this).name()))
    throw MyException("falscher Typname !");
  else
  { delete [] const_cast<char*>(cpName);
    cpName=typeid(*this).name();
  }
}

Streamable* Streamable::refObject(istream& in, unsigned long id)  // protected
{ Streamable* found=NULL;                                         // und static
  unsigned i=0;
  while (found==NULL && i<inList.getAnz())
    if (inList.entry(i)->getID()==id)
      found=inList.entry(i);
    else
      i++;
  while (found==NULL)
  { Streamable* str=instreamObject(in);
    inList.insInStreamList(str);
    if (str->getID()==id)
      found=str;
  }
  return found;
}

Streamable* Streamable::instreamObject(istream& in)   //private und static
{ Streamable* str=new Streamable(in);
  ClassList::CreateFunc crfunc = clList.getCreateFunc(str->typeName());
  if (crfunc!=NULL)
    str=crfunc(in, str);
  return str;
}
```

Anwendung des Realisierungsbeispiels zur Persistenz in C++ (1)

- **Klassendiagramm zum Realisierungsbeispiel**



Anwendung des Realisierungsbeispiels zur Persistenz in C++ (2)

- **Definition von polymorphen persistenten Klassen**

```
#include <iostream>
using namespace std;

class PObject
{ public:
    virtual ~PObject() {};
   // weitere Memberfunktionen
    virtual bool equals(const PObject& a) const = 0;
    virtual void output(ostream& out) const = 0;
    virtual void input(istream& in) = 0;
     virtual PObject& operator=(const PObject&) =0;
};
```

```
#include "PObject.h"
#include "streamable.h"

class SObject : public PObject, public Streamable
{ public :
    SObject() {};
    SObject(istream& in, Streamable* st=NULL) : Streamable(in, st) {};
};
```

```
#include "sobject.h"

class SDouble : public SObject
{ public :
    SDouble(double d=0);
    SDouble(const SDouble&);
    SDouble(istream&, Streamable* =NULL);             // für Persistenz (Laden)
    // weitere Memberfunktionen als Erbe von PObject und eigener Funktionalität
    virtual void doStream(ostream&);                  // für Persistenz (Abspeich.)
    static Streamable* create(istream&, Streamable*); // für Persistenz (Laden)
  private :
    double m_dVal;
};
```

```
#include "sobject.h"
#include "sdouble.h"

class SComplex : public SObject
{ public :
    SComplex(SDouble=0.0, SDouble=0.0);
    SComplex(const SComplex&);
    SComplex(istream&, Streamable* =NULL);            // für Persistenz (Laden)
    // weitere Memberfunktionen als Erbe von PObject und eigener Funktionalität
    virtual void doStream(ostream&);                  // für Persistenz (Abspeich.)
    static Streamable* create(istream&, Streamable*); // für Persistenz (Laden)
  private :
    SDouble m_cReal;
    SDouble m_cImag;
};
```

Anwendung des Realisierungsbeispiels zur Persistenz in C++ (3)

- **Definition einer persistenten Mengenklasse (Mengen-Objekte enthalten Verweise auf andere Objekte)**

```
#include "pobject.h"

#define START_SIZE 20

class PSet
{ public :
    PSet(unsigned = START_SIZE);
    ~PSet();
    PSet& insert(PObject&);
    PSet& remove(PObject&);
    int contains(PObject&) const;
    PObject* element(unsigned i);
    unsigned getAnz() const { return m_uAnz;}
    void clear();
    void resize(unsigned);
  private :
    PObject** m_pMembers;
    unsigned m_uSize;
    unsigned m_uAnz;
};
```

```
#include <iostream>
using namespace std;

#include "pset.h"
#include "streamable.h"
#include "sobject.h"

#define START_SIZE 20

class SSet : public PSet, public Streamable
{ public :
    SSet(unsigned = START_SIZE);
    SSet(istream&);
    SSet& insert(SObject&);
    SSet& remove(SObject&);
    int contains(SObject&) const;
    void doStream(ostream&);
};
```

Anwendung des Realisierungsbeispiels zur Persistenz in C++ (4)

- **Auszugsweise Implementierung der persistenten Klassen**

```
#include "SDouble.h"

void SDouble::doStream(ostream& out)
{ out << m_dVal << endl;
}

SDouble::SDouble(istream& in, Streamable* st) : SObject(in, st)
{ chkName();
  in >> m_dVal;
}

Streamable* SDouble::create(istream& in, Streamable* st)
{ return new SDouble(in, st);
}
```

```
#include "scomplex.h"

void SComplex::doStream(ostream& out)
{ m_cReal.outstreamObject(out);
  m_cImag.outstreamObject(out);
}

SComplex::SComplex(istream& in, Streamable* st)
  : SObject(in, st), m_cReal(in), m_cImag(in)
{ chkName();
}

Streamable* SComplex::create(istream& in, Streamable* st)
{ return new SComplex(in, st);
}
```

```
#include "sset.h"

void SSet::doStream(ostream& out)
{ unsigned anz=getAnz();
  out << anz << endl;
  for (unsigned i=0; i<anz; i++)
    (dynamic_cast<Streamable*>(element(i)))->streamOut(out);
}

SSet::SSet(istream& in) : Streamable(in), PSet(0)
{ chkName();
  unsigned anz;
  in >> anz;
  resize(anz);
  unsigned long* ids=new unsigned long[anz];
  for (unsigned i=0; i<anz; i++)
  { ids[i]=instreamID(in);
    delete [] const_cast<char*>(instreamName(in));
  }
  for (i=0; i<anz; i++)
  { insert(dynamic_cast<SObject&>(*refObject(in, ids[i])));
  }
  delete[] ids;
}
```

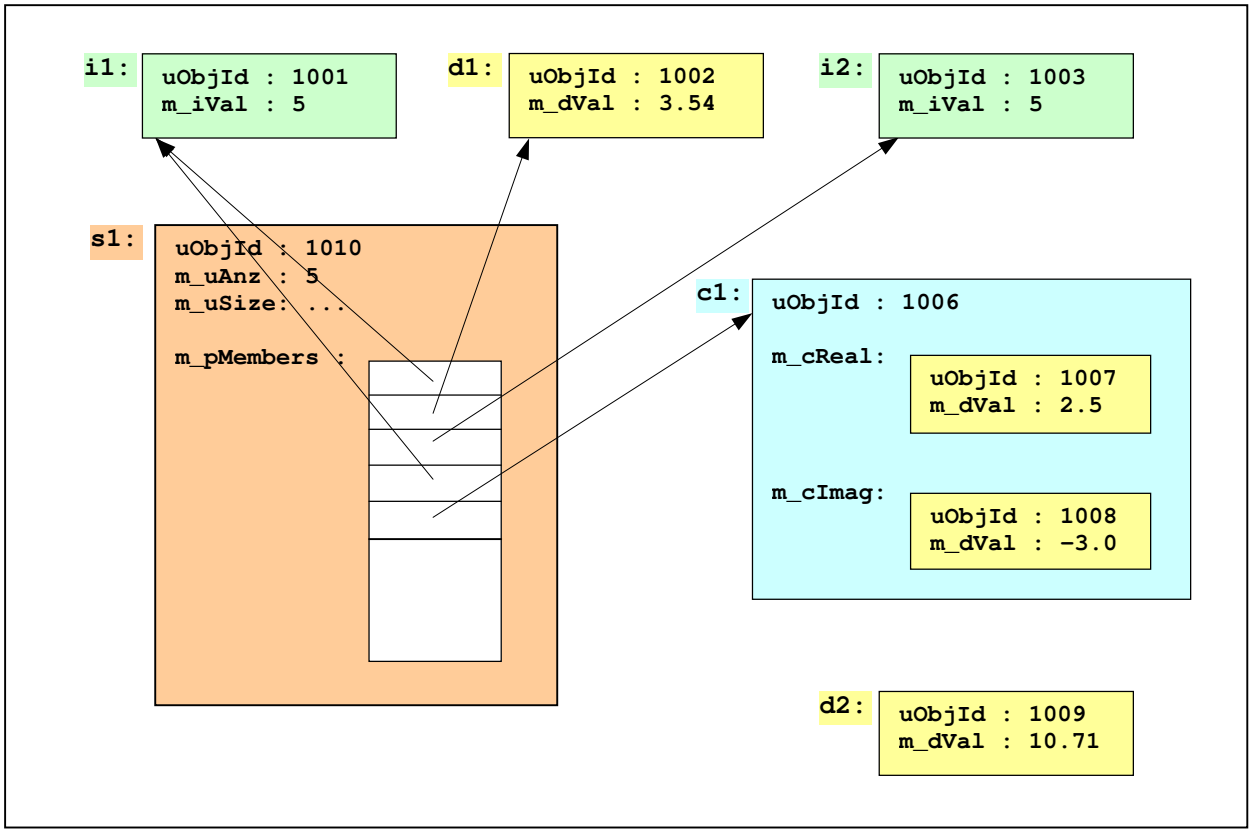
Anwendung des Realisierungsbeispiels zur Persistenz in C++ (5)

- **Implementierung eines kleinen Testprogramms (Datei `persistex_m.cpp`)**

```
#include <fstream>
using namespace std;

#include "sset.h"
#include "sinteger.h"
#include "sdouble.h"
#include "scomplex.h"
#include "myexception.h"

// ---------------------------------------------------------------------------

void streamOutTest(ostream& out)
{ SInteger i1(5);
  SDouble d1(3.54);
  SInteger i2(i1);
  SComplex c1(2.5, -3.0);
  SDouble d2(10.71);
  SSet s1;
  s1.insert(i1);
  s1.insert(d1);
  s1.insert(i2);
  s1.insert(i1);
  s1.insert(c1);
  d2.streamOut(out);
  s1.streamOut(out);
}

// ---------------------------------------------------------------------------

void streamInTest(istream& in)
{ Streamable::emptyAllStreamLists();
  SDouble d2(in);
  SSet s1(in);
  d2.streamOut(cout);          // Ausgabe zu Testzwecken in Standardausgabe
  s1.streamOut(cout);          // Ausgabe zu Testzwecken in Standardausgabe
}

// ---------------------------------------------------------------------------

int main(void)
{ int ret=0;
  ofstream out("save.dat");
  streamOutTest(out);
  out.close();
  try
  { ifstream in("save.dat");
    streamInTest(in);
  }
  catch(MyException e)
  { err << endl << e.getReason() << endl;
    ret=1;
  }
  return ret;
}
```

Anwendung des Realisierungsbeispiels zur Persistenz in C++ (6)

- **In der Funktion `streamOutTest()` erzeugte Objekte**



- **Abgespeicherte Objekte (Inhalt der Sicherungsdatei `save.dat`)**

```
1009 class SDouble
10.71

1010 class SSet
5
1001 class SInteger
1002 class SDouble
1003 class SInteger
1001 class SInteger
1006 class SComplex

1001 class SInteger
5

1002 class SDouble
3.54

1003 class SInteger
5

1006 class SComplex
1007 class SDouble
2.5
1008 class SDouble
-3
```