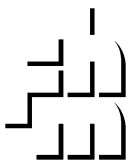
Objektorientiertes Programmieren

mit C++

für Fortgeschrittene

Unterlagen zur Lehrveranstaltung "Programmieren 3"

R. Thomas



Objektorientierte Programmieren mit C++

für Fortgeschrittene

Kapitel-Überblick

Programmieren 3 - Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Überblick

1. Ergänzungen zu Klassen und Objekten

1.1. Pointer auf Klassenkomponenten
1.2. Unions als Klassen
1.3. Innere und lokale Klassen

2. Ergänzungen zum Überladen von Operatoren

2.1. Increment- u. Decrement-Operator
2.2. Funktionsaufruf-Operator
2.3. Delegations-Operator (`->`)
2.4. Typkonvertierung

3. Mehrfachvererbung

3.1. Eigenschaften und Problematik
3.2. Virtuelle Basisklassen

4. Ergänzungen zur Laufzeitpolymorphie

4.1. Abstrakte Klassen
4.2. Laufzeittypinformation
4.3. Typkonvertierungs-Operator `dynamic_cast`

5. Funktions- und Klassen-Templates

5.1. Generische Funktionen (Funktions-Templates)
5.2. Generische Klassen (Klassen-Templates)

6. Ausnahmebehandlung (*Exception Handling*)

6.1. Allgemeines
6.2. Werfen und Fangen von Exceptions
6.3. Beispiele

7. Entwicklung von OOP-Programmen

7.1. Entwicklungsprozeß
7.2 Modellierung (UML)

8. Einige Klassen für spezielle Design-Zwecke

8.1 Smart-Pointer
8.2. Interface-Klassen
8.3. Handle-Klassen und Referenzzählung
8.4. Proxy-Klassen
8.5. Iteratoren
8.6. Persistenz

9. Entwurfsmuster (*Design Pattern*)

9.1. Allgemeines und Überblick
9.2. Beispiele

10. Ausgewählte Komponenten der Standardbibliothek

10.1. Überblick über die Standardbibliothek
10.2. Standard-Exception-Klassen
10.3. Auto-Pointer
10.4. Datentyp für Wertepaare
10.5. Strings
10.6. Funktionsobjekte

11. Standard-Template-Library (STL)

11.1. Überblick
11.2. Container
11.3. Iteratoren
11.4. Algorithmen

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Literaturhinweise

(1) Bjarne Stroustrup
The C++ Programming Language
(deutsch : **Die C++ Programmiersprache**)
Addison Wesley Publishing Company

(2) International Standard ISO/IEC 14882:1998
Programming languages – C++
American National Standards Institute

(3) H.M. Deitel / P.J. Deitel
C++ How to Program
Prentice Hall

(4) Nicolai Josuttis
Objektorientiertes Programmieren in C++
Addison Wesley Publishing Company

(5) Ulla Kirch-Prinz / Peter Prinz
C++ Lernen und professionell anwenden
mitp Verlag

(6) Nicolai Josuttis
Die C++-Standardbibliothek
Addison Wesley Publishing Company

(7) Rolf Isernhagen
Softwaretechnik in C und C++
Hanser-Verlag

(8) Ulrich Breymann
C++ Einführung und professionelle Programmierung
Hanser-Verlag

(9) E. Gamma / R. Helm / R. Johnson / J. Vlissides
Design Patterns
Addison Wesley Publishing Company

(10) M. Fowler / K. Scott
UML konzentriert
Addison Wesley Publishing Company

(11) Terry Quatrani
Visual Modelling with Rational Rose and UML
Addison Wesley Publishing Company

(12) P. Stevens / R. Pooley
UML Softwareentwicklung mit Objekten und Komponenten
Pearson Studium

(13) Horst A. Neumann
Objektorientierte Softwareentwicklung
mit der Unified Modelling Language
Hanser-Verlag

(14) Mario Jeckle u.a.
UML2 glasklar
Hanser-Verlag

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 1

1. Ergänzungen zu Klassen und Objekten

1.1. Pointer auf Klassenkomponenten

1.2. Unions als Klassen

1.3. Innere und lokale Klassen

Pointer auf Klassenkomponenten in C++ (1)

- **Komponentenzeiger**
  - ◇ C++ kennt auch den abgeleiteten Typ "**Pointer auf Klassenkomponenten eines bestimmten Typs**" (→ **Komponentenzeiger,** *pointer to class ,member*).
  - ◇ Komponentenzeiger sind an eine **Klasse** und einen **Komponententyp** gebunden. Sie sind sowohl für **Datenkomponenten** als auch für **Funktionskomponenten** (Member-Funktionen) möglich.
  - ◇ **Werte** eines Komponentenzeiger-Typs identifizieren Komponenten eines bestimmten Typs innerhalb eines Objekts einer bestimmten Klasse. Es handelt sich bei diesen "Adressen" **nicht** um normale **Speicher-Adressen**, sondern um eine **Art Index** in eine – gedachte – Tabelle der Komponenten. **Zeiger auf Datenkomponenten** können als **Offset** innerhalb eines Objekts der entsprechenden Klasse aufgefasst werden.
  - ◇ Komponentenzeiger können **nicht** auf **statische Komponenten** zeigen.

- **Vereinbarung von Komponentenzeiger-Typen und -Variablen**
  - ◇ Zusätzliche Angabe des **Scope Resolution Operators** zusammen mit dem **Klassennamen** :
    - ▻ Typangabe für einen **Pointer auf Datenkomponenten** :

      `komponententyp klassenname::*`
    - ▻ Typangabe für einen **Pointer auf Funktionskomponenten** :

      `funktionstyp (klassenname::* )(parameterliste)`
  - ◇ **Beispiel :**

```
class Complex
{ public:
    Complex(float, float);
    Complex mal(const Complex& x);
    Complex plus(const Complex& x);
    float betrag(void);
    float re, im;
};
// ...

float Complex::*pfcomp;         // pfcomp ist Zeiger auf eine Komponente
                                // vom Typ float der Klasse Complex

Complex (Complex::*pcfunc)(const Complex&);
                              // pcfunc ist Zeiger auf eine Member-Funktion der
                              // Klasse Complex mit einer Complex-Referenz als
                              // Parameter und dem Funktionstyp Complex
```

  - ◇ Mittels **`typedef`** kann auch ein **Typname** für einen Komponentenzeiger-Typ vereinbart werden, der dann für Variablenvereinbarungen nutzbar ist.

    Beispiel :
```
typedef float (Complex::*PMFF)(void);
// PMFF ist ein Komponentenzeiger-Typ
```

- **Initialisierung von und Wertzuweisung an Komponentenzeigervariablen**
  - ◇ Es dürfen nur die "**Adressen**" **von Komponenten**, die vom **entsprechenden Typ** sind und auf die auch **Zugriff besteht**, zugewiesen werden (`public` bzw innerhalb Member- oder Freund-Funktionen)
    Auch hierbei ist dem **Komponentennamen** der **Scope Resolution Operator** zusammen mit dem **Klassennamen** voranzustellen.
  - ◇ **Beispiel :**
```
int main(void)
{ PMFF pffunc = &Complex::betrag;
  pfcomp = &Complex::re;
  pcfunc = &Complex::plus;
  //...
}
```

Pointer auf Klassenkomponenten in C++ (2)

- **Die Komponentenauswahloperatoren `.*` und `->*`**

  ◇ Nach Zuweisung einer Komponenten-"Adresse" an eine Komponentenzeigervariable kann über diese zu der **Komponente zugegriffen werden**.
  Dies ist aber nur in der **Bindung an ein konkretes Objekt** der Klasse möglich.
  Hierzu dienen die speziellen **Komponentenauswahl-Operatoren** `.*` und `->*`

  ◇ Beide Operatoren liegen in der **Priorität** zwischen den Multiplikations-Operatoren und den unären Operatoren.

  ◇ Der **Operator** `.*` wird auf **Objekte** angewendet, während der **Operator** `->*` auf **Objektzeiger** angewendet wird :

  ▻ Zugriff zu **Datenkomponenten** : **`objekt.*datenkomp_zeiger`**
  `objektzeiger->*datenkomp_zeiger`

  ▻ Zugriff zu **Funktionskomponenten**: **`(objekt.*funktionskomp_zeiger)(parameterliste)`**
  `(objektzeiger->*funktionskomp_zeiger)(parameterliste`)

  ◇ **Beispiel :**

```
class Complex;              // wie vorher definiert

typedef float Complex::*PDF;
typedef Complex (Complex::*PMFC)(const Complex&);

int main(void)
{
  Complex c1(1,1), c2(0,0);
  Complex *cptr=&c2;
  PDF pfcomp;               // Zeiger auf float-Komp der Klasse Complex
  PMFC pcfunc;              // Zeiger auf Funktions-Komp. von Complex

  pfcomp=&Complex::re;
  pcfunc=&Complex::plus;

  c1.*pfcomp=3.5;                 // c1.re=3.5;
  cptr->*pfcomp=7.3;              // cptr->re (=c2.re) =7.3;
  c1=(c1.*pcfunc)(c2);            // Klammerung notwendig, da () höhere
  c2=(cptr->*pcfunc)(c1);         // Priorität als .* bzw ->* hat
  // ...
}
```

- **Hinweis :**

  ◇ Der Zeigertyp "**Pointer auf Klassenkomponenten eines bestimmten Typs**" muß sorgfältig vom Zeigertyp "**Pointer auf bestimmten Typ**" unterschieden werden :

  ◇ So sind (**`float Complex::*`**) und (**`float *`**) **verschiedene** – weder implizit noch explizit ineinander konvertierbare – **Typen.**

  ◇ **Beispiel :**

```
void func(float Complex::*pfcomp);

void falsch(void)
{
  float a=5.8;
  func(&a);           // Fehler , falscher Parametertyp !
}
```

Pointer auf Klassenkomponenten in C++ (3)

- **Anmerkungen zu "Adressen" von Member-Funktionen**

  ◇ "**Adressen" von Funktionskomponenten** (Member-Funktionen) können nur über die **Bindung an die Klasse** (Scope Resolution Operator und Klassenangabe) ermittelt werden.
  Die Ermittlung der Adresse über die **Bindung** an ein **konkretes Objekt** der Klasse ist **nicht zulässig**.

  Beispiel :

```
int main(void)
{
  Complex c1(1.0,1.0);  // Klasse Complex wie vorher definiert
  float (Complex::*pffunc)(void);
  pffunc=&Complex::betrag;          // zulässig
  pffunc=&c1.betrag;                // unzulässig
  // ...
}
```

  ◇ Die **Zuweisung** der "**Adresse"** einer **nicht-statischen** Member-Funktion an eine "**einfache**" – nicht mit einer Klassenangabe versehenen – **Funktionspointervariable** ist **nicht zulässig**. Es ist **keine Speicheradresse** !

  Beispiel :

```
float (*pf)(void);
pf=&Complex::betrag;                // unzulässig
```

  ◇ **Statische** Member-Funktionen werden dagegen ähnlich wie freie – nicht an eine Klasse gebundene – Funktionen behandelt.
  Die **Adresse** einer derartigen Funktion darf einer "**einfachen**" **Funktionspointervariablen** zugewiesen werden.
  Es handelt sich um eine echte Speicheradresse.
  Die **Zuweisung** an eine entsprechende **Komponentenzeigervariable** ist dagegen **unzulässig**.
  Natürlich kann die Adresse **nur** über die **Bindung an die Klasse** ermittelt werden.
  (Gilt **analog** auch für **statische Datenkomponenten**.)

  Beispiel :

```
class Complex
{ public:
    // ...
    static float betrag(void);
    // ...
};

int main(void)
{
  float (*pf)(void);
  pf=&Complex::betrag;              // zulässig
  // ...
}
```

- **Anmerkungen zu Adressen von Datenkomponenten**

  ◇ Neben der Ermittlung der "Adresse" einer Datenkomponente über die Bindung an die Klasse, läßt sich auch die echte **Speicheradresse einer Datenkomponente** eines **konkreten Objekts** ermitteln. Eine derartige Adresse kann dann **nur** "**einfachen**" **Datenpointer-Variablen** zugewiesen werden.

  Beispiel :

```
int main(void)
{
  Complex c1(1.5,2.0);   // Klasse Complex wie vorher definiert
  float *pflt = &c1.re;  // zulässig
  // ...
}
```

Demonstrationsprogramm zu Pointer auf Klassenkomponenten in C++

```
/* ------------------------------------------------------------------ */
/* Programm kompptrdemo.   (C++-Quelldatei kpdemo_m.cpp)              */
/* ------------------------------------------------------------------ */
/* Demonstrationsprogramm zu Pointern auf Klassenkomponenten          */
/* ------------------------------------------------------------------ */

#include <iostream>

using namespace std;

class Demo
{
  public:
    void func1(void);
    void func2(void);
};

void Demo::func1(void)
{
  cout << "\nAufruf von func1()\n";
}

void Demo::func2(void)
{
  cout << "\nAufruf von func2()\n";
}

typedef void (Demo::*PDFUNC)(void);   // PDFUNC ist Komponentenzeigertyp

int main(void)
{
  Demo d;               // Objekt der Klasse Demo
  Demo *pd=&d;          // Pointer auf Objekt der Klasse Demo
  PDFUNC fptr;          // fptr ist Komponentenzeigervariable

  fptr=&Demo::func1;    // Zuweisung der Adresse einer Member-Funktion
  (d.*fptr)();          // Aufruf Member-Funktion für konkretes Objekt
  fptr=&Demo::func2;    // Zuweisung der Adresse einer anderen Member-Fkt.
  (pd->*fptr)();        // Aufruf Member-Funktion für konkretes Objekt
  return 0;
}
```

- **Ausgabe des Programms :**

```
Aufruf von func1()

Aufruf von func2()
```

Unions als Klassen in C++

- **Eigenschaften**
  - ◇ **Unions in C++** sind **abwärtskompatibel zu Unions in C**, d.h. eine gültige C-Union ist auch eine gültige C++-Union (aber nicht notwendigerweise umgekehrt).
  - ◇ **Unions in C++** sind **auch Klassen**.
    Ihre **Datenkomponenten** belegen alle den **gleichen Speicherplatz** (genauer: sie beginnen alle an der gleichen Speicheradresse).
    Neben Datenkomponenten können sie **auch Funktionskomponenten** (Member-Funktionen) - einschließlich Konstruktoren und Destruktor - besitzen.
  - ◇ **Defaultmäßig** sind alle Komponenten **`public`**.
    Einzelne oder alle Komponenten können **auch `private`** oder **`protected`** deklariert werden.
    Dies bedeutet, daß es möglich ist, den Speicherplatz für die Datenkomponenten über eine öffentliche Komponente von außen anzusprechen, während der gleiche Speicherplatz über eine private Komponente nicht von außen zugänglich ist.

- **Initialisierung von Union-Objekten :**
  - ◇ **alle** Datenkomponenten sind **`public:`** wie in C (nur erste Komponente) oder über **geeignete Konstruktoren**, die auch eine andere als die erste Komponente initialisieren können
  - ◇ **wenigstens eine `private`** Datenkomponente : **nur** über **geeignete Konstruktoren**

- **Beispiel :**

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

union Bdouble
{   Bdouble(double);                    // Konstruktor           (public !)
    void showBytes(void);               // Ausgabe als Bytefolge (public !)
  private:
    double d;
    unsigned char c[sizeof(double)];
};

Bdouble::Bdouble(double w) { d=w; }     // Konstruktor

void Bdouble::showBytes(void)           // Ausgabe double-Wert als Byte-Folge
{ cout << endl << hex << setfill('0');
  for (int i=sizeof(double)-1; i>=0; i--)
    cout << setw(2) << (c[i]&0xff) << ' ';
  cout << endl << dec;
}

int main(void)
{ Bdouble dobj(1991.829);
  dobj.showBytes();
  return 0;
}
```

```
40 9f 1f 50 e5 60 41 89
```

- **Einschränkungen für Unions :**
  - ◇ Unions können **nicht** von einer anderen Klasse a**bgeleitet** sein.
  - ◇ Unions können **keine Basisklasse** sein.
  - ◇ Eine Union darf **keine Komponente** besitzen, die einen **Konstruktor** oder **Destruktor** oder **selbstdefinierten Zuweisungs-Operator** hat.
  - ◇ Eine Union darf **keine statischen Datenkomponenten** besitzen.
  - ◇ Eine Union darf **keine virtuellen Member-Funktionen** haben.
  - ◇ **Anonyme** Unions dürfen **keine Funktionskomponenten** und **nur `public`**–Datenkomponenten aufweisen.

Innere Klassen in C++ (1)

- **Definition geschachtelter Klassen**
  - ◇ **Klassendefinitionen** können **geschachtelt** werden → Definition einer Klasse innerhalb einer anderen Klasse.
  - ◇ Eine innerhalb einer anderen Klasse definierte Klasse heißt **innere Klasse** oder eingebettete Klasse (***nested class***). Eine Klasse, innerhalb der eine andere Klasse definiert ist, wird **äußere** oder **umschließende Klasse** (***enclosing class***) genannt.
  - ◇ **Beispiel :** Rückwärts verkettete Liste, Definition des Typs der Listenelemente als innere Klasse

```
class Set                                          // äußere Klasse
{
  public :
    Set() { last=NULL; }
    void insert(int val) { last = new SetMember(val, last); }

    // ...

  private :

    class SetMember                                // innere Klasse
    {
      public :
        SetMember(int val, SetMember* n)
        { memVal=val;
          next=n;
        }
      private :
        int memVal;
        SetMember* next;
    };

    SetMember* last;
};
```

  - ◇ Die Definition der inneren Klasse kann im **`public`-Teil oder** im **`private`-Teil** der äußeren Klasse erfolgen. Im **`private`-Teil** definierte innere Klassen können **nur innerhalb** der **äußeren Klasse** (sowie von Freunden) einschließlich weiterer innerer Klassen verwendet werden, im `public`-Teil definierte Klassen können dagegen auch außerhalb benutzt werden.
  - ◇ Der Name der inneren Klasse ist **lokal** zur äußeren Klasse.
    Die **innere** Klasse befindet sich damit **im Sichtbarkeitsbereich** (Verfügbarkeitsbereich, ***scope***) der **äußeren** Klasse.
    → eine **Verwendung** des Klassennamens **außerhalb der umschließenden** Klasse erfordert die **Qualifizierung** mit deren Namen.

```
class Outer
{
  public :
    class Inner
    {
       // ...
    };
  // ...
};

void func(void)                // Verwendung der inneren Klasse von außerhalb
{
  Inner yourObj;               // fehlerhaft : Name Inner ist nicht sichtbar
  Outer::Inner myObj;          // ok
  // ...
}
```

Innere Klassen in C++ (2)

- **Beziehungen zwischen innerer und äußerer Klasse**

  ◇ Die Beziehungen zwischen innerer und äußerer Klasse **entsprechen** denen zwischen **separaten Klassen** mit der **Besonderheit**, dass sich die **innere** Klasse im **Sichtbarkeitsbereich** (*scope*) der **äußeren** Klasse befindet.
  Das bedeutet, dass die **innere Klasse** die in der **äußeren Klasse** definierten **Namen direkt verwenden** kann, der **gegenseitige Komponentenzugriff** sich aber nach den **üblichen Zugriffsrechteregeln** richtet.

  ⇨ ▻ Es existiert **keine** gegenseitige **`friend`-Eigenschaft** zwischen äußerer und innerer Klasse.
  ▻ **Memberfunktionen** der **äußeren** Klasse haben **keine besonderen Zugriffsrechte** zu den Komponenten der **inneren** Klasse **und umgekehrt**
  ▻ **Freunde** der einen Klasse sind **nicht** automatisch Freunde der anderen Klasse

  ◇ In Memberfunktionen und sonstigen **Vereinbarungen** der **inneren Klasse** dürfen unter Beachtung der Zugriffsrechte ohne konkreten Objektbezug nur **Typnamen**, **statische Komponenten** (Daten und Funktionen) sowie **Aufzählungskonstante** der **äußeren** Klasse verwendet werden.
  Eine Verwendung **nichtstatischer** Daten- oder Funktionskomponenten erfordert einen **konkreten Objektbezug** (Objekt, Referenz oder Pointer auf Objekt)

  ◇ Entsprechendes gilt für **Vereinbarungen** in der **äußeren** Klasse. Dabei müssen Typnamen, sowie die Namen von statischen Komponenten und Aufzählungskonstanten aus der **inneren Klasse** mit deren Namen **qualifiziert** werden.

  ◇ **Beispiel :**

```
class Outer
{
    enum State                                  // private !
    { READY, RUNNING, WAITING, STOPPED };

  public :
    int ioPub;
    static int ioStatPub;

    class Inner
    { public :
        int iiPub;
        static int iiStatPub;
        State st;              // ok : privater Typname kann benutzt werden,
                               //      wenn Name vorher vereinbart ist
        void doSomeIn(int i, Outer* op)
        { st=READY;     // Fehler : private Aufzaehlungskonstante von Outer
          ioPub=i;      // Fehler : nichtstatische Komponente von Outer
          ioStatPub=i;  // ok     : statische public-Komponente von Outer
          ioPriv=i;     // Fehler : private-Komponente von Outer u. nicht statisch
          ioStatPriv=i; // Fehler : private-Komponente von Outer
          op->ioPub=i;  // ok     : Zugriff zu public-Komp über Objekt-Pointer
          op->ioPriv=i; // Fehler : Zugriff zu private-Komponente
        }

      private :
        int iiPriv;
        static int iiStatPriv;
    };

    void doSomeOut(int i, Inner* ip)
    { iiStatPub=i;         // Fehler : Qualifikation mit Klassenname fehlt
      Inner::iiStatPub=i;  // ok     : statische Komponente von Inner
      Inner::iiPub=i;      // Fehler : nichtstatische Komponente von Inner
      Inner::iiStatPriv=i; // Fehler : private Komponente von Inner
      ip->iiPub=i;         // ok     : Zugriff zu public-Komp über Objekt-Pointer
      ip->iiPriv=i;        // Fehler : Zugriff zu private-Komponente
    }

  private :
    int ioPriv;
    static int ioStatPriv;
};
```

Innere Klassen in C++ (3)

- **Komponentendefinition außerhalb der Klassendefinition**

  ◇ **Memberfunktionen** innerer Klassen können auch **außerhalb der Klassendefinition** definiert werden.
  Statische Datenkomponenten müssen sogar außerhalb der Klassendefinition definiert werden.
  In beiden Fällen muß der **vollqualifizierte** Komponentenname (doppelte Qualifikation !) angegeben werden.

```
class Outer
{
  public :
    class Inner
    {
      public :
        void doSomeIn(int i, Outer* op);
        // ...
      private :
        static int iiStatPriv;
        // ...
    };
  //...
};

void Outer::Inner::doSomeIn(int i, Outer* op)
{ // ... }

int Outer::Inner::iiStatPriv=1;
```

  ◇ Dabei ist zu berücksichtigen, dass der **Sichtbarkeitsbereich** (*scope*) einer **Klasse** mit ihrem Scope-**Qualifier beginnt** (für die innere Klasse oben : mit `Outer::Inner::`) und sich **bis zum Ende** der jeweiligen Komponenten-**Definition erstreckt**.
  Das hat z.B. Konsequenzen für die Angabe des Rückgabetyps von Memberfunktionen, wenn dies ein innerhalb der äußeren Klasse definierter Typ ist.

```
class Outer
{
  public :
    class In1
    {
      // ...
    };

    class In2
    {
      public :
        In1 func(In2);
        // ...
    };
  // ...
};


In1 Outer::In2::func(In2 x)     // fehlerhaft : Qualifikation für In1 fehlt
{ // ...
}

Outer::In1 Outer::In2::func(In2 x)   // ok !
{ // ...
  return *(new In1);                 // ok : innerhalb der Funktionsdefini-
                                     //      tion ist Qualifikation für In1
}                                    //      nicht notwendig
```

Innere Klassen in C++ (4)

- **Vorwärtsdeklaration von inneren Klassen**
  - ◇ Eine **innere Klasse** kann innerhalb der Definition einer äußeren Klasse auch zunächst nur **deklariert** werden.
    → **Vorwärtsdeklaration** !
  - ◇ Die **Definition** der **inneren** Klasse muß dann entweder **innerhalb** der **äußeren** Klasse **später** oder **außerhalb** deren Klassendefinition erfolgen.
  - ◇ Bei der zweiten Möglichkeit kann die **Definition** der **inneren Klasse** nach **außen** (gegenüber dem Anwender) **verborgen** werden, da diese in einer eigenen Headerdatei, die dem Anwender nicht zugänglich gemacht wird, enthalten sein kann.
  - ◇ **Beispiel :**

```
class OuterFwd
{
  class InnerFwd1;            // Vorwärtsdeklaration
  class InnerFwd2;            // Vorwärtsdeklaration
  // ...
  class InnerFwd1             // spätere Definition
  {
   // ...
  };
  // ...
};


class OuterFwd::InnerFwd2     // Definition in eigener Headerdatei
{
  // ...
};
```

- **Gründe für die Schachtelung von Klassen**
  - ◇ Wenn eine Klasse **nur innerhalb einer anderen Klasse benötigt** wird, kann sie als **innere Klasse** der anderen Klasse definiert werden. Sie tritt **nach außen** überhaupt **nicht in Erscheinung**.
    Dadurch wird die Anzahl der globalen Namen vermindert.
    Sinnvollerweise sollte die Definition der inneren Klasse im **`private`**-Teil der äußeren Klasse erfolgen
    Beispiel : Referenzzählung
  - ◇ Wenn eine Klasse **funktionell zu einer anderen Klasse** gehört, nur im Zusammenhang mit dieser Klasse sinnvoll eingesetzt werden kann, aber durchaus **außerhalb** dieser Klasse **verwendet** wird, sollte sie als **innere Klasse** im **`public`**-Teil der anderen Klasse definiert werden.
    Beispiel : Iteratoren

Lokale Klassen in C++

- **Definition lokaler Klassen**
  - ◇ Eine Klasse kann auch **innerhalb** einer **Funktion** definiert werden → **lokale Klasse**.
  - ◇ Die Funktion, die die Definition einer lokalen Klasse enthält, wird als **umschließende Funktion** (*enclosing function*) bezeichnet.
  - ◇ **Memberfunktionen** einer lokalen Klasse müssen **innerhalb** der **Klassendefinition definiert** werden.
  - ◇ Eine lokale Klasse darf **weder statische Datenkomponenten noch statische Memberfunktionen** enthalten.
  - ◇ **Beispiel :**

```
#include <iostream>

using namespace std;


int iext=2;


void func(int i)
{
  static int is=i;
  int ia=i;

  class LocClass
  { public :
      LocClass(int i=0)  { setVal(i);}
      void setVal(int i) { iVal=(i==0?iext:i); }
      int getVal()       { return iVal; }
    private :
      int iVal;
  };

  LocClass myLoc1(ia);
  cout << "\nvalue : " << myLoc1.getVal() << endl;
  LocClass myLoc2(is);
  cout << "\nvalue : " << myLoc2.getVal() << endl;
  LocClass myLoc3(5);
  cout << "\nvalue : " << myLoc3.getVal() << endl;
  LocClass myLoc4;
  cout << "\nvalue : " << myLoc4.getVal() << endl;
}
```

- **Verwendungsbeschränkungen**
  - ◇ Der Name einer lokalen Klasse ist nur innerhalb der Funktion, in der die Klasse definiert ist, bekannt → die **Klasse** kann **nur innerhalb** dieser **Funktion verwendet** werden.
  - ◇ Die Funktion, in der die lokale Klasse definiert ist, hat **keine besondere Zugriffsberechtigung zu den Komponenten** der lokalen Klasse. Es gelten die üblichen Zugriffsrechte.
  - ◇ Innerhalb einer lokalen Klasse kann **nur zu globalen** und zu **statisch-lokalen Variablen** der umschließenden Funktion zugegriffen werden. Ein Zugriff zu lokalen `auto`-Variablen der umschließenden Funktion ist nicht möglich.
    Weiterhin können alle übrigen globalen und lokalen Namen (Funktionen, Typen, Aufzählungskonstante) der umschließenden Funktion (sofern vorher vereinbart) innerhalb der lokalen Klasse verwendet werden.
  - ◇ **Anmerkung zu Visual-C++** (6.0) : Abweichend von ANSI-C++ ist ein Zugriff zu statisch-lokalen Variablen der umschließenden Funktion nicht zulässig.

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 2

2. Ergänzungen zum Überladen von Operatoren

2.1. Increment- und Decrement-Operator

2.2. Funktionsaufruf-Operator

2.3. Delegations-Operator (->)

2.4. Typkonvertierung

Überladen des Increment- und Decrement-Operators in C++

- **Notationsformen des Increment- bzw Decrement-Operators**
  - ◇ Die **unären Operatoren** ++ (**Increment**) und -- (**Decrement**) existieren für die Standard-Datentypen jeweils in **2 Formen** :
    - ▻ **Prefix-Notation** : **++x**; Ausdruckswert : veränderter Operandenwert
    - ▻ **Postfix-Notation**: **x++**; Ausdruckswert : alter Operandenwert
  - ◇ In **früheren** C++-Versionen konnte beim Überladen dieser Operatoren **nicht** zwischen **Prefix**- und **Postfix**-Notation **unterschieden** werden.
  - ◇ Im **ANSI/ISO-C++-**Standard ist jedoch eine **Unterscheidungsmöglichkeit** vorgesehen :
    - ▻ Die **Prefix-Notation** wird - wie bei den anderen unären Operatoren - durch eine **Operatorfunktion** mit **einem Parameter** (der bei der Definition als Member-Funktion implizit als `this`–Pointer übergeben wird) realisiert.
    - ▻ Die **Postfix-Notation** dagegen wird durch eine **Operatorfunktion** mit **zwei Parametern** (von denen bei der Definition als Member-Funktion der erste implizit als `this`–Pointer übergeben wird) realisiert.
      Der **zweite** Parameter ist ein **Dummy-Parameter**. Er muß vom Typ **int** sein.
      Beim Aufruf der Operatorfunktion wird für ihn i.a. der Wert 0 übergeben.
  - ◇ Damit sich die überladenen Operatoren bezüglich Prefix- und Postfix-Notation wie die Standard-Operatoren verhalten – was sinnvollerweise der Fall sein sollte – , sollten sich die **beiden** jeweiligen Funktionen lediglich im **Rückgabewert** unterscheiden.
    Dabei ist zu berücksichtigen, daß die Standard-Increment-und Decrement-Operatoren in der **Prefix-**Notation einen ***Lvalue*** (→ Rückgabe einer Referenz) und in der **Postfix-**Notation einen ***Rvalue*** (→ Rückgabe eines Wertes) zurückgeben.

- **Beispiel** :

```
class Ratio
{ public:
    //  ...
    Ratio& operator++(void);     // Überladen Increment-Operator (Prefix)
    Ratio operator++(int);       // Überladen Increment-Operator (Postfix)
  //  ...
};

Ratio& Ratio::operator++(void)       // Prefix
{ zaehler+=nenner;
  return *this;                      // Rückgabe neuer Wert (als Referenz)
}

Ratio Ratio::operator++(int dummy)   // Postfix, dummy ist Dummy-Parameter
{ Ratio temp=*this;
  zaehler+=nenner;
  return temp;                       // Rückgabe alter Wert
}

int main(void)
{
  Ratio a(4,5), b, c;

  b=a++;       // a.operator++(0);   ==>   b ⇐ (4,5),   a ⇐ (9,5)
  c=++a;       // a.operator++();    ==>   c ⇐ (14,5),  a ⇐ (14,5)
  //  ...
}
```

- **Anmerkung :**

  Die **Prefix-Notation** realisiert i.a. eine **schnellere Operation** als die Postfix-Notation (zweimaliges Kopieren des Objekts) → sie sollte – wenn nur der Inc- bzw Dec-Effekt benötigt wird – in der Regel **bevorzugt verwendet** werden.

Funktionsaufruf-Operator in C++ (1)

- **Funktionsaufruf**
  - ◇ Ein **Funktionsaufruf** hat die Form
    `expression(expression-list)`
  - ◇ Er kann formal als **zweistellige** (binäre) Operation aufgefasst werden :
    - **`()`** ist der – binäre – **Funktionsaufruf-Operator**
    - **`expression`** ist der **linke Operand**.
      Er muß eine Funktion referieren oder einen Funktionspointer als Wert ergeben.
    - **`expression-list`** ist der **rechte Operand**.
      Er besteht aus der durch Kommata separierten Liste der aktuellen Parameter, die auch leer sein kann.

- **Überladen des Funktionsaufruf-Operators `()`**
  - ◇ Auch der Funktionsaufruf-Operator lässt sich für Objekte selbstdefinierter Klassen überladen.
    → In diesem Fall ist der linke Operand keine Funktion (bzw Funktionspointer) sondern ein Objekt
  - ◇ Die Operatorfunktion **`operator()`** muß eine **nichtstatische Memberfunktion** sein, die beliebig viele Parameter (auch gar keine) haben kann.
    Default-Werte für die Parameter sind **zulässig**.
  - ◇ **Beispiel** :

```
class Gerade           // y= m*x + t
{
  public :
    Gerade(double, double=0);
    double operator()(double=0);
  private :
    double m_dM;       // Steigung m
    double m_dT;       // Achsenabschnitt t
};

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

Gerade::Gerade(double m, double t)
{ m_dM=m;
  m_dT=t;
}

double Gerade::operator()(double x)
{ return x*m_dM + m_dT;
}

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

int main(void)
{ Gerade line(0.5, 2);
  double arg;
  cout << endl << "Achsenabschnitt : " << line() << endl ;
  while (cout << "? ", cin >> arg)
    cout << "Wert : " << line(arg) << endl;
  cout << endl;
  return 0;
}
```

  - ◇ Die Operatorfunktion **`operator()`** kann für eine Klasse auch **mehrfach überladen** werden (unterschiedliche Parameterlisten !)

Funktionsaufruf-Operator in C++ (2)

• Funktionsobjekte

◇ Objekte von Klassen, für die der Funktionsaufrufoperator überladen ist, bezeichnet man als **Funktionsobjekte**.

◇ Es sind **Objekte**, die **wie Funktionen verwendet** werden können :
Der Ausdruck

`object(expression-list)`

ist kein Aufruf der Funktion `object()`,
sondern ein Aufruf der Memberfunktion `operator()()` für das Objekt `object.`:

`object.operator()(expression_list)`

• Anwendung von Funktionsobjekten

◇ Funktionsobjekte stellen häufig eine sinnvolle **objektorientierte Alternative zu freien Funktionen** dar.
Insbesondere dann, wenn mehrere gleichartig strukturierte aber mit unterschiedlichen Kenngrößen (z.B. Koeffizienten von Polynomen) arbeitende Funktionen benötigt werden.
Statt diese Kenngrößen bei jedem Funktionsaufruf zusätzlich zu den eigentlichen Funktionsparametern zu übergeben, werden sie durch einen Konstruktor in Datenkomponenten von Funktionsobjekten abgelegt.

◇ Das Überladen der Operatorfunktion `operator()()` ist weiterhin auch immer dann sinnvoll, wenn für eine Klasse nur **eine einzige** oder eine wichtige **überwiegend angewendete Funktionalität** existiert.
Beispiele : - Implementierung von einfachen **Iteratoren** (→ Programmbeispiel s. Kap. 8.5 "Iteratoren")
- **Applikatorklassen** als eine Möglichkeit zur Realisierung von Manipulatoren mit Parametern

◇ Beispiele anderer gebräuchlicher Anwendungen des überladenen Funktionsaufruf-Operators sind :
- Einsatz als **Substring-Operator**
- **Indizierung mehrdimensionaler Arrays** bzw von Datenstrukturen, die als mehrdimensionale Arrays behandelt werden können

◇ **Beispiel** zur Indizierung eines mehrdimensionalen Arrays

```
class IntMatrix
{ public :
    IntMatrix(unsigned, unsigned);
    int& operator()(unsigned, unsigned);
  private :
    unsigned m_uRows;
    unsigned m_uCols;
    int* m_piMatr;
};


IntMatrix::IntMatrix(unsigned rows, unsigned cols)
{ m_uRows=rows;
  m_uCols=cols;
  m_piMatr=new int[m_uRows*m_uCols];
  // Initialisierung aller Komponenten mit 0
}

int& IntMatrix::operator()(unsigned row, unsigned col)
{ row=row%m_uRows;
  col=col%m_uCols;
  return m_piMatr[row*m_uCols+col];
}


int main(void
{
  IntMatrix matr(5, 10);
  matr(2,3)=25;
  // ...
}
```

Operator -> in C++ (1)

- **Interpretation des Operators `->`**

  ◇ **Standardmäßig** wird der Operator **`->`** als **binärer Operator** verwendet.
  Er dient zum Zugriff zu Komponenten eines Objekts.
  Dabei muß der **linke Operand** ein **Pointer auf ein Objekt** und der **rechte Operand** der **Name einer Komponente** dieses Objekts sein.
  Der "**Wert**" eines derartigen Ausdrucks ist die dadurch **referierte Objekt-Komponente**.
  → **dereferenzierender Komponenten-Operator**.

  ◇ Der Operator lässt sich für Klassen so **überladen**, dass er direkt auf Objekte (statt auf Objekt-Pointer) angewendet werden kann. :
  In dieser überladenen Form stellt er einen **unären Operator** dar, der bei einem Objekt als Operanden – direkt oder indirekt – einen Objekt-Pointer zurückliefern muß.

  → Ein Ausdruck **`object->name`**
  wird interpretiert als **`(object.operator->())->name`**

  ◇ Der zurückgelieferte Pointer muß nicht das Objekt referieren, auf das der Operator angewendet wurde.
  Vielmehr wird er sich bei realen Anwendungen im allgemeinen auf ein anderes Objekt beziehen.
  Die durch `name` bezeichnete Objektkomponente muß dann eine Komponente dieses Objekts und nicht eine des Operanden-Objekts sein.
  → Der Komponentenzugriff wird an ein anderes Objekt **delegiert**.
  → **Delegations-Operator**.

- **Operatoratorfunktion `operator->`**

  ◇ Eine den Operator **`->`** überladende Funktion muß eine **nichtstatische Memberfunktion ohne Parameter** sein.

  ◇ Sie sollte einen **Pointer auf ein Objekt** als **Funktionswert** zurückgeben.

  ◇ Falls die Funktion statt eines Objekt-Pointers ein Objekt (oder eine Referenz auf ein Objekt) zurückgibt, wird die **Auswertung** eines **-> - Ausdrucks rekursiv** fortgesetzt : Für das zurückgelieferte Objekt wird wiederum eine Operatorfunktion `->` aufgerufen. Falls für dessen Klasse keine definiert ist, endet die Auswertung fehlerhaft.

  ◇ Ein expliziter Aufruf der Operatorfunktion `operator->` muß ohne Parameter erfolgen (das entspricht **einem Operanden**). Eine Verwendung von `->` in einem Ausdruck dagegen erfordert **zwei Operanden**.

```
class Abc
{ public :
    // ...
    X* operator->();          // X sei eine andere Klasse
  private :
    // ...
};

void func(Abc p)
{
  X* px1 = p.operator->();    // ok
  X* px2 = p->;               // Fehler !
  // ...
}
```

- **Beispiele für Anwendungen**

  ▻ Realisierung von **"*smart*" Pointern**.
  Hierbei handelt es sich um Objekte, die wie Pointer verwendet werden können, bei jedem Komponentenzugriff aber eine zusätzliche Funktionalität realisieren. (Kapselung der eigentlichen Pointer in dem Objekt)

  ▻ **Ausdehnung der Fähigkeiten** einer vorhandenen Klasse auf eine andere Klasse, in die sie eingekapselt wird, ohne Vererbung sondern mittels **Delegation**.

Operator -> in C++ (2)

- **Beispiel zur Delegation mittels Operatorfunktion `operator->`**

```
class Worker
{ public :
    Worker(int=0);
    // ...
    void doWork();
  private :
    // ...
    int m_id;
};


#include <iostream>
using namespace std;


inline Worker::Worker(int id)
{
  m_id=id;
}


inline void Worker::doWork()
{
  cout << "\nIch, der Worker mit id=" << m_id << ", arbeite gerade !\n";
}

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

class Chief
{ public :
    // ...
    void hireWorker(Worker&);
    Worker* operator->();
  private :
    Worker* myEmpl;
};


inline void Chief::hireWorker(Worker& fred)
{
  myEmpl=&fred;
}


inline Worker* Chief::operator ->()
{
  return myEmpl;
}

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

int main(void)
{ Chief moi;
  Worker tu(1);
  moi.hireWorker(tu);
  moi->doWork();          // Delegation von doWork() an Worker-Objekt
  return 0;
}
```

Ausgabe des Programms :

```
Ich, der Worker mit id=1, arbeite gerade !
```

Konvertierungsfunktionen in C++

- **Funktionen zur Typkonvertierung :**
  - ◇ **Konstruktoren**, die mit **einem Parameter** aufgerufen werden können, können als **Funktionen zur Konvertierung** des Parameter-Typs **in den Klassen-Typ** betrachtet werden.
  - ◇ Zur **entgegengesetzten Konvertierung** - **aus dem Klassentyp** in einen anderen Typ - dienen **Konvertierungs-funktionen** (*conversion functions*).
  - ◇ Im Prinzip handelt es sich bei **einer Konvertierungs-Funktion** um eine **Operatorfunktion** zum **Überladen beider Formen der** einfachen **Typkonvertierungs-Operatoren** (Cast-Operator `(typ)` und Werterzeugungs-Operator `typ()`) sowie des Operators **`static_cast`**.

- **Eigenschaften einer Typkonvertierungsfunktion :**
  - ◇ Sie muß eine **nichtstatische Member-Funktion** sein.
  - ◇ Da die Typkonvertierungs-Operatoren **unäre Operatoren** sind, hat sie **keine expliziten Parameter**.
  - ◇ Der Typ ihres Rückgabewertes ist implizit durch den Ziel-Typ, der nach dem Schlüsselwort `operator` anzugeben ist, festgelegt. Eine **explizite Festlegung des Funktionstyps** erfolgt daher **nicht**.

    → **Allgemeine Syntax** für die **Funktionsdeklaration :**

    ```
    operator typ();        // typ ist der Ziel-Typ
    ```
  - ◇ Der Ziel-Typ **`typ`** kann ein **beliebiger Typ**, auch ein anderer Klassentyp, sein.
  - ◇ Eine Konvertierungsfunktion wird **nicht nur** bei **expliziter Anwendung** eines **Typkonvertierungs-Operators** sondern auch in Fällen **impliziter Typkonvertierung** aufgerufen.

- **Beispiel :**

```
class Ratio
{ public:
    // ...
    operator double(void);     // Konvertierungsfunktion Ratio --> double
    // ...
};

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

Ratio::operator double(void)
{
  return (double)zaehler/nenner;
}

// ...

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

int main(void)
{
  Ratio a(3,-2), b(4,9);
  double z, v, w;
  // ...
  z=(double)a;           // Cast-Operator     --> Aufruf von a.operator double()
  w=double(b);           // Werterzeug.-Op    --> Aufruf von b.operator double()
  u=static_cast<double>(b); // static_cast    --> Aufruf von b.operator double()
  v=a;                   // impliz.Typ-Konv.  --> Aufruf von a.operator double()
  // ...
}
```

Anmerkungen zur impliziten Typkonvertierung in C++ (1)

- **Implizite Typkonvertierungen :**

  Stimmen bei einer **Zuweisung** oder **Initialisierung** die **Typen** des **Ziel-Objekts** und des **Quell-Wertes nicht überein**, versucht der Compiler eine **automatische** - **implizite** - **Typkonvertierung** des Quell-Wertes in den Typ des Ziel-Objekts vorzunehmen.

  Anmerkung : Diese Typkonvertierung findet auch bei **Parameterübergaben** in Funktionsaufrufen und in **Ausdrücken**, die ja in Aufrufe von Operatorfunktionen umgesetzt werden, statt
  (Die Übergabe von Wertparametern stellt eine Erzeugung und Initialisierung temporärer Objekte dar)

- **Regeln für die implizite Typkonvertierung**

  ◇ Für die implizite – automatische - Typkonvertierung werden die folgenden - vereinfacht dargestellten - **Regeln** in der **angegebenen Reihenfolge** angewendet (**absteigende Priorität**) :

  1. Anwendung **trivialer Typumwandlungen**
     (z.B. Array-Name → Pointer, Funktionsname → Pointer, `T` → `T&`, `T&` → `T`, `T` → `const T`, `T*` → `const T*`)
  2. Anwendung der **informationserhaltenden Standard-Typumwandlungen**
     - Integral Promotion :
       `char`, `short`, `enum`, Bitfeld → `int`,
       `unsigned char`, `unsigned short`, `unsigned` Bitfeld → `(unsigned) int`
     - Umwandlung `float` → `double`
  3. Anwendung der **übrigen Standard-Typumwandlungen**
     - Standard-Typumwandlungen von C
       (z.B. `int` → `double`, `int` → `unsigned` , usw)
     - Umwandlung von Referenzen und Pointer auf abgeleitete Klassen in Referenzen und Pointer auf Basisklassen
  4. Anwendung **benutzerdefinierter Typumwandlungen**
     - **Konstruktoren**
     - **Konvertierungsfunktionen**

  ◇ Eine Typkonvertierung kann **über mehrere Umwandlungsschritte** erfolgen.
  Existieren **mehrere Konvertierungswege**, wählt der Compiler den **kürzesten** aus.
  Existieren **zwei oder mehr gleichberechtigte Wege**, erzeugt er eine **Fehlermeldung** .

- **Implizite Anwendung benutzerdefinierter Typumwandlungen**

  Es gelten die folgenden Regeln :

  ◇ Sie werden **nur dann** versucht, wenn **keine** der **Standard-Typumwandlungen** zum **Erfolg** führt.

  ◇ An einer Typkonvertierung darf **maximal eine benutzerdefinierte Typumwandlung** beteiligt sein,
  Standard-Typumwandlungen können zusätzlich ohne Einschränkung angewendet werden.

  ◇ Es darf **nur einen Konvertierungsweg** geben, an dem **eine benutzerdefinierte Typumwandlung beteiligt** ist (unterschiedliche Weglängen spielen hierbei keine Rolle). → Konvertierung muß eindeutig sein.

  ◇ **Konstruktoren** und **Konvertierungsfunktionen** sind **gleichberechtigt**.

Anmerkungen zur impliziten Typkonvertierung in C++ (2)

- Die **automatische benutzerdefinierte Typkonvertierung** kann z.B dazu genutzt werden, um **typgemischte Ausdrücke** unter Beteiligung von Objekten selbstdefinierter Klassen zu ermöglichen.

  Statt die jeweilige Operatorfunktion mehrfach - für die zulässigen Typkombinationen - zu überladen, benötigt man **nur** jeweils **eine Operatorfunktion** und **adäquate Konstruktoren**.
  Soll eine **"symmetrische" Typ-Mischung** möglich sein, so muß die Operatorfunktion auch hier eine **"freie"** – gegebenenfalls befreundete – **Funktion** sein.

  Beispiel :

```
class Ratio
{ public:
    // ...
    Ratio(double);                                  // alternativer Konstruktor
    // ...
    friend Ratio operator+(const Ratio&, const Ratio&);
  // ...                                                    // nicht überladen
};

// ...

int main(void)
{
  Ratio a(4,9), b, c;
  // ...
  b=3.7+a;                          // Aufruf von operator+(Ratio(3.7), a)
  c=a+4.3;                          // Aufruf von operator+(a, Ratio(4.3))
  // ...
}
```

- **Probleme**

  Funktionen zur **automatischen Typkonvertierung** können auch **Probleme** erzeugen.
  Wird im obigen Beispiel zusätzlich eine Konvertierungsfunktion

  `Ratio::operator double()`

  die ein `Ratio`-Objekt in einen `double`-Wert umwandelt, definiert, sind die **gemischten Additions-Ausdrücke nicht** mehr **eindeutig** :

Beispielsweise kann	**`3.7 + a`**	dann mittels Typkonvertierung
als	**`Ratio(3.7) + a`**	
und als	**`3.7 + double(a)`**	
interpretiert werden.		

  → **Beide Konvertierungen** verwenden eine **benutzerdefinierte Umwandlung**, was **nicht zulässig** ist.

  => Konstruktoren und Konvertierungsfunktionen sind nur sehr wohl durchdacht zu definieren.

  Insbesondere muß darauf geachtet werden, daß **keine zyklischen Typkonvertierungen** möglich werden.
  Die ist z.B. der Fall,
  - ▻ wenn - wie im obigen Beispiel - **in einer Klasse** ein **Konstruktor** und eine **Konvertierungsfunktion** für genau **entgegengesetzte Typumwandlungen** definiert werden,
  - ▻ oder in **zwei Klassen** jeweils ein **Konstruktor** vorhanden ist, der eine **Typumwandlung aus der jeweiligen anderen Klasse** vornimmt.

Ergänzungen zur Typkonvertierung in C++

- **Weiteres Beispiel zur Mehrdeutigkeit bei automatischer Typkonvertierung**

  Die **Konvertierung** eines Objekts der **Klasse X** in ein Objekt der **Klasse T** kann erreicht werden
  - ▻ entweder durch die **Konvertierungsfunktion** : `X::operator T()` `// Member von X`
  - ▻ oder durch die **Konstruktor-Funktion** : `T::T(X)` `// Member von T`

  Wenn **beide Funktionen definiert** sind, sind **automatische Typkonvertierungen** ebenfalls **nicht mehr eindeutig** → Compiler-Fehler

  Beispiel :

```
class X
{ // ...
  operator T();            // Konvertierungsfunktion
  // ...
};

class T
{ // ...
  T(X);                    // Konstruktor
  T(T&);                   // Copy-Konstruktor
  // ...
};

void main(void)
{
  X xobj;
  T tobj=xobj;           // Konstruktor oder Konvertierungsfunktion ?
   // ...
}
```

- **Vermeidung der Problematik von Mehrdeutigkeiten**

  Wegen der **Gefahr von Mehrdeutigkeiten** ist es häufig **besser**, durch **Vermeiden von Konvertierungsfunktionen** automatische Typkonvertierungen zu reduzieren.
  Trotzdem **benötigte Typkonvertierungen** lassen sich i.a. **problemlos** durch **explizite Aufrufe** entsprechend definierter **normaler Member-Funktionen** realisieren.
  Diesen Funktionen sollte man dann entsprechend **aussagekräftige Namen** geben.

  Beispiel :

```
class Ratio
{ public:
    // ...
    double asDouble(void);     // Funktion zur expliziten Typkonvertierung
  // ...
 }

// ...

void main(void)
{ Ratio a(5,13);
  double z = a.asDouble();     // expliziter Aufruf der Typkonvertierung
  // ...
}
```

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 3

3. Mehrfachvererbung

3.1. Eigenschaften und Problematik

3.2. Virtuelle Basisklassen

Mehrfachvererbung in C++ (1)

- **Eigenschaften**
  - ◇ **Mehrfachvererbung** liegt vor, wenn eine abgeleitete Klasse **mehr als eine direkte Basisklasse** hat, d.h. in ihrer Abstammungsliste zwei oder mehr Basisklassen aufgeführt sind.
  - ◇ **Beispiel :**

```
class Auto
{ public:
    void fahreWeiter(int d) { km+=d; }  // Fortbewegung um d km
    // ...
  protected:
    int km;                             // insgesamt gefahrene Kilometer
    // ...
};


class Boot
{ public:
    void fahreWeiter(int d) { sm+=d; }  // Fortbewegung um d sm
    // ...
  protected:
    int sm;                             // insgesamt gefahrene Seemeilen
    // ...
};


enum antrieb { reifen, schraube };


class AmphFahrz : public Auto, public Boot
{ public:
    void fahre(antrieb, int);
    // ...
  protected:
    antrieb aktAntr;                    // akt. Antrieb
    // ...
};


void AmphFahrz::fahre(antrieb a, int d)
{
  aktAntr=a;
  if (a==reifen)
     Auto::fahreWeiter(d);         // fahreWeiter(d) allein nicht zulässig
  else                             // da mehrdeutig
     Boot::fahreWeiter(d);
}
```

  - ◇ Bei Mehrfachvererbung **erbt** die **abgeleitete Klasse** die **Komponenten aller ihrer Basisklassen**.
    Die "***is a ..*** "-Beziehung gilt zwischen der abgeleiteten Klasse und **jeder ihrer Basisklassen**.
    → Ein **Objekt der abgeleiteten Klasse** kann zugleich auch als ein **Objekt von jeder der Basisklassen** betrachtet werden ("**sowohl** ... **als auch**...").
    Zu obigem Beispiel : Ein `AmphFahrz` ist sowohl ein `Auto` als auch ein `Boot`.

- **Namenskonflikte**
  - ◇ Die - jederzeit mögliche - Verwendung eines **gleichen Komponentennamens** in **verschiedenen Basisklassen** führt zu **Namenskonflikten** : in der **abgeleiteten Klasse** ist der **Komponentenname** allein **nicht mehr eindeutig**.
  - ◇ **Lösung :** Zur eindeutigen Auswahl einer bestimmten Komponente dieses Namens muß ihr **vollqualifizierter Name** angegeben werden.

Mehrfachvererbung in C++ (2)

- **Teilobjekte der Basisklassen**
  - ◇ Ein **Objekt einer mehrfach abgeleiteten Klasse** besitzt - neben den neu definierten spezifischen Komponenten – **Teilobjekte von jeder ihrer Basisklassen**.
  - ◇ Die Anordnungs-Reihenfolge der Teil-Objekte im Arbeitsspeicher ist implementierungsabhängig. Typischerweise entspricht sie der Reihenfolge, in der die Basisklassen in der Abstammungsliste angegeben sind. (Anmerkung : nur die Datenkomponenten belegen Speicherplatz im Objekt)
  - ◇ **Beispiel** :

```
class Auto
{ // ...
};


class Boot
{ // ...
};


class AmphFahrz : public Auto, public Boot
{ // spezifische
  // Komponenten
};
```

Objekt der Klasse `AmphFahrz`

- **(Teil-)Objektadressen**
  - ◇ Ein **Pointer** oder eine **Referenz** auf ein **Objekt der abgeleiteten Klasse** kann - **implizit oder explizit** - in einen **Pointer** oder eine **Referenz** auf ein **Objekt jeder ihrer Basisklassen umgewandelt** werden.
  Der **umgewandelte Zeiger** (bzw die umgewandelte Referenz) **referiert** das **entsprechende Teil-Objekt** innerhalb des Gesamt-Objekts.
  Die **Anfangsadressen der einzelnen Teilobjekte** sind aber **unterschiedlich**.
  Das bedeutet, daß bei **Mehrfachableitung** sich durch die o.a. Typkonvertierung die **Objekt-Adresse ändern** kann.
  Bei einfacher Ableitung bleibt die Adresse dagegen gleich.
  - ◇ **Beispiel :**

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

int main(void)
{ AmphFahrz a;
  Auto* ap=&a;         // implizite Typkonvertierung AmphFahrz* --> Auto*
  Boot* bp=&a;         // implizite Typkonvertierung AmphFahrz* --> Boot*

  cout << "\nAdresse als AmphFahrz : " << &a;
  cout << "\nAdresse als Auto      : " << ap;
  cout << "\nAdresse als Boot      : " << bp << '\n';
  return 0;
}
```

Erzeugte Ausgabe :

```
Adresse als AmphFahrz : 0012FF74
Adresse als Auto      : 0012FF74
Adresse als Boot      : 0012FF78
```

Mehrfachvererbung in C++ (3)

- **Mehrfachvererbung derselben Basisklasse**

  ◇ Es ist **nicht zulässig**, ein und dieselbe Basisklasse **mehrfach direkt zu erben**.

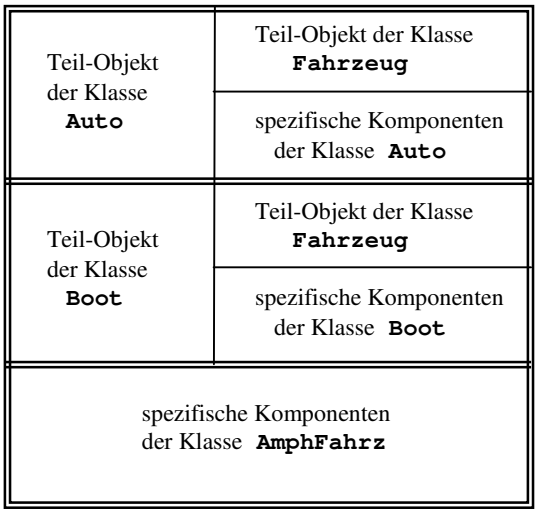
  Beispiel :
  ```
  class A { /* ... */ };
  class B : public A, public A     // unzulässig !
  { /* ... */ };
  ```

  ◇ Es ist jedoch möglich, ein und dieselbe Basisklasse über **unterschiedliche Ableitungswege** (mehrstufige Ableitungen) **mehrfach indirekt** zu erben.

  Beispiel :
  ```
  class Fahrzeug { protected : int baujahr; /* ... */ };
  class Auto : public Fahrzeug { /* ... */ };
  class Boot : public Fahrzeug { /* ... */ };
  class AmphFahrz : public Auto, public Boot { /* ... */ };
  // AmphFahrz erbt die Klasse Fahrzeug zweimal,
  // einmal über Auto und einmal über Boot.
  ```

  In einem derartigen Fall ist in einem Objekt der zuletzt abgeleiteten Klasse ein **Teil-Objekt** der **mehrfach geerbten indirekten Basisklasse** - mit allen seinen Daten-Komponenten - **mehrfach enthalten** (für jeden Ableitungsweg einmal).

  Zu obigen Beispiel : **Objekt der Klasse `AmphFahrz`**



  ◇ Für die **Komponenten** der **mehrfach geerbten Basisklasse** ergeben sich dadurch **Mehrdeutigkeitsprobleme** :
  Eine derartige Komponente kann weder über ihren Komponentennamen allein noch über ihren - den Basisklassennamen verwendenden - voll-qualifizierten Namen eindeutig angesprochen werden.
  Eindeutigkeit ist nur gegeben, wenn ein **voll-qualifizierter Name**, der sich auf eine der **Zwischenklassen** bezieht, benutzt wird.

  Zu obigen Beispiel :
  ```
  // In einer Member-Funktion der Klasse AmphFahrz :
  baujahr=1900;               // unzulässig
  Fahrzeug::baujahr=1900;     // unzulässig
  Auto::baujahr=1900;         // eindeutig --> zulässig
  ```

Virtuelle Basisklassen in C++ (1)

• Virtuelle Ableitung

Eine Basisklasse kann **virtuell abgeleitet** werden :
Ergänzung des Basisklassen-Eintrags in der Abstammungsliste um das Schlüsselwort **`virtual`** (vor oder nach dem Zugriffs-Specifier).
→ **virtuelle Basisklasse**

Beispiel :

```
class Fahrzeug { /* ... */ };
class Auto : virtual public Fahrzeug { /* ... */ };
```

• Aufbau eines Objekts mit einer virtuellen Basisklasse

◇ Ein Objekt einer Klasse mit einer virtuellen Basisklasse besitzt einen **anderen Aufbau** als ein entsprechendes Objekt mit nicht-virtueller Basisklasse :
Es enthält - im Unterschied zu einer nicht-virtuellen Ableitung - einen **Pointer auf das Teil-Objekt der Basisklasse**. Dieses wird üblicherweise **nach** den **spezifischen Komponenten der abgeleiteten Klasse** angelegt.

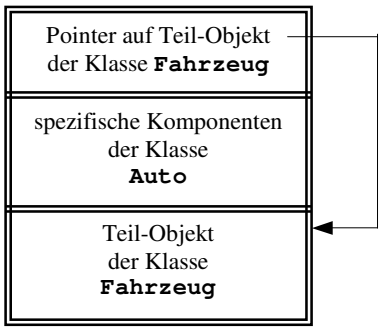
Beispiel :

nicht-virtuelle Ableitung

```
class Fahrzeug { /* ... */ };
class Auto : public Fahrzeug
{ /* ... */ };
```

virtuelle Ableitung

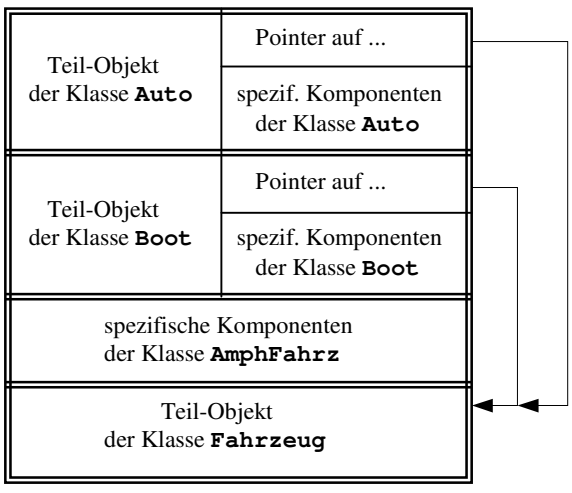
```
class Fahrzeug { /* ... */ };
class Auto : virtual public Fahrzeug
{ /* ... */ };
```



◇ Von einer virtuellen Basisklasse ist bei **indirekter Mehrfachvererbung** in abgeleiteten Klassen immer **nur ein Teil-Objekt** vorhanden
(Voraussetzung : **alle Ableitungen** der betreffenden Klasse müssen **virtuell** sein)

Beispiel :

```
class Fahrzeug
{ /* ... */ };

class Auto :
virtual public Fahrzeug
{ /* ... */ };

class Boot :
virtual public Fahrzeug
{ /* ... */ };

class AmphFahrz :
public Auto, public Boot
{ /* ... */ };
```



Virtuelle Basisklassen in C++ (2)

- Für die **Komponenten** einer **mehrfach geerbten virtuellen Basisklasse** existieren **keine Mehrdeutigkeitsprobleme**.
  → Sie können **sowohl** über ihren **Komponentennamen allein** als auch über ihren - den Basisklassennamen verwendenden - **voll-qualifizierten Namen** eindeutig angesprochen werden.

  Zu obigen Beispiel :

```
// In einer Member-Funktion der Klasse AmphFahrz
baujahr=1900;             // eindeutig --> zulässig
Fahrzeug::baujahr=1900;   // eindeutig --> zulässig
Auto::baujahr=1900;       // eindeutig --> zulässig
```

- **Initialisierung von Objekten einer Klasse mit mehrfach indirekt abgeleiteten virtuellen Basisklassen :**

  ◇ Da jedes (Teil-) Objekt nur einmal initialisiert werden darf (Konstruktoraufruf), ergeben sich Änderungen in der **Initialisierungs-Reihenfolge** gegenüber der indirekten Mehrfachableitung nicht-virtueller Basisklassen :
  - ▻ Die **Konstruktoren virtueller Basisklassen** werden **immer zuerst** ausgeführt (pro Klasse ein Konstruktoraufruf !).
  - ▻ Anschließend werden die Konstruktoren der übrigen direkten bzw indirekten Basisklassen in der Reihenfolge der Abstammungsliste bzw der Ableitung aufgerufen.
  - ▻ Zuletzt wird der Konstruktor der zuletzt abgeleiteten Klasse (der Klasse des anzulegenden Objekts) ausgeführt.

  Beispiel :

```
class Besitz { /* ... */ };
class Fahrzeug { /* ... */ };
class Auto : virtual public Fahrzeug { /* ... */ };
class Boot : virtual public Fahrzeug { /* ... */ };
class AmphFahrz : public Besitz, public Auto, public Boot
{ /* ... */ };
```

  Reihenfolge der Konstruktoraufrufe für ein Objekt von `Amphfahrz` :

```
Konstruktor von Fahrzeug
Konstruktor von Besitz
Konstruktor von Auto
Konstruktor von Boot
Konstruktor von AmphFahrz
```

  Aufruf-Reihenfolge bei nicht-virtueller Basisklasse Fahrzeug :

```
Konstruktor von Besitz
Konstruktor von Fahrzeug
Konstruktor von Auto
Konstruktor von Fahrzeug
Konstruktor von Boot
Konstruktor von AmphFahrz
```

  ◇ Vom **Konstruktor** einer **virtuellen Basisklasse benötigte Initialisierungswerte** müssen diesem **direkt vom Konstruktor** der **zuletzt abgeleiteten Klasse übergeben** werden.
  Eine eventuelle Übergabe durch Zwischenklassen wird ignoriert. Eine **stufenweise Übergabe über Zwischenklassen** (wie bei nicht-virtuellen Basisklassen erforderlich) ist daher **nicht möglich**.
  Dies kann bei der Ableitung von Klassen, bei denen das Vorhandensein virtueller Basisklassen nicht bekannt ist (z.B. nicht selbst geschriebene Klassen), zu **Problemen** führen.

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 4

4. Ergänzungen zur Laufzeitpolymorphie

4.1. Abstrakte Klassen

4.2. Laufzeittypinformation

4.3. Typkonvertierungsoperator `dynamic_cast`

Abstrakte Klassen in C++

- **Eigenschaften**

  ◇ Eine Klasse, für die wenigstens eine **rein-virtuelle Funktion** deklariert ist, wird als **abstrakte Klasse** bezeichnet.

  ◇ Eine **rein-virtuelle Funktion** besitzt **keine Definition** in der Basisklasse. Für sie ist **lediglich** das **Interface** (Parameter und Funktionstyp) jedoch keine konkrete Implementierung festgelegt.
  Eine **abstrakte Klasse** ist damit **unvollständig** definiert. Von ihr lassen sich **keine konkreten Objekte anlegen**.
  Sie kann **nur als Basisklasse** zur Ableitung anderer Klassen verwendet werden.

  ◇ Jede von einer abstrakten Klasse abgeleitete Klasse, von der konkrete Objekte erzeugt werden sollen, muß für sämtliche geerbten rein-virtuellen Funktionen Definitionen enthalten (→ **konkrete Klasse**).
  Eine rein-virtuelle Funktion, die in einer abgeleiteten Klasse nicht definiert wird, bleibt rein-virtuell für diese Klasse.
  Die abgeleitete Klasse ist damit ebenfalls abstrakt.

  ◇ Eine **abstrakte Klasse** darf **nicht als Parameter-Typ** und **nicht als Funktions-Rückgabe-Typ** verwendet werden und darf **nicht in** einer **expliziten Typ-Konvertierung** auftreten.

  ◇ **Pointer** und **Referenzen** auf **abstrakte Klassen** sind dagegen **zulässig**.

- **Anwendung :**

  Abstrakte Klassen dienen in einer Klassenhierarchie zur **Zusammenfassung gemeinsamer Eigenschaften** unterschiedlicher konkreter Objekte unter einem **Oberbegriff** und zur Bereitstellung eines **gemeinsamen Methoden-Interfaces** ohne Implementierungs-Details festzulegen.
  Die Implementierung der Methoden kann **geändert** oder **ergänzt** werden (neue abgeleitete Klassen), **ohne** daß dies **Auswirkungen auf die Schnittstelle** und damit auf die Anwendung der Methoden hat.

- **Beispiel :**

```
class Punkt { private: int x, y; /* ... */ };

class GeoObj                                          // abstrakte Klasse
{ public:
    virtual void move(const Punkt&) = 0;
    virtual void draw(void) = 0;
    // ...
  protected:
    GeoObj(const Punkt& p) : refpunkt(p) {}
    Punkt refpunkt;
};

class Linie : public GeoObj                           // konkrete Klasse
{ public:
    void move(const Punkt& p) { /* Verschiebe Linie ... */ }
    void draw(void)           { /* Zeichne Linie    ... */ }
    // ...
};

class Kreis : public GeoObj                           // konkrete Klasse
{ public:
    void move(const Punkt& p) { /* Verschiebe Kreis ... */ }
    void draw(void)           { /* Zeichne Kreis    ... */ }
    // ...
};
```

Demonstrationsprogramm zu abstrakten Klassen in C++

```
// ------------------------------------------------------------------------
// Programm ABCLBSP
// ------------------------------------------------------------------------
// Demonstrationsbeispiel zu abstrakten Klassen
// ------------------------------------------------------------------------

#include <iostream>
using namespace std;

class Num                                           // abstrakte Klasse
{ public:
    Num(int i=0) { wert=i;}
    virtual void showNum(void) = 0;
  protected:
    int wert;
};

class DecNum : public Num                           // konkrete Klasse
{ public:
    DecNum(int i=0) : Num(i) { }
    void showNum(void)
    { cout <<"wert dezimal   : " << dec << wert << '\n'; }
};

class HexNum : public Num                           // konkrete Klasse
{ public:
    HexNum(int i=0) : Num(i) { }
    void showNum(void)
    { cout <<"wert sedezimal : " << hex << wert << '\n'; }
};

class OctNum : public Num                           // konkrete Klasse
{ public:
    OctNum(int i=0) : Num(i) { }
    void showNum(void)
    { cout <<"wert oktal     : " << oct << wert << '\n'; }
};

void main(void)
{ Num *baseptr[3];
  DecNum wd(10);
  HexNum wh(511);
  OctNum wo(63);
  baseptr[0]=&wd;            // implizite Typwandlung DecNum* --> Num*
  baseptr[1]=&wh;            // implizite Typwandlung HexNum* --> Num*
  baseptr[2]=&wo;            // implizite Typwandlung OctNum* --> Num*
  cout << '\n';
  for (int i=0; i<3; i++)
    baseptr[i]->showNum();
}
```

- **Ausgabe des Programms :**

```
wert dezimal   : 10
wert sedezimal : 1ff
wert oktal     : 77
```

Laufzeit-Typinformation in C++ (1)

- **Einführung**

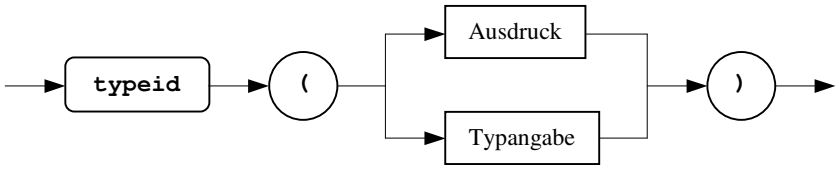
  Ein **Zeiger** bzw eine **Referenz** auf ein Objekt einer **abgeleiteten Klasse** kann implizit oder explizit in einen Zeiger bzw eine Referenz auf ein Objekt einer - eindeutigen - **Basisklasse umgewandelt** werden.
  Dadurch wird es möglich, mittels Basisklassen-Pointern bzw -Referenzen Objekte unterschiedlichen Typs - allerdings aus derselben Vererbungshierarchie - zu verwalten. Das bedeutet, daß der gleiche Basisklassen-Pointer (bzw -Referenz) dynamisch änderbar auf Objekte unterschiedlichen Typs zeigen kann.
  Manchmal ist es wünschenswert, während der Laufzeit den **tatsächlichen Typ** des Objekts, auf das ein Basisklassen-Pointer bzw -Referenz zeigt, zu ermitteln.
  → **Laufzeit-Typinformation** (*Runtime Type Information* = **RTTI**)

- **`typeid`-Operator**

  ◇ **unärer** Operator

  ◇ ermöglicht die **Ermittlung von Typinformationen während der Laufzeit**

  ◇ **Syntax :**



  ◇ **Beispiele :**
```
typeid(*baseptr[1])
typeid(int)
```

  ◇ Der **Wert** eines **`typeid`-Ausdrucks** ist vom **Typ `const type_info&`**

  → Ein `typeid`-Ausdruck liefert als Ergebnis eine Referenz auf ein Objekt der Klasse `type_info`, durch das der Typ des Operanden-Ausdrucks bzw der Operanden-Typangabe repräsentiert wird.

  Die Klasse **`type_info`** ist Bestandteil der ANSI/ISO-C++-Standardbibliothek
  . Sie ist in der Header-Datei **`<typeinfo>`** (bzw `<typeinfo.h>`) definiert.

  Der **Compiler** legt **für jeden Datentyp ein Objekt dieser Klasse** an.

  Wenn der **Operanden-Ausdruck** eine **Referenz** oder ein **dereferenzierter Pointer != NULL** auf Objekte einer **polymorphen Klasse** ist, ist das **Ergebnis** eine **Referenz** auf das **`type_info`-Objekt**, das den **tatsächlichen Typ** des aktuell referierten vollständigen Objekts referiert.
  → Ermittlung des **dynamischen** (zur Laufzeit änderbaren) **Typs**

  Ist der **Operanden-Ausdruck** ein **dereferenzierter NULL-Pointer** auf Objekte einer polymorphen Klasse wird die **Exception `bad_typeid`** geworfen.

  Für **jeden anderen Typ des Operanden-Ausdrucks** (sowie für eine **Typangabe als Operand**) ist das Ergebnis eine **Referenz** auf das **`type_info`-Objekt**, das diesen **(statischen) Typ** repräsentiert.

  ⇒ eine (dynamische) **Laufzeit-Typinformation läßt sich nur für Objekte polymorpher Klassen** ermitteln.

- **Hinweis :**

  In **Visual-C++** muß die Unterstützung der Laufzeit-Typinformation durch Setzen eines **Compiler-Schalters** explizit **aktiviert** werden.
  Menu : **Projekt → Einstellungen** → Reiter : **C++** → Kategorie : **Programmiersprache C++**→ Checkbox : **RTTI aktiv**.

Laufzeit-Typinformation in C++ (2)

- **Beispiele für das Ergebnis eines `typeid`-Ausdrucks :**

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

class Fahrzeug { /* ... */ };                    // nicht-polymorphe Klasse

class LandFahrz : public Fahrzeug                // polymorphe Klasse
{ public:
    virtual void fahren(float) { /* ... */ }
    // ...
};

class Auto : public LandFahrz { /* ... */ };

class Fahrrad : public LandFahrz { /* ... */ };

class Boot : public Fahrzeug  { /* ... */ };


void main(void)
{
  Fahrzeug *pclFahr;
  LandFahrz *pclLandF;
  LandFahrz clLaFz;
  Auto a;
  LandFahrz& rLaFz=a;

  pclLandF=new Fahrrad;
  typeid(*pclLandF);     // --> type_info-Objekt von Fahrrad   (dynamisch)
  pclLandF=&a;
  typeid(*pclLandF);     // --> type_info-Objekt von Auto      (dynamisch)
  typeid(rLaFz);         // --> type_info-Objekt von Auto      (dynamisch)

  pclFahr=&clLaFz;
  typeid(*pclFahr);      // --> type_info-Objekt von Fahrzeug  (statisch)
  pclFahr=new Boot;
  typeid(*pclFahr);      // --> type_info-Objekt von Fahrzeug  (statisch)
  typeid(clLaFz);        // --> type_info-Objekt von LandFahrz (statisch)
  typeid(LandFahrz);     // --> type_info-Objekt von LandFahrz (statisch)
}
```

- **Realisierung der Ermittlung der (dynamischen) Laufzeit-Typinformation**

Für **jede polymorphe Klasse** wird ein **Pointer** auf das die Klasse repräsentierende **`type_info`-Objekt** als **zusätzlicher Eintrag** mit in die **virtuelle Methoden-Tabelle** (VMT) aufgenommen.
Damit ist dieses Objekt über den **VMT-Pointer** erreichbar.
Da eine VMT nur für polymorphe Klassen existiert, ist die Ermittlung der (dynamischen) Laufzeit-Typinformation auf Objekte derartiger Klassen beschränkt.

Laufzeit-Typinformation in C++ (3)

- **Die Klasse `type_info`**

  ◇ Objekte dieser Klasse **repräsentieren Typen**

  ◇ Die Klasse ist in der C++-Header-Datei **`<typeinfo>`** (bzw `<typeinfo.h>`) definiert.

  ◇ Für **jeden Datentyp** legt der Compiler **ein Objekt** dieser Klasse an.
  Dieses enthält **implementierungsabhängige Datenkomponenten** zur Speicherung des **Typnamens** sowie eines **codierten Wertes**, der es gestattet, Typen in eine **Sortierreihenfolge** anzuordnen sowie zwei Typen auf **Gleichheit** zu überprüfen.
  Die Klasse überlädt die **Operatoren `==`** und **`!=`** und definiert eine **Memberfunktion zur Ermittlung des Typnamens** sowie eine **Memberfunktion zum Vergleich des "Reihenfolgekriteriums"** zweier Typen.

  ◇ In **ANSI/ISO-C++** ist folgende **prinzipielle Definition** der Klasse vorgesehen :

```
class type_info
{ public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
  private:
    // implementierungsabhängige Datenkomponenten zur
    // Speicherung des Typnamens und eines "Reihenfolgekriteriums"
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
};
```

  ◇ Da der **Copy-Konstruktor** und die **Zuweisungsoperator-Funktion** dieser Klasse **`private`** sind, lassen sich `type_info`-Objekte **nicht kopieren**.

- **Beispiele zur Anwendung der Memberfunktionen der Klasse `type_info`** :

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

{ //  ...
  LandFahrz *apclFuhrpark[ANZ-1];
  int iAnzVelo=0;
  //  ...
  for (int i=0; i<ANZ; i++)
    if (typeid(*apclFuhrpark[i]) == typeid(Fahrrad))
      iAnzVelo++;
  //  ...
}

{ //  ...
  LandFahrz *pclFahr;
  //  ...
  cout << typeid(*pclFahr).name();
  //  ...
}
```

Der Typkonvertierungsoperator `dynamic_cast` in C++ (1)

- **`dynamic_cast`**

  ◇ Dieser Typkonvertierungsoperator ermöglicht **sichere Typkonvertierungen innerhalb von Klassenhierarchien**; insbesondere eine **sichere Rückwandlung** eines **Pointers** (Referenz) auf **Basisklasse** in **Pointer** (Referenz) auf **abgeleitete Klasse**.
  Eine eventuelle **`const`-**Eigenschaft läßt sich mit ihm **nicht entfernen**.

  ◇ Der Ausdruck **`dynamic_cast<T>(e)`**

  bewirkt die Konvertierung des Ausdrucks **`e`** in den Typ **`T`**.

  Der **Zieltyp `T`** muß ein **Pointer** oder eine **Referenz** auf eine vollständig definierte Klasse bzw der Typ **`void*`** sein. Ensprechend muß der **Quellausdruck `e`** ein **Pointe**r auf ein Klassen-Objekt oder ein **Lvalue** eines Klassentyps sein.

  ◇ **Folgende Fälle** sind **zulässig :**

  a) Der **Quellausdruck `e`** ist ein Pointer auf ein Objekt bzw ein Lvalue einer - von einer Basisklasse `B` **abgeleiteten** - **Klasse `D`.** Dabei muß `B` eine zugreifbare (`public`) und eindeutige Basisklasse von `D` sein.
  `T` ist ein Pointer bzw eine Referenz auf diese **Klasse `B`.**
  In diesem Fall ist das **Ergebnis** ein Pointer bzw eine Referenz auf das im **`D`-Objekt enthaltene Teil-Objekt der Klasse `B`.**
  → dieser Fall **entspricht** der **impliziten Standardkonvertierung**

  Beispiel :

```
class B { /* ... */ };

class D : public B { /* ... */ };

// ...
D clD;
B* pclB1 = dynamic_cast<B*>(&clD);
B* pclB2 = &clD;                    // pclB1 == pclB2 !
```

  b) Der **Quellausdruck `e`** ist ein Pointer auf ein Objekt einer **polymorphen Klasse** und **`T`** ist der Typ **`void*`.**
  In diesem Fall ist das **Ergebnis** ein Pointer auf das **vollständige** durch **`e`** tatsächlich referierte Objekt.

  c) Der **Quellausdruck `e`** ist ein Pointer auf ein Objekt bzw ein Lvalue einer **polymorphen Klasse `B`** und **`T`** ist **nicht** der Typ **`void*`.**
  In diesem Fall wird mittels einer **Laufzeit-Typprüfung** des tatsächlich referierten Objekts **geprüft**, ob der Quellausdruck **`e`** in den Zieltyp **`T`** **umgewandelt** werden kann :

  ▻ Ist **`T`** ein Pointer bzw eine Referenz auf eine **von `B public` abgeleitete Klasse `D`** und wird **durch `e`** tatsächlich ein **Objekt dieser Klasse `D`** oder einer **von `D` abgeleiteten Klasse** referiert, so ist das **Ergebnis** ein Pointer bzw eine Referenz auf das **`D`-(Teil-)Objekt**.

  ▻ Ist **`T`** ein Pointer bzw eine Referenz auf eine **Klasse `A`,** die zugreifbare und eindeutige **Basisklasse** des **durch `e` tatsächlich referierten Objekts** ist, so ist das **Ergebnis** ein Pointer bzw eine Referenz auf das **`A`-Teil-Objekt** des von **`e`** referierten Objekts.

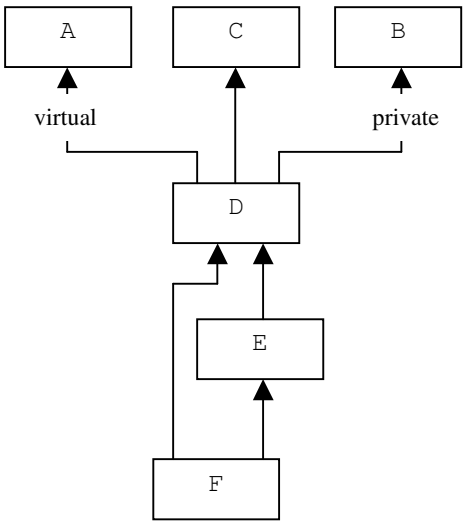
  ▻ In allen **übrigen Fällen** ist die **gewünschte Typumwandlung nicht möglich**.
  → Wenn **`T`** ein **Pointer-Typ** ist, wird als **Ergebnis** der **`NULL`-Pointer** erzeugt;
  wenn **`T`** eine **Referenz** ist, wird die **Exception `bad_cast`** geworfen.

Der Typkonvertierungsoperator `dynamic_cast` in C++ (2)

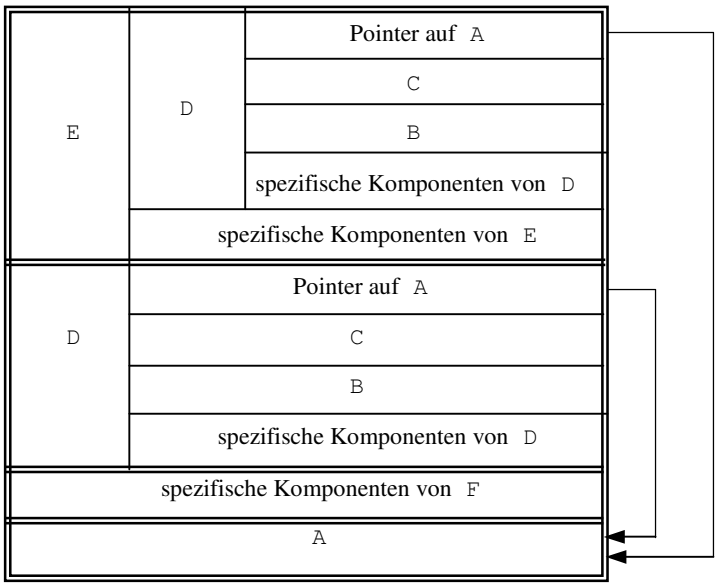
- **Beispiel für Typwandlungen bei polymorphen Klassen** :

```
class A { virtual void f(); /* ...*/ };

class B { virtual void g(); /* ...*/ };

class C { virtual void h(); /* ... */ };

class D : public virtual A, public C , private B { /* ... */ };

class E : public D { /* ... */ };

class F : public E, public D { /* ...*/ };    // in Visual-C++ 6.0 nicht zulaessig

void func(void)
{ A  a;
  D  d;
  E  e;
  F  f;
  A* ap = &a;
  B* bp = dynamic_cast<B*>(&d);    // D* → B*  Fehlschlag (private)
  D* dp = dynamic_cast<D*>(ap);    // A* → D*  Fehlschlag (ap zeigt auf A)
  C* cp = dynamic_cast<C*>(ap);    // A* → C*  Fehlschlag (ap zeigt auf A)
  ap = &d;                         // D* → A*  implizit
  dp = dynamic_cast<D*>(ap);       // A* → D*  hier o.k.(ap zeigt auf D)
  D& dr2 = dynamic_cast<D&>(*ap); // A  → D&  o.k. (ap zeigt auf D)
  cp = dynamic_cast<C*>(ap);       // A* → C*  o.k. (ap zeigt auf D, C ist auch
                                   //           Basisklasse von D)
  bp = dynamic_cast<B*>(ap);       // A* → B*  Fehlschlag (ap zeigt auf D,
  ap = &e;                         //           B ist private Basisklasse von D)
  dp = dynamic_cast<D*>(ap);       // A* → D*  o.k.(ap zeigt auf E,
                                   //           D ist public-Basisklasse von E)
  ap = &f;                         // o.k.,     nur ein A in F enthalten
                                   //           (virtuelle Basisklasse)
  dp = dynamic_cast<D*>(ap);       // A* → D*  Fehlschlag (mehrdeutig)
  E* ep1 = (E*)ap;                 // Fehler,   C-compatibler cast von
                                   //           virtueller Basis
  E* ep2 = dynamic_cast<E*>(ap);   // A* → E*  o.k.(ap zeigt auf F,
                                   //           E ist public-Basisklasse von F)
 }
```

Klassendiagramm



Objekt der Klasse `F`



Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 5

5. Funktions- und Klassen-Templates

5.1. Generische Funktionen (Funktions-Templates)

5.2. Generische Klassen (Klassen-Templates)

Generische Funktionen (Funktions-Templates) in C++ (1)

- **Allgemeines** :

  ◇ Generische Funktionen sind **Funktions-Schablonen**, die ein Muster für eine ganze **Gruppe von Funktionen**, die den **prinzipiell gleichen Algorithmus** für jeweils **unterschiedliche Parametertypen** ausführen, definieren.

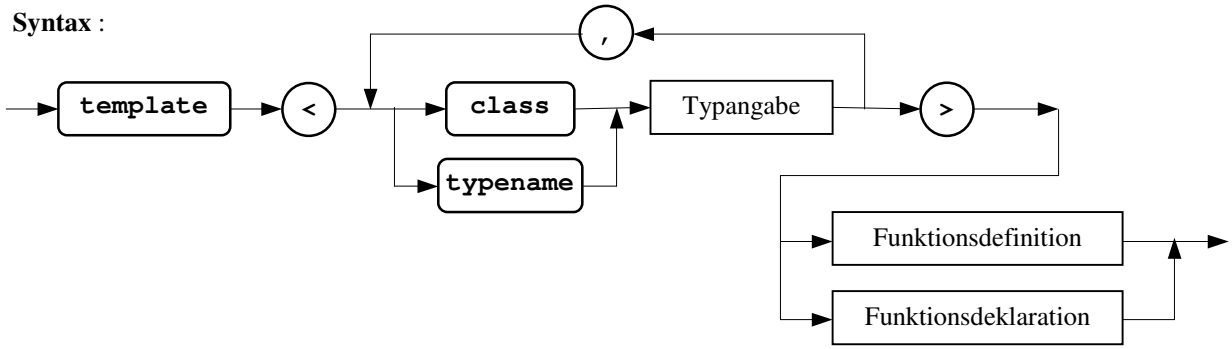
  ◇ Die Definition einer generischen Funktion (Funktions-Template, ***function template***) erzeugt noch keinen Code. Passender – den Typen der jeweiligen aktuellen Parameter entsprechender – **Code** wird **vom Compiler** erst **bei der erstmaligen Verwendung der generischen Funktion** mit diesen Parameter-Typen erzeugt (**Instantiierung** des Templates → **instantiierte Funktion**, **Template-Funktion**, ***template function***).

  ◇ Im Prinzip stellt eine generische Funktion eine Funktion dar, die sich **automatisch selbst überladen** kann. Im Unterschied zu allgemeinen überladenen Funktionen, bei denen jede Funktion prinzipiell einen anderen Algorithmus realisieren und eine unterschiedliche Anzahl von Parametern haben kann, haben alle Versionen einer generischen Funktion die gleiche Anzahl von Parametern und realisieren den gleichen Algorithmus - nur eben mit unterschiedlichen Parametertypen.

  ◇ In der Vereinbarung einer generischen Funktion werden die **Parametertypen**, die sich **ändern** können, als (Template-) **Parameter** festgelegt.

- **Vereinbarung einer generischen Funktion** :

  ◇ **Syntax** :



  ◇ Typangabe ist ein **formaler Typ-Parameter** (**Template-Parameter**). Jeder Typ-Parameter kann innerhalb der **Parameterliste der Funktionsvereinbarung** als Platzhalter für eine **aktuelle Typangabe** verwendet werden (Festlegung des Typs eines Funktionsarguments).

  ◇ Ein Template-Parameter kann auch verwendet werden
  - ▻ zur **Festlegung** des **Rückgabetyps** der Funktion
  - ▻ zur **Festlegung** des Typs **lokaler Variabler** der Funktion

  ◇ **Beispiele** :

```
template <class T> void swap(T& a, T&b)
{
  T h=a;
  a=b;
  b=h;
}

template <typename TYPE1, typename TYPE2>
void myfunc(TYPE1 x, TYPE2 y)
{
  cout << x << "   " << y << '\n';
}

template <class T1, class T2>
T2 convert(T1 w)
{
  return (T2)w;
}
```

Generische Funktionen (Funktions-Templates) in C++ (2)

• Aufruf von Template-Funktionen

◇ Eine Template-Funktion wird **bei** ihrem **erstmaligen Aufruf generiert**. (→ **Implizite Instantiierung**)
Für die **Angabe des Funktionsnamens** existieren hierfür zwei Möglichkeiten :
- ▻ Verwendung nur des Template-Namens als Funktionsnamen.
- ▻ Ergänzung des Template-Namens um die Angabe aktueller Template-Parameter

◇ Bei **Aufruf** einer Template-Funktion **allein** mit dem **Template-Namen** müssen die zu verwendenden aktuellen Template-Parameter aus den aktuellen Funktions-Parametern (Funktions-Argumenten) ermittelbar sein.
Das bedeutet,
- sämtliche Template-Parameter müssen zur Festlegung des Typs von Funktions-Parametern dienen
- die Typen der aktuellen Funktions-Parameter müssen – ohne Anwendung impliziter Typkonvertierungen – eine **eindeutige Template-Instantiierung** ermöglichen.

◇ Bei **Aufruf** einer Template-Funktion mit dem um eine **aktuelle Parameterliste ergänzten Template-Namen** legt die Parameterliste die aktuellen Template-Parameter fest.
In diesem Fall können gegebenenfalls implizite Typkonvertierungen der aktuellen Funktions-Parameter in die aktuellen Template-Parameter stattfinden.
Dienen Template-Parameter allein zur Festlegung des Rückgabetyps der Funktion und/oder allein zur Festlegung des Typs von lokalen Funktions-Variablen muß diese Form des Aufrufs verwendet werden.

• Templates bei mehreren Programm-Modulen

◇ Bei der Übersetzung eines Template-Funktions-Aufrufs (d.h. bei der Erzeugung einer Template-Funktion) muss dem Compiler der Code des Funktions-Templates vorliegen.

◇ Soll dasselbe Funktions-Template in mehreren Modulen verwendet werden, ist es zweckmässig den **Template-Code** (also die Template-Definition) in eine **Headerdatei** aufzunehmen, die dann von den verwendenden Modulen eingebunden werden muss.

◇ Eine Template-Funktion, die nicht als `inline` definiert worden ist, wird wie eine normale globale Funktion behandelt. Sie lässt sich daher aus verschiedenen Modulen aufrufen. Andererseits darf sie in einem Programm nur einmal definiert sein.
Der o.a. Mechanismus bewirkt aber zunächst, dass eine Template-Funktion in jedem Modul, in dem sie aufgerufen wird, auch erzeugt (d.h. definiert) wird.
Der Linker ist dafür verantwortlich, dass – bis auf eine – alle weiteren Instanzen einer derartigen Funktion wieder entfernt werden.

• Überladen generischer Funktionen :

◇ Ein Funktions-Template läßt sich auch **explizit überladen**.

◇ Das Überladen kann erfolgen mit
- einem **weiteren Funktions-Template**, das eine **unterschiedliche Funktions-Parameter-Liste** besitzt.
- einer **Nicht-Template-Funktion**, die eine von dem Funktions-Template **abweichende Parameter-Liste** besitzt.
- einer **Nicht-Template-Funktion**, deren **Parameter-Liste** mit der Parameter-Liste einer auch aus dem Template instantiierbaren Template-Funktion **übereinstimmt**.
  In diesem Fall "überschreibt" die explizit überladende Funktion die spezielle – ihren Parametertypen entsprechende – Template-Funktion (Instanz des Funktions-Templates).
  → Definition einer **expliziten Spezialisierung** des Funktions-Templates.

◇ ANSI-C++ sieht zur **Kennzeichnung** einer **expliziten Template-Spezialisierung** den Vorsatz

```
template<>
```

vor, der der Funktions-Vereinbarung vorangestellt werden kann.

Generische Funktionen (Funktions-Templates) in C++ (3)

- **Demonstrationsprogramm `genfunc1`**

```
// ------------------------------------------------------------------
// Programm genfunc1     (C++-Quelldatei genfunc1_m.cpp)
// Beispiel zu generischen Funktionen und zu zusätzlicher explizit
// überladender Funktion (explizite Spezialisierung)
// ------------------------------------------------------------------

#include <iostream>
using namespace std;


template <class TY> TY max(TY a, TY b)
{
  return a>b ? a : b;
}


template <typename TYPE1, typename TYPE2>
TYPE2 convert(TYPE1 w)
{
  return (TYPE2)w;
}


template<>                          // explizite Spezialisierung :
char *max(char *a, char *b)         // "überschreibt" generische Version
{                                   // von char *max(char *, char *)
  int i=0;
  while (a[i]==b[i] && a[i]!=0)
    i++;
  return a[i]>b[i] ? a : b;
}


int main(void)
{
  cout << "\nint-max        : " << max<int>(7, -2);
  cout << "\ndouble-max     : " << max(3.14, 27.9);
  cout << "\nchar-max       : " << max('a', 'z');
  cout << "\nstring-max     : " << max("Hausdach", "Haus");
  cout << "\nlong-max       : " << max(-256000L, 4L);
  cout << "\nlong-max       : " << max<long>(-256000L, 4);
  cout << "\nconvert (d->i) : " << convert<double, int>(45.14);
  cout << '\n';
  return 0;
}
```

- **Ausgabe des Programms**

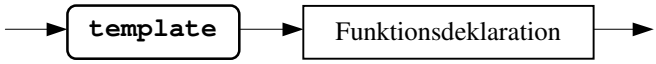
```
int-max        : 7
double-max     : 27.9
char-max       : z
string-max     : Hausdach
long-max       : 4
long-max       : 4
convert (d->i) : 45
```

Generische Funktionen (Funktions-Templates) in C++ (4)

• Explizite Instantiierung eines Funktions-Templates

◇ ANSI-C++ sieht auch eine explizite Instantiierung eines Funktions-Templates vor.
Hierbei wird der Code einer Template-Funktion erzeugt, ohne dass die Funktion aufgerufen wird.

◇ **Syntax** :



Beispiel :
```
template <class TY>            // Template-Definition
TY max(TY a, TY b) { return a>b ? a : b;}
template int max(int, int);    // Template-Instantiierung
```

◇ Eine Explizite Template-Instantiierung sollte zweckmässigerweise in einer `cpp`-Datei und nicht in einer Headerdatei stehen. Diese `cpp`-Datei kann dann auch die Template-Definition enthalten.

◇ Eine explizit instantiierte Funktion kann **in** anderen **Übersetzungseinheiten verwendet** (aufgerufen) werden, **ohne** dass sie **erneut generiert** werden muss.
Für ihre Verwendung ist lediglich eine – häufig in eine Headerdatei gestellte – **Extern-Deklaration** erforderlich :
- entweder der jeweiligen Template-Funktion
- bzw (als gemeinsameDeklaration für alle Template-Funktionen) des Funktions-Templates

• Beispiel zur expliziten Instantiierung eines Funktions-Templates

```
// C++-Headerdatei templmax2.h
// Nur Deklaration des Funktions-Templates max<>

template <class TY> TY max(TY a, TY b);
```

```
//C++-Quelldatei templmax.cpp
//Explizite Instantiierung eines Funktions-Templates

#include "templmax2.h"     // Einbinden der Headerdatei hier nicht erforderlich

template <class TY> TY max(TY a, TY b)
{ return a>b ? a : b;}

template int max(int, int);
template char max(char, char);
template double max(double, double);
template long max(long, long);
```

```
// Programm genfunc2     (C++-Quelldatei genfunc2_m.cpp)
// Beispiel zur Verwendung der expliziten Instantiierung eines Funktions-Templates

#include <iostream>
using namespace std;

#include "templmax2.h"

int main(void)
{ cout << "\nint-max    : " << max(7, -2);
  cout << "\ndouble-max : " << max(3.14, 27.9);
  cout << "\nchar-max   : " << max('a', 'z');
  //cout << "\nstring-max : " << max("Hausdach", "Haus"); // keine Instantiierung !
  cout << "\nlong-max   : " << max<long>(-256000L, 4);
  cout << '\n';
  return 0;
}
```

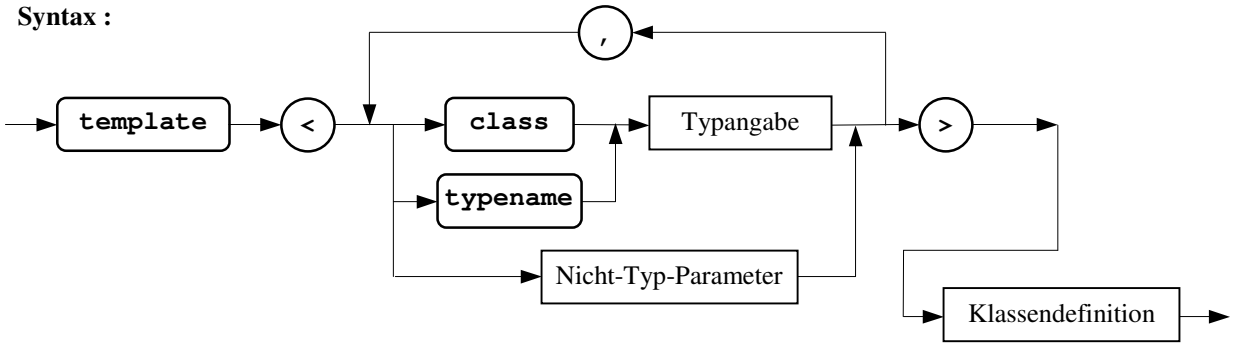
Generische Klassen (Klassen-Templates) in C++ (1)

- **Allgemeines :**

  Generische Klassen (Klassen-Templates, *class templates*) sind **Schablonen**, die ein **Definitions-Muster** für eine ganze **Gruppe von Klassen**, die **prinzipiell gleich aufgebaut** sind und deren **Member-Funktionen** den jeweils **prinzipiell gleichen Algorithmus** realisieren, festlegen.
  Die einzelnen nach diesem Muster konstruierbaren Klassen unterscheiden sich lediglich im **Typ einer oder mehrerer Komponenten** (und in den dadurch gegebenen Auswirkungen auf die Member-Funktionen).
  Die **Typen**, in denen sich die **einzelnen Klassen unterscheiden**, werden bei der **Definition** einer generischen Klasse als **formale Parameter** festgelegt. (Generische Klassen werden daher auch als **parameterisierte Typen** bezeichnet).

- **Definition einer generischen Klasse :**

  ◇ **Syntax :**



  ◇ **Beispiel :**

```
template <class T> class Stack  // Generische Klasse "Stack für Typ T"
{                               // (Klassen-Template)
  public:
    Stack (int);          // Konstruktor
    void push(T);         // Ablage eines Elements auf Stack
    T pop(void);          // Rückholen eines Elements vom Stack
  private:
    int size;             // Groesse des Stacks
    T *stck;              // Pointer auf Speicherbereich des Stacks
    int tos;              // Index des Top des Stacks (Stackpointer)
};
```

- **Template-Klassen :**

  ◇ Die **Definition einer generischen Klasse** generiert noch **keine konkrete Klasse** (und erzeugt noch keinen Code für Member-Funktionen). Dies erfolgt erst bei der **erstmaligen Verwendung** ihres Namens (des **Klassen-Template-Namens**) zusammen **mit aktuellen Parametern**
  → **implizite** (z.B. in einer Objekt-Vereinbarung) oder **explizite Instantiierung** des Klassen-Templates.
  Jeder **unterschiedliche** Satz **aktueller Parameter** definiert eine **neue Klasse**
  (→ instantiierte Klasse, **Template-Klasse**, *template class*).

  ◇ Der **Klassen-Template-Name** gefolgt von **in spitzen Klammern eingeschlossenen aktuellen Parametern** (Parameter-Zusatz) stellt den **Namen einer konkreten Klasse** dar und kann genauso wie jeder andere Klassenname verwendet werden (→ **Template-Klassen-Name**).

  ◇ **Syntax :** **`klassen-template-name<akt_par_liste>`**

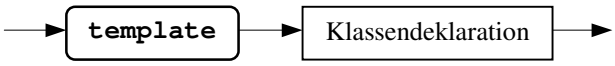
  ◇ **Beispiele :**

```
// implizite Template-Instantiierungen
Stack<int>    is(20);       // Stack fuer int-Werte
Stack<char*>  cps(30);      // Stack fuer char-Pointer
Stack<double> ds(50);       // Stack fuer double-Werte
```

Generische Klassen (Klassen-Templates) in C++ (2)

- **Explizite Instantiierung eines Klassen-Templates**

  ◇ **Syntax** :

  Beispiel : `template class Stack<int>;` `// Stack fuer int-Werte`

- **Verwendung des Klassen-Template-Namens**

  ◇ **Außerhalb** der **Definition der generischen Klasse** darf der **Klassen-Template-Name nicht allein** sondern nur **zusammen mit** einem **Parameter-Zusatz** verwendet werden (gegebenenfalls mit den formalen Parametern, s. Definition der Member-Funktionen generischer Klassen).

  ◇ **Innerhalb** der Klassen-Definition **darf** - muß aber nicht - bei **Eindeutigkeit** der **Parameter-Zusatz** mit den formalen Parametern **auch weggelassen** werden.
  Lediglich bei **Konstruktor**- und **Destruktor**-Namen **darf** der Parameter-Zusatz **nicht** angegeben werden.

  ◇ **Beispiel :**

```
template <class T> class List       // Generische Klasse : "Element für
{                                   // lineare Liste von Werten des Typs T"
  public :
    List(T);                        // Konstruktor
    ~List(void);                    // Destruktor
    void add(List<T> *node);        // oder :  ...(List* node);
    List<T> *getNext(void) const;   // oder :  List* get...
    T getData(void)const;
  private :
    T data;
    List<T> *next;                  // oder :  List* next;
};
```

- **Explizite Spezialisierung einer generischen Klasse**

  ◇ Eine **generische Klasse** kann durch die **explizite Definition** einer **konkreten Klasse** für einen speziellen aktuellen Parametersatz "**überladen**" werden. Kennzeichnung nach ANSI-C++ : Vorsatz **`template<>`**
  Sofern die explizite Definition vor Anwendung der konkreten Klasse erfolgt, hat sie **Vorrang** vor einer Instantiierung der generischen Klasse für den speziellen Parametersatz.

  ◇ Innerhalb einer derartigen Klassendefinition müssen **Konstruktornamen** - **nicht** jedoch **Destruktornamen** - um einen **Parameter-Zusatz** (mit dem speziellen aktuellen Parametersatz) **ergänzt** werden.

  ◇ Für eine explizit definierte konkrete Klasse müssen auch **alle Member-Funktionen** – auch wenn sie im Namen und Aufbau denen der generischen Klasse entsprechen – **gesondert definiert** werden.

  ◇ **Beispiel :**

```
template <class T> class List    // Definition generische Klasse
{ /* ... */ };

template<>                       // Kennzeichnung einer expliziten Spezialis.
class List<char*>                // explizite Spezialisierung
{ public:                        // "Überladen" der generischen Klasse
    List<char *>(char *);
    ~List();
    void add(List<char*> *node);
    List<char*> *getNext(void) const;
    char *getData(void) const;
  private:
    char *data;
    List<char*> *next;
};
```

Generische Klassen (Klassen-Templates) in C++ (3)

• Member-Funktionen generischer Klassen

◇ Für **jede** aus einer generischen Klasse erzeugten **konkreten Klasse** wird ein **eigener Satz Member-Funktionen** angelegt.

◇ Jede Member-Funktion einer generischen Klasse ist **implizit** eine **generische Funktion** (Funktions-Template) mit der **gleichen Typ-Parameterliste** wie die **generische Klasse**.
Bei ihrer **Definition außerhalb der Klassendefinition** muß sie daher **als Funktions-Template** (Beginn ebenfalls mit **`template < ... >`**) formuliert werden.

Beispiel :

```
template <class T> class List
{ public:
    List(T);                          // Konstruktor
    void add(List<T> *node);          // Element zur Liste hinzufügen
    T getData(void)const;             // Rückgabe Komponente data
  // ...
};

template <class T> List<T>::List(T d)  // generischer Konstruktor
{ data=d; next=NULL; }

template <class T> void List<T>::add(List<T> *node)
{ node->next=this; next=NULL; }

template <class T> T List<T>::getData(void) const
{ return data; }
```

◇ Für **spezielle aktuelle Parameter** - also für eine spezielle konkrete Klasse (spezielle Instanz des Klassen-Templates) – kann eine **Member-Funktion** auch **überladen** werden.
Bei einer expliziten Spezialisierung müssen alle Memberfunktionen gesondert definiert, d.h. überladen werden.

Beispiel :

```
List<char *>::List(char *s)     // Konstruktor für konkrete Klasse
{ data=new char[strlen(s)+1];   // Überladen des generischen Konstruktors
  strcpy(data, s);
  next=NULL;
}
```

◇ **Explizite Funktions-Templates** als Member-Funktionen sind auch **möglich**.

• Befreundete Funktionen generischer Klassen

◇ Sind **nicht implizit** generische Funktionen.

◇ Eine befreundete Funktion kann **von Template-Parametern unabhängig** und damit für alle aus dem Klassen-Template erzeugbaren **konkreten Klassen dieselbe** Freund-Funktion sein.

◇ Eine befreundete Funktion kann aber auch **von Template-Parametern abhängen** und damit für **jede konkrete Klasse unterschiedlich** sein. In diesem Fall sollte sie **explizit als generische Funktion** definiert werden.

Demonstrationsbeispiel zu generischen Klassen in C++ (1)

```
// ------------------------------------------------------------------------
// Header-Datei  StackTempl.h
// ------------------------------------------------------------------------
// Definition eines Klassen-Templates Stack
// ("einfacher Stack für unterschiedliche Objekte")
// ------------------------------------------------------------------------

#ifndef STACK_TEMPL_H
#define STACK_TEMPL_H

#define DEF_SIZE 10

template <class T> class Stack   // Generische Klasse "Stack für ..."
{                                // (Klassen-Template)
  public:
    Stack (int=DEF_SIZE);        // Konstruktor
    ~Stack();                    // Destruktor
    void push(T);                // Ablage eines Elements auf Stack
    T pop(void);                 // Rückholen eines Elements vom Stack
  private:
    int size;                    // Größe (Tiefe) des Stacks
    T* stck;                     // Pointer auf Speicherbereich des Stacks
    int tos;                     // Index des Top of Stack (Stackpointer)
};

#endif
```

```
// ------------------------------------------------------------------------
// C++-Quell-Datei  StackTempl.cpp
// Implementierung des Klassen-Templates Stack
// ------------------------------------------------------------------------

#include "StackTempl.h"

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

template <class T> Stack<T>::Stack(int s)
{ stck=new T[size=s];
  tos=0;
}

template <class T> Stack<T>::~Stack()
{ delete [] stck;
  stck=NULL;
  size=tos=0;
}

template <class T> void Stack<T>::push(T c)
{ if (tos==size)
    cout << "Stack ist voll !\n";
  else
  { stck[tos]=c;
    tos++;
  }
}

template <class T> T Stack<T>::pop(void)
{ if (tos==0)
  { cout << "Stack ist leer !\n";
    return (T)0;
  }
  else
  { tos--;
    return stck[tos];
  }
}
```

Demonstrationsbeispiel zu generischen Klassen in C++ (2)

```
// ------------------------------------------------------------------------
// C++-Quell-Datei templstack_m.cpp
// ------------------------------------------------------------------------
// Programm templstack
// ------------------------------------------------------------------------
// Einfaches Demonstrationsprogramm zu Klassen-Templates in C++
// ------------------------------------------------------------------------
// Verwendung des Klassen-Templates Stack
// ------------------------------------------------------------------------

#include "StackTempl.h"
#include "StackTempl.cpp"   // nur bei impliziter Template-Instantiierung

#include <iostream>
using namespace std;

int main(void)
{
  Stack<char> cs1, cs2(20);
  int i;

  cs1.push('A');
  cs2.push('b');
  cs1.push('C');
  cs2.push('D');
  for (i=1; i<3; i++)
    cout << '\n' << cs1.pop();
  cout << '\n';
  for (i=1; i<3; i++)
    cout << '\n' << cs2.pop();

  Stack<double> ds1(30), ds2;
  ds1.push(3.75);
  ds2.push(-4.7);
  ds1.push(2.3);
  ds2.push(5.83);
  cout << '\n';
  for (i=1; i<3; i++)
    cout << '\n' << ds1.pop();
  cout << '\n';
  for (i=1; i<3; i++)
    cout << '\n' << ds2.pop();
  cout << '\n';
  return 0;
}
```

- **Ausgabe des Programms :**

```

C
A

D
b

2.3
3.75

5.83
-4.7

```

Generische Klassen (Klassen-Templates) in C++ (4)

• Andere Template-Parameter (Nicht-Typ-Parameter)

◇ Klassen-Templates können auch **Parameter** besitzen, die **keine Typen** sind, sondern normalen Funktionsparametern entsprechen.
Als entsprechende aktuelle Parameter sind nur zulässig :
- konstante Ganzzahl- oder Aufzählungstyp-Ausdrücke (bei Nicht-Referenz-Parametern)
- Adressen von global zugreifbaren Objekten und Funktionen (bei Pointer-Parametern)
- Referenzen auf Objekte, die global (Speicherklasse `extern`) oder als `static`-Komponente definiert wurden (bei Referenz-Parametern)

Sinnvoll sind solche Nicht-Typ-Parameter, wenn über sie bestimmte Eigenschaften einer konkreten Klasse oder ihrer Komponenten festgelegt werden können (z.B. die Größe eines Arrays o.ä.)

◇ **Beispiel :**

```
template <class T, int i> class Stack
{ public:
    Stack (void);
    void push(T);
    T pop(void);
  private:
    int size;
    T stck[i];  // Größe des Stacks durch Template-Parameter bestimmt
    int tos;
};

template <class T, int i> Stack<T,i>::Stack(void)
{
  size=i;
  tos=0;
}

// ...

int main(void)
{
  Stack<char,20> cs1;   // cs1 und cs2 sind Objekte
  Stack<char,50> cs2;   // unterschiedlicher Klassen
  Stack<char,2*25> cs3; // cs2 u. cs3 sind Objekte der gleichen Klasse
  Stack<double,30> ds;
  // ...
}
```

◇ **Zwei** aus **einem Klassen-Template** erzeugte **konkrete Klassen** (Template-Klassen) sind nur **dann gleich**, wenn sie in den **aktuellen Typ-Parametern übereinstimmen** und ihre aktuellen **Nicht-Typ-Parameter gleiche Werte** haben (s. obiges Beispiel).

• `typedef`-Namen für Template-Klassen

◇ Häufig eingeführt, um die wiederholte Auflistung von Template-Parametern zu vermeiden.

◇ **Beispiel :**

```
typedef Stack<char,20>   CharStack20;
typedef Stack<char,50>   CharStack50;
typedef Stack<double,30> DoubleStack30;
// ...
CharStack20 cs1;
CharStack50 cs2;
DoubleStack30 ds;
```

Generische Klassen (Klassen-Templates) in C++ (5)

• Generische Klassen mit statischen Komponenten

◇ Wenn eine generische Klasse statische Komponenten besitzt, werden für **jede** von ihr erzeugte **Template-Klasse eigene Vertreter** dieser **statischen Komponenten** angelegt.

◇ Die **Definition** (Speicherplatz-Reservierung) für die **statischen Datenkomponenten** kann
- für alle Template-Klassen mittels einer **Template-Vereinbarung** gemeinsam formuliert werden oder
- für jede Template-Klasse gesondert erfolgen (z.B. zur getrennten Angabe von Initialisierungswerten)

◇ **Beispiel** :

```
template <class T> class X
{ //  ...
  public :
    static T s;
  //  ...
};

template <class T> T X<T>::s;     // Template-Vereinbarung für stat. Datenkomp.

double X<double>::s = 5.26;

int main(void)
{
  X<int> xi;        // X<int> besitzt statische. Komponente s vom Typ int
  X<char*> xcp;     // X<char*> besitzt statische Komponente s vom Typ char*

  X<int>::s=12;
  X<char*>::s="Hallo !";

  cout << endl << "s von X<int>    : " << X<int>::s;
  cout << endl << "s von X<char*>  : " << X<char*>::s;
  cout << endl << "s von X<double> : " << X<double>::s;
  cout << endl;

  return 0;
}
```

• Klassen-Templates als Template-Parameter

◇ Eine generische Klasse (oder eine generische Funktion) kann auch **Parameter** besitzen, die selbst **Klassen-Templates** sind (→ **Verschachteln von Templates**).

◇ **Beispiel :** `Stack<Stack<int> > istackstack;  // Stack von int-Stacks`

◇ **Anmerkung** :Die beiden spitzen Klammern '**>**' müssen durch mindestens ein Leerzeichen getrennt werden, damit sie nicht als Operator '**>>**' interpretiert werden.

• Ort von Template-Definitionen

Klassen- (und Funktions-)Templates dürfen **nur** auf der **globalen Ebene** oder **innerhalb** einer **Klasse** bzw eines **Klassen-Templates** (***member templates***) definiert werden.
→ Klassen-Templates können also auch **innere Klassen**, **nicht** jedoch **lokale Klassen**, beschreiben.

Generische Klassen (Klassen-Templates) in C++ (6)

• Default-Argumente von Templates

◇ Für Template-Parameter (sowohl von Klassen-Templates als auch Funktions-Templates) können auch **Default-Werte** festgelegt werden.
Dies gilt sowohl für Typ-Parameter als auch für Nicht-Typ-Parameter.

◇ Die Festlegung von Default-Parametern erfolgt
- entweder in der **Template-Definition**
- oder in einer **Template-Deklaration** in einem Modul

Beispiel :
```
template <class T =double, int n = 256>
  class Array
  { /* ... */ };
```

◇ In der **gleichen Übersetzungseinheit** (Modul) darf für ein- und denselben Template-Parameter **nur eine Default-Festlegung** angegeben werden.

◇ Default-Werte für Template-Parameter müssen – wie bei Funktions-Parametern – immer **am Ende der Parameterliste** angegeben werden.
Wird für einen Parameter ein Default-Wert festgelegt, so müssen für alle folgenden Parameter ebenfalls Default-Werte angegeben werden.

Beispiel :
```
template <class T1 = int, class T2> class B;     // Fehler !
```

◇ Bei der **Instantiierung** eines Templates mit Default-Parametern können dann die **entsprechenden aktuellen Parameter weggelassen** werden.

Beispiel :
```
Array<char> buffer;     // Typ : Array<char, 256>
```

◇ Existieren für alle Template-Parameter Default-Festlegungen und werden auch alle verwendet, kann bei der Angabe der Template-Klasse eine **leere Parameterliste** angegeben werden.

Beispiel :
```
Array<> polvec;         // Typ : Array<double, 256>
```

Achtung : Die leere Parameterliste (öffnende und schliessende spitze Klammern) muss angegeben werden.

• Anwendung von Klassen-Templates

◇ Die Hauptanwendung von Klassen-Templates liegt in der Implementierung von **Container-Klassen**.
Container-Klassen sind Klassen, deren Objekte zur **Verwaltung anderer Objekte** dienen. Die verwalteten Objekte sind entweder als Komponenten enthalten oder werden über enthaltene Pointer-Komponenten referiert.
Typische Beispiele hierfür sind Stacks, Listen, assoziative Arrays, Mengen u.ä.

◇ Container-Klassen besitzen die **besondere Eigenschaft**, daß der **Typ der Objekte**, die sie verwalten, für die Definition der Klasse von **untergeordnetem Interesse** ist.
Im **Vordergrund** stehen die **Operationen**, die mit den Objekten - unabhängig von ihrem Typ - auszuführen sind.
Mit entsprechend definierten Klassen-Templates lassen sich dann gleichartige Container-Klassen für "Verwaltungs"-Objekte unterschiedlichsten Typs realisieren.

◇ In der **ANSI-C++-Standardbibliothek** sind mehrere derartige Klassen-Templates enthalten :
u.a. **deque**, **list**, **vector, stack** (→ **STL – Standard Template Library**)

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 6

6. Ausnahmebehandlung (Exception Handling)

6.1. Allgemeines

6.2. Werfen und Fangen von Exceptions

6.3. Beispiele

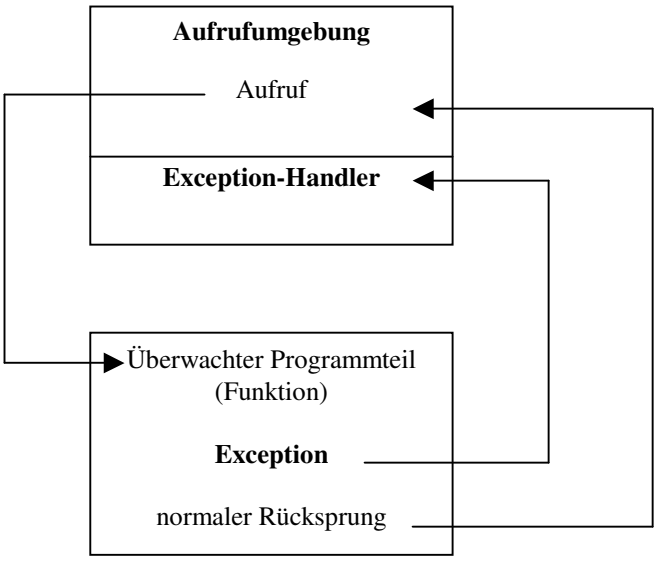
Ausnahmebehandlung in C++ - Allgemeines

- **Ausnahmesituationen**

  C++ stellt einen speziellen Mechanismus zur Behandlung von Ausnahmesituationen (Fehlerfällen, Exceptions) zur Verfügung. ⇒ **Exception Handling**.
  Ausnahmesituationen in diesem Sinne sind **Fehler** oder sonstige unerwünschte **Sonderfälle** (z.B. Dateizugriffsfehler, Fehler bei der dynamischen Speicherallokation, Bereichsfehler usw), die im **normalen Programmablauf** nicht auftreten sollten aber auftreten können.
  Sie werden vom Programmierer festgelegt.
  → Es handelt sich **nicht** um einen Mechanismus zur Behandlung von externen oder internen **Interrupts**.

- **Grundprinzip :**

  Das in C++ implementierte Exception-Handling beruht auf der **Trennung** von der im normalen Programmablauf auftretenden **Fehlererkennung** und der **Fehlerbehandlung** :
  Der Programmteil (i.a.eine Funktion), der einen Problemfall (Fehlerfall) entdeckt, bearbeitet diesen nicht selbst, sondern "**wirft**" **eine Exception** ("**throws an exception"**).
  In der Aufrufumgebung dieses Programmteils sollte eine Bearbeitungsroutine für diese Exception (→ **Exception-Handler**) vorhanden sein. Der Exception Handler "**fängt**" die Exception und behandelt den Problemfall.



- **Exception-Klassen**

  Eine derartige Exception wird durch einen Ausdruck beschrieben, der prinzipiell einen beliebigen Wert ergeben kann.
  Im einfachsten Fall kann dies der Wert eines Standard-Datentyps oder ein char-Pointer (C-String) sein, meist wird es sich dabei aber um ein Klassen-Objekt handeln.
  Durch den Typ des erzeugten Werts bzw Objekts lassen sich verschiedene **Exception-Arten** unterscheiden.
  Zweckmäßigerweise definiert man für die verschiedenen Exceptions spezielle **Exception-Klassen** (Ausnahme-Klassen, Fehler-Klassen, → **Exception-Typ**).
  Beim Auftritt einer Exception wird dann ein Objekt der entsprechenden Klasse erzeugt und geworfen.
  ⇒ die **Exception** wird als **Objekt** behandelt.

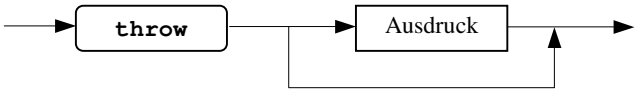
  Wenn eine Exception-Klasse allein zur Anzeige der Exception-Art dienen soll, muß sie keine Komponenten besitzen.
  I.a. wird sie jedoch Komponenten haben, die der Fehlerbehandlung genauere Informationen über die Fehlerursache zur Verfügung stellen.

Ausnahmebehandlung in C++ - "Werfen" einer Exception

• `throw`-Ausdruck

◇ Das **"Werfen" einer Exception** erfolgt durch einen `throw`-Ausdruck. Dieser wird mit dem **unären `throw`-Operator** gebildet, dessen Präzedenz zwischen Komma-Operator und den Zuweisungsoperatoren liegt.

◇ Der **Typ** eines `throw`-Ausdrucks ist **`void`** → ein `throw`-Ausdruck ist nur in der Form einer Ausdrucksanweisung möglich.



◇ Der throw-Ausdruck initialisiert ein **temporäres Objekt** seines Operanden-Typs (=**Exception-Typ**) mit dem Wert seines Operanden (→ "geworfene" Exception), **sucht** den passenden **Exception-Handler**, initialisiert gegebenenfalls dessen Parameter mit diesem Wert und übergibt die Programmfortsetzung an diesen.
→ Der Handler hat die Exception "**gefangen**".
Dabei findet eine **Stackbereinigung** ("*stack unwinding*") statt : Sämtliche `auto`-Objekte (und einfache `auto`-Variable), die seit Eintritt in die zum Exception-Handler gehörende Aufrufumgebung angelegt worden sind, werden entfernt.

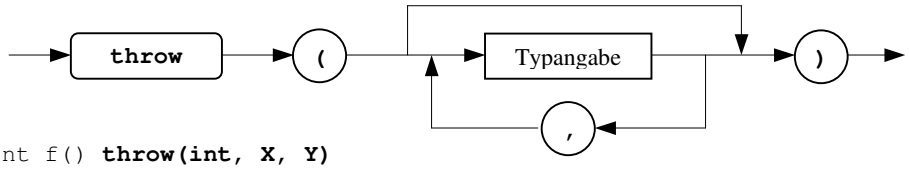
◇ Das vom `throw`-Ausdruck angelegte temporäre Objekt existiert solange, solange ein Exception-Handler für diese Exception ausgeführt wird. Erst nach Beendigung des (letzten) Exception-Handlers für diese Exception wird das Objekt zerstört.

◇ Die Form des `throw`-Ausdrucks **ohne Operanden** ist nur **innerhalb eines Exception-Handlers** (bzw innerhalb einer von einem Exception-Handler aufgerufenen Funktion) zulässig .
Ein derartiger `throw`-Ausdruck bewirkt, daß eine weitere Exception mit dem vorhandenen temporären Objekt geworfen wird, d.h. es wird versucht, die Programmfortsetzung an einen weiteren Exception-Handler zu übergeben (→ "***rethrow***" *the exception*).

◇ Kann durch den `throw`-Ausdruck kein passender Exception-Handler gefunden werden, so wird die (Standardbibliotheks-)Funktion **`terminate()`** aufgerufen.

• Exception-Specification

◇ Ergänzung einer **Funktionsdeklaration** bzw des Funktionskopfes einer **Funktionsdefinition** um eine **Auflistung der Exception-Typen**, die von der **Funktion** – direkt oder indirekt – **geworfen** werden können bzw dürfen.
Eine Exception-Specification kann auch in einer **Funktionspointer-Vereinbarung** enthalten sein.



◇ **Beispiele** :

```
int f() throw(int, X, Y)
{
..// Funktion darf Exceptions vom Typ int, X und Y werfen
}

void (*pf)() throw(A);
```

◇ Enthält eine Vereinbarung (Deklaration oder Definition) einer Funktion eine Exception-Specification, so müssen alle anderen Vereinbarungen derselben Funktion ebenfalls eine Exception-Specification mit derselben Typ-Liste enthalten.

◇ Eine Exception-Specification mit **leerer Typ-Liste** bedeutet, daß die entsprechende Funktion **keine Exceptions** werfen darf. Eine **ohne Exception-Specification** vereinbarte Funktion darf **beliebige Exceptions** werfen.

◇ Wenn eine Typ-Liste den Typ `X` enthält, so darf die betreffende Funktion neben Exceptions dieses Typs auch Exceptions aller Typen, die sich von `X` öffentlich und eindeutig ableiten lassen,werfen

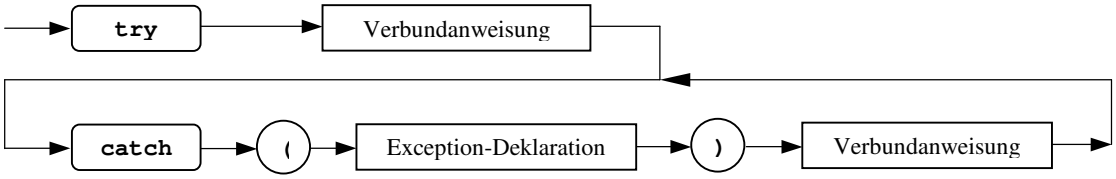
◇ Wird während der Abarbeitung einer Funktion eine Exception geworfen, deren Typ nicht in der Typ-Liste enthalten ist, so wird die Standardbibliotheks-Funktion **`unexpected()`** aufgerufen.

◇ **Anmerkung** : Die Exception-Specification hat bei **Visual-C++ keine Wirkung**.

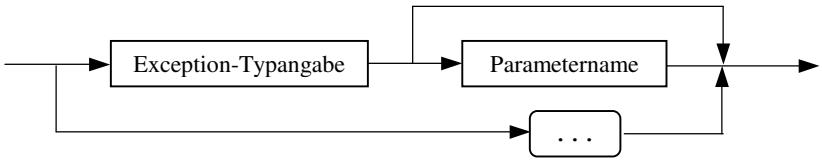
Ausnahmebehandlung in C++ - "Fangen" von Exceptions

- **`try`-Anweisung**

  ◇ Sie legt die Aufrufumgebung und damit den Gültigkeitsbereich einer Ausnahmebehandlung und die dazugehörigen Exception-Handler fest.
  ⇒ Sie besteht aus
  - einer **Verbundanweisung** (**`try`-Block**), die aus den Anweisungen besteht, in denen die Fehlererkennung wirksam ist (Häufig sind Aufrufe von Funktionen enthalten, in denen die eigentliche Fehlererkennung stattfindet).
  - den **Exception-Handlern** (**`catch`-Blöcken**), die den verschiedenen jeweils fangbaren Exception-Typen zugeordnet sind. Für jeden Exception-Typ ist ein eigener Handler vorzusehen.



Exception-Deklaration :



  ◇ **`try`-Anweisungen** können **geschachtelt** werden.

  ◇ Der in der Exception-Deklaration eines Handlers angegebene Typ wird als der **Typ des Handlers** bezeichnet.

- **"Fangen" von Exceptions**

  ◇ Der `throw`-Ausdruck bewirkt eine **Suche nach einem Handler**, der die geworfene Exception "fangen", d.h. bearbeiten kann. Die Suche erfolgt in der Reihenfolge der `catch`-Blöcke.
  Dem **ersten Handler**, der die Exception bearbeiten kann, wird das geworfene **Fehlerobjekt übergeben**.
  Die Übergabe erfolgt **analog zur Parameterübergabe** bei Funktionen, allerdings finden **keine impliziten Typkonvertierungen** für **Standard-Datentypen** statt.

  ◇ Ein Handler kann eine **Exception vom Typ `E`** bearbeiten,
  - wenn der **Typ des Handlers** der Typ **`T`**, oder **`T&`** (einschließlich einer eventuellen Qualifikation mit **`const`** und/oder **`volatile`**) ist und
    - **`E`** und **`T`** der **gleiche Typ** sind (ohne Berücksichtigung der Qualifikation mit `const` bzw `volatile`) oder
    - **`T`** zugreifbare und eindeutige **Basisklasse von `E`** ist
  - oder wenn der **Typ des Handlers** ein **Pointer-Typ** (einschließlich einer eventuellen Qualifikation mit **`const`** und/oder **`volatile`**) ist und **`E`** ebenfalls ein **Pointer-Typ** ist, der mittels Standardkonversion in den Typ des Handlers umgewandelt werden kann (Pointer auf abgeleitete Klasse → Pointer auf Basisklasse).

  ◇ Ein Handler, in dessen Exception-Deklaration statt eines Typs drei Punkte (**`...`**) angegeben ist, kann **jede beliebige Exception** fangen. Falls vorhanden, muß er der **letzte Handler** einer try-Anweisung sein.

  ◇ Wird kein passender Handler gefunden, wird die Suche in der nächsten umfassenden `try`-Anweisung – sofern vorhanden – fortgesetzt usw.
  Wird auch auf diese Weise **kein passender Handler** gefunden, wird die (Standardbibliotheks-) Funktion **`terminate()`** aufgerufen.

  ◇ **Nach** erfolgter **Abarbeitung eines Handlers** wird – sofern der Handler nicht das Programm beendet oder erneut eine Exception geworfen hat – das **Programm** mit der Anweisung, die **auf die `try`-Anweisung folgt**, zu der der Handler gehört, **fortgesetzt**. Es findet also **keine Rückkehr** zu der **Stelle**, an der die **Exception geworfen** wurde, statt.

Einfaches Demonstrationsprogramm zur Ausnahmebehandlung in C++

- **Demonstrationsprogramm `exhdemo` :**

```
//-----------------------------------------------------------------------------
// Programm EXHDEMO
// ----------------------------------------------------------------------------
// Einfaches Demonstrationsprogramm zum Exception Handling
// ----------------------------------------------------------------------------

#include <iostream>
using namespace std;

class IntVec
{ public:
    IntVec(int);
    ~IntVec() {  delete[] ip;  }
    int& operator[](int);
    //  ...
  private:
    int *ip;
    int len;
};

class RangeError                                    // Fehlerklasse
{ public:
    RangeError(int i) : ind(i) { }
    operator int() { return ind; }
  private:
    int ind;                                        // fehlerhafter Index
};

IntVec::IntVec(int i)
{  ip=new int[i];  len=i;  for(i=0; i<len; i++)  ip[i]=i;  }

int& IntVec::operator[](int i)
{ if (i<0 || i>=len)
    throw RangeError(i);                            // "Werfen" der Exception
  else
    return ip[i];
};

int main(void)
{ IntVec iv(20);
  try                                               // try-Block
  { cout << "\niv[10] : " << iv[10] << '\n';
    cout <<   "iv[30] : " << iv[30] << '\n';
    cout << "Ende try-Block\n";
  }
  catch (RangeError& err)                           // catch-Block
  { cout << "Index [" << int(err) << "] out of range\n";
  }
  cout << "Programmende\n";
  return 0;
}
```

- **Ausgabe des Programms :**

```
iv[10] : 10
Index [30] out of range
Programmende
```

Beispiel- und Demonstrationsprogramm zur Ausnahmebehandlung in C++ (1)

- **Erzeugung und Behandlung von Exceptions bei Zugriff zu einem Stack**

  ◊ Definition der folgenden **Exception-Klassen** :
  - ◈ **`MyException`** als Basisklasse für die weiteren Exception-Klassen.
    Dem `protected` Konstruktor ist ein konstanter String, der die Exception-Ursache beschreiben sollte, zu übergeben. Dieser String wird nicht in einen neu allokierten Speicherbereich kopiert, sondern es wird nur seine Adresse in einer Datenkomponente gespeichert.
    Um ein Kopieren von `MyException`-Objekten zu vermeiden, sind Copy-Konstruktor und Zuweisungsoperator als `private` deklariert.
    Mittels der `public` Memberfunktion `getReason()` kann die Adresse des Ursachen-Strings ermittelt werden.
  - ◈ **`StackEmptyException`**
  - ◈ **`StackFullException`** (Klassen-Template)
    Template-Parameter ist der jeweilige Typ der Stack-Elemente
    Dem Konstruktor ist ein Wert vom Typ der Stack-Elemente als Parameter zu übergeben. Es sollte sich um den Wert, bei dessen Ablage-Versuch die Exception aufgetreten ist, handeln. Der übergebende Parameter wird in einer Datenkomponente gespeichert
    Mittels der `public` Memberfunktion `getValue()` kann zu diesem Wert lesend zugegriffen werden.
  - ◈ **`HeapFullException`**

  ◊ Realisierung eines **"allgemeinen" Stacks** durch ein Klassen-Template **`Stack<T>`**
  Template-Parameter ist der Typ der jeweiligen Stack-Elemente.
  Dem Konstruktor ist die (Anfangs-)Kapazität (Größe) des Stacks als Parameter zu übergeben.
  Mittels der `public` Memberfunktion `incCapacity()` läßt sich diese Kapazität um einen bestimmten Bruchteil der aktuellen Kapazität erhöhen. Der Erhöhungs-Bruchteil ist als Parameter zu übergeben
  Sowohl der **Konstruktor** als auch **`incCapacity()`** werfen die Exception **`HeapFullException`**, wenn der benötigte Speicher nicht allokiert werden kann.
  Die `public` Memberfunktion **`push()`** wirft die Exception **`StackFullException`**, wenn der Stack voll ist, die `public` Memberfunktion **`pop()`** wirft die Exception **`StackEmptyException`**, wenn sich kein Element im Stack befindet
  Weiterhin ist die `public` Memberfunktion **`outStack()`** zur Ausgabe des Stackinhalts nach `cout` definiert.

  ◊ Demonstration des Exception Handling mittels eines kleinen **Testprogramms** :
  Die Funktion `main()` ruft eine Funktion **`testStack()`** auf.
  In `testStack()` wird ein **`char`-Stack** untersucht.
  In einer `do`-Schleife wird eine **`try`-Anweisung** ausgeführt, in der nach Ausgabe des jeweils aktuellen Stack-Inhalts die Eingabe der jeweils nächsten auszuführenden Operation angefordert und diese dann ausgeführt wird
  Bei Eingabe von `'q'` wird die `do`-Schleife beendet.
  Die `try`-Anweisung definiert **zwei Exception-Handler** : einen, der **`StackFullException`**s fängt, einen zweiten, der **`StackEmptyException`**s fängt.
  Der Handler für `StackEmptyException`s gibt lediglich die Exception-Ursache aus, anschließend wird die `do`-Schleife fortgesetzt.
  Der Handler für `StackFullException`s versucht durch Aufruf von `incCapacity()` den Stack zu vergrößern, um dann das Element, das zunächst keinen Platz hatte, im vergrößerten Stack abzulegen. Bei Erfolg wird die `do`-Schleife fortgesetzt.
  Kann der Stack nicht vergrößert werden, so wird durch `incCapacity()` die Exception **`HeapFullException`** geworfen, die von einem Exception-Handler in einer übergeordneten `try`-anweisung gefangen werden muß.
  Diese **`try`-Anweisung** ist in **`main()`** enthalten . Sie umschließt den Aufruf von `testStack()`.
  Der durch sie definierte **Exception-Handler** fängt Exceptions vom Typ **`MyException`**, damit auch Exceptions vom Typ **`HeapFullException`**.
  Nach Ausgabe der Exception-Ursache beendet dieser Handler die Funktion `main()` und damit das Programm.

Beispiel- und Demonstrationsprogramm zur Ausnahmebehandlung in C++ (2)

- **Definition der Exception-Klassen** (Header-Datei `"Exceptbsp.h"`)

```
// ------------------------------------------------------------------------
// Header-Datei Exceptbsp.h
// ------------------------------------------------------------------------
// Definition von Eception-Klassen
// ------------------------------------------------------------------------
// für Programm ExceptBsp -
// Beispiel- u. Demonstrationsprogramm zum Exception-Handling
// ------------------------------------------------------------------------

#ifndef EXCEPTBSP_H
#define EXCEPTBSP_H


class MyException
{
  public :
    const char* getReason() const { return m_cpReason; }
  protected :
    explicit MyException(const char *str) { m_cpReason=str;}
  private :
    const char* m_cpReason;                        // Exception-Ursache
    MyException& operator=(const MyException&);    // privater Zuweisungsoperator
};


class HeapFullException : public MyException
{
  public :
    HeapFullException() : MyException("Heap is full") { }
};


class StackEmptyException : public MyException
{
  public :
    StackEmptyException() : MyException("Stack is empty") { }
};


template<class T> class StackFullException : public MyException
{
  public :
    StackFullException(T val) :
      MyException("Stack is full"), m_tValue(val) { }
    T getValue() const { return m_tValue; }
  private :
    T m_tValue;                                    // nicht mehr speicherbarer Wert
  };


#endif
```

Beispiel- und Demonstrationsprogramm zur Ausnahmebehandlung in C++ (3)

- **Definition und Implementierung des Klassen-Templates stack<T>** (Header-Datei `"Stack.h"`)

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

#define DEF_INC   0.2
#define DEF_SIZE  1000

template <class T> class Stack
{
  public :
    explicit Stack(int=DEF_SIZE);
    ~Stack() { delete[] m_tContent; }
    void push(T);
    T pop();
    void incCapacity(double=DEF_INC);
    // ...
    void outStack();
  private :
    T* m_tContent;    // Pointer auf Speicherplatz des Stacks
    int m_iTos;       // Index des ersten freien Speicherplatzes ("Top of Stack")
    int m_iCap;       // Kapazität des Stacks
};

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

template <class T> Stack<T>::Stack(int cap)
{
  m_iTos=0;
  m_iCap=cap;
  if ((m_tContent=new T[cap])==0)
    throw HeapFullException();
}

template <class T> void Stack<T>::push(T val)
{
  if (m_iTos>=m_iCap)
    throw StackFullException<T>(val);
  else
    m_tContent[m_iTos++]=val;
}

template <class T> T Stack<T>::pop()
{
  if (m_iTos<=0)
    throw StackEmptyException();
  else
    return m_tContent[--m_iTos];
}

template <class T> void Stack<T>::incCapacity(double inc)
{
  T* temp=new T[m_iCap=(unsigned)((1+inc)*m_iCap)+1];
  if (temp==0)
    throw HeapFullException();
  else
  { for (int i=0; i<m_iTos; i++)
      temp[i]=m_tContent[i];
    delete [] m_tContent;
    m_tContent=temp;
  }
}

template <class T> void Stack<T>::outStack()
{
  for (int i=m_iTos-1; i>=0; i--)
    cout << m_tContent[i] << " ";
}
```

Beispiel- und Demonstrationsprogramm zur Ausnahmebehandlung in C++ (4)

- **Implementierung des Testprogramms** (C++-Quell-Datei `"Stacktest.cpp"`)

```
// ------------------------------------------------------------------------
// Programm EXCEPTBSP
// ------------------------------------------------------------------------
// Beispiel- u. Demonstrationsprogramm zum Exception-Handling
// ------------------------------------------------------------------------

#include "Exceptbsp.h"
#include "Stack.h"

void testStack()
{
  Stack<char> cStack(5);
  char c;
  do
  { try
    {
      cout << "\nStackinhalt : ";
      cStack.outStack();
      cout << "\npop(-) oder push(+) oder quit(q) : ";
      cin >> c;
      if (c=='-')
        cout << cStack.pop() << endl;
      else
        if (c=='+')
        {
          char cNeu;
          cout << "Zeichen ? ";
          cin >> cNeu;
          cStack.push(cNeu);
        }
    }
    catch (StackFullException<char>& ex)
    {
      cout << ex.getReason() << endl;
      cStack.incCapacity();
      cout << "Stack capacity increased\n";
      cStack.push(ex.getValue());
    }
    catch (StackEmptyException& ex)
    {
      cout << ex.getReason() << endl << endl;
    }
  } while (c!='q');
  cout << endl;
}

int main()
{
  try
  {
    testStack();
    return 0;
  }
  catch (MyException& ex)
  {
    cout << ex.getReason() << endl << endl;
    return 1;
  }
}
```

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 7

7. Entwicklung von OOP-Programmen

7.1. Entwicklungsprozeß

7.2. Modellierung (UML)

Der Softwareentwicklungsprozeß (Überblick)

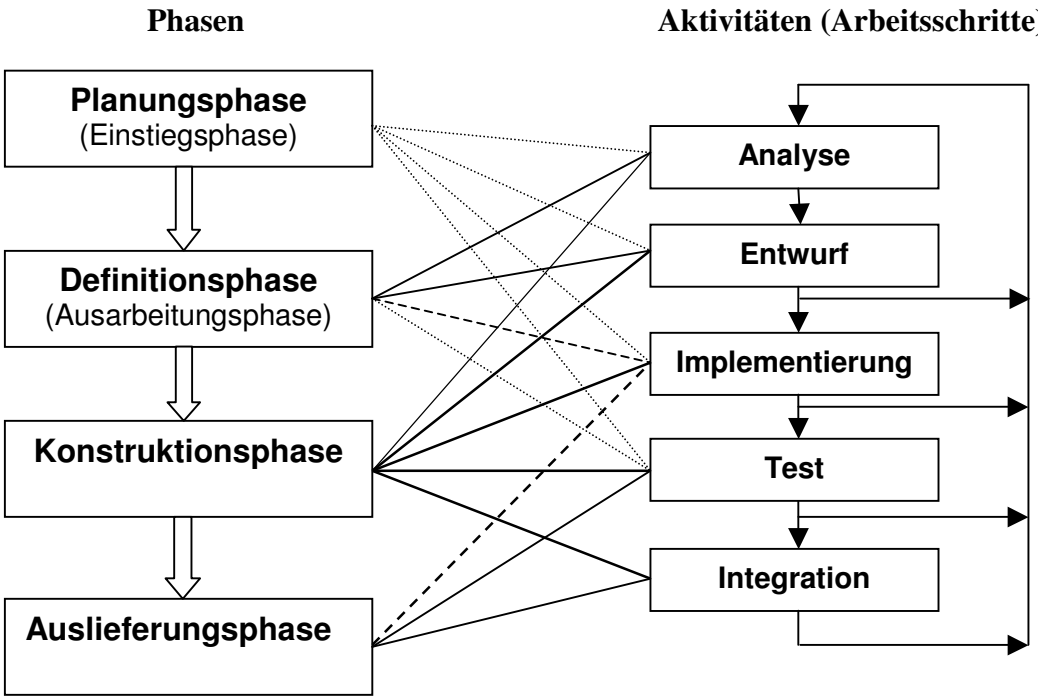
- **Entwicklungsziel** :

  ◇ Das **Ergebnis eines erfolgreichen Softwarentwicklungsprozesses** soll ein **Software-System** sein, dass
  - ▻ die vom Benutzer gestellten **funktionalen** und **nichtfunktionalen Anforderungen und Erwartungen** erfüllt
  - ▻ **zeitgerecht** und **wirtschaftlich** entwickelt wurde
  - ▻ **änderungs**- und **anpassungsfähig** ist

  ◇ Ein derartiger Prozeß muß einerseits ein ausreichendes Maß an **Kreativität** und **Innovation** unterstützen, andererseits muß er die notwendige **Überprüfung** und **Steuerung des Entwicklungsfortschritts** sicherstellen.

  ◇ Diese Forderungen werden von einem **iterativen** und **inkrementellen** Entwicklungsprozeß erfüllt.
  Ein derartiger **Prozeß** besteht aus zeitlich nacheinander ablaufenden **Phasen**, innerhalb denen bestimmte **Aktivitäts-Zyklen** (Arbeitsschrittzyklen) **wiederholt** - mit teilweise unterschiedlicher Gewichtung - ausgeführt werden.

- **Phasen und Aktivitäten des Entwicklungsprozesses** :



  ◇ Insbesondere die **Konstruktionsphase** besteht aus zahlreichen **Iterationen**.
  In jeder Iteration wird ein vollständiger **Aktivitätszyklus** - z.Teil in sich auch wieder zyklisch – durchlaufen, wobei jeweils Software erstellt, getestet und integriert wird, die eine **Teilmenge der Gesamtanforderungen** des Projekts abdeckt.
  → Schrittweise Weiterentwicklung von **Systemprototypen** mit zunächst unvollständiger Funktionalität bis zum fertigen System.
  Ein vollständiger Aktivitätszyklus kann aber auch bereits in der **Planungsphase** und der **Definitionsphase** zur schnellen Erzeugung **explorativer Prototypen** durchlaufen werden.

  ◇ In jeder Entwicklungsphase können sich **Rückwirkungen** auf Ergebnisse und Festlegungen **früherer Phasen** ergeben, insbesondere kann derartig die Konstruktionsphase auf die Definitionsphase rückwirken.

  ◇ In **allen Phasen** und **Tätigkeiten** sollten die jeweiligen Ergebnisse und Festlegungen geeignet **dokumentiert** werden.

Die Phasen des Softwareentwicklungsprozesses

- **Planungsphase** (Einstiegsphase, Anfangsphase,Vorphase, inception phase)

  ◇ **Ziel** : Prüfung, ob ein Projekt realisiert werden soll.

  ◇ Festlegung von Zweck und Ziel eines Projektes,
  Durchführung von Trendstudien und Marktanalysen,
  Untersuchung der Durchführbarkeit
  (Aufwands-, Bedarfs- und Termin-Abschätzung, Risikoanalyse, Witschaftlichkeitsrechnung)
  ⇒ **Durchführbarkeitsstudie** (feasability study)
  bestehend aus : Lastenheft (grobes Pflichtenheft), Projektkalkulation und Projektplan

- **Definitionsphase** (Ausarbeitungsphase, Elaborationsphase, elaboration phase)

  ◇ **Ziel** : Festlegung der vollständigen, konsistenten, eindeutigen und durchführbaren Projektanforderungen

  ◇ Ermittlung und Analyse der Anforderungen (Systemanalyse)
  Problembereichanalyse (→erste Exploration von Klassen), Abgrenzung des Software-Systems zur Systemumgebung
  → Problembereichsmodell
  Identifikation der **Nutzungsfälle** (Use Cases) → Nutzungsfallmodell
  ⇒ Anforderungsspezifikation (Pflichtenheft)

  Entwurf der grundlegenden **Systemarchitektur** (→weitere Exploration von **Klassen**)
  Konzipierung der externen Schnittstellen (z.B. Benutzeroberfläche)
  Riskoanalyse und Priorisierung der Nutzungsfälle
  Festlegung der Entwurfsstrategie
  Planung der Systementwicklung in inkrementellen Entwicklungsschritten (→ Evolutionsplan, Entwurfsmodell)
  ⇒ **Entwurfsspezifikation**

- **Konstruktionsphase** (construction phase)

  ◇ **Ziel** : Erstellung eines die festgelegten Anforderungen erfüllenden Softwaresystems (Programm(e))

  ◇ Entwicklung einer detaillierten Systemstruktur
  Identifikation und Definition weiterer **Klassen/Objekte**
  Konzipierung und Analyse der statischen Klassenbeziehungen → **Klassendiagramme**
  Konzipierung und Analyse der dynamischen Objektinteraktion → **Interaktionsdiagramme**, **Kollaborationsdiagramme**
  → Erstellung **statischer und dynamischer Modelle**

  ◇ Erstellung des Systems in einer Folge von Iterationen (**zyklische Evolution**)
  (jeweils Analyse, Entwurf, Implementierung, Test und Integration)
  → inkrementelles Hinzufügen weiterer Funktionalitäten
  → iterative Änderung des Codes (z.B. Umstrukturierung)
  Validierung jeder Iteration durch einen geeigneten Test
  ⇒ **ausgetesteter Code mit Dokumentation**
  Benutzer- und Installationshandbücher
  Prüf- und Testprotokolle

- **Auslieferungsphase** (Abnahme- u. Einführungsphase, Überleitungsphase, transition phase)

  ◇ **Ziel** : Auslieferung eines fertiggestellten Produkts an den Kunden/Anwender

  ◇ Abnahme des Produkts durch den Kunden (→ Abnahmetest → **Abnahmeprotokoll**)
  Gegebenenfalls Korrektur von festgestellten Fehlern
  Installation und Inbetriebnahme des Produkts
  Schulung der Benutzer

Klassenkategorien

• Allgemeines

◇ Ein **objektorientiertes Softwaresystem** (OO-Programm) ist als eine Ansammlung **interagierender Objekte** organisiert.

◇ Die einzelnen Objekte decken dabei unterschiedliche Aufgabenbereiche ab. Entsprechend ihrem jeweiligen Aufgabenbereich können sie und damit die sie beschreibenden Klassen unterschiedlichen **Kategorien** zugeordnet werden.
→ Objekte und Klassen besitzen einen **Stereotype**.

◇ Auch die gegebenenfalls zwischen Klassen bestehenden **Beziehungen** lassen sich in **verschiedene Arten** einteilen.

• Gebräuchliche Klassenkategorien

▻ **Entity-Klassen**
Das sind Klassen, deren Objekte Bestandteil des Problembereichs sind (**Domänen-Klassen**).
Sie reflektieren entweder **reale Objekte** des Problembereichs oder sie werden zur Wahrnehmung **interner Aufgaben** des Systems benötigt.
Es kann sich dabei um **konkrete technische Objekte** (z.B. Pkw) handeln, es können aber auch **Personen** und deren **Rollen** (z.B. Student, Kunde) **Orte**, (z.B.Hörsaal), **Organisationseinheiten** (z.B.Fachbereich), **Ereignisse** (z.B. Unfall), **Informationen über (Inter-)Aktionen** (z.B. Kaufvertrag), **Konzepte** (z.B. Entwicklungsplan), sonstige **allgemeine** oder **problembereichsbezogene Begriffe** (z.B.Lehrveranstaltung) **usw**. sein
Objekte von Entity-Klassen sind typischerweise **unabhängig von der Systemumgebung** und von der Art und Weise, wie das System mit der Umgebung kommuniziert.
Häufig sind sie auch **applikationsunabhängig**, d.h. sie lassen sich in mehreren Applikationen einsetzen.

▻ **Interface-Klassen** (*Boundary Classes*)
Objekte dieser Klassen sind für die **Kommunikation** zwischen der Systemumgebung und dem Inneren des Systems zuständig.
Sie realisieren die **Schnittstelle** des Systems **zum Sytembenutzer** (Benutzeroberfläche) bzw **zu anderen Systemen**.
→ Sie bilden den **umgebungsabhängigen** Teil eines Systems.

▻ **Controller-Klassen** (*Control Classes*)
Objekte von Controller-Klassen **steuern** den **Ablauf**, d.h. die Zusammenarbeit der übrigen Objekte zur Realisierung eines oder mehrerer Use Cases.
Es handelt sich um **aktive** Objekte.
Typischerweise sind sie **applikationsspezifisch**.

▻ **Service-Klassen**
Objekte von Service-Klassen stellen anderen Objekten (insbesondere den Controller-Objekten) **Dienste** zur Verfügung.
Häufig handelt es um Klassen aus einer **Bibliothek**.
Sie können aber auch durch **Auslagerung von Funktionalitäten** anderer Klassen gebildet werden.

▻ **Utility-Klassen**
Sie **kapseln** Daten und Operationen (Funktionen), die **global** verfügbar sein sollen.
Im allgemeinen werden sie **nicht instanziiert**.
Beispiel : mathematische Konstanten u. mathematische Funktionen.

Die konsequente Unterscheidung und Trennung von Entity-, Interface- und Controller-Klassen führt zur "***Model-View-Controller***"-Architektur (MVC-Architektur), ein allgemein eingeführtes OOP-Paradigma.
Das User-Interface (View) ist von der eigentlichen Problembearbeitung (Model) getrennt und weitgehend entkoppelt.
Die Verbindung zwischen beiden wird über einen Controller hergestellt.

Beziehungen zwischen Klassen

- **Vererbungsbeziehung**

  ◇ Beziehung zwischen Klassen, deren Komponenten sich teilweise überdecken

  ◇ Eine abgeleitete Klasse erbt die Eigenschaften und Fähigkeiten (Komponenten) der Basisklasse(n).
  "ist"-Beziehung → ein Objekt der abgeleiteten Klasse ist auch ein Objekt der Basisklasse(n)

  ◇ Ordnungsprinzip bei der Spezifikation von Klassen.
  → **Generalisierung / Spezialisierung**

- **Nutzungsbeziehungen**

  ◇ Unter einer (statischen) Nutzungsbeziehung versteht man eine in einem konkreten Anwendungsbereich geltende Beziehung zwischen Klassen, deren Instanzen voneinander Kenntnis haben und die dadurch miteinander **kommunizieren** können
  → Nutzungsbeziehungen sind notwendig für die **Interaktion von Objekten**

  ◇ **Assoziation**
  - Spezielle Beziehung zwischen Klassen bzw Objekten, bei der die Objekte **unabhängig** voneinander existieren und **lose** miteinander **gekoppelt** sind
    Beispiel : einem Objekt wird ein anderes Objekt als Parameter übergeben.
  - **Name** : Kennzeichnung der Semantik der Beziehung zwischen den Klasseninstanzen
  - **Navigationsrichtung** : legt die Kommunikationsrichtung und die Richtung, in der ein Objekt der einen Klasse ein Objekt der anderen Klasse referieren kann, fest.
    bidirektional (Kommunikation in beiden Richtungen möglich) oder unidirektional
  - **Rolle** : Ein Name für die Aufgabe, die ein Objekt der assoziierten Klasse aus der Sicht eines Objekts der assoziierenden Klasse wahrnimmt.
  - **Kardinalität** (*multiplicity*) : bezeichnet die mögliche Anzahl der an der Assoziation beteiligten Instanzen einer Klasse.

  ◇ **Aggregation**
  - Spezielle Beziehung zwischen Klassen bzw Objekten, bei der die Objekte der einen Klasse **Bestandteile** (Komponenten) eines oder mehrerer Objekte der anderen Klasse sind. → zwischen den Objekten besteht eine **feste Kopplung**
  - **"hat"**-Beziehung bzw **"ist Teil von"**-Beziehung
  - Das "umschließende" Objekt bildet einen Container für das bzw die enthaltene(n) Objekt(e)
  - Aggregation kann als **Spezialfall der Assoziation** aufgefaßt werden.
    → die für Assoziationen möglichen Kennungen (Name usw.) lassen sich auch für Aggregationen verwenden.
  - eine Aggregation ist i.a. aber eine **unidirektionale** Beziehung (Navigation vom umschließenden Objekt zu den Komponenten-Objekten)

  Je nach dem **Grad der Kopplung** unterscheidet man :

  ▻ **einfache Aggregation**
  Das umschließende Objekt (Aggregat) und die Komponenten sind **nicht existenzabhängig**
  Eine Komponente kann zusätzlich noch **weiteren** Aggregaten der gleichen oder einer anderen Klasse zugeordnet sein.
  Bei Löschung des Aggregats bleiben die Komponenten unabhängig vom Aggregat erhalten.
  Beispiel : Klasse mit dynamisch erzeugten Komponenten

  ▻ **echte Aggregation** (Komposition, *composite aggregation)*
  Die Komponenten können **nur einem** Aggregat zugeordnet sein und nur **innerhalb** des Aggregats **existieren**.
  Bei Löschung des Aggregats werden auch die Komponenten gelöscht.
  Beispiel : Klasse mit statisch allozierten Komponenten

  ◇ **Anmerkung** :
  In der Praxis kann es im Einzelfall sehr schwierig sein, zwischen Assoziation und Aggregation und den verschiedenen Formen der Aggregation zu unterscheiden.

Identifikation von Klassen und ihren Beziehungen

• Allgemeines

◇ Das **Identifizieren** von **Klassen** und ihren **Beziehungen** ist eine **zentrale Aufgabe** der **Analyse-**Tätigkeit innerhalb des OO-Entwicklungsprozesses.
Es handelt sich um eine sehr schwierige, äußerst kreative Tätigkeit, die methodisch kaum unterstützt wird und für die kein allgemeingültiges "Kochrezept" angegeben werden kann.
Zum Erwerb der notwendigen eigenen Erfahrungen kann man sich lediglich von Empfehlungen, Tips und Tricks (**Heuristiken**) , die einschlägige Experten mittlerweile in der Literatur veröffentlich haben, leiten lassen.

◇ Da der Entwicklungsprozeß iterativ verläuft, wird sich die **Liste der gefundenen Klassen** im Laufe dieses Prozesses **ändern** → die im ersten Schritt gefundenen Klassen, werden i.a. nicht der endgültig realisierten Liste entsprechen, neue Klassen werden hinzukommen, bereits identifizierte Klassen werden modifiziert (z.B. geteilt oder zusammengelegt) bzw wieder verworfen.
Analoges gilt für die **Attribute** (Datenkomponenten) und **Operationen** (Funktionskomponenten) der Klassen, sowie für die **Beziehungen** zwischen den Klassen

• Identifikation von Klassen

▻ Ermittlung der **konkreten technischen Objekte** des Problembereichs (**Fachklassen**).

▻ Ermittlung von Objekten/Klassen aus im Problembereich eingesetzten **Formularen** (Formularanalyse)

▻ Untersuchung der Beschreibung der Szenarien der einzelnen *Use Cases* (oder sonstiger verbal formulierter Anforderungen an das System, wie z.B. das Pflichtenheft) nach **Hauptworten**.Diese sind i.a. Kandidaten für Klassen.

▻ Einsatz von **CRC-Karten** (CRC – *Class*, *Responsibilities*, *Collaborations*)
Anlegen kleiner Karten, auf denen der Klassenname, die Zuständigkeiten (*responsibilties*) und die mit ihnen kollaborierenden Klassen (zu denen also Assoziationen bestehen müssen) vermerkt werden.
Überprüfung der einzelnen Anwendungsfälle mittels der CRC-Karten auf Erfüllung der geforderten Funktionalität.
Übervolle Karten bedeuten entweder schlecht ausgearbeitete Zuständigkeiten oder nichtkohäsive Klassen.

▻ Untersuchung jedes **Aktor-/Scenario-Paares** zur Ermittlung der **Interface-Klassen**.

▻ Hinzufügen je einer **Controller-Klasse** für jedes **Aktor/Scenario-Paar**.

▻ Zuordnung der gefundenen Klassen zu den verschiedenen **Klassenkategorien**

▻ Wählen **aussagekräftiger Namen** für die gefundenen Klassen

▻ Zuordnen von **Operationen** und **Attributen**, im ersten Schritt allerdings nur soweit es zur **Unterscheidung** der Klassen und zur Abdeckung der Zuständigkeiten notwendig ist.

▻ **Filterung** der gefundenen Klassenmenge hinsichtlich **tatsächlicher Problembereichszugehörigkeit**, echter **Substanz**, eindeutiger **Abgrenzung**, **Überschneidungen**, **Redundanz** usw.

▻ Gegebenenfalls sind Klassen wieder zu verwerfen, zusammenzufassen, zu teilen oder Komponenten in gemeinsame Basisklassen auszulagern (**Refaktorierung**).

• Identifikation der Klassenbeziehungen

▻ Untersuchung der gefundenen Klassen auf Gemeinsamkeiten (→ Vererbungsbeziehungen)

▻ Ermittlung der Nutzungsbeziehungen kann parallel zur Identifikation der Klassen beginnen (CRC-Karten !)

▻ Untersuchung der einzelnen Scenarios auf **Verben**. Diese können auf zwischen Objekten auszutauschende Botschaften hinweisen. Der Austausch von Botschaften erfordert Nutzungsbeziehungen.

▻ Rangordnung der beteiligten Klassen überprüfen
Über-/Unterordung kann auf Aggregation hinweisen.

▻ Überprüfung, ob "hat"- bzw "ist Teil von"-Beziehung vorliegt

▻ Im Zweifelsfall Assoziation wählen

▻ Kennungen (Name, Navigationsrichtung, Rollenname, Kardinalität usw) der gefundenen Beziehungen festlegen.

Modellierung

• Allgemeines

◇ Das zu entwickelnde **Software-System** sowie der von ihm abzudeckende **Problembereich** sind häufig so **komplex**, dass sie sich nur mit Hilfe geeigneter **Hilfsmittel** ausreichend erfassen lassen.
Ein derartiges Hilfsmittel stellt die **Modellierung** dar.

◇ I.a. ist es wenig sinnvoll das Gesamtsystem in einem einzigen – alle Einzelheiten erfassenden – Modell darzustellen. Vielmehr setzt man **unterschiedliche Modelle** ein, die jeweils verschiedene Teilaspekte des Systems repräsentieren. Diese Modelle erleichtern nicht nur das Problembereichs- und Systemverständnis, sondern bilden auch ein wesentliches Kommunikationsmittel aller an der Systementwicklung beteiligten Personen und stellen damit auch einen wichtigen Bestandteil der Systemdokumentation dar.

◇ Voraussetzung für die Bildung adäquater, ausagekräftiger, eindeutiger und leicht verständlicher Modelle ist eine **geeignete Notation** zur Modelbeschreibung.
Für den Bereich der objektorientierten Systementwicklung ist dies die **Unified Modeling Language** (**UML**), mit der Modelle in überwiegend **graphischer** Notation (→ **Diagramme**) dargestellt werden können.

◇ Die **UML** stellt Sprachmittel zur Formulierung zahlreicher **unterschiedlicher Diagramme** zur Verfügung, die zur Beschreibung der verschiedenen Modelle geeignet sind.

• Modelle und zugeordnete UML-Diagramme

◇ Modell der Systemnutzung

▻ **Nutzungsfallmodell** (Anwendungsfallmodell) → **Use-Case-Diagramm**

◇ Logisches Modell

▻ **Statisches Modell**
grobe Systemarchitektur (Aufbaustruktur) → **Paketdiagramm**
detaillierte Systemarchitektur → **Klassendiagramm**

▻ **Dynamisches Modell**
Zusammenarbeit mehrerer Objekte (Objekt-Interaktion) → Interaktionsdiagramme :
Sequenzdiagramm
Kollaborationsdiagramm
Objektverhalten (in mehreren Use Cases) → **Zustandsdiagramm**
Systemverhalten → **Aktivitätsdiagramm**

◇ Physikalisches Modell

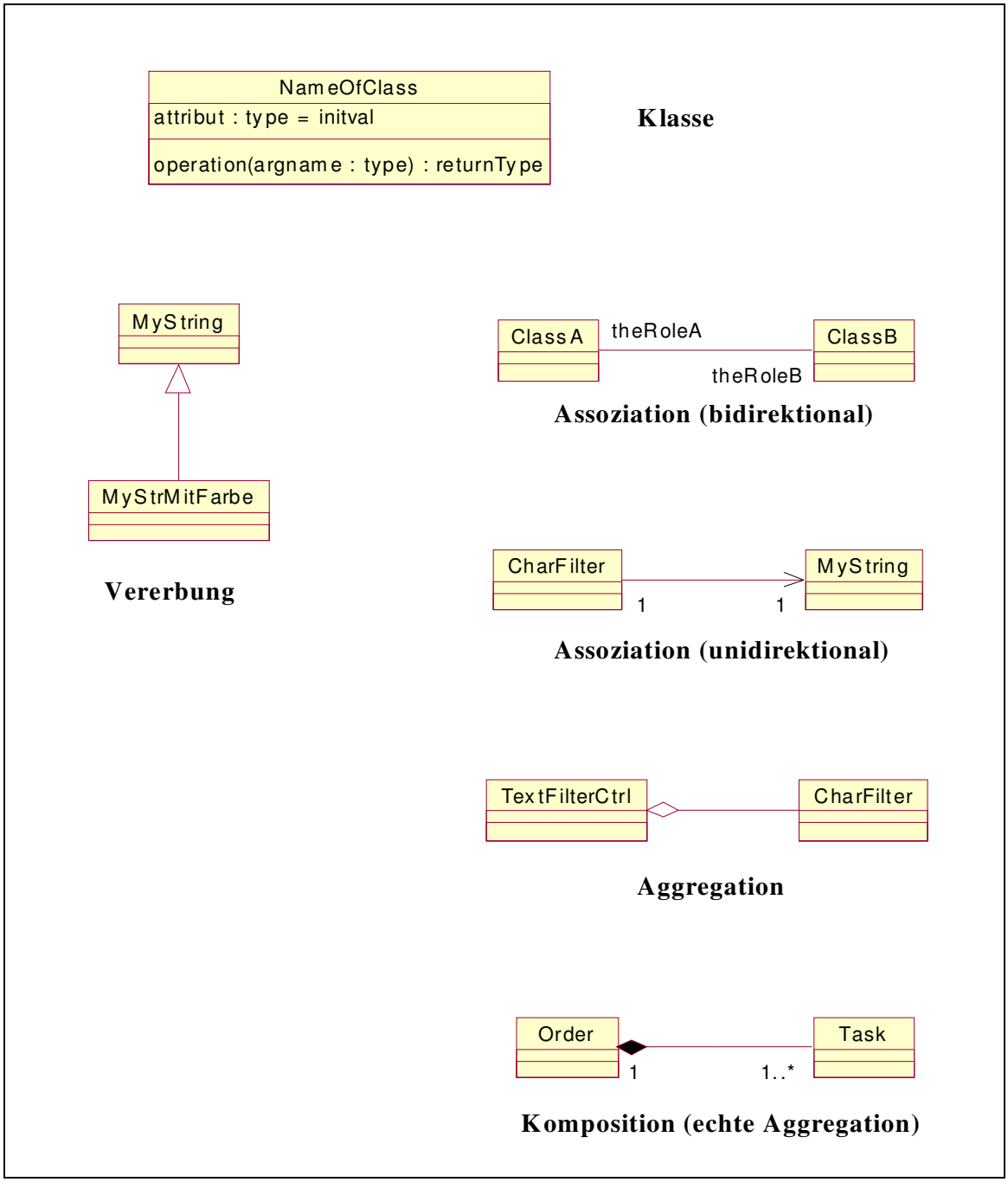
▻ **Implementierungsmodell** → **Komponentendiagramm** (Moduldiagramm)

▻ **Konfigurierungsmodell** → **Verteilungsdiagramm** (Deployment Diagram)
(topologische Strukturierung und Verteilung/Zuordnung der Soft- und Hardwarekomponenten des Gesamtsystems)

Unified Modelling Language (UML) (1)

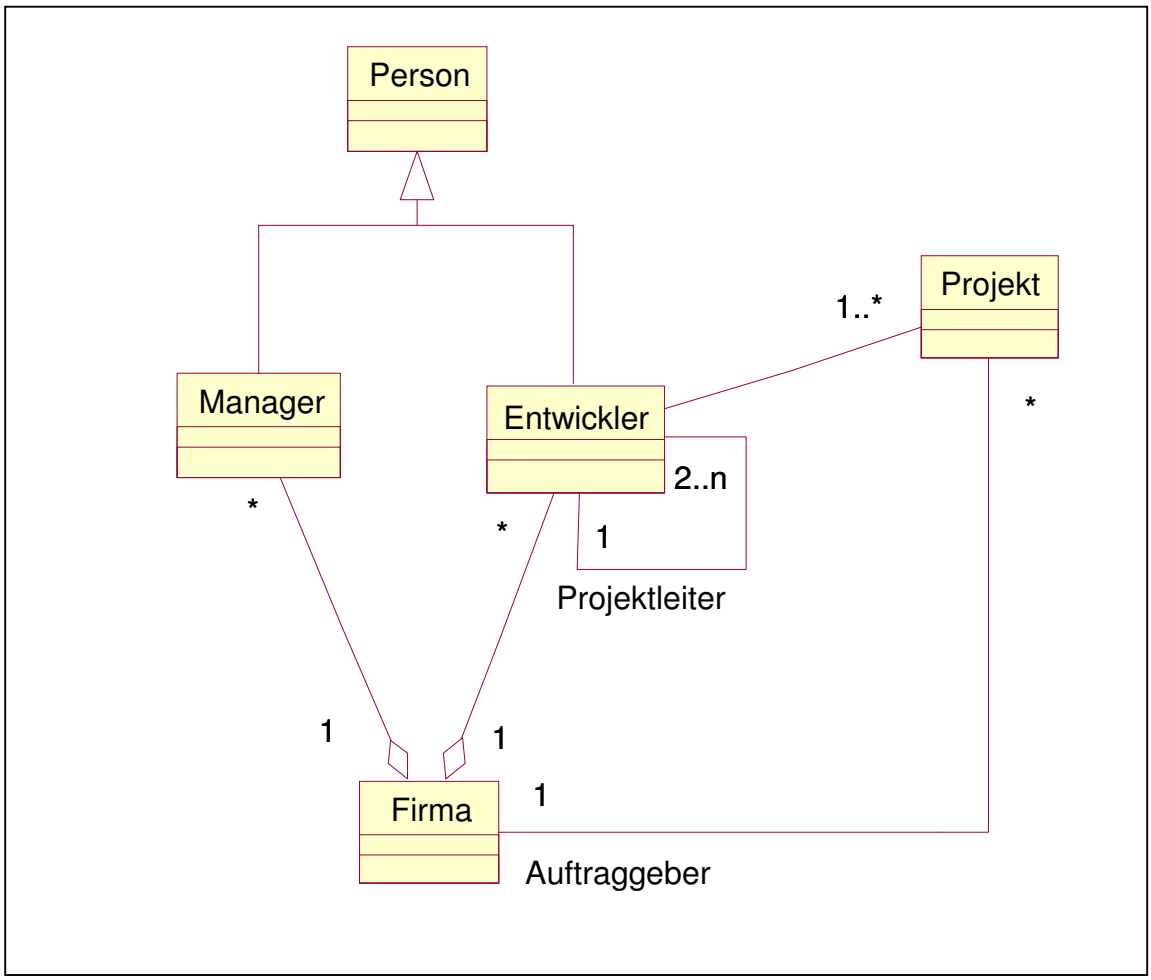
- **Klassendiagramm**
  - ◇ Ein Klassendiagramm beschreibt die im System eingesetzten **Klassen** und ihre statischen **Beziehungen** (Vererbungsbeziehungen und Nutzungsbeziehungen).→**detailliertes statisches Systemmodell**
  - ◇ Zur Erhöhung der Übersichtlichkeit kann die Gesamtheit der Klassen eines Systems auf mehrere **Teildiagramme** aufgeteilt sein

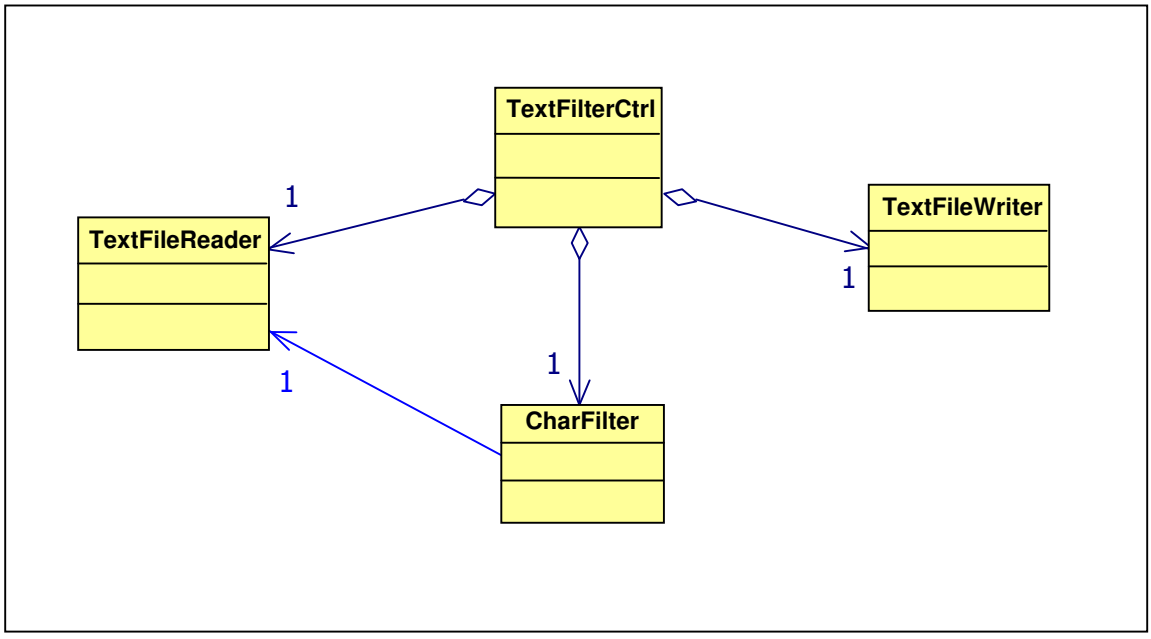
- **Elemente des Klassendiagramms**



Unified Modelling Language (UML) (2)

- **Beispiel für Klassendiagramme**

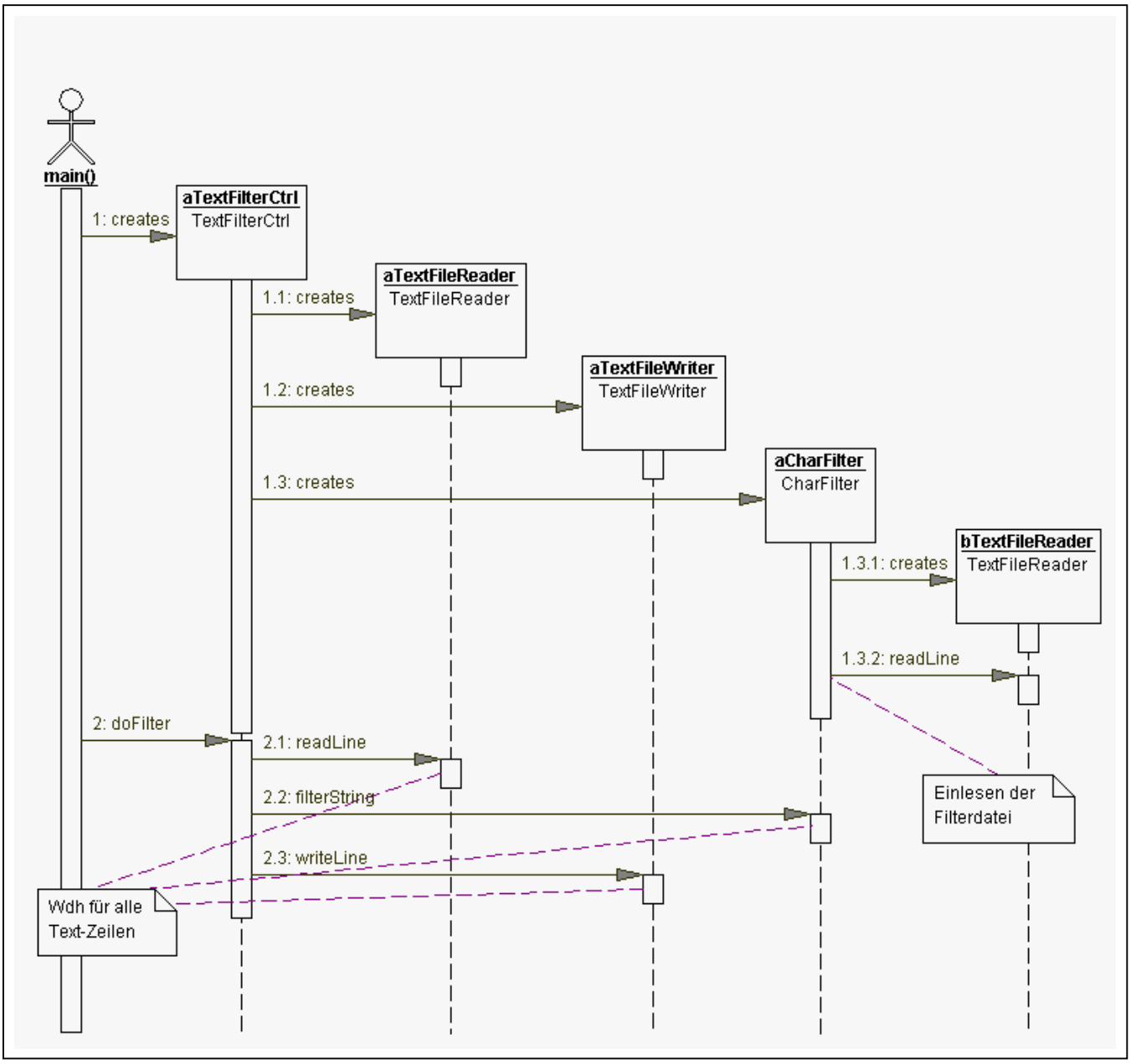


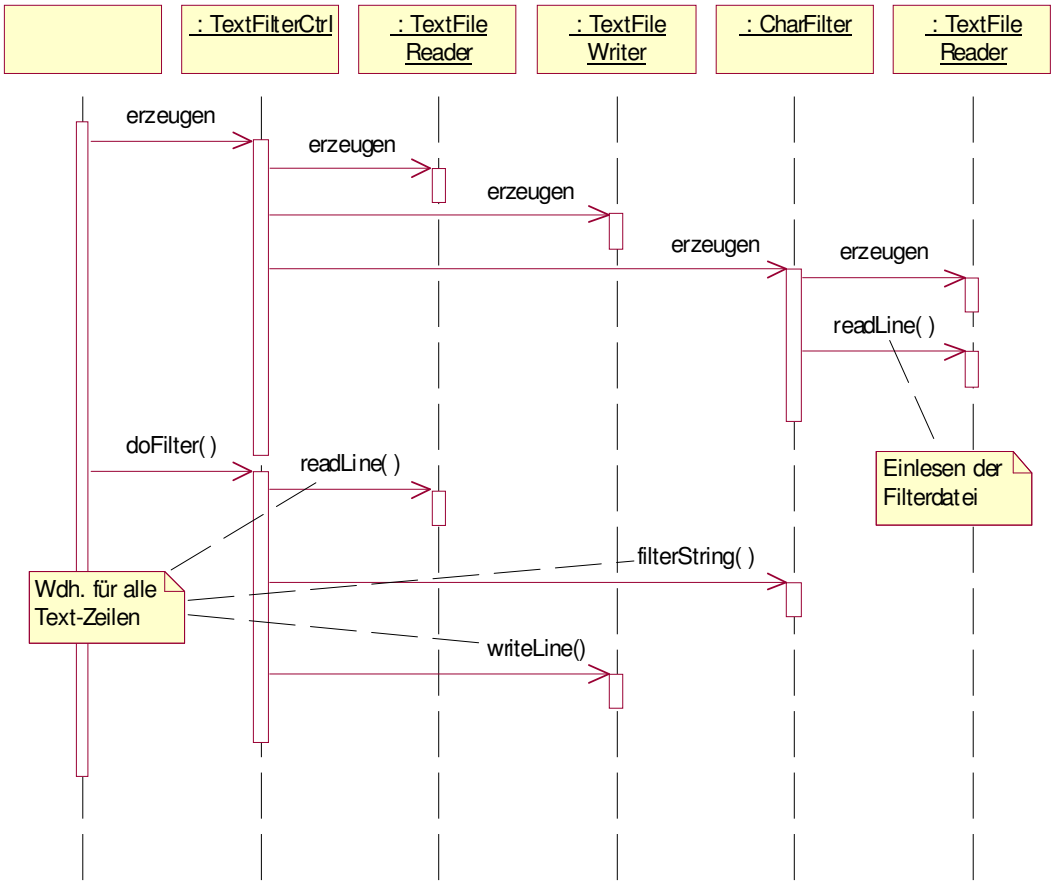


Unified Modelling Language (UML) (3)

- **Sequenzdiagramm**
  - ◇ Ein Sequenzdiagramm beschreibt die **Interaktion mehrerer Objekte**.
  - ◇ Für jedes Scenario jedes Use Cases sollte ein eigenes Diagramm erstellt werden.

- **Beispiel für ein Sequenzdiagramm**




: TextFilterCtrl
: TextFile Reader
: TextFile Writer
: CharFilter
: TextFile Reader
erzeugen
erzeugen
erzeugen
erzeugen
erzeugen
readLine( )
Einlesen der Filterdatei
doFilter( )
readLine( )
filterString( )
Wdh. für alle Text-Zeilen
writeLine()

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.

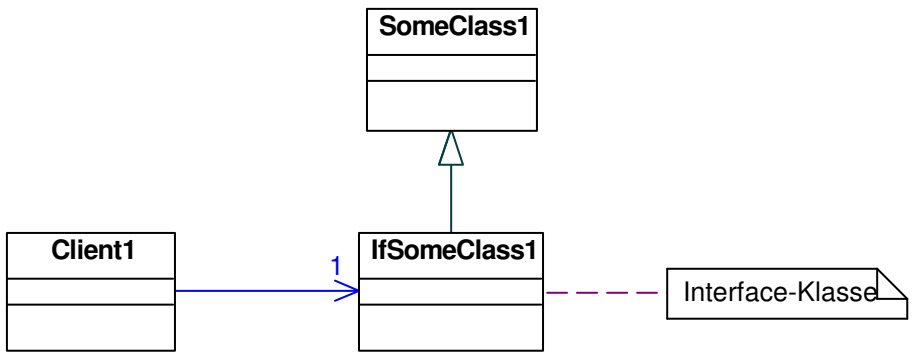
  ◇ Sehr häufig ist eine Interface-Klasse von der Klasse, deren Schnittstelle sie anpassen soll, **abgeleitet**.

  ◇ 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**).

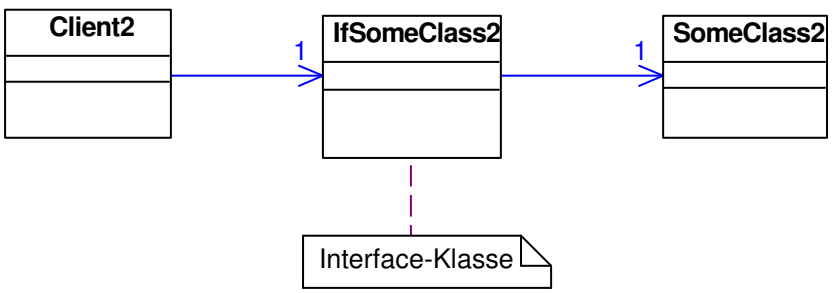
  ◇ In **allgemeinerer Form** wird die Anpassung eines Klassen-Interfaces durch das **Design-Pattern Adapter** realisiert.

- **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.

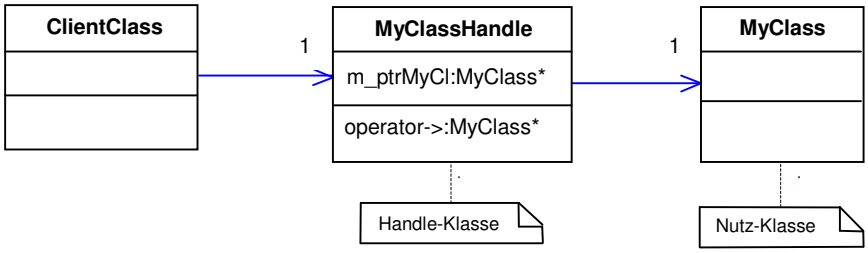
  ◇ Die **Verbindung** zwischen einem Handle-Objekt und seinem Nutz-Objekt wird typischerweise durch einen **Pointer** im Handle-Objekt realisiert.
  Häufig – aber nicht immer – ist dieser Pointer die einzige Datenkomponente eines Handle-Objekts.
  Er wird i.a. dem Konstruktor der Handle-Klasse als Parameter 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 incrementiert, bei jeder Freigabe des Objekts wird der Referenzzähler decrementiert.
  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 **Decrementierung** 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 Decrementieren von dessen Referenzzähler), setzt dann seinen Verbindungspointer auf das Nutz-Objekt des Quell-Handle-Objekts und incrementiert 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 allokierte 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 Incrementieren und Decrementieren desselben zur Verfügung.
  Wird beim Decrementieren 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 `simplestring.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;
}
```

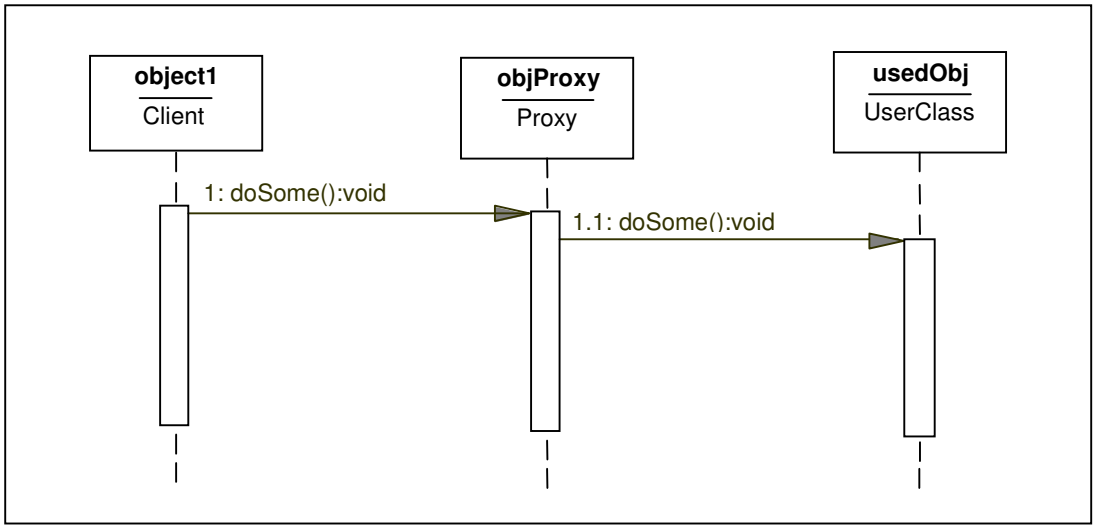
Beispiel 2 zur Referenzzählung in C++ (3)

- **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 Proxyklasse** 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 Abstammungliste 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&)`**
    zentraleVerwaltungs- 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 :

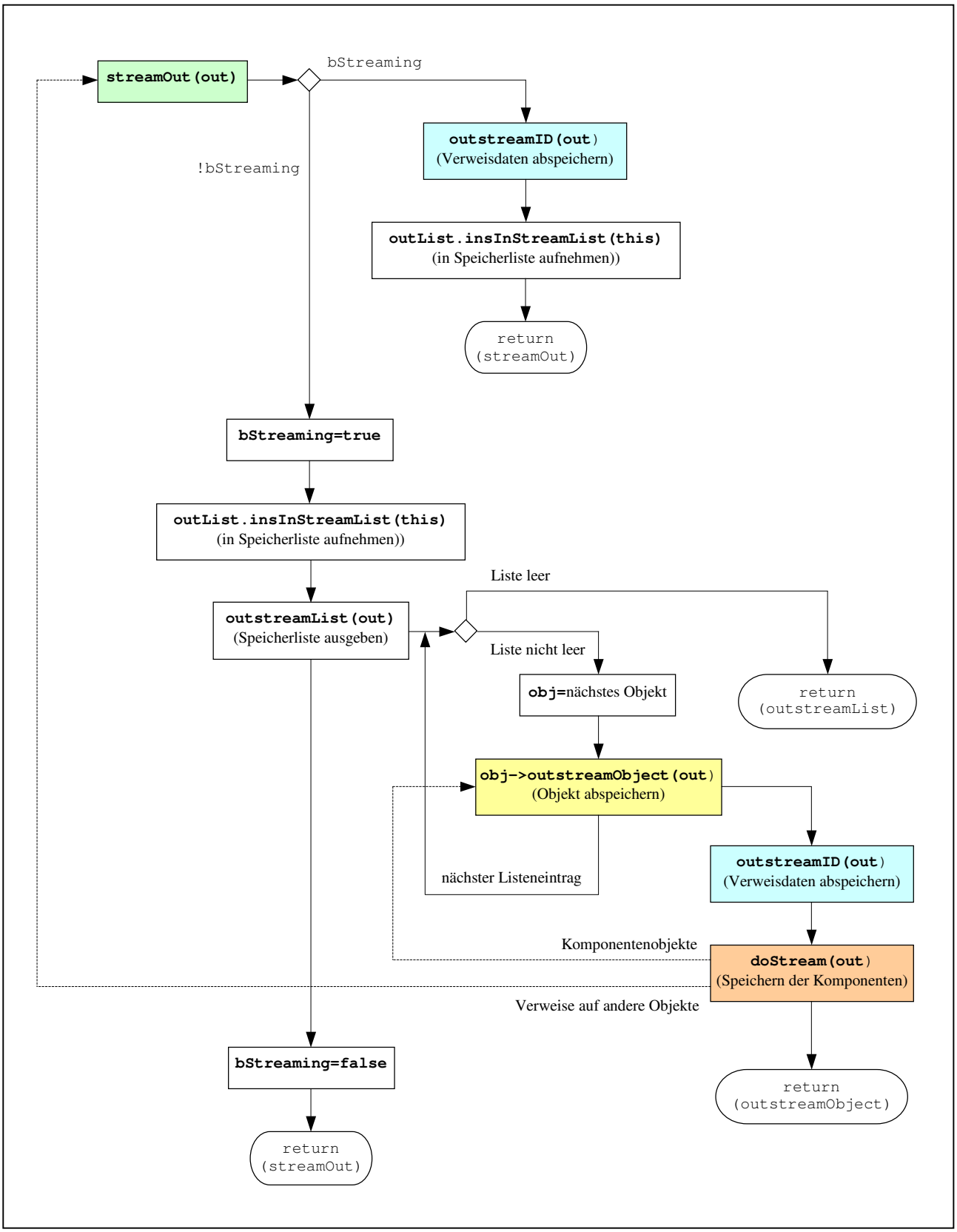
- ▻ geeigneter **Konstruktor** mit einem Parameter vom Typ `istream&` ("Lade"-Konstruktor).
  Dieser muß in einer Initialisierungsliste die "Lade"-Konstruktoren seiner persistenten Basisklassen und Komponentenklassen aufrufen, bei direkter Ableitung von `Streamable` also auch den "Lade"-Konstruktor dieser 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.

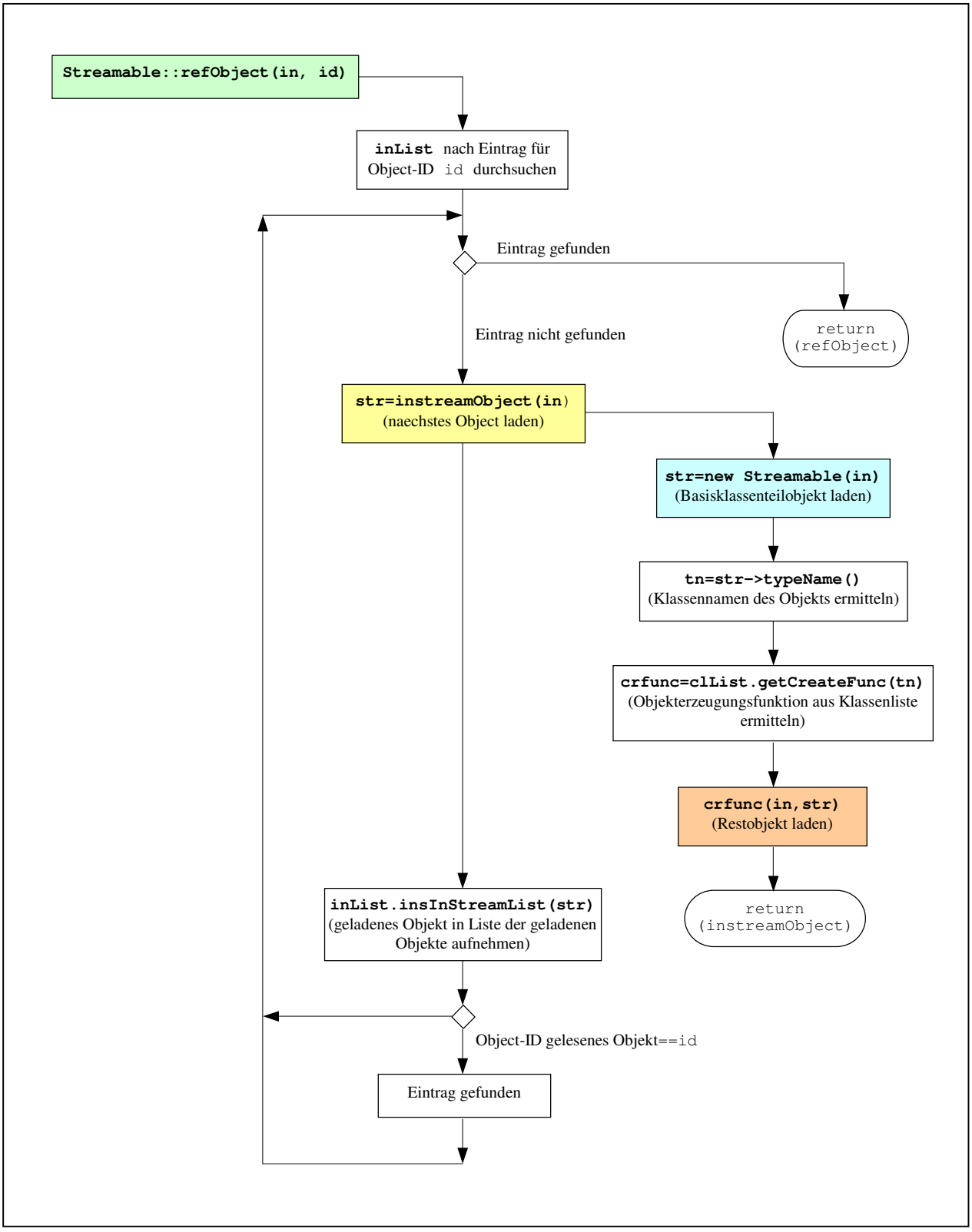
Realisierungsbeispiel zur Persistenz in C++ (2)

- **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;
}
```

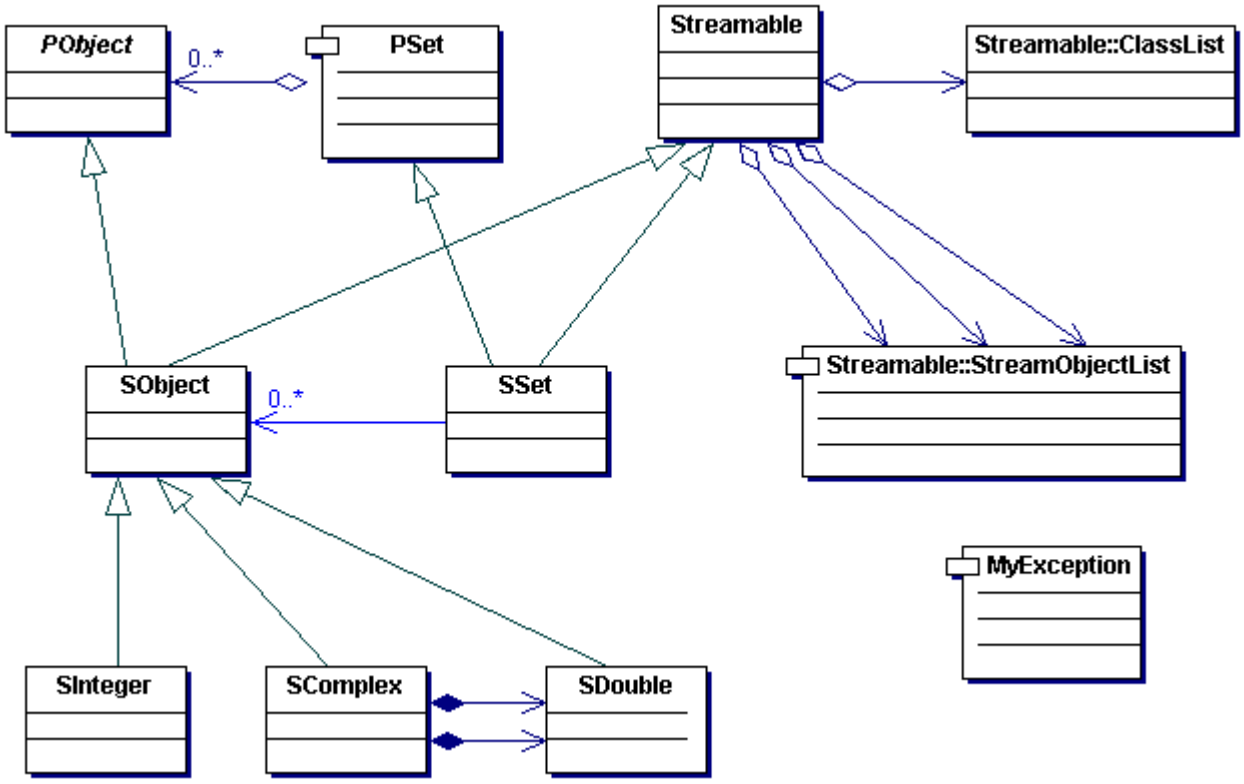
Realisierungsbeispiel zur Persistenz in C++ (6)

- **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;
}
```

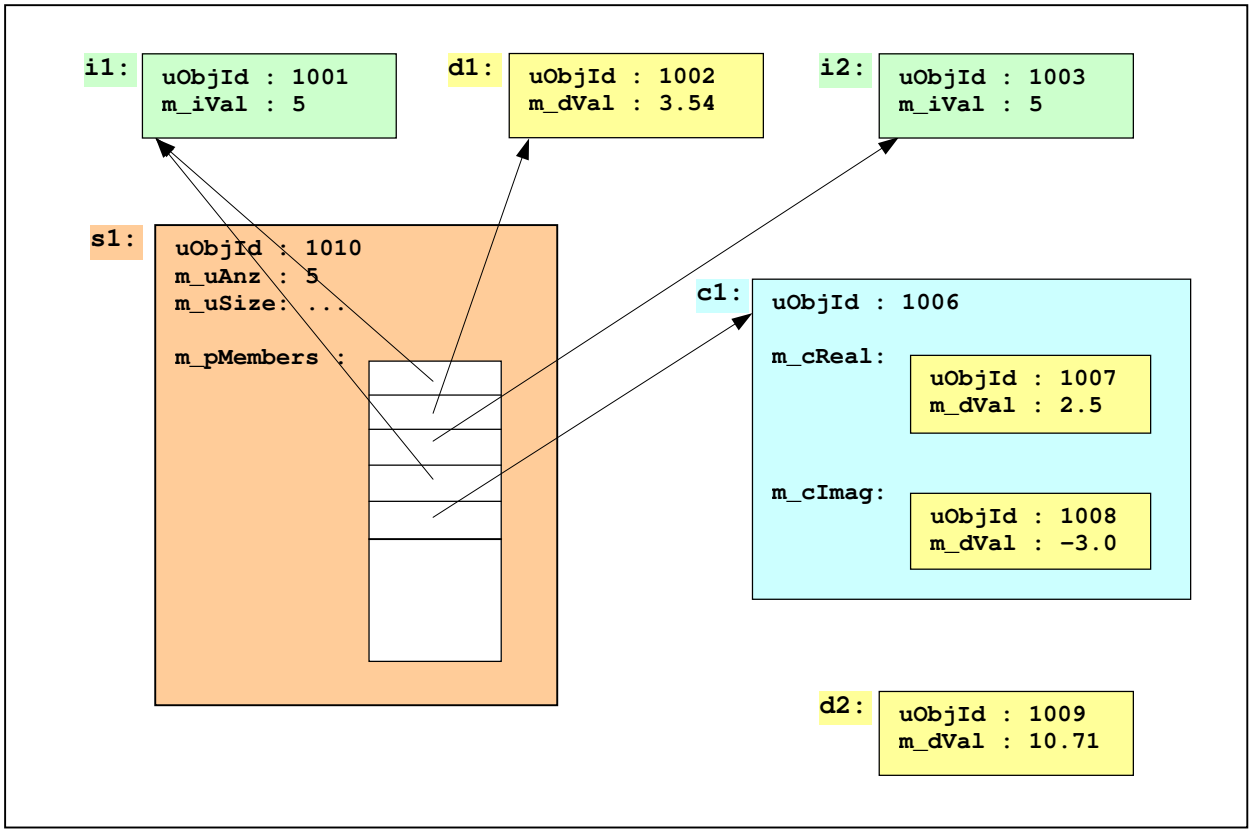
Anwendung des Realisierungsbeispiels zur Persistenz in C++ (5)

- **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 Sicherunsgdatei `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
```

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 9

9. Entwurfsmuster (*Design Pattern*)

9.1. Allgemeines und Überblick

9.2. Beispiele

Entwurfsmuster – Allgemeines

• Allgemeines

◇ **Entwurfsmuster** (**Design Pattern**) sind wiederverwendbare **bewährte** generische **Löungen** für bestimmte immer wiederkehrende Entwurfsprobleme. Dabei ist charakteristisch, daß in **unterschiedlichen Anwendungsbereichen** auftretende **gleichartige Probleme** durch Anwendung des gleichen Entwurfsmusters **gleichartig gelöst** werden können.

◇ Entwurfsmuster können auf unterschiedlichen Abstraktionsebenen gebildet werden.
Im engeren Sinn versteht man darunter *Beschreibungen von interagierenden **Objekten und Klassen, die so aufeinander abgestimmt sind, daß sie ein allgemeines Entwurfsproblem in einem bestimmten Kontext lösen*** können

◇ Ein Entwurfsmuster **identifizier**t und **abstrahiert** die **Kern-Aspekte** einer allgemeinen Entwurfsstruktur.
Es identifiziert und benennt die beteiligten Klassen und Objekte, ihre Rollen, Verantwortlichkeiten und ihre Beziehungen.

> Entwurfsmuster **dokumentieren** erprobte **Entwurfserfahrungen**.
> Sie fördern dadurch in besonderen Maße die **Wiederverwendbarkeit** bewährter Lösungsstrukturen und ermöglichen einen Entwurf auf **höherem Abstraktionsniveau**.
> Ihre Kenntnis und sinnvolle Anwendung **erleichtert** und **beschleunigt** den **Entwicklungsprozeß** und **verhindert** gleichzeitig den Einsatz **"schlechterer" Lösungsalternativen**.

• Beschreibungselemente

◇ Im Laufe der Zeit sind zahlreiche Entwurfsmuster "entdeckt" und in Büchern und Fachzeitschriften veröffentlicht worden.
Neue Muster kommen laufend hinzu.

◇ Die Beschreibung eines Entwurfsmusters sollte wenigstens die folgenden vier Elemente umfassen :

▻ **Name des Musters** (*pattern name*)
Prägnante Zusammenfassung des Entwurfsproblems, seiner Lösung und deren Anwendungskonsequenzen in ein oder zwei Worten.

▻ **Problembeschreibung** (*problem*)
Darstellung des Anwendungsbereichs des Entwurfsmusters durch Erläuterung des Problems, seines Kontexts und gegebenenfalls spezifischer Entwurfsprobleme

▻ **Problemlösung** (*solution*)
Beschreibung der Komponenten (Objekte und Klassen) der Entwurfsstruktur, ihrer Verantwortlichkeiten, ihrer Beziehungen und ihrer Zusammenarbeit.
Dabei bezieht sich die Beschreibung weder auf einen konkreten Entwurf noch auf eine konkrete Implementierung, da ein Entwurfsmuster wie eine Schablone in verschiedenen – aber strukturell ähnlichen - Situationen anwendbar sein soll.
Entwurfsmuster beruhen auf einer abstrakten Darstellung eines Entwurfsproblems und stellen eine allgemeine Anordnung von Elementen (Klassen und Objekten), die das Problem lösen kann, zurVerfügung.
Prinzipielle Realisierungsbeispiele – in einer konkreten Implementierungssprache – können gegebenenfalls angegeben werden.

▻ **Anwendungskonsequenzen** (*consequences*)
Auflistung der Ergebnisse und Folgen ("*trade-offs*"), die sich aus der Anwendung des Entwurfsmusters ergeben.
Die Kenntnis der Konsequenzen sind für eine kritische Untersuchung von Entwurfsalternativen und die Abwägung von Kosten und Nutzen einer Lösung wichtig.
Häufig betreffen die Konsequenzen Speicher- und/oder Zeiteffizienz.
Sie können sich aber auch auf Sprach- und Implementierungsgesichtspunkte beziehen.
Darüberhinaus ist es grundsätzlich wichtig, den Einfluß eines Entwurfsmusters auf die Flexibilität, die Erweiterbarkeit und die Portabilität eines Systems zu kennen.

Entwurfsmuster – Klassifizierung

- **Klassifikation**

  ◇ Die verschiedenen bisher veröffentlichten Entwurfsmuster **differieren** bezüglich ihres **Anwendungsbereiches** und ihrer **strukturellen Auflösung** (*granularity*).
  Sie lassen sich nach **unterschiedlichen Gesichtspunkten klassifizieren.**

  ◇ In Anlehnung an das Standardwerk von Gamma/Helm/Johnson/Vlissides ("Design Patterns") ist vor allem eine Klassifizierung nach zwei Gesichtspunkten üblich :

  - ▻ **Einsatzzweck** (*purpose*)
  - ▻ **Geltungsbereich** (*scope*)

  ◇ Der **Einsatzzweck** spiegelt wieder, ***was*** ein Entwurfsmuster ***bewirkt***. Man unterscheidet :

  - ▻ **Erzeugende Muster** (*creational patterns*)
    Sie beziehen sich auf die Erzeugung von Objekten
  - ▻ **Strukturelle Muster** (*structural patterns*)
    Sie befassen sich mit der Zusammensetzung (*composition*) von Klassen und Objekten
  - ▻ **Verhaltensmuster** (*behavioral patterns*)
    Sie bestimmen die Zusammenarbeit zwischen Klassen bzw. Objekten und die Verteilung ihrer Verantwortlichkeiten

  ◇ Der **Geltungsbereich** legt fest, ***worauf*** sich ein Entwurfsmuster primär ***bezieht*** (auf Klassen oder auf Objekte):

  - ▻ **Klassenmuster** (*class patterns*)
    Sie befassen sich primär mit Vererbungsbeziehungen zwischen Klassen.
    Diese Beziehungen sind statisch und liegen zur Compile-Zeit fest.
  - ▻ **Objektmuster** (*object patterns*)
    Sie befassen sich primär mit den Nutzungsbeziehungen zwischen Objekten.
    Diese Beziehungen sind i.a. zur Laufzeit änderbar und daher dynamisch.
    Die meisten Entwurfsmuster gehören dieser Kategorie an.

- **Klassifizierung der Standard-Entwurfsmuster nach Gamma (u.a.)**

		Geltungsbereich	
		Klassenmuster	**Objektmuster**
Einsatzzweck	**Erzeugende Muster**	Factory Method	Abstract Factory Builder Prototype Singleton
	Strukturelle Muster	Adapter (class)	Adapter (Object) Bridge Composite Decorator Facade Flyweigth Proxy
	Verhaltensmuster	Interpreter Template Method	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategie Visitor

Entwurfsmuster – Überblick (1)

- **Überblick über die Standard-Entwurfsmuster nach Gamma (u.a.)**

Erzeugende Muster (*Creational Patterns*)

◇ **Abstract Factory** (*Kit*) : Stellt ein Interface zur Erzeugung ganzer Familien verwandter oder voneinander abhängiger Objekte zur Verfügung, ohne daß deren konkrete Klassen spezifiziert werden müssen

◇ **Builder** : Trennt die Konstruktion eines komplexen Objekts von seiner Repräsentation, so daß derselbe Konstruktionsprozeß zur Erzeugung unterschiedlicher Repräsentationen eingesetzt werden kann.

◇ **Factory Method** (*Virtual Constructor*) : Definiert ein Interface zur Erzeugung eines Objekts, überträgt aber Subklassen die Entscheidung welches konkrete Objekt tatsächlich erzeugt wird.

◇ **Prototype** : Instantiiert Objekt-Prototypen und erzeugt neue Objekte durch Kopieren dieser Prototypen.
Dadurch wird die Erzeugung von erst zur Laufzeit spezifizierter Objekte ermöglicht.

◇ **Singleton** :Stellt sicher, daß nur eine Instanz einer Klasse erzeugt wird und stellt einen globale Zugriffsmöglichkeit zu dieser Instanz zur Verfügung.

Strukturelle Muster (*Structural Patterns*)

◇ **Adapter** (*Wrapper*) : Wandelt das Interface einer Klasse in ein anderes von einem Client erwartetes Interface um. Adapter ermöglichen die Zusammenarbeit von Klassen, die sonst wegen inkompatibler Interfaces nicht zusammenarbeiten könnten.

◇ **Bridge** (*Handle, Body*) : Entkoppelt eine Abstraktion (Klassenhierarchie) von ihrer Implementierung (Klassenhierarchie), so daß beide unabhängig voneinander variiert werden können.

◇ **Composite** (*Recursive Composition*) : Ermöglicht die Repräsentation von Teil-Ganzheits-Beziehungen durch den Aufbau baumartiger Objektstrukturen. Dadurch können Clients individuelle Objekte und Zusammensetzungen von Objekten gleichartig behandeln.

◇ **Decorator** : Ergänzt ein Objekt dynamisch um zusätzliche Fähigkeiten.
Das Entwurfsmuster stellt bezüglich dieser Erweiterung eine flexible Alternative zur Vererbung dar.

◇ **Facade** : Kapselt einen Satz von Interfaces innerhalb eines Subsystems nach außen durch ein vereinfachtes einheitliches Interface. Dadurch wird ein Interface höherer Ebene definiert, das die Benutzung des Subsystems erleichtert.

◇ **Flyweight** : Ermöglicht die Mehrfachnutzung von Objekten mit gleichem inneren Zustand aber unterschiedlichem äußeren Zustand (Kontext). Dadurch werden Strukturen mit einer großen Anzahl "einfacherer" Objekte effektiv unterstützt.

◇ **Proxy** (*Surrogate*) : Stellt einen Platzhalter (Stellvertreter) für ein Objekt zur Verfügung, um den Zugriff zu diesem Objekt zu kontrollieren.

Entwurfsmuster – Überblick (2)

- **Überblick über die Standard-Entwurfsmuster nach Gamma (u.a.)**, Forts.

Verhaltensmuster (*Behavioral Patterns*)

◇ **Chain of Responsibility** : Ermöglicht, daß mehr als ein Objekt die Gelegenheit zur Reaktion auf eine Botschaft erhält. Die möglichen Botschaftempfänger werden miteinander verkettet und die Botschaft wird die Kette entlang gereicht, bis ein Objekt diese bearbeitet.

◇ **Command** (*Action, Transaction*) : Kapselt eine Botschaft als Objekt. Dadurch werden Sender und Empfänger einer Botschaft voneinander entkoppelt.Dies ermöglicht u.a. die Parameterisierung von Client-Objekten (Anwendungen) mit unterschiedlichen Botschaften für unterschiedliche daraus folgende Operationen.

◇ **Interpreter** : Beschreibt die Definition der Grammatik einer einfachen Sprache sowie die Darstellung von Sätzen in der Sprache zusammen mit einem Interpreter zu ihrer Interpretation.

◇ **Iterator** (*Cursor*) : Ermöglicht den sequentiellen Zugriff zu den Elementen eines Aggregat-Objekts ohne dessen internen Aufbau offenzulegen.

◇ **Mediator** : Definiert ein Objekt, das die Interaktion einer Objektmenge kapselt. Das Entwurfsmuster unterstützt eine lose Objektkopplung durch Verhinderung eines expliziten Bezugs der Objekte aufeinander. Es ermöglicht eine unabhängige Änderung der Objektinteraktion.

◇ **Memento** (*Token*) : Ermöglicht das Festhalten und die äußere Darstellung des inneren Zustands eines Objekts zu seiner späteren Wiederherstellung, ohne die Objektkapselung zu verletzen.

◇ **Observer** (*Dependents, Publish-Subscribe*) : Ermöglicht die dynamische Registrierung von Objektabhängigkeiten, so daß bei einem Zustandswechsel eines Objekts alle von ihm abhängigen Objekte benachrichtigt werden.

◇ **State** (*Object for States*) : Kapselt die Zustände eines Objekts in separate Objekte, die jeweils zu einer Zustandsklasse gehören, die von einer gemeinsamen abstrakten Basis-Zustandsklasse abgeleitet sind. Dieses Entwurfsmuster ermöglicht einem Objekt, sein Verhalten bei einer Zustandsänderung zu ändern. Nach außen scheint das Objekt dadurch seine Klasse zu ändern.

◇ **Strategy** (*Policy*) : Kapselt verwandte Algorithmen in jeweils einer eigenen Klasse, die von einer gemeinsamen Basisklasse abgeleitet ist. Dies ermöglicht die Auswahl und Änderung der Algorithmen zur Laufzeit ohne daß Änderungen an den Clients, die sie benutzen, notwendig sind.

◇ **Template Method** : Definiert das Gerüst eines Algorithmus in einer abstrakten Basisklasse, wobei die Konkretisierung einiger Algorithmus-Schritte an abgeleitete Klassen übertragen wird, ohne daß dadurch die Struktur des Algorithmus geändert werden muß.

◇ **Visitor** : Implementiert eine Operation, die mehrere Objekte in einer komplexen Struktur betrifft, in einem separaten Objekt. Das Entwurfsmuster ermöglicht die Definition neuer Operationen ohne Änderung der Klassen der Objekte, auf die sich die Operation bezieht.

Entwurfsmuster : Singleton (1)

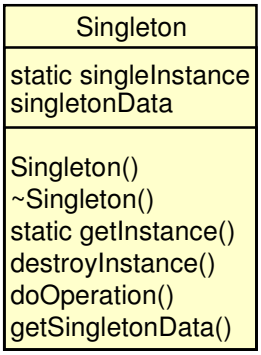
- **Name : Singleton**

- **Kurzbeschreibung**
  Stellt sicher, daß **nur eine Instanz** einer Klasse erzeugt wird und stellt einen **globale Zugriffsmöglichkeit** zu dieser Instanz zur Verfügung.

- **Problembeschreibung**
  Oft ist es wichtig, daß von einer Klasse nur eine einzige Instanz erzeugt wird.
  Zusätzlich besteht häufig die Forderung, daß diese Instanz global zugreifbar sein soll.
  Beispielsweise sollte in einem zentralisierten Workflow-Managementsystem nur ein Manager-Objekt vorhanden sein.

- **Problemlösung**
  In einer sinnvollen Lösung ist die Klasse selbst dafür verantwortlich, daß sie nur einmal instantiiert wird.
  Dies wird dadurch erreicht, daß alle Konstruktoren der Klasse privat sind
  Zur Objekterzeugung und zum Zugriff auf das Singleton-Objekt dient eine öffentliche statische Memberfunktion.
  Diese überprüft zuerst, ob bereits eine Instanz der Klasse angelegt ist.
  Falls nein wird eine Instanz – durch Aufruf eines privaten Konstruktors - angelegt und ein Pointer auf diese in einer privaten statischen Membervariablen abgelegt. Dieser Pointer wird gleichzeitig als Funktionswert zurückgegeben.
  Falls ja, gibt sie den in der statischen Membervariablen gespeicherten Pointer auf die Instanz zurück.

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Da die Singleton-Klasse ihre einzige Instanz kapselt, hat sie die strikte Kontrolle über den Zugriff zu dieser
  - ▻ Die Singleton-Klasse kann leicht zu einer Klasse erweitert werden, die eine definierte Anzahl von Instanzen größer als eins erzeugen kann.
  - ▻ Der Einsatz einer Singleton-Klasse als Basisklasse kann problematisch sein.
    In diesem Fall darf der Konstruktor nicht privat sein. Außerdem läßt sich die statische Objekterzeugungs-Funktion nicht überschreiben.

Entwurfsmuster : Singleton (2)

- **Implementierungsbeispiel**

  ◇ **Definition der Singleton-Klasse :**

```
class Singleton
{
  public :
    static Singleton* getInstance();
    void destroyInstance();
    // weitere nicht-statische Memberfunktionen
  private :
    static Singleton* singleInstance;
    // weitere nicht-statische Datenkomponenten
   Singleton();
  ~Singleton();
 };
```

  ◇ **Implementierung der Singleton-Klasse :**

```
Singleton* Singleton::singleInstance = 0;   // == NULL-Pointer

Singleton::Singleton()
{
  // privater Konstruktor
}

Singleton::~Singleton()
{
  // privater Destruktor
}

Singleton* Singleton::getInstance()
{
  if (singleInstance==0)
    singleInstance = new Singleton;
  return singleInstance;
}

void Singleton::destroyInstance()
{
  delete this;
  singleInstance=0;       // == NULL-Pointer
}
```

Entwurfsmuster : Factory Method (1)

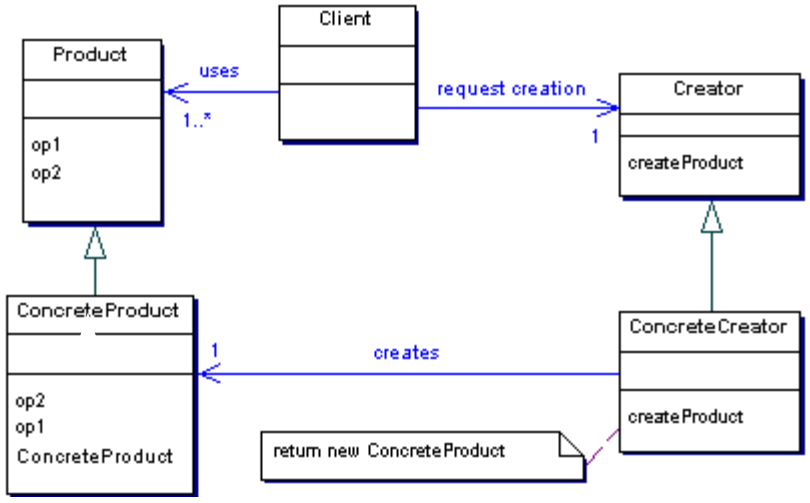
- **Name : Factory Method** (*Virtual Constructor*)

- **Kurzbeschreibung**
  Ermöglicht die Erzeugung **applikationsspezifischer Objekte** durch **applikationsunabhängigen** und damit auch von den Klassen der erzeugten Objekte unabhängigen **Code**.

- **Problembeschreibung**
  Es gibt zahlreiche Situationen, in denen ein Objekt einer Klasse (Client) – je nach Applikation in der sie eingesetzt wird – mit unterschiedlichen Objekten, die aber alle eine gemeinsame Basisklasse haben, arbeiten soll. Das jeweils zu verwendende Objekt muß von dem verwendenden Objekt (Client) erzeugt werden. Die Klasse des verwendenden Objekts (Client) kennt zwar den Zeitpunkt der Objekterzeugung, jedoch nicht dessen konkrete Klasse. Um unabhängig von der Klasse des zu erzeugenden Objekts zu sein und damit allgemein verwendbar zu bleiben, muß die eigentliche Objekterzeugung entweder in eine eigene Erzeugerklasse ausgelagert werden oder durch Unterklassen realisiert werden.
  Typisch für solche Situationen sind z.B. Frameworks, die es ermöglichen ganze Gruppen verwandter Anwendungen mit gemeinsam nutzbaren Klassen, zu realisieren. Beispiel sei ein Framework für Desktop-Applikationen, die jeweils mit unterschiedlichen Dokumenten-Objekten arbeiten. Die Framework-Klassen sollten unabhängig von der Klasse des konkret zu erzeugenden applikationsspezifischen Dokumenten-Objekts sein.

- **Problemlösung**
  Definition eines **Interfaces** in einer – häufig abstrakten – Klasse **`Creator`** zur Erzeugung eines Objekts der Klasse **`Product`** mittels einer (rein) **virtuellen Funktion** (→ ***factory method***). Diese Methode wird in von `Creator` abgeleiteten Klassen (**`ConcreteCreator`**) so **überschrieben**, dass das jeweils benötigte konkrete Objekt (abgeleitete Klasse **`ConcreteProduct`**) erzeugt wird. Die `Creator`–Klasse verschiebt also die konkrete Objekterzeugung zu ihren abgeleiteten Klassen, die somit die Applikationsabhängigkeit enthalten.
  Der applikationsunabhängige Code (der sich in einer eigenen Anwendungsklasse **`Client`** befindet oder auch in der `Creator`–Klasse angesiedelt sein kann) ruft zur Erzeugung eines Objekts über das Interface tatsächlich die überschreibende Methode auf.

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Der die Objekterzeugung anfordernde Code (Client) ist unabhängig von der Klasse des konkret erzeugten Objekts. Er ist damit abstrakter und wiederverwendbarer.
  - ▻ Es existieren **zwei Hauptvarianten** dieses Entwurfsmusters :
    - Der die Objekterzeugung anfordernde Code befindet sich in einer von `Creator` getrennten eigenen Klasse
    - Der die Objekterzeugung anfordernde Code befindet sich in der `Creator`–Klasse (`Client`–Klasse existiert nicht)
  - ▻ Die `Creator`–Klasse kann abstrakt sein (*factory method* ist rein virtuell). Sie kann aber auch eine konkrete Klasse sein und eine Default-Implementierung dieser Methode bereitstellen.
  - ▻ Das Entwurfsmuster erhöht die Klassen-Komplexität, da Subklassen allein zur Objekterzeugung benötigt werden.
  - ▻ Die Objekterzeugungsfunktion (*factory method*) kann auch **parameterisiert** werden. Die Parameter können entweder an den Konstruktor der Klasse des zu erzeugenden Objekts weitergereicht werden oder/und zur Auswahl zwischen verschiedenen Klassen dienen → Erzeugung von Objekten unterschiedlichen Typs durch eine Methode.

Entwurfsmuster : Factory Method (2)

- **Implementierungsbeispiel** (Teil 1)

  ◇ **Definition einer – abstrakten – Produkt-Klasse** (**`Product`**)

```
class RawDiskReader
{ public :
    virtual ~RawDiskReader() {};
    virtual bool openDisk(char*) = 0;
    virtual bool readSector(unsigned long, char*) = 0;
};
```

◇ **Definition einer konkreten Produkt-Klasse** (**`ConcreteProduct`**)

```
class WinRawDiskReader : public RawDiskReader
{ public :
    WinRawDiskReader(char* = NULL);
    ~WinRawDiskReader();
    bool openDisk(char*);
    bool readSector(unsigned long, char*);
  private :
    // Datenkomponenten und private Memberfunktionen
};
```

◇ **Definition einer – abstrakten – Creator-Klasse** (**`Creator`**)

```
class RDRCreator
{ public :
    virtual RawDiskReader* createRDR()=0;   // factory method (Interface)
};
```

◇ **Definition einer konkreten Creator-Klasse** (**`ConcreteCreator`**)

```
class WinRDRCreator : public RDRCreator
{ public :
    RawDiskReader* createRDR();             // factory method
};
```

◇ **Definition einer Anwendungsklasse** (**`Client`***)*

```
class DiskDump
{ public :
    DiskDump(RDRCreator*);
    ~DiskDump();
    void doDump(char*, unsigned);
    // weitere öffentliche Memberfunktionen
  private :
    RDRCreator* m_pCre;        // Pointer auf Creator-Objekt
    // weitere Datenkomponenten und private Memberfunktionen
};
```

Entwurfsmuster : Factory Method (3)

- **Implementierungsbeispiel** (Teil 2)

◇ **Implementierung der Klasse `WinRawDiskReader`** (Auszug) (applikationsabhängig)

```
WinRawDiskReader::WinRawDiskReader(char* diskname)
{ if (diskname!=NULL)
    openDisk(diskname);
  else
  { m_hDisk=INVALID_HANDLE_VALUE;
    m_pcDiskName=NULL;
  }
}

WinRawDiskReader::~WinRawDiskReader()
{ CloseHandle(m_hDisk);
  delete [] m_pcDiskName;
  // ...
}

bool WinRawDiskReader::openDisk(char* diskname) { /* ... */ }

bool WinRawDiskReader::readSector(unsigned long sec, char* buff) { /* ... */ }
```

◇ **Implementierung der Klasse `WinRDRCreator`** (applikationsabhängig)

```
RawDiskReader* WinRDRCreator::createRDR()     // factory method
{
  return new WinRawDiskReader();
}
```

◇ **Implementierung der Klasse `DiskDump`** (Auszug) (applikationsunabhängig)

```
DiskDump::DiskDump(RDRCreator* cr)
{ m_pCre=cr;
}

void DiskDump::doDump(char* dname, unsigned size)
{ // ...
  RawDiskReader* pRDR=m_pCre->createRDR();   // Erzeugung eines Reader-Objekts
  if (pRDR->openDisk(dname))
  // ...
}
```

◇ **Beispiel für verwendenden Code** (Funktion **`main()`** eines Programms `diskdump`) (applikationsabhängig)

```
int main(int argc, char** argv)
{ if (argc==1)
    cout << "\nAufruf : diskdump <diskname>\n";
  else
  { RDRCreator* cr = new WinRDRCreator;   // einzige applikationsabhängige Zeile
    DiskDump dd(cr);
    dd.doDump(argv[1], SEC_SIZE);
  }
  return 0;
}
```

Entwurfsmuster : Composite (1)

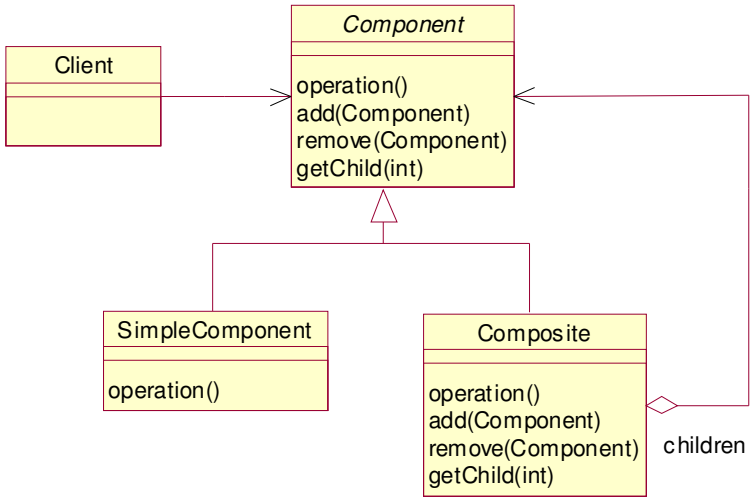
- **Name : Composite** (*Recursive Composition*)

- **Kurzbeschreibung**
  Ermöglicht die Repräsentation von Teil-Ganzheits-Beziehungen durch den Aufbau baumartiger Objektstrukturen. Dadurch können Clients individuelle Objekte und Zusammensetzungen von Objekten gleichartig behandeln.

- **Problembeschreibung**
  In vielen Anwendungsbereichen treten Strukturen auf, die aus einfachen Objekten und zusammengesetzten Objekten (Containern) aufgebaut sind. Dabei können i.a. die zusammengesetzten Objekte wiederum zusammengesetzte Objekte als Komponenten enthalten ⇒ Rekursive Teil-Ganzheits-Hierarchien.
  (Beispiel : Fenster-System → Ein Fenster enthält u.a. Text, einfache Steuerungselemente und "Unter"-Fenster).
  Es ist sinnvoll und effektiv, wenn Code, der die entsprechenden Klassen benutzt (Client), einfache Objekte und zusammengesetzte Objekt völlig gleichartig behandeln kann, d.h. beim Aufruf von Komponenten-Methoden nicht zwischen einfachen und zusammengesetzten Objekten unterscheiden muß.

- **Problemlösung**
  Die Klassen für einfache Komponenten und für zusammengesetzte Komponenten werden von einer gemeinsamen abstrakten Basisklasse **`Component`** abgeleitet.
  Diese definiert sowohl das "allgemeine" Komponenten-Anwendungs-Interface als auch das spezielle "Management"-Interface für zusammengesetzte Komponenten.
  Von der gemeinsamen Komponenten-Basisklasse **`Component`** können durchaus mehrere Klassen für einfache Komponenten abgeleitet sein (im untenstehenden Klassendiagramm nur die Klasse **`SimpleComponent`**).
  Die Klasse für zusammengesetzte Komponenten **`Composite`** kann selbst wieder eine (abstrakte) Basisklasse für mehrere (konkrete) Composite-Klassen sein.
  Die Klasse für zusammengesetzte Komponenten **`Composite`** verwaltet ihre enthaltenen Objekte (Kindkomponenten, "children") als Instanzen der gemeinsamen Komponenten-Basisklasse **`Component.`**
  An sie gerichtete Anwendungs-Methodenaufrufe ("`operation()`") delegiert sie an alle in ihr enthaltenen Komponenten.

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Client-Code kann immer dann, wenn er ein einfaches Objekt erwartet auch mit einem zusammengesetzten Objekt arbeiten.
  - ▻ Da Clients einfache und zusammengesetzte Objekte gleich behandeln können, vereinfacht sich der Client-Code (z.B keine `switch-case`-Anweisungen zu ihrer Unterscheidung notwendig).
  - ▻ Neue Komponenten-Klassen können einfach ohne Änderungen des Client-Codes hinzugefügt werden.
  - ▻ Andererseits ist es schwierig, Kindkomponenten auf bestimmte Typen zu begrenzen (→ zusätzliche Laufzeitüberprüfungen notwendig).

Entwurfsmuster : Composite (2)

- **Implementierungsbeispiel** (Teil 1)

  ◇ **Definition einer Komponenten-Basisklasse** (**`Component`**) **:**

```
class Equipment
{
  public :
    virtual ~Equipment ();
    const char* getName() const { return m_szName; }
    virtual double getPrice() const = 0;
    // weitere Anwendungs-Memberfunktionen
    virtual void add(Equipment*);
    virtual void remove(Equipment*);
    virtual Iterator<Equipment*>* createIterator();
  protected :
    Equipment (const char*);
  private :
    const char* m_szName;
};
```

  ◇ **Definition einer einfachen Komponentenklasse** (**`SimpleComponent`**) **:**

```
class BasicPart : public Equipment
{
  public :
    BasicPart(const char*, double);
    ~BasicPart();
    virtual double getPrice() const;
    // weitere Anwendungs-Memberfunktionen
  private :
    double m_dPrice;
};
```

  ◇ **Definition einer zusammengesetzten Komponentenklasse** (**`Composite`**) **:**

```
class CompositeEquipment : public Equipment
{
  public :
    CompositeEquipment(const char*);
    virtual ~CompositeEquipment();
    virtual double getPrice() const;
    // weitere Anwendungs-Memberfunktionen
    virtual void add(Equipment*);
    virtual void remove(Equipment*);
    virtual Iterator<Equipment*>* createIterator();
  private :
    List<Equipment*> m_pclEquipment;
};
```

Entwurfsmuster : Composite (3)

- **Implementierungsbeispiel** (Teil 2)

◇ **Implementierung der Klasse `Equipment` (Auszug):**

```
Equipment:: Equipment(const char* name)
{
  m_szName=name;
}
```

◇ **Implementierung der Klasse `BasicPart` (Auszug):**

```
BasicPart::BasicPart(const char* name, double price) : Equipment(name)
{
  m_dPrice=price;
}

double BasicPart::getPrice() const
{
  return m_dPrice;
}
```

◇ **Implementierung der Klasse `CompositeEquipment` (Auszug):**

```
CompositeEquipment::CompositeEquipment(const char* name) :
  Equipment(name) { }

double CompositeEquipment::getPrice() const
{
  Iterator<Equipment*>* i = createIterator();
  double sum=0.0;
  for (i->first(); i->hasMore(); i->next())
    sum+=i->current()->getPrice();
  delete i;
  return sum;
}
```

◇ **Beispiel für Anwendungscode :**

```
CompositeEquipment* cabinet  = new CompositeEquipment("PC Cabinet");
CompositeEquipment* motherboard = new CompositeEquipment("PC Motherboard");
BasicPart* emptyCabinet  = new BasicPart("Cabinet (empty)", 110.00);
BasicPart* emptyBoard     = new BasicPart("ASUS-Board Pentium II 350", 370.00 );

cabinet->add(emptyCabinet);
cabinet->add(motherboard);
motherboard->add(emptyBoard);

CompositeEquipment* bus = new CompositeEquipment("PCI-Bus");
bus->add(new BasicPart("ATI Graphic Card", 50.00));
motherboard->add(bus);
motherboard->add(new BasicPart("3.5 Floppy Disc", 65.00));

cout << endl << "The total price is : " << cabinet->getPrice() << endl;
```

Entwurfsmuster : Strategy (1)

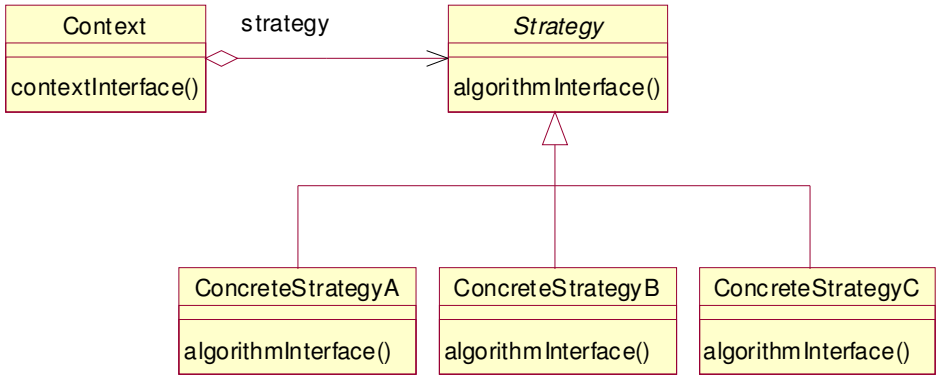
- **Name : Strategy** (*Policy*)

- **Kurzbeschreibung**
  Kapselt verwandte Algorithmen in jeweils einer eigenen Klasse, die von einer gemeinsamen Basisklasse abgeleitet ist. Dies ermöglicht die Auswahl und Änderung der Algorithmen zur Laufzeit ohne daß Änderungen an den Clients, die sie benutzen, notwendig sind.

- **Problembeschreibung**
  Häufig sollen zur Realisierung bestimmter Funktionalitäten einer Klasse unterschiedliche Algorithmen einsetzbar sein. (Beispiel : Unterschiedliche Auswahlstrategien eines Workflow-Managers zur Bestimmung der nächsten zu bearbeitenden Task.)
  Eine Codierung sämtlicher möglichen Algorithmen in einer Klasse führt zu einer aufwendigen, umständlichen und wenig änderungsfreundlichen Implementierung.
  Die Erzeugung mehrerer ähnlicher, sich nur im jeweils verwendeten Algorithmus unterscheidenden, Klassen stellt ebenfalls eine unhandliche und inflexible Lösung dar.

- **Problemlösung**
  Auslagerung der Funktionalität in eine eigene (abstrakte) Basisklasse **`Strategy`**, von der für jeden in Frage kommenden Algorithmus eine eigene konkrete Strategie-Klasse abgeleitet wird (**`ConcreteStrategyA`**, **`ConcreteStrategyB`** usw).
  Die Klasse, die diese Funktionalität eigentlich ausführen soll, (**`Context`**) wird mit einer Referenz (oder einem Pointer) auf ein konkretes Strategie-Objekt konfiguriert, an das sie Anforderungen zur Ausführung der betreffenden Funktionalität weiterleitet.

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Die Isolierung der Algorithmen-"Familie" in einer eigenen Klassenhierarchie erlaubt ihre einfachere Wiederverwendung.
  - ▻ Das Entwurfsmuster stellt eine sehr flexible Alternative zur Vererbung dar : Es wird nur eine Kontext-Klasse benötigt die ohne Änderungen an ihr vorzunehmen, mit unterschiedlichen Strategien konfiguriert werden kann.
  - ▻ Die jeweils verwendete Strategie kann leicht – auch zur Laufzeit - ausgetauscht werden.
  - ▻ Es ist möglich, daß das von der Klasse **`Strategy`** definierte Interface, Parameter enthält, die nicht von allen konkreten Strategie-Objekten benötigt werden.⇒ Kommunikations-Overhead zwischen **`Context`** und **`Strategy`**
    Abhilfe : **`Context`** definiert ein Interface für den Zugriff durch **`Strategy`**, über das sich **`Strategy`** nur die wirklich benötigte Information beschafft. (→ engere Kopplung zwischen **`Strategy`** und **`Context`**).

Entwurfsmuster : Strategy (2)

- **Implementierungsbeispiel** (Teil 1)

  ◇ **Definition einer abstrakten Strategy-Basisklasse (`Strategy`) :**

```
class TaskScheduler
{
  public :
    virtual ~TaskScheduler();
    virtual Task* determineNextTask(TaskList*) = 0;
    // gegebenenfalls weitere Memberfunktionen
  protected :
    TaskScheduler(WFManager*);
  private :
    WFManager* m_pclManager;
};
```

  ◇ **Definition einer konkreten StrategyKlasse (`ConcreteStrategyA`) :**

```
class SimpleTaskScheduler : public TaskScheduler
{
  public :
    SimpleTaskScheduler(WFManager*);
    ~SimpleTaskScheduler();
    virtual Task* determineNextTask(TaskList*);
    // gegebenenfalls weitere Memberfunktionen
  private :
   // gegebenenfalls weitere Datenkomponenten;
};
```

  ◇ **Definition einer Context-Klasse (`Context`) :**

```
class WFManager
{
  public :
    WFManager();
    WFManager(TaskScheduler*);
    ~WFManager();
    void setTaskScheduler(TaskScheduler*);
    // weitere Memberfunktionen
  private :
    void handleWaitingTasks();
    TaskScheduler* m_pclScheduler;
    TaskList* m_pclTasks;
    // weitere Datenkomponenten
};
```

Entwurfsmuster : Strategy (3)

- **Implementierungsbeispiel** (Teil 2)

  ◇ **Implementierung der Klasse `TaskScheduler`** (Auszug)**:**

```
TaskScheduler ::TaskScheduler(WFManager* manager)
{
  m_pclManager=manager;
}
```

  ◇ **Implementierung der Klasse `SimpleTaskScheduler`** (Auszug)**:**

```
SimpleTaskScheduler::SimpleTaskScheduler(WFManager* manager) :
  TaskScheduler(manager) { }

Task* SimpleTaskScheduler::determineNextTask(TaskList* tasks)
{ Task* nextTask;
  // Implementierung eines einfachen Auswahlalgorithmus (z.B.FIFO)
  return nextTask;
}
```

  ◇ **Implementierung der Klasse `WFManager`** (Auszug)**:**

```
WFManager::WFManager()
{
  // Realisierung des Default-Konstruktors
}

WFManager::WFManager(TaskScheduler* scheduler)
{
  m_pclScheduler=scheduler;
  // restliche Initialisierung
}

WFManager::~WFManager()
{
  delete m_pclScheduler;
  // weitere Funktionalität des Destruktors
}

void WFManager::setTaskScheduler(TaskScheduler* scheduler)
{ if (m_pclScheduler!=0)
    delete m_pclScheduler;
  m_pclScheduler=scheduler;
}

void WFManager::handleWaitingTasks()
{ // ...
  Task* nextTask;
  nextTask=m_pclScheduler->determineNextTask(m_pclTasks);
  // ...
}
```

Entwurfsmuster : Observer (1)

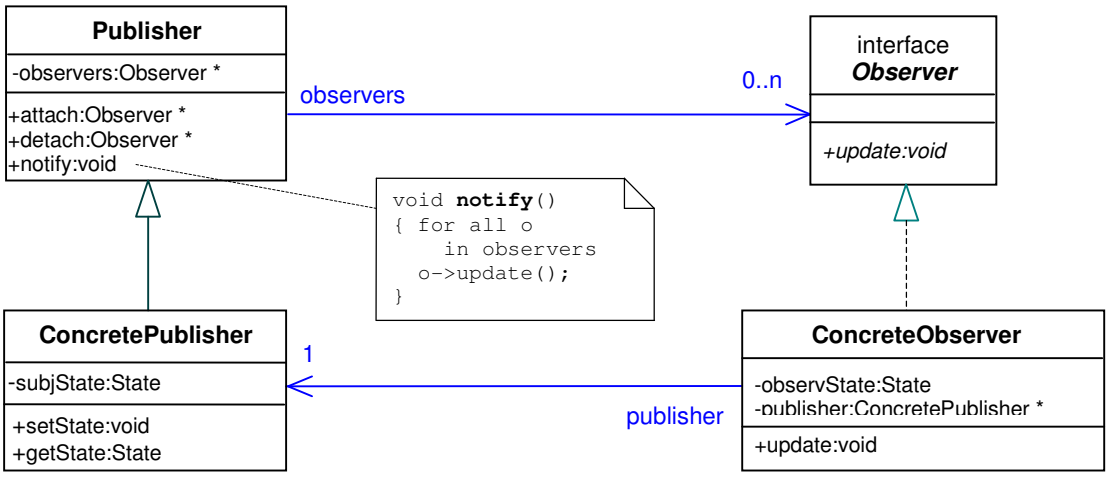
- **Name : Observer** (*Publish-Subscribe*, *Dependents*)

- **Kurzbeschreibung**
  Ermöglicht die **dynamische Registrierung** von **Objektabhängigkeiten**, so daß bei einem **Zustandswechsel** eines Objekts alle von ihm **abhängigen Objekte benachrichtigt** werden.

- **Problembeschreibung**
  In vielen Anwendungsbereichen soll sich der Zustandswechsel eines Objekts direkt auf den Zustand bzw das Verhalten anderer Objekte auswirken.
  Beispiel : Ein Datenbestand und seine – u.U. gleichzeitige – Darstellung in verschiedenen Formaten in einem GUI-System (z. B. Tabelle, Balkendiagramm, Tortendiagramm). Jede Änderung des Datenbestandes muß sich umgehend auf alle Darstellungen auswirken.
  Eine enge Kopplung der beteiligten Objekte (→ die beteiligten Objekte haben voneinander Kenntnis) ist meist nicht wünschenswert, da dadurch die unabhängige Verwendung und Modifikation ihrer jeweiligen Klassen stark eingeschränkt wird. Außerdem müssen in diesem Fall die Abhängigkeiten bereits zur Compilezeit bekannt sein, was eine flexible dynamische Anpassung zur Laufzeit verhindert.

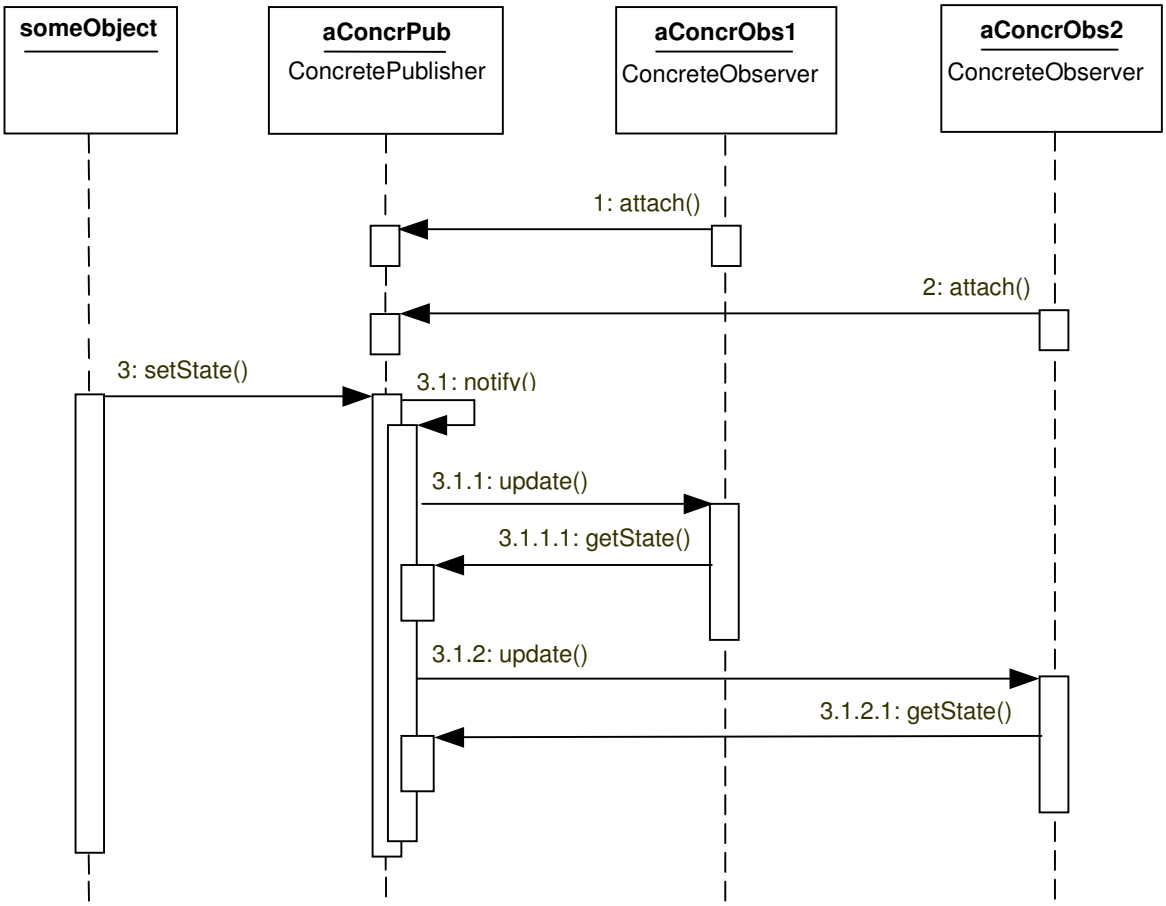
- **Problemlösung**
  Bei dem Objekt, von dessen Zustand andere Objekte abhängen (***Publisher*-Objekt**), wird eine **Liste der abhängigen Objekte** geführt. In diese Liste tragen sich alle Objekte, die über den Zustand dieses Objekts auf dem laufenden gehalten werden wollen (weil sie von ihm abhängen), ein (***Subscriber*-Objekte**, ***Observer*-Objekte**).
  Bei einem **Wechsel** seines Zustands **informiert** das *Publisher*-Objekt alle eingetragenen *Observer*-Objekt hierüber (***notify***).
  Diese können dann den neuen Zustand vom *Publisher*-Objekt ermitteln und entsprechend der eingetretenen Änderung reagieren (z.B. ihren eigenen Zustand an den Zustand des *Publisher*-Objektes anpassen).
  Die Anzahl der *Observer*-Objekte muß dem *Publisher*-Objekt a priori nicht bekannt sein. Bei ihm können sich zur Laufzeit beliebig viele *Observer*-Objekte an- bzw auch wieder abmelden.
  Publisher-Objekt und *Observer*-Objekte sind von einander entkoppelt und können unabhängig voneinander modifiziert werden.
  Die ***Publisher*-Schnittstelle** wird durch eine geeignete Klasse (**`Publisher`**) zur Verfügung gestellt. Diese Klasse dient als **Basisklasse** für konkrete *Publisher*-Klassen (**`ConcretePublisher`**), die die Datenkomponenten für den jeweiligen Zustand und die Methoden zum Setzen und Ermitteln derselben bereitstellen.
  Das **Interface** zum Informieren von *Observer*-Objekten wird durch eine abstrakte Klasse (***`Observer`***) definiert
  Dieses Interface wird von konkreten *Observer*-Klassen (**`ConcreteObserver`**) implementiert. Objekte dieser Klassen enthalten – neben ihren eigentlichen Zustands-Datenkomponenten – eine Referenz auf das konkrete *Publisher*-Objekt, bei dem sie sich eingetragen haben.

- **Struktur**



Entwurfsmuster : Observer (2)

- **Ablauf der Interaktion**



- **Anwendungskonsequenzen**
  - ***Publisher*** und ***Observer*** können **unabhängig** voneinander **geändert** und **ausgetauscht** werden.
    Publisher können wiederbenutzt werden, ohne ihre *Observer* wiederzubenutzen und umgekehrt.
    Observer können hinzugefügt werden, ohne den *Publisher* oder andere *Observer* zu verändern.
  - Die **Kopplung** zwischen *Publisher* und *Observer* ist **abstrakt** und **minimal**.
    Ein *Publisher*-Objekt weiss lediglich, dass es eine Liste von *Observer*-Objekten besitzt, die das *Observer*-Interface implementieren. Es kennt nicht die konkrete Klasse seiner *Observer*.
    Publisher und Observer können unterschiedlichen Abstraktions-Ebenen (Schichtenmodell !) angehören.
  - *Observer*-Objekte können gegebenenfalls selbst auch einen Zustandswechsel beim *Publisher*-Objekt bewirken
  - Die Information über den Zustandswechsel (**Notifikation**) durch das *Publisher*-Objekt ist automatisch eine **Broadcast-Kommunikation**. Es werden immer alle eingetragenen *Observer*-Objekte informiert, unabhängig davon, wieviel es sind. Es obliegt einem *Observer*-Objekt, eine Notifikation gegebenenfalls zu ignorieren.
  - Falls **sehr viele** *Observer*-Objekte eingetragen sind, kann eine Notifikation und der dadurch ausgelöste *Observer*-Update u.U. eine **längere Zeit** dauern.
  - Da das eine Zustandsänderung bewirkende Objekt keine Information über die *Observer* und die mit diesen verbundenen "Update"-Kosten hat, können **"leichtfertige" Zustandsänderungen** einen erheblichen zeit- und damit kostenintensiven **Aufwand** bewirken.
  - Das **einfache Notifikations-Protokoll** liefert **keine Informationen** darüber, **was** sich im Publisher-Objekt **geändert** hat. Dies festzustellen, obliegt dem Observer-Objekt, was den Update-Aufwand gegebenenfalls noch erhöht.
    Als Alternative kann u.U. ein **erweitertes Protokoll** definiert werden, dass detailliertere Zustandsänderungs-Info liefert.
  - Es gibt Fälle, bei denen sinnvoll ist, dass sich **ein *Observer***-Objekt bei **mehreren *Publisher*-**Objekten für eine Notifikation einträgt. In diesen Fällen muß die Update-Botschaft eine Information über das absendende Publisher-Objekt enthalten

Entwurfsmuster : Observer (3)

- **Implementierungsbeispiel** (Teil1)

  ◇ **Definition einer Publisher-Basisklasse** (Publisher)

```
class Publisher
{ public :
    virtual ~Publisher() {};
    virtual void attach(Observer*);
    virtual void detach(Observer*);
    virtual void notify();
  protected :
    Publisher() {};
  private :
    Set<Observer*> observs;       // Liste der eingetragenen Observer-Objekte
};
```

  ◇ **Implementierung der Klasse `Publisher`**

```
void Publisher::attach(Observer* no)
{ observs.insert(no);
}

void Publisher::detach(Observer* no)
{ observs.remove(no);
}

void Publisher::notify()
{ Observer** ppo;
  for(ppo=observs.first(); ppo; ppo=observs.next())
    (*ppo)->update(this);
}
```

  ◇ **Definition einer konkreten Publisher-Klasse** (ConcretePublisher)

```
class ClockTimer : public Publisher
{ public :
    ClockTimer();
    ~ClockTimer();
    int getHour();
    int getMinute();
    int getSecond();
    void tick();          // Veränderung des Objekt-Zustands
  private :
    // Datenkomponente(n) zur Speicherung der Zeit
};
```

  ◇ **Implementierung der Klasse `ClockTimer`**

```
// Implementierung der uebrigen Memberfunktionen

void ClockTimer::tick()
{
  // update der privaten Datenkomponente(n) zur Speicherung der Zeit
  notify();
}
```

Entwurfsmuster : Observer (4)

- **Implementierungsbeispiel** (Teil2)

  ◇ **Definition einer abstrakten Observer-Basisklasse** (*Observer*)

```
class Publisher;

class Observer
{ public :
    virtual ~Observer() { };
    virtual void update(Publisher*) = 0;
  protected :
    Observer() {};
};
```

  ◇ **Definition einer konkreten Observer-Klasse** (ConcreteObserver)

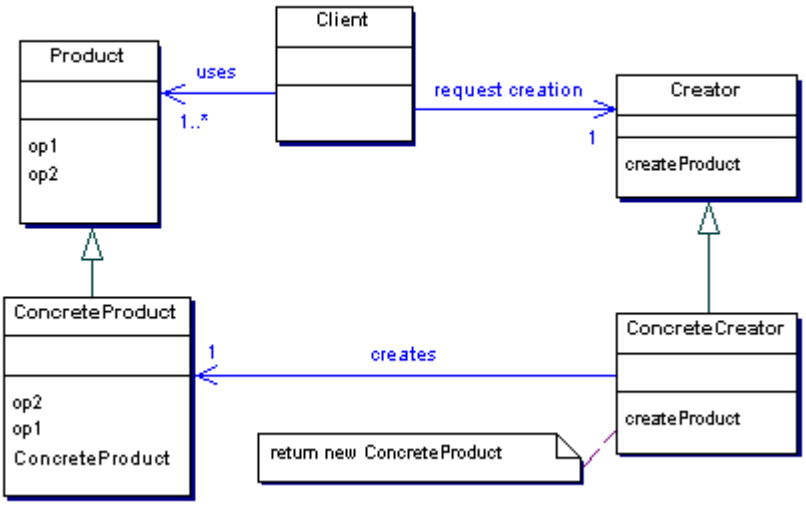
```
class Widget;      // Fenster-Basisklasse mit graph. Fähigkeiten
class ClockTimer;

class DigitalClock : public Observer, public Widget
{ public :
    DigitalClock(ClockTimer*);
    ~DigitalClock();
    void update(Publisher*);
    void draw();             // virtuelle Methode von Widget
  private :
    ClockTimer* m_publ;
};
```

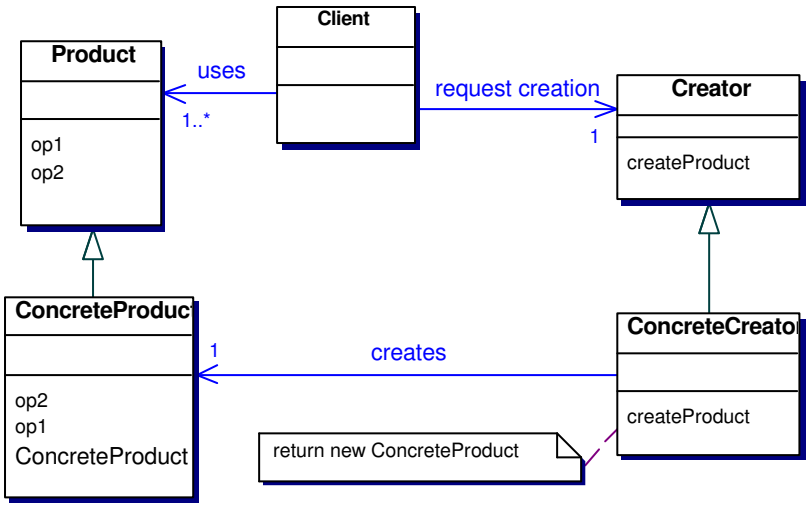
  ◇ **Implementierung der Klasse `DigitalClock`**

```
DigitalClock::DigitalClock(ClockTimer* ct)
{ m_publ=ct;
  m_publ->attach(this);
}

DigitalClock::~DigitalClock()
{ m_publ->detach(this);
}

void DigitalClock::update(Publisher* pub)
{ if (pub==m_publ)
    draw();
}

void DigitalClock::draw()              // virtuelle Methode von Widget
{ int h = m_publ->getHour();
  int m = m_publ->getMinute();

  // Zeichnen der Digital-Uhr
}
```

  ◇ **Beispiel für Anwendungscode** (Auszug)

```
  ClockTimer timer;
  DigitalClock digClock(&timer);
  // bei jedem Aufruf von timer.tick() stellt digClock die neue Zeit dar
```


Client
Product
uses
1..*
request creation
1
Creator
op1
op2
createProduct
ConcreteProduct
op2
op1
ConcreteProduct
1
creates
ConcreteCreator
createProduct
return new ConcreteProduct


Client
Product
uses
1..*
request creation
1
Creator
op1
op2
createProduct
ConcreteProduct
op2
op1
ConcreteProduct
1
creates
ConcreteCreator
createProduct
return new ConcreteProduct

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 10

10. Ausgewählte Komponenten der Standardbibliothek

ANSI/ISO-C++-Standardbibliothek - Allgemeines

• Grundsätzliches

◇ Mit dem **ANSI/ISO-Standard** für C++ ist neben der eigentlichen Sprache auch eine dazugehörige **Standard-Bibliothek** genormt worden.

◇ Die C++-Standardbibliothek verwendet eine Reihe neuerer erst **relativ spät in die Sprache übernommener Sprachmittel** (z.B. Namespaces, Exception Handling, Templates, Datentyp `bool`, neue Typwandlungs-Operatoren). Daher ist sie u.U.. noch nicht in allen vorhandenen C++-Systemen vollständig realisiert.

◇ Die Standardbibliothek wurde nicht von Grund auf neu entwickelt, sondern faßt zahlreiche schon vorher weitgehend unabhängig voneinander entstandene Komponenten und Ansätze zusammen, die soweit wie möglich aneinander angepaßt, sowie um zahlreiche Vorschläge und Mechanismen ergänzt und erweitert wurden.
Zusätzlich wurde auch die **ANSI-C-Standardbibliothek übernommen**.
Diese ist somit - z.Tl. geringfügig angepaßt - Bestandteil der ANSI-C++- Standardbibliothek.
Die **C++-Standardbibliothek** stellt daher **kein homoges Gebilde** dar, sondern besteht aus **verschiedenen Teilen**, die weitgehend unabhängig voneinander sind, die aber doch in einigen Details miteinander verbunden sind.

◇ Alle in der C++-Standardbibliothek definierten **globalen Namen** - auch die in der integrierten ANSI-C-Standardbibliothek definierten - sind im **Namensbereich `std`** definiert.

• Konventionen für Header-Dateien :

◇ Header-Dateien werden in der `#include`-Anweisung **ohne Namens-Extension** angegeben, z.B.

```
#include <iostream>
```

◇ Auch die **Header-Dateien der C-Standardbibliothek** werden **ohne Extension** angegeben, allerdings wird ihnen zur Unterscheidung **ein c vorangestellt**, z.B.

```
#include <cstdlib>        // C-Header-Datei stdlib.h
```

◇ Die Angabe der Header-Dateinamen ohne Extension bedeutet nicht, daß Header-Dateien generell keine Namens-Extension mehr besitzen.
Vielmehr ist die Umsetzung des in der `#include`-Anweisung verwendeten Dateinamens in den tatsächlich im jeweiligen C++-System verwendeten Dateinamen implementierungsabhängig.
Z.B. kann

```
#include <iostream>
```

vom Preprozessor umgesetzt werden in

```
namespace std
{
    #include <iostream.h>
}
```

◇ Aus **Kompatibilitätsgründen** können für die **C-Header-Dateien** auch die **"normalen" Namen mit Extension** verwendet werden.
Dies gilt in der Praxis - wenn auch nicht im Standard festgelegt - ebenfalls für C++-spezifische Header-Dateien, die bereits vor der Definition der Standardbibliothek verwendet wurden, z.B

```
#include <iostream.h>
```

Werden **Header-Datei-Namen mit Extension** verwendet, so befinden sich die definierten globalen Namen im **globalen Namensbereich** (nicht `std`)

ANSI/ISO-C++-Standardbibliothek - Überblick

- **Bestandteile der Standardbibliothek** :

  ◇ **Sprachunterstützungs-Bibliothek (*Language support library*)**

  Funktionen und Typen, die zur Realisierung bestimmter Sprachelemente benötigt werden, wie z.B.
  Funktionen, die während der Ausführung von C++-Programmen implizit aufgerufen werden, Definition zugehöriger verwendeter Typen, Festlegung implementierungsabhängiger Eigenschaften der Standardtypen
  (u.a. Klassen-Template **`numeric_limits`**)

  ◇ **Diagnose-Bibliothek (*Diagnostics library*)**

  Komponenten zum Entdecken und Melden von Fehler-Bedingungen,
  u.a. Hierarchie von Fehlerklassen (***exception classes***), Macro **`assert`**, Fehlernummern-Macros

  ◇ **Utility-Bibliothek (*General utility library*)**

  Ergänzende Hilfsmittel, u.a. Datentypen für Wertepaare (Klassen-Template **`pair`**) und "sichere" Pointer (Klassen-Template **`auto_ptr`**), Default-Allokator-Klasse **`allocator`** (Allokator-Objekte repräsentieren Speichermodelle)

  ◇ **String-Bibliothek (*Strings library*)**

  Klassen-Template **`basic_string`** und konkrete Realisierungen (Template-Klassen) **`string`** und **`wstring`**

  ◇ ***Localization library***

  Komponenten zur portablen Unterstützung nationaler Eigenheiten
  (z.B. Zeichendarstellung und Zeichensatz, Datums-, Währungs-, Zahlformate), u.a. Klasse **`locale`**

  ◇ **STL -** ***Standard Template Library***

  Bibliothek zur Verwaltung und Bearbeitung von Mengen beliebigen Typs.
  Sie besteht aus 3 Teilen :

  ▻ ***Containers library***
  Container-Klassen (Objekte dieser Klassen speichern und verwalten andere Objekte)

  ▻ ***Iterators library***
  Iterator-Klassen (Iteratoren dienen zum Zugriff zu einzelnen Elementen einer Objektmenge, insbesondere zu Elementen von Containern)

  ▻ ***Algorithms library***
  "freie" Funktionen zur Realisierung von Algorithmen, mit denen Mengen und Elemente in Mengen bearbeitet werden können.

  In diesem Teil der Bibliothek sind auch Komponenten enthalten, die nicht direkt auf Objekt-Mengen bezogen sind, z.B.
  - Klassen-Template **`bitset`** zur Verwaltung und Bearbeitung von Bit-Feldern fester aber beliebig wählbarer Größe (*Containers library*)
  - "freie" Funktions-Templates zur Realisierung häufig benötigter Hilfsfunktionen, wie z.B `max(), min(), swap()` (*Algorithms library*)

  ◇ **Numerische Bibliothek (*Numerics library*)**

  Klassen-Templates für komplexe Zahlen (**`complex`**) und numerische Felder, wie z.B. Vektoren und Matrizen (**`valarray`**),
  Erweiterung der numerischen Funktionen der C-Standard-Bibliothek durch Überladen (freie Funktionen)

  ◇ **Stream-I/O-Bibliothek (*Input/Output library*)**

  Stream-Klassen (einschließlich File- u. String-Streams), Standard-Stream-Objekte, Stream-Buffer-Klassen, Standard-Manipulatoren

ANSI/ISO-C++-Standardbibliothek – Überblick Header-Dateien

Language support library	
Typen	**<cstddef>**
Implementierungsabhängige Eigenschaften	**<limits>, <climits>, <cfloat>**
Programm-Start u. Beendigung	**<cstdlib>**
Dynamische Speicherverwaltung	**<new>**
Typidentifikation	**<typeinfo>**
Ausnahmebehandlung	**<exception>**
weitere Laufzeitunterstützung	**<cstdarg>, <csetjmp>, <ctime>, <csignal>, <cstdlib>**
Diagnostics library	
Exeception-Klassen	**<stdexcept>**
Assertions	**<cassert>**
Fehlernummern	**<cerrno>**
General utilities library	
Utility-Komponenten	**<utility>**
Funktions-Objekte	**<functional>**
Speicher	**<memory>**
Datum und Zeit	**<ctime>**
Strings library	
Zeicheneigenschaften (*char traits*)	**<string>**
String-Klassen	**<string>**
NUL-terminierte Zeichenfolgen	**<cctype>, <cwctype>, cstring>, <cwchar>, <cstdlib>**
Localization library	
Locales-Komponenten u. Kategorien	**<locale>**
C-Bibliotheks-Locales	**<clocale>**
Containers library	
Folgen (sequences)	**<deque>, <list>, <queue>, <stack>, <vector>**
Assoziative Container	**<map>, <set>**
Bitset	**<bitset>**
Iterators library	
Iteratoren	**<iterator>**
Algorithms library	
Diverse Algorithmen	**<algorithm>**
C-Bibliotheks-Algorithmen	**<cstdlib>**
Numerics library	
Komplexe Zahlen	**<complex>**
Numerische Felder	**<valarray>**
Numerische Operationen	**<numeric>**
Numerische C-Bibliothek	**<cmath>, <cstdlib>**
Input/Output library	
Forwärts-Deklarationen	**<iosfwd>**
Standard-Stream-Objekte	**<iostream>**
I/O-Stream-Basisklassen	**<ios>**
Stream-Buffer	**<streambuf>**
Formatierung und Manipulatoren	**<istream>, <ostream>, <iomanip>**
String-Streams	**<sstream>, <cstdlib>**
File-Streams	**<fstream>, <cstdio>, <cwchar>**

ANSI/ISO-C++-Standardbibliothek : Standard-Exception-Klassen (1)

• Grundsätzliches

◇ Die ANSI-C++-Standardbibliothek stellt auch einige Exception-Klassen zur Verfügung.
Diese bilden eine **Klassen-Hierarchie**, die von der Klasse **`exception`** abgeleitet ist.

◇ Einige dieser Exception-Klassen gehören zur **Sprachunterstützungsbibliothek** (*Language Support Library*).
Sie werden von **Sprachkonstrukten** verwendet (z.B. von `new, dynamic_cast, typeid`)

◇ Die übrigen Exception-Klassen werden von der **Standardbibibliothek selbst** benutzt. Sie sind – mit einer Ausnahme – in der **Diagnosebibliothek** (*Diagnostics Library*) definiert.
Objekte dieser Exception-Klassen können auch im **Anwender-Code** geworfen werden.
Darüberhinaus lassen sie sich auch als **Basisklassen** für **selbstdefinierte Exception-Klassen** einsetzen.

• Die Exception-Basisklasse `exception`

◇ Bestandteil der Sprachunterstützungsbibliothek.

◇ Sie ist die Basisklasse für alle Objekte, die als Exceptions von einigen Sprachausdrücken sowie von Bestandteilen der Standardbibliothek geworfen werden können.

◇ Ihre **öffentliche Schnittstelle** ist in der **Headerdatei `<exception>`** wie folgt **definiert :**

```
class exception
{
  public :
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
  private :
    // ...
};
```

◇ Die virtuelle **Methode** `const char* what()` dient zur Ermittlung einer Information über die **Ursache** einer Exception. Der von ihr für die Klasse `exception` zurückgelieferte C-String ist **implementierungsabhängig**.
Häufig wird in den von `exception` abgeleiteten Bibliotheks-Klassen diese Methode jeweils – ebenfalls implementierungsabhängig – überschrieben.
Auch in selbst definierten Exception-Klassen, die von einer der Bibliotheks-Exception-Klassen abgeleitet sind, kann sie – und muß sie gegegebenenfalls – in geeigneter Weise überschrieben werden.
Bei den Exception-Klassen, deren Instanzen in Sprachausdrücken geworfen werden können, ist die zurückgelieferte Information häufig direkt in der Methode `what()` implementiert, bei den von Komponenten der Standardbibliothek verwendeten Klassen lässt sich diese Information bei der Exception-Objekt-Erzeugung dem Konstruktor übergeben.

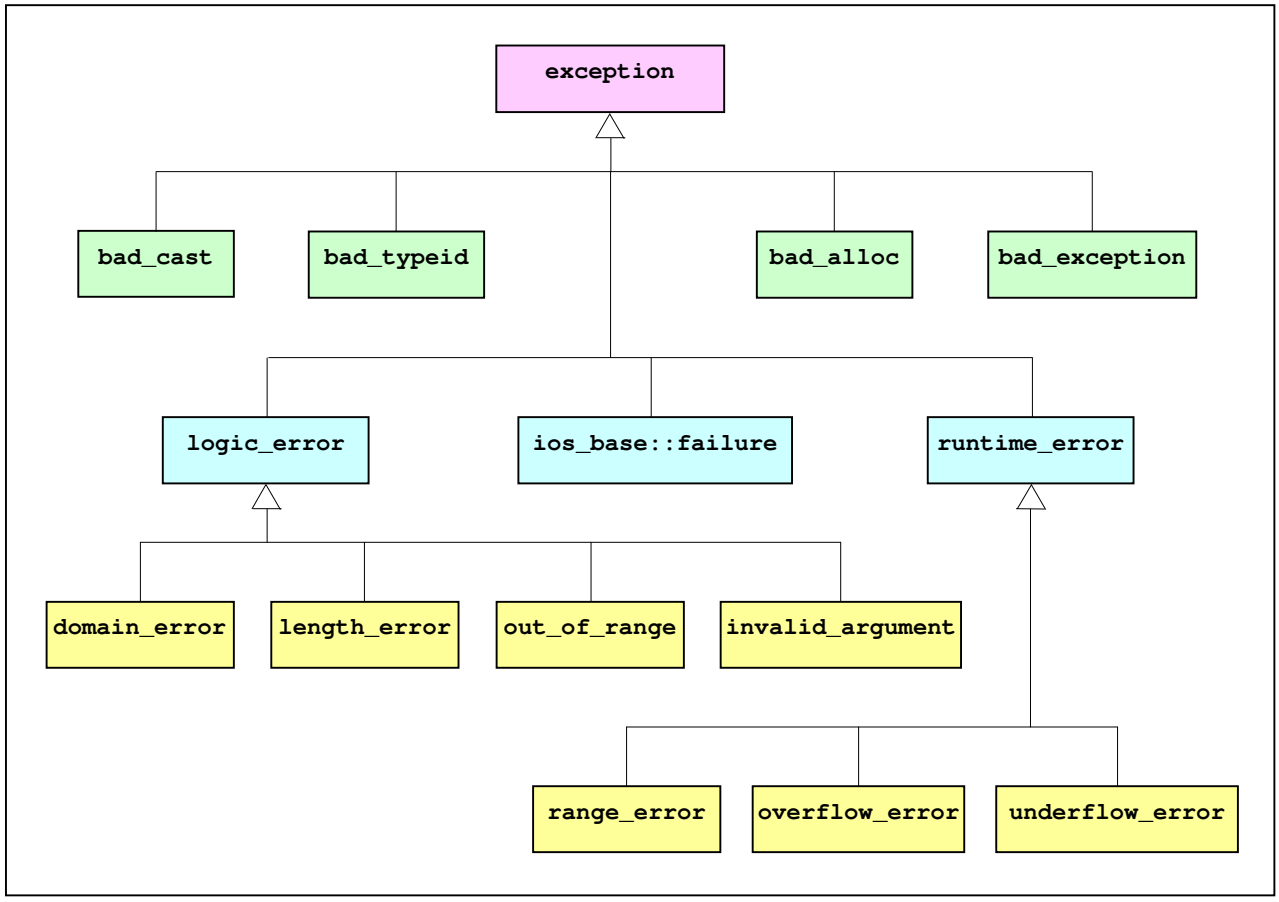
→ portable Auswertung der Exception-Information :

```
try
{
  // ...
}
catch (const exception& ex)
{
  cerr << endl << ex.what() << endl;
  // ...
}
```

◇ Die **Exception-Spezifikation `throw()`** der verschiedenen Memberfunktionen der Klasse `exception` legen fest, dass diese **selbst keine Exceptions werfen** dürfen.

ANSI/ISO-C++-Standardbibliothek : Standard-Exception-Klassen (2)

- **Hierarchie der Bibliotheks-Exception-Klassen**



- **Die Exception-Klassen zur Sprachunterstützung**

Klassenname	Headerdatei	geworfen von ... bei
bad_cast	<typeinfo>	Operator **dynamic_cast** bei ungültigem Referenz-Cast
bad_typeid	<typeinfo>	Operator **typeid** bei dereferenziertem NULL-Pointer im Operandenausdruck
bad_alloc	<new>	Operator **new** bei fehlgeschlagener Allokation (falls entsprechend implementiert)
bad_exception	<exception>	unter gewissen Umständen durch die Funktion `unexpected()`, wenn durch eine Funktion eine **nicht vorgesehene Exception** (Exception-Spec) geworfen wird

◇ Die **öffentliche Schnittstelle** für alle Exception-Klassen dieser Gruppe entspricht der Schnittstelle von `exception` :

- ▻ Konstruktor ohne Parameter
- ▻ Copy-Konstruktor
- ▻ Zuweisungsoperatorfunktion
- ▻ virtueller Destruktor
- ▻ virtuelle Memberfunktion `const char* what() const throw()` zur Ermittlung von Information über die Exception-Ursache

In einigen Implementierungen existiert darüberhinaus zusätzlich ein **Konstruktor**, dem eine **genauere Information** über die **Exception-Ursache** (bei Visual-C++ z.B. als C-String) übergeben werden kann.

ANSI/ISO-C++-Standardbibliothek : Standard-Exception-Klassen (3)

- **Die in der Standardbibliothek eingesetzten Exception-Klassen**

  ◇ Aus dieser Gruppe sind **drei Klassen direkt** von der Klasse **`exception`** **abgeleitet** , zwei in der Diagnosebibliothek und eine in der IO-Bibliothek. Von den Klassen der Diagnosebibliothek sind weitere abgeleitete Klassen definiert.

  ◇ Klasse **`logic_error`**

  ▻ definiert in der Headerdatei **`<stdexcept>`** (Diagnosebibibliothek)

  ▻ dient als **Basisklasse** für **weitere Exceptionklassen**, die den Auftritt **logischer Fehler** charakterisieren.
  Unter logischen Fehlern werden solche Fehler verstanden, die in der Programmlogik liegen und damit prinzipiell schon **vor dem Programmstart** oder durch das Überprüfen von Funktionsparametern gefunden werden können.

  Die folgenden von dieser Klasse **abgeleiteteten Klassen** sind ebenfalls in der Headerdatei **`<stdexcept>`** definiert :

  ▻ Klasse **`domain_error`**
  Objekte dieser Klasse werden beim Auftritt von Domänenfehlern (Wertebereichsfehlern) geworfen.

  ▻ Klasse **`length_error`**
  Objekte dieser Klasse werden geworfen, wenn versucht wird, ein Objekt zu erzeugen, dessen Größe die maximal zulässige Größe überschreiten würde (z.B. durch einige Memberfunktionnen der Bibliotheksklasse `string`).

  ▻ Klasse **`out_of_range`**
  Objekte dieser Klasse können geworfen werden, wenn ein Funktionsparameter ausserhalb des zulässigen Bereichs liegt (vor allem bei Positionsangaben, z.B. durch einige Memberfunktionen der Bibliotheksklasse `string` zur Auswahl von Stringelementen)

  ▻ Klasse **`invalid_argument`**
  Objekte dieser Klasse können geworfen werden, wenn einer Funktion ein Parameter mit einem ungültigen Wert übergeben wird (z.B. durch den Konstruktor des Klassen-Templates `bitset<>,` wenn ihm ein Stringparameter übergeben wird, der andere Zeichen als `'0'` und `'1'` enthält)

  ◇ Klasse **`runtime_error`**

  ▻ ebenfalls definiert in der Headerdatei **`<stdexcept>`** (Diagnosebibliothek)

  ▻ dient als Basisklasse für weitere Exceptionklassen, die den Auftritt von **Laufzeitfehlern** kennzeichnen.
  Laufzeitfehler sind Fehler, die prinzipiell **erst zur Laufzeit** erkannt werden können (z.B. Divison durch `0`).

  Die folgenden von dieser Klasse **abgeleiteteten Klassen** sind ebenfalls in der Headerdatei **`<stdexcept>`** definiert :

  ▻ Klasse **`range_error`**
  Objekte dieser Klasse kennzeichnen Bereichsfehler, die bei arithmetischen Berechnungen auftreten können.

  ▻ Klasse **`overflow_error`**
  Objekte dieser Klasse können geworfen werden, wenn ein arithmetischer Überlauf auftritt
  (z.B. in einer Memberfunktion des Klassen-Templates `bitset<>,` mit der ein Bitset in einen `unsigned long` Wert umgewandelt werden soll, wenn das Bitset nicht als `unsigned long` dargestellt werden kann)

  ▻ Klasse **`underflow_error`**
  Objekte dieser Klasse können geworfen werden, wenn ein arithmetischer Unterlauf auftritt

  ◇ Klasse **`ios_base::failure`**

  ▻ definiert in der Headerdatei **`<ios>`** innerhalb der Klasse `ios_base` (IO-Bibliothek)
  Die Headerdatei `<ios>` wird durch die Headerdatei `<iostream>` eingebunden.

  ▻ Objekte dieser Klasse (und davon abgeleiteter Klassen) werden zur Kennzeichnung von Fehlersituationen in der IO-Bibliothek geworfen.

  ◇ Für alle Klassen dieser Gruppe ist ein **Konstruktor** definiert, dem ein **`string`-Objekt**, das die **Fehlerursache** kennzeichnet, als Parameter zu übergeben ist.
  Beispiel : **`explicit logic_error(const string& what_arg);`**

  In realen Implementierungen sind i.a. zusätzlich definiert :

  ▻ virtueller Destruktor

  ▻ virtuelle Memberfunktion `const char*` **`what()`** `const throw()`

  Für die Klasse `ios_base::failure` sind diese Memberfunktionen auch in der ANSI-Norm vorgesehen.

Standard-Exception-Klassen - einfaches Demonstrationsprogramm (1)

- **Modul mit exception-werfender Funktion (C++-Quelldatei `testbibexcept_func.cpp`)**

```
#include <iostream>
#include <fstream>
#include <exception>
#include <stdexcept>
#include <typeinfo>
#include <new>
using namespace std;


class Base
{ public : virtual ~Base() {}; /* ... */ };

class Derived : public Base
{ public : virtual ~Derived() {}; /* ... */ };

class Der2 : public Base
{ public : virtual ~Der2() {}; /* ... */ };


void func1(int i)
{ switch(i)
  { case 0 : cout << "\ncase 0 : ";
             throw(exception());
             break;
    case 1 : cout << "\ncase 1 : ";
             throw(bad_exception());
             break;
    case 2 : cout << "\ncase 2 : ";
             { Base* bp=new Der2;
               dynamic_cast<Derived&>(*bp);    // throw(bad_cast());
             }
             break;
    case 3 : cout << "\ncase 3 : ";
             { Base* bp=NULL;
               typeid(*bp);                   // throw(bad_typeid());
             }
             break;
    case 4 : cout << "\ncase 4 : ";
             new char[0x7fffffff];            // throw(bad_alloc());
             break;
    case 5 : cout << "\ncase 5 : ";
             { ifstream istrm;
               istrm.exceptions(ios::eofbit | ios::failbit | ios::badbit);
               istrm.setstate(ios::eofbit);   // throw(ios_base::failure());
             }
             break;
    case 6 : cout << "\ncase 6 : ";
             throw(ios_base::failure("mein IO-Fehler"));
             break;
    case 7 : cout << "\ncase 7 : ";
             throw(logic_error("Exception wg. logischem Fehler"));
             break;
    case 8 : cout << "\ncase 8 : ";
             throw(runtime_error("Exception wg. Laufzeit-Fehler"));
             break;
    case 9 : cout << "\ncase 9 : ";
             throw(range_error("Exception wg. Range Error"));
             break;
    default: cout << "\ndefault: ";
             throw("Keine Bibliotheks-Exception");
             break;
  }
}
```

Standard-Exception-Klassen - einfaches Demonstrationsprogramm (2)

- **Modul mit exception-fangender `main()`-Funktion (C++-Quelldatei `testbibexcept_m.cpp`)**

```
#include <iostream>
#include <exception>
#include <new.h>       // Visual-C++-spezifisch, nur fuer _set_new_handler(_PNH)
using namespace std;

#define MAX_NR 10

extern void func1(int);

int mynewhdlr(size_t)
{ throw(bad_alloc());
}


int main(void)
{ _set_new_handler(mynewhdlr);      // Visual-C++-spezifische Funktion
  for (int i=0; i<=MAX_NR; i++)
  { try
    { func1(i);
      cout << "func(" << i << ") normal beendet" << endl;
    }
    catch(const exception& e)
    { cout << e.what() << endl;
    }
    catch(const char* cpe)
    { cout << cpe << endl;
    }
  }
  cout << "\nWeiter gehts !\n";
  return 0;
}
```

- **Ausgabe des Programms**

```
case 0 : Unknown exception

case 1 : bad exception

case 2 : Bad dynamic_cast!

case 3 : Attempted a typeid of NULL pointer!

case 4 : bad allocation

case 5 : ios::eofbit set

case 6 : mein IO-Fehler

case 7 : Exception wg. logischem Fehler

case 8 : Exception wg. Laufzeit-Fehler

case 9 : Exception wg. Range Error

default: Keine Bibliotheks-Exception

Weiter gehts !
```

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `auto_ptr` (1)

- **Eigenschaften**

  ◇ Das in der **Utility-Bibliothek** enthaltene Klassen-Template **`auto_ptr`** ist die Standard-Bibliotheks-Implementierung von **Smart-Pointern**.
  Smart-Pointer sind Objekte, die auf andere Objekte verweisen und die wie Pointer verwendet werden können, gegenüber diesen aber zusätzliche Eigenschaften besitzen.

  ◇ `auto_ptr`-Objekte enthalten einen **Pointer** auf das durch sie jeweils **referierte ("verwaltete") Objekt**.
  Das **referierte Objekt** muß **dynamisch** (mit `new`) **alloziert** worden sein.
  `auto_ptr`-Objekte dagegen werden meist statisch alloziert.
  Beim **Zerstören** eines **`auto_ptr`-Objekts** (z.B. beim Verlassen seines Gültigkeitsbereichs) wird das von ihm referierte Objekt **automatisch zerstört** (mit `delete`).

  ◇ Für `auto_ptr`-Objekte gilt eine **strikte Besitzsemantik** (*semantics of strict ownership*) : Ein von `auto_ptr`-Objekten verwaltetes Objekt darf immer nur von **einem einzigen** `auto_ptr`-Objekt referiert werden. Dieses referierende Objekt "**besitzt**" das referierte Objekt.
  Dies erfordert eine **andere Kopiersemantik** als die, die für "normale" Pointer gilt : Beim Kopieren zweier `auto_ptr`-Objekte (Copy-Konstruktor, Zuweisungsoperator) geht der "Besitz" des referierten Objekts vom Quell-`auto_ptr` zum Ziel-`auto_ptr` über. Der Quell-`auto_ptr` zeigt danach auf nichts (enthält den `NULL`-Pointer).
  → **zerstörendes Kopieren**.
  Wegen dieser Änderung des Quell-`auto_ptr`-Objekts darf es sich bei diesem nicht um ein konstantes Objekt handeln.
  Wird ein Objekt von mehr als einem `auto_ptr`-Objekt "besessen", ergibt sich ein undefiniertes Verhalten.

  ◇ Copy-Konstruktor und Zuweisungsoperator sind so überladen, dass auch das **Kopieren von `auto_ptr`-Objekten** ermöglicht wird ,deren **referierter Objekt-Typ nicht identisch** ist. Es muß lediglich eine implizite Konvertierung zwischen den normalen Pointern auf die beteiligten Typen möglich sein.
  Damit wird die **Polymorphie zwischen Pointern** (Pointer auf Basisklasse – Pointer auf abgeleitete Klasse) auch **auf `auto_ptr`-Objekte übertragen**.

  ◇ **Weitere Funktionalitäten**, die von Smart-Pointern sinnvoll ausgeführt werden können, sind **nicht implementiert**.

- **Schnittstelle**

  ◇ Das Klasen-Template `auto_ptr` ist in der Headerdatei **`<memory>`** definiert.

  ◇ In der ANSI/ISO-Norm ist folgende **Schnittstelle** festgelegt :

```
template <class X> class auto_ptr
{   template <class Y> struct auto_ptr_ref { /* ... */ }; // Hilfsklasse
    // ... private Komponenten, z.B. X* ptr
  public :
    typedef X element_type;
    explicit auto_ptr(X* p = 0) throw();                  // Konstruktor
    ~auto_ptr() throw();

    auto_ptr(auto_ptr&) throw();                          // Copy-Konstruktor
    template <class Y> auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr&) throw();               // Zuweisungsoperator
    template <class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();

    X& operator*() const throw();
    X* operator->() const throw();
    X* get() const throw();          // Rückgabe des Pointers auf ref. Objekt
    X* release() throw();            // Freigabe des referierten Objekts
    void reset(X* p = 0) throw();    // Änderung des referierten Objekts

    auto_ptr(auto_ptr_ref<X>) throw();    // Konstruktor zur Typkonvertierung
    template <class Y> operator auto_ptr_ref<Y>() throw();
    template <class Y> operator auto_ptr<Y>() throw();    // Typkonvertierung
};
```

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `auto_ptr` (2)

- **Kurzbeschreibung der Memberfunktionen, 1. Teil**

Anmerkungen :

- ▻ Alle Memberfunktionen von `auto_ptr` werfen keine Exceptions (Exception-Deklaration `throw()`)
- ▻ Der Datentyp `X` ist formaler Template-Parameter

◇ **`auto_ptr(X* p = 0)`** (Konstruktor)

- ▻ Initialisiert ein neues `auto_ptr`-Objekt mit dem Pointer `p` (Default `p = NULL`-Pointer)
  → das `auto_ptr`-Objekt besitzt das durch `p` referierte Objekt
- ▻ `p` muß auf ein dynamisch erzeugtes Objekt zeigen (oder der `NULL`-Pointer sein)

◇ **`~auto_ptr()`** (Destruktor)

- ▻ Falls das aktuelle `auto_ptr`-Objekt ein Objekt referiert,wird dieses mittels `delete` zerstört

---

◇ **`auto_ptr(auto_ptr& ap)`** (Copy-Konstruktor)

- ▻ Initialisiert ein neues `auto_ptr`-Objekt mit dem bisher von `ap` gehaltenen Pointer.
  → das bisher von `ap` besessene Objekt geht in den Besitz des neuen `auto_ptr`-Objektes über
- ▻ `ap` verweist danach auf kein Objekt mehr

◇ **`template <class Y> auto_ptr(auto_ptr<Y>& ap)`** (überladener Copy-Konstruktor)

- ▻ Initialisiert ein neues `auto_ptr`-Objekt mit dem bisher in `ap` enthaltenen Pointer.
  → das bisher von `ap` besessene Objekt geht in den Besitz des neuen `auto_ptr`-Objektes über
- ▻ `ap` verweist danach auf kein Objekt mehr
- ▻ Ermöglicht das Kopieren von `auto_ptr`-Objekten, die Objekte unterschiedlichen Typs referieren.
- ▻ `Y*` muss implizit in `X*` konvertierbar sein.

◇ **`auto_ptr& operator=(auto_ptr& ap)`** (Zuweisungsoperator)

- ▻ Falls das aktuelle `auto_ptr`-Objekt ein Objekt referiert,wird dieses mittels `delete` zerstört
- ▻ Weist dem aktuellen `auto_ptr`-Objekt den bisher von `ap` gehaltenen Pointer zu
  → das bisher von `ap` besessene Objekt geht in den Besitz des aktuellen `auto_ptr`-Objektes über
- ▻ `ap` verweist danach auf kein Objekt mehr

◇ **`template <class Y> auto_ptr& operator=(auto_ptr<Y>& ap)`** (überladener Zuweisungsoperator)

- ▻ Falls das aktuelle `auto_ptr`-Objekt ein Objekt referiert, wird dieses mittels `delete` zerstört
- ▻ Weist dem aktuellen `auto_ptr`-Objekt den bisher von `ap` gehaltenen Pointer zu
  → das bisher von `ap` besessene Objekt geht in den Besitz des aktuellen `auto_ptr`-Objektes über
- ▻ `ap` verweist danach auf kein Objekt mehr
- ▻ Ermöglicht das Kopieren von `auto_ptr`-Objekten, die Objekte unterschiedlichen Typs referieren.
- ▻ `Y*` muss implizit in `X*` konvertierbar sein.

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `auto_ptr` (3)

- **Kurzbeschreibung der Memberfunktionen, 2. Teil**

Anmerkungen :

- ▻ Alle Memberfunktionen von `auto_ptr` werfen keine Exceptions (Exception-Deklaration `throw()`)
- ▻ Der Datentyp `X` ist formaler Template-Parameter

◇ **`X& operator*() const`** (Dereferenzierungs-Operator)

- ▻ Gibt eine Referenz auf das vom aktuellen `auto_ptr`-Objekt besessene Objekt zurück
- ▻ Erfordert, dass das aktuelle `auto_ptr`-Objekt auch tatsächlich ein Objekt referiert

---

◇ **`X* operator->() const`** (Delegations-Operator)

- ▻ Gibt den vom aktuellen `auto_ptr`-Objekt gehaltenen Pointer zurück
  (Pointer auf besessenes Objekt bzw `NULL`-Pointer)

◇ **`X* get() const`**

- ▻ Gibt den vom aktuellen `auto_ptr`-Objekt gehaltenen Pointer zurück
  (Pointer auf besessenes Objekt bzw `NULL`-Pointer)

◇ **`X* release()`**

- ▻ Gibt den Besitz am bisher referierten Objekt frei
  → der bisher gehaltene Pointer wird durch den `NULL`-Pointer ersetzt.
- ▻ Das bisher referierte Objekt wird nicht zerstört.
- ▻ Gibt den vom aktuellen `auto_ptr`-Objekt bisher gehaltenen Pointer zurück

---

◇ **`void reset(X* p = 0)`**

- ▻ Falls der vom aktuellen `auto_ptr`-Objekt bisher gehaltene Pointer `!= p` ist, wird `p` zum neuen gehaltenen Pointer
  → das aktuelle `auto_ptr`-Objekt ändert seinen Besitz vom bisher referierten Objekt in das durch `p` referierte Objekt
- ▻ Das vom bisher gehaltenen Pointer referierte Objekt (das bisher besessene Objekt) wird zerstört
- ▻ `p` muß auf ein dynamisch erzeugtes Objekt zeigen (oder der `NULL`-Pointer sein)

◇ **`template <class Y> operator auto_ptr<Y>()`** (Typkonvertierung)

- ▻ Erzeugt ein `auto_ptr`-Objekt, das das vom aktuellen `auto_ptr`-Objekt bisher besessene Objekt referiert.
- ▻ Das aktuelle `auto_ptr`-Objekt gibt das bisher besessene Objekt frei
  → der bisher gehaltene Pointer wird durch den `NULL`-Pointer ersetzt.
- ▻ Das aktuelle und das neu erzeugte `auto_ptr`-Objekt können Objekte unterschiedlichen Typs (`X` bzw `Y`) referieren.
  `X*` muß aber in `Y*` konvertierbar sein. (Übertragung der Pointer-Polymorphie auf `auto_ptr`-Objekte)

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `auto_ptr` (4)

- **Abweichende Implementierungen**

  ◇ In einigen Systemen (z.B. auch **Visual-C++**) ist das Konzept der **strikten Besitzsemantik abweichend** vom ANSI/ISO-Standard **implementiert** :

  ▻ bei der **Aufgabe** (`release()`) bzw **Übergabe** (Copy-Konstruktor, Zuweisung) des **Besitzes** an einem referierten Objekt wird der gehaltene **Pointer beibehalten** und nicht durch den `NULL`-Pointer ersetzt.
  → Das bisherige besitzende Objekt zeigt weiterhin auf das Objekt, das ihm nicht mehr gehört

  ▻ In einer weiteren Datenkomponente ist vermerkt, ob durch den gehaltenen Pointer ein besessenes Objekt referiert wird.
  Dadurch wird sichergestellt, dass ein referiertes Objekt tatsächlich immer nur von einem einzigen `auto_ptr`-Objekt zerstört werden kann.

  ◇ Weiterhin **fehlen** in einigen Implementierungen (z.B auch in **Visual-C++**)

  ▻ die innere Hilfsklasse `template <class Y> struct auto_ptr_ref`

  ▻ sowie die Memberfunktionen :

  - `X* reset()`
  - `auto_ptr(auto_ptr_ref<X>)`
  - `template <class Y> operator auto_ptr_ref<Y>()`
  - `template <class Y> operator auto_ptr<Y>()`
  - `template <class Y> auto_ptr(auto_ptr<Y>&)`
  - `template <class Y> auto_ptr& operator=(auto_ptr<Y>&)`

    ⇒ Übertragung der zwischen Pointern bestehenden Polymorphie auf `auto_ptr`-Objekte ist nicht implementiert

Einfaches Demonstrationsprogramm 1 zum Klassen-Template `auto_ptr`

- **C++-Quelldatei `demobibap1_m.cpp`**

```
// Demonstrationsprogramm 1 zum Bibliotheks-Klassen-Template auto_ptr

#include <memory>
#include <iostream>
#include <exception>
using namespace std;


class Integer
{ public :
    Integer(int i = 0) : m_iDat(i)
      { cout << "\nKonstruktor Integer(" << m_iDat << ")\n"; }
    ~Integer() { cout << "\nDestruktor Integer(" << m_iDat << ")\n"; }
    void set(int i) { m_iDat=i; }
    int get() const { return m_iDat; }
  private :
    int m_iDat;
};

ostream& operator<<(ostream& out, const Integer& iv)
{ out << iv.get() ; return out; }


void func1()
{  try
  { auto_ptr<Integer> api(new Integer(25));
    Integer * pi = new Integer(4);
    cout << "\nWert von *api in func1 : " << *api;
    cout << "\nWert von  *pi in func1 : " << *pi << endl;
    throw(*new exception);
    delete pi;
  }
  catch (exception& e)
  { cout << endl << e.what() << endl;
  }
}


int main(void)
{ func1();
  cout << "\nmain() beendet\n";
  return 0;
}
```

- **Ausgabe des Programms `demobibap1`**

```
Konstruktor Integer(25)

Konstruktor Integer(4)

Wert von *api in func1 : 25
Wert von  *pi in func1 : 4

Destruktor Integer(25)

Unknown exception

main() beendet
```

Einfaches Demonstrationsprogramm 2 zum Klassen-Template `auto_ptr` (1)

- **C++-Quelldatei `demobibap2_m.cpp`**

```
// Demonstrationsprogramm 2 zum Bibliotheks-Klassen-Template auto_ptr

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


class Integer
{ public :
    Integer(int i = 0) : m_iDat(i)
      { cout << "\nKonstruktor Integer(" << m_iDat << ")\n"; }
    ~Integer() { cout << "\nDestruktor Integer(" << m_iDat << ")\n"; }
    void set(int i) { m_iDat=i; }
    int get() const { return m_iDat; }
  private :
    int m_iDat;
};


ostream& operator<<(ostream& out, const Integer& iv)
{ out << iv.get() ; return out; }


template <class T>
void refObjOut(ostream& out, const auto_ptr<T>& ap)
{
  if (ap.get()==NULL)
    out << "NULL";
  else
    out << *ap;
}


void func2(void)
{
  auto_ptr<Integer> api1(new Integer(2003));
  auto_ptr<Integer> api2(api1);
  auto_ptr<Integer> api3(new Integer(6));
  auto_ptr<Integer> api4(new Integer(2));

  cout << "\nWert von api1 in func2 : "; refObjOut(cout, api1);
  cout << "\nWert von api2 in func2 : "; refObjOut(cout, api2);
  cout << "\nWert von api3 in func2 : "; refObjOut(cout, api3);
  cout << "\nWert von api4 in func2 : "; refObjOut(cout, api4); cout << endl;
  api3=api2;
  cout << "\nnach Zuweisung api3=api2 :";
  cout << "\nWert von api2 in func2 : "; refObjOut(cout, api2);
  cout << "\nWert von api3 in func2 : "; refObjOut(cout, api3); cout << endl;
  api3.release();
  cout << "\nnach api3.release() :";
  cout << "\nWert von api2 in func2 : "; refObjOut(cout, api2);
  cout << "\nWert von api3 in func2 : "; refObjOut(cout, api3); cout << endl;
}

int main(void)
{
  func2();
  return 0;
}
```

Einfaches Demonstrationsprogramm 2 zum Klassen-Template `auto_ptr` (2)

- **Ausgabe des Programms `demobibap2` (übersetzt mit Visual-C++ 6.0)**

```
Konstruktor Integer(2003)

Konstruktor Integer(6)

Konstruktor Integer(2)

Wert von api1 in func2 : 2003
Wert von api2 in func2 : 2003
Wert von api3 in func2 : 6
Wert von api4 in func2 : 2

Destruktor Integer(6)

nach Zuweisung api3=api2 :
Wert von api2 in func2 : 2003
Wert von api3 in func2 : 2003

nach api3.release() :
Wert von api2 in func2 : 2003
Wert von api3 in func2 : 2003

Destruktor Integer(2)
```

- **Ausgabe des Programms `demobibap2` (übersetzt mit Borland-C++ 5.5)**

```
Konstruktor Integer(2003)

Konstruktor Integer(6)

Konstruktor Integer(2)

Wert von api1 in func2 : NULL
Wert von api2 in func2 : 2003
Wert von api3 in func2 : 6
Wert von api4 in func2 : 2

Destruktor Integer(6)

nach Zuweisung api3=api2 :
Wert von api2 in func2 : NULL
Wert von api3 in func2 : 2003

nach api3.release() :
Wert von api2 in func2 : NULL
Wert von api3 in func2 : NULL

Destruktor Integer(2)
```

Einfaches Demonstrationsprogramm 3 zum Klassen-Template `auto_ptr` (1)

- **C++-Quelldatei `demobibap3_m.cpp`**

```
// Demonstrationsprogramm 3 zum Bibliotheks-Klassen-Template auto_ptr
// hier : Demo für Kopieren von auto_ptr-Objekten für unterschiedliche Typen der
//        referierten Objekte
//        sowie Aufruf der Memberfunktion reset()
//        läuft nicht unter Visual-C++

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


class Zahl
{ public :
    virtual ~Zahl() { cout << "Destruktor Zahl\n"; }
    virtual const char* className() const { return "Zahl"; }
    virtual void output(ostream&) const = 0;
  protected :
    Zahl() { }
};


class Integer : public Zahl
{ public :
    Integer(int i = 0) : m_iDat(i)
      { cout << "\nKonstruktor Integer(" << m_iDat << ")\n"; }
    ~Integer() { cout << "\nDestruktor Integer(" << m_iDat << ")\n"; }
    void output(ostream& out) const { out << m_iDat; }
    const char* className() const { return "Integer"; }
    void set(int i) { m_iDat=i; }
    int get() const { return m_iDat; }
  private :
    int m_iDat;
};

ostream& operator<<(ostream& out, const Zahl& iv)
{ iv.output(out) ; return out; }


template <class T>
void refObjOut(ostream& out, const auto_ptr<T>& ap)
{
  if (ap.get()==NULL)
    out << "NULL";
  else
    out << *ap << " (ref. Objekt : " << ap->className() << ')';
}


int main(void)
{
  auto_ptr<Zahl> apz;
  auto_ptr<Integer> api(new Integer(2003));
  apz=api;
  cout << "\nWert von api : "; refObjOut(cout, api); cout << endl;
  cout << "\nWert von apz : "; refObjOut(cout, apz); cout << endl;
  apz.reset(new Integer(30));
  cout << "\nWert von apz : "; refObjOut(cout, apz); cout << endl;
  return 0;
}
```

Einfaches Demonstrationsprogramm 3 zum Klassen-Template `auto_ptr` (2)

- **Ausgabe des Programms `demobibap3` (übersetzt mit Borland-C++ 5.5)**

```
Konstruktor Integer(2003)

Wert von api : NULL

Wert von apz : 2003 (ref. Objekt : Integer)

Konstruktor Integer(30)

Destruktor Integer(2003)
Destruktor Zahl

Wert von apz : 30 (ref. Objekt : Integer)

Destruktor Integer(30)
Destruktor Zahl
```

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `pair` (1)

- **Definition des Klassen-Templates `pair`**

  ◇ An einigen Stellen der Standardbibliothek werden **Wertepaare** verwendet.
  Beispielsweise verwalten die in der STL definierten Container-Klassen **`Map`** und **`Multimap`** Mengen, deren Elemente Wertepaare (Schlüssel/Wert) sind.

  ◇ Zur Unterstützung der Arbeit mit **Wertepaaren** ist in der **Utility-Bibliothek** als generischer Datentyp das – als **`struct`** realisierte – **Klassen-Template `pair`** definiert.

  ◇ Die **Definition** befindet sich in der Headerdatei **`<utility>`** :

```
template <class T1, class T2>
struct pair
{
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;                          // erster Wert des Paares
  T2 second;                         // zweiter Wert des Paares
  pair();                            // Default-Konstruktor
  pair(const T1& x, const T2& y);    // Konstruktor mit Initialisierungswerten
  template <class U, class V>
    pair(const pair<U, V>& p);       // Copy-Konstruktor
};
```

  ◇ Die **beiden Werte** eines Wertepaares sind durch zwei entsprechende **Datenkomponenten** realisiert.
  Diese sind **öffentlich** zugänglich (`public` ist Default bei `struct`), zu ihnen kann also direkt zugegriffen werden.

  ◇ Drei Konstruktoren bilden die einzigen Memberfunktionen.

- **Beschreibung der Konstruktoren**

  ◇ **`pair();`** (Default-Konstruktor)
  - ▻ Initialisierung der beiden Datenkomponenten durch Aufruf des Default-Konstruktors ihres jeweiligen Typs
  - ▻ typische Implementierung innerhalb der Klassendefinition :

```
pair() : first(T1()), second(T2()) {}
```

  ◇ **`pair(const T1& x, const T2& y);`** (Konstruktor mit Initialisierungswerten)
  - ▻ Initialisierung der beiden Datenkomponenten mit den übergebenen Parametern `x` und `y`
  - ▻ typische Implementierung innerhalb der Klassendefinition :

```
pair(const T1& x, const T2& y) : first(x), second(y) {}
```

  ◇ **`template <class U, class V> pair(const pair<U, V>& p);`** (Copy-Konstruktor)
  - ▻ Initialisierung der beiden Datenkomponenten mit den entsprechenden Datenkomponenten des Parameters `p`, wobei gegebenenfalls implizite Typkonvertierungen durchgeführt werden
  - ▻ typische Implementierung innerhalb der Klassendefinition :

```
template<class U, class V> pair(const pair<U, V> &p)
: first(p.first), second(p.second) {}
```

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `pair` (2)

- **Zusätzlich definierte freie Funktionen :**

  ◇ Die Utility-Bibliothek enthält zusätzlich **7 freie Funktionen** zur Verwendung mit dem Klassen-Template `pair`. Typischerweise sind diese Funktionen in der Headerdatei `<utility>` als **`inline`-Funktionen** definiert. 6 dieser Funktionen sind **Vergleichsfunktionen,** die 7. dient zur **Erzeugung eines `pair`-Objektes**.

◇
```
template <class T1, class T2>
  inline bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return x.first == y.first && x.second == y.second; }
```
▻ Test zweier `pair`-Objekte auf Gleichheit

▻ Zwei `pair`-Objekte sind genau dann gleich, wenn ihre beiden entsprechenden Komponenten jeweils gleich sind

---

◇
```
template <class T1, class T2>
  inline bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return x.first < y.first || (!(y.first < x.first) && x.second < y.second); }
```
▻ Test zweier `pair`-Objekte auf "kleiner als"

▻ Für den Vergleich wird primär die erste Komponente überprüft. Wenn diese für beide Objekte gleich ist, wird das Vergleichsergebnis durch Vergleich der zweiten Komponente ermittelt.

◇
```
template <class T1, class T2>
  inline bool operator!=(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return !(x == y); }
```
▻ Test zweier `pair`-Objekte auf Ungleichheit

◇
```
template <class T1, class T2>
  inline bool operator>(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return y < x; }
```
▻ Test zweier `pair`-Objekte auf "größer als"

◇
```
template <class T1, class T2>
  inline bool operator>=(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return !(x < y); }
```
▻ Test zweier `pair`-Objekte auf "größer gleich"

◇
```
template <class T1, class T2>
  inline bool operator<=(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return !(y < x); }
```
▻ Test zweier `pair`-Objekte auf "kleiner gleich"

◇
```
template <class T1, class T2>
  inline pair<T1, T2> make_pair(const T1& x, const T2& y)
  { return pair<T1, T2>(x, y); }
```
▻ Erzeugung eines `pair`-Objektes aus zwei Komponenten ohne explizite Angabe der Komponententypen

Beispiel : `return` **`make_pair(5, 3.728);`** statt `return pair<int, double>(5, 3.728);`

ANSI/ISO-C++-Standardbibliothek : String-Bibliothek Überblick (1)

- **String-Bibliothek**
  - ◇ Zur ANSI-C++-Standardbibliothek gehörende Teilbibliothek für die **String-Unterstützung**.
  - ◇ Sie stellt ein **Klassen-Template** zur Verfügung, dessen Instanzen, die **String-Klassen**, eine sehr **komfortabl**e und **sichere** (Speicherverwaltung und Bereichskontrolle) Bearbeitung von Strings ermöglichen.
    Wesentlicher **Template-Parameter** ist der **Datentyp** zur Darstellung von **Zeichen**.
  - ◇ Strings werden also – im Unterschied zu C-Strings – als **Objekte** dargestellt und verwendet.
    Die in den Objekten enthaltene Zeichenfolge ist nicht mit einem speziellen Endzeichen (z.B. `'\0'`–Character) abgeschlossen. Vielmehr kann sie jedes Zeichen des verwendeten Zeichen-Datentyps enthalten.
    Der für die Ablage der Zeichenfolge benötigte Speicherplatz wird dem jeweiligen Bedarf entsprechend durch die Memberfunktionen des Klassen-Templates alloziert bzw freigegeben.
  - ◇ Alle für die Anwendung der String-Bibliothek notwendigen Vereinbarungen sind in der Headerdatei `<string>` enthalten. Sie liegen alle im Namensraum **`std.`**

- **Klassen-Template `basic_string`**
  - ◇ Zentrales Klassen-Template für die String-Klassen, das deren Verhalten und Fähigkeiten definiert.
  - ◇ Die aus dem Template instanzierbaren String-Klassen sind durch drei **Template-Parameter** bestimmt :
    - ▻ **`charT`** : **Datentyp** zur Darstellung der **Zeichen** (Zeichentyp)
    - ▻ **`traits`** : **Klasse** zur Beschreibung von **Eigenschaften** (Merkmalen) des **Zeichentyps** (*character traits*).
      Als **Default** ist hierfür die Instanz des ebenfalls mit dem Zeichen-Datentyp `charT` parameterisierten Klassen-Templates `char_traits`, das ist **`char_traits<charT>`**, festgelegt.
      Das Klassen-Template `char_traits` ist auch in der Header-Datei `<string>` definiert.
    - ▻ **`Allokator`** : **Allokator-Klasse**, die das Speichermodell beschreibt, das die zu verwendende dynamische Speicherverwaltung festlegt.
      Als **Default** ist die Instanz des ebenfalls mit dem Zeichen-Datentyp `charT` parameterisierten Klassen-Templates `allocator`, das ist **`allocator<charT>`**, vorgesehen.
      Das Klassen-Template `allocator` bestimmt die im System eingesetzten Standard-Allokator-Klassen, die `new` und `delete` zur Speicherallokation verwenden. Seine Definition befindet sich in der Headerdatei `<memory>`.
  - ◇ **Definition** :

```
template <class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
  class basic_string { /* ... */ };
```

- **String-Klassen `string` und `wstring`**
  - ◇ Standardmäßig sind in `<string>` zwei String-Klassen als **Instanzen** von `basic_string` vordefiniert.
  - ◇ Klasse **`string`** für den Datentyp **`char`** ("normale" Ein-Byte-Zeichen) unter Verwendung der Defaults für die anderen Parameter :

```
typedef basic_string<char> string;
```

  - ◇ Klasse **`wstring`** für den Datentyp **`wchar_t`** (Mehr-Byte-Zeichen, z.B. Unicode-Zeichen) unter Verwendung der Defaults für die anderen Parameter :

```
typedef basic_string<wchar_t> wstring;
```

ANSI/ISO-C++-Standardbibliothek : String-Bibliothek Überblick (2)

• Überblick über die Operationen mit Strings

◇ Die String-Bibliothek von C++ erlaubt es, dass mit Strings – im Gegensatz zu C-Strings – wie mit elementaren Datentypen gearbeitet werden kann.
Defacto steht mit der Bibliotheks-Klasse `string` (bzw `wstring` bzw jeder anderen Klassen-Instanz von `basic_string<>`) ein quasi-fundamentaler Datentyp zur Verfügung.

◇ Das Klassen-Template **`basic_string<>`** definiert

- ▻ grundlegende Fähigkeiten zum **Anlegen** (Initialisieren) und **Kopieren** von Strings
- ▻ zahlreiche Methoden zum **Manipulieren** und **Bearbeiten** von Strings, wie Konkatenieren, Einfügen, Löschen, Suchen, Ersetzen, Vergleichen und Teilstringbildung.
  Die meisten dieser Methoden sind mehrfach so überladen, dass sich die entsprechenden Operationen – wo sinnvoll – auch im Zusammenhang mit C-Strings und Einzelzeichen durchführen lassen.
  Für einige der Bearbeitungsoperationen existieren neben Memberfunktionen mit einem geeigneten Namen auch entsprechende – z. T. freie – Operatorfunktionen (s. unten)
- ▻ Methoden zum **Zugriff** zu den **einzelnen Zeichen** eines Strings (String-Elementen)
- ▻ Methoden zum **Umwandeln** von Strings in **C-Strings**
- ▻ Methoden zum Bereitstellen von **String-Iteratoren**
- ▻ Methoden zur **Ermittlung** und **Änderung** der **Stringlänge** und **Stringkapazität**.

◇ Weiterhin sind in der Stringbibliothek zahlreiche **freie Operatorfunktionen** für Strings definiert.
Diese ermöglichen

- ▻ die Stream-**Ein-** und **Ausgabe** von Strings
- ▻ die **Konkatenation** von Strings, C-Strings und Einzelzeichen an Strings
  sowie die Konkatenation von Strings an Strings, C-Strings und Einzelzeichen
- ▻ den **Vergleich** von Strings mit Strings, Strings mit C-Strings und C-Strings mit Strings

◇ In vielen – aber nicht allen – Funktionen, in denen die Möglichkeit zu einem Speicherzugriffsfehler besteht, können **Exceptions** geworfen werden :

- ▻ Exception der Klasse **`out_of_range`** beim Zugriff zu einer unzulässigen String-Position
- ▻ Exception der Klasse **`length_error`** beim Versuch einen String über die maximal mögliche Länge hinaus zu verlängern

Beide Exception-Klassen sind in der Standard-Bibliothek definiert. Sie sind von der Exception-Klasse `logic_error` abgeleitet. Diese wiederum ist von der Exception-Basis-Klasse `exception` abgeleitet.

• Anmerkungen zur Beschreibung der String-Funktionen

◇ Zur Vereinfachung und Erhöhung der Übersichtlichkeit werden die einzelnen String-Funktionen im folgenden nicht in der originalen Template-Form sondern in einer für die – am häufigsten verwendete – **Klassen-Instanz `string` spezialisierten Darstellung** angegeben :

- ▻ statt dem Klassen-Template `basic_string` wird `string` eingesetzt
- ▻ statt dem Template-Parameter `charT` wird der Zeichen-Datentyp `char` eingesetzt

◇ **Beispiele** :

▻ statt :
```
basic_string& append(const charT* s);
```
(Memberfunktion)
wird formuliert :
```
string& append(const char* s);
```

▻ statt :
```
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is,
               (basic_string<charT, traits, Allocator>& str);
```
wird formuliert :
```
istream& operator>>(istream& is, string& str);
```

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (1)

- **Datentypen als Klassenelemente**

  ◇ Innerhalb der Klasse `string` sind mehrere Datentypen – meist indirekt – definiert. Es sind `public`-Komponenten . Sie stehen damit zur Verwendung ausserhalb der Klasse `string` zur Verfügung.
  (Achtung : ausserhalb Qualifikation mit dem Klassennamen `string::` notwendig)

  ◇ Einer dieser Datentypen ist : **`size_type`**
  Es handelt sich um einen vorzeichenlosen ganzzahligen Datentyp, der zur Angabe von String-Positionen, Anzahl-, Längen- und Größenangaben verwendet wird.

- **`public`-Datenkomponente der Klasse `string`**

  ◇ **`static const size_type npos = -1;`**

  Definition einer Konstanten, die – je nach Anwendung – als (**Default-)Parameterwert** für "**alle Zeichen**" ("bis zum Stringende") oder zur Kennzeichnung einer **ungültigen Position** verwendet wird.
  Da der Typ der Konstante **`(size_type)`** als vorzeichenloser ganzzahliger Datentyp definiert ist, entspricht der Wert **`-1`** dem **größtmöglichen Wert** dieses Typs.

- **Konstruktoren und Destruktor der Klasse `string`**

`string();`	Erzeugung eines Leer-Strings (Länge `0`)
`string(const string& str, size_type pos=0, size_type n= npos);`	Erzeugung eines Strings, der mit `n` Zeichen ab der Position `pos` des Strings `str` initialisiert wird. Für die Default-Parameterwerte : Copy-Konstruktor
`string(const char* s);`	Erzeugung eines Strings, der mit dem C-String `s` initialisiert wird
`string(const char* s, size_type n);`	Erzeugung eines Strings, der mit den ersten `n` (höchstens aber allen) Zeichen des C-Strings `s` initialisiert wird
`string(size_type n, char c);`	Erzeugung eines Strings der Länge `n.` Alle Komponenten werden mit dem Zeichen `c` initialisiert
`~string();`	Destruktor

- **Memberfunktionen zur Ermittlung der Länge und Kapazität**

`size_type size() const;`	Ermittlung der aktuellen Stringlänge (Anzahl der Zeichen im String)
`size_type length() const;`	Ermittlung der aktuellen Stringlänge (Anzahl der Zeichen im String)
`size_type capacity() const;`	Ermittlung der maximal möglichen Stringlänge ohne Neuallokation
`size_type max_size() const;`	Ermittlung der maximalen Größe, die ein String haben kann
`bool empty() const;`	Ermittlung, ob String leer ist, wenn ja Rückgabe von `true`, sonst `false`

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (2)

- **Memberfunktionen zur Änderung der Länge und Kapazität**

`void resize(size_type n, char c);`	Falls `n<=max_size()` : Änderung der Stringlänge auf den Wert `n` bei Stringverlängerung : Auffüllen mit dem Zeichen `c` Falls `n>max_size()` : Werfen der Exception **`length_error`**
`void resize(size_type n);`	Falls `n<=max_size()` : Änderung der Stringlänge auf den Wert `n` bei Stringverlängerung : die zusätzlichen Zeichen bleiben undefiniert Falls `n>max_size()` : Werfen der Exception **`length_error`**
`void clear();`	Löschen aller Zeichen im String, neue Stringlänge = `0`
`void reserve(size_type res=0);`	Falls `res>capacity()` : Vergrößerung der Kapazitaet Falls `res<=max_size()` : neue Kapazität `>= res` Falls `res>max_size()` : Werfen der Exception **`length_error`** Falls `res<=capacity()` : keine Wirkung

- **Memberfunktionen zur Zuweisung**

Rückgabewert für alle Zuweisungsfunktionen : **Referenz** auf das **aktuelle** String-Objekt (`*this`)	
`string& operator=(const string& str);`	Zuweisung des Strings `str`
`string& operator=(const char* s);`	Zuweisung des C-Strings `s`
`string& operator=(char c);`	Zuweisung des Einzelzeichens `c` (neue Stringlänge `= 1`)
`string& assign(const string& str);`	Zuweisung des Strings `str`
`string& assign(const string& str` **`size_type pos, size_type n);`**	Falls `pos<str.size()` : Zuweisung von `n` (höchstens aber allen) Zeichen ab der Position `pos` aus dem String `str` Falls `pos==str.size()` : keine Wirkung Falls `pos>str.size()` : Werfen der Exception **`out_of_range`**
`string& assign(const char* s);`	Zuweisung des C-Strings `s`
`string& assign(const char* s, size_type n);`	Zuweisung der ersten `n` (höchstens aber alle) Zeichen des C-Strings `s`
`string& assign(size_type n, char c);`	Zuweisung eines Strings der Länge `n,` bei dem alle Zeichen gleich dem Zeichen `c` sind

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (3)

- **Memberfunktionen zum Zugriff zu Einzel-Zeichen im String (String-Elementen)**

`const char& operator[](size_type pos) const;` `char& operator[](size_type pos);`	Falls `pos<size()` : Rückgabe des Zeichens (bzw Referenz auf das Zeichen) an der Position (Index) `pos` Falls `pos>=size()` : undefiniert
`const char& at(size_type pos) const;` `char& at(size_type pos);`	Falls `pos<size()` : Rückgabe des Zeichens (bzw Referenz auf das Zeichen) an der Position (Index) `pos` Falls `pos>=size()` : Werfen der Exception **`out_of_range`**

- **Memberfunktionen zur Konvertierung von Strings in `char`-Arrays bzw C-Strings**

`size_type copy(char* s, size_type n, size_type pos = 0) const;`	Kopieren von `n` Zeichen ab der Position `pos` (höchstens jedoch bis zum Stringende) in das durch `s` referierte `char`-Array. Es wird kein `'\0'`-Character angehängt Rückgabewert : Anzahl kopierter Zeichen
`const char* data() const;`	Rückgabe eines Pointers auf ein `char`-Array, dessen Elemente mit den Zeichen des aktuellen Strings übereinstimmen. Das `char`-Array ist nicht mit dem `'\0'`-Character abgeschlossen
`const char* c_str() const;`	Rückgabe eines Pointers auf einen C-String (Abschluß mit `'\0'`-Character), dessen Zeichen mit den Zeichen des aktuellen Strings übereinstimmen.

- **Memberfunktionen zum Vergleich von Strings bzw Teilstrings**

Rückgabewert für **alle Vergleichsfunktionen** : `<0`, wenn akt. (Teil-) String `< str` (bzw `s`) `0`, wenn akt. (Teil-) String `== str` (bzw `s`) `>0`, wenn akt. (Teil-) String `> str` (bzw `s`)	
`int compare(const string& str) const;`	Vergleich aktueller String mit dem String `str`
`int compare(size_type pos, size_type n, const string& str) const;`	Vergleich `n` Zeichen des aktuellen Strings ab Position `pos` mit dem String `str`
`int compare(size_type pos1, size_type n1, const string& str, size_type pos2, size_type n2) const;`	Vergleich `n1` Zeichen des aktuellen Strings ab Position `pos1` mit `n2` Zeichen des Strings `str` ab Position `pos2`
`int compare(const char* s) const;`	Vergleich aktueller String mit dem C-String `s`
`int compare(size_type pos1, size_type n1, const char* s, size_type n2=npos) const;`	Vergleich `n1` Zeichen des aktuellen Strings ab Position `pos1` mit `n2` Zeichen (default : bis String-Ende) des C-Strings `s`

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (4)

• Memberfunktionen zum Anhängen (Konkatenation)

Für **alle Funktionen zum Anhängen** gilt :	**Rückgabewert** ist **Referenz** auf das **aktuelle** String-Objekt (`*this`) Falls die neue Länge des Strings den maximal möglichen Wert für die Stringlänge übersteigen würde, wird die Exception **`length_error`** geworfen
`string& operator+=(const string& str);`	Anhängen des Strings `str` an den aktuellen String
`string& operator+=(const char* s);`	Anhängen des C-Strings `s` an den aktuellen String
`string& operator+=(char c);`	Anhängen des Zeichens `c` an den aktuellen String
`string& append(const string& str);`	Anhängen des Strings `str` an den aktuellen String
`string& append(const string& str,` **`size_type n,`** **`size_type pos);`**	Falls `pos<=str.size()` : Anhängen von `n` Zeichen des Strings `str` ab der Position `pos` (aber maximal bis zum Stringende) an den aktuellen String. Falls `pos>str.size()` : Werfen der Exception **`out_of_range`**
`string& append(const char* s);`	Anhängen des C-Strings `s` an den aktuellen String
`string& append(const char* s,` **`size_type n);`**	Anhängen der ersten `n` (höchstens aber allen) Zeichen des C-Strings `s` an den aktuellen String
`string& append(size_type n, char c);`	Anhängen von `n`-mal das Zeichen `c` an den aktuellen String

• Memberfunktionen zum Einfügen

Für **alle Einfügefunktionen** gilt :	**Rückgabewert** ist **Referenz** auf das **aktuelle** String-Objekt (`*this`) Falls eine Positionsangabe größer als die jeweilige Stringlänge ist, wird die Exception **`out_of_range`** geworfen Falls die neue Länge des Strings den maximal möglichen Wert für die Stringlänge (`max_size()`) übersteigen würde, wird die Exception **`length_error`** geworfen
`string& insert(size_type pos,` **`const string& str);`**	Einfügen des Strings `str` in den aktuellen String ab der Position `pos`
`string& insert(size_type pos1,` **`const string& str,`** **`size_type pos2, size_type n);`**	Einfügen von maximal `n` Zeichen ab der Position `pos2` aus dem String `str` in den aktuellen String ab der Position `pos1`
`string& insert(size_type pos, const char* s);`	Einfügen des C-Strings `s` in den aktuellen String ab der Position `pos`
`string& insert(size_type pos, const char* s,` **`size_type n);`**	Einfügen der ersten `n` (maximal allen) Zeichen des C-Strings `s` in den aktuellen String ab Pos. `pos`
`string& insert(size_type pos,` **`size_type n, char c);`**	Einfügen `von n`-mal das Zeichen `c` in den aktuellen String ab der Position `pos`

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (5)

- **Memberfunktionen zum Ersetzen und Löschen**

`void swap(string& str);`	Vertauschen des Inhalts des aktuellen Strings mit dem Inhalt des Strings `str`
Für **alle** folgenden **Funktionen** gilt : **Rückgabewert** ist **Referenz** auf das **aktuelle** String-Objekt (`*this`) Falls eine Positionsangabe größer als die jeweilige Stringlänge ist, wird die Exception **`out_of_range`** geworfen Für die **Ersetzungs**-Funktionen gilt : Die Anzahl der ersetzenden Zeichen kann kleiner, größer oder gleich der Anzahl der ersetzten Zeichen sein Falls die neue Länge des Strings den maximal möglichen Wert für die Stringlänge (`max_size()`) übersteigen würde, wird die Exception **`length_error`** geworfen	
`string& replace(size_type pos, size_type n, const string& str);`	Ersetzen von max. `n` Zeichen des aktuellen Strings ab der Position `pos` durch alle Zeichen des Strings `str`
`string& replace(size_type pos1,size_type n1, const string& str, size_type pos2, size_type n2);`	Ersetzen von max. `n1` Zeichen des aktuellen Strings ab der Position `pos1` durch max. `n2` Zeichen des Strings `str` ab der Position `pos2`
`string& replace(size_type pos, size_type n, const char* s);`	Ersetzen von max. `n` Zeichen des aktuellen Strings ab der Position `pos` durch alle Zeichen des C-Strings `s`
`string& replace(size_type pos, size_type n1, const char* s, size_type n2);`	Ersetzen von max. `n1` Zeichen des aktuellen Strings ab der Position `pos` durch max. `n2` Zeichen des `C`–Strings `s`
`string& replace(size_type pos, size_type n1, size_type n2, char c);`	Ersetzen von max. `n1` Zeichen des aktuellen Strings ab der Position `pos` durch `n2`–mal das Zeichen `c`
`string& erase(size_type pos = 0, size_type n = npos);`	Löschen von max. `n` Zeichen des aktuellen Strings ab der Position `pos` Für Default-Parameterwerte : Löschen aller Zeichen

- **Memberfunktion zur Ermittlung eines Teilstrings**

`string substr(size_type pos = 0, size_type n = npos);`	Bildung und Rückgabe eines neuen Strings (`string`–Objekt), der aus max. `n` Zeichen des aktuellen Strings ab der Position `pos` besteht Für Default-Parameterwerte : Rückgabe einer Kopie des aktuellen Strings Falls `pos>size()` ist, Werfen der Exception **`out_of_range`**

- **Memberfunktion zur Ermittlung und Verwendung von Iteratoren**

  ◇ Es existieren eine Reihe von Memberfunktionen zur Ermittlung von Iteratoren für das aktuelle `string`–Objekt

  ◇ Darüberhinaus existieren für viele der o.a. String-Bearbeitungsfunktionen Versionen, die Iteratoren statt Positions-(Index-) Angaben als Parameter verwenden

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (6)

- **Memberfunktionen zum Suchen**

  ◇ Es existieren insgesamt **24 Memberfunktionen** zum Suchen in Strings

  ◇ Es kann nach **Teilstrings**, **C-Teilstrings**, **Einzelzeichen** oder **Zeichen aus einer Zeichenmenge** gesucht werden.
  Es kann auch nach dem Auftritt von Zeichen gesucht werden, die in einer **Zeichenmenge nicht enthalten** sind

  ◇ Es ist eine Suche nach dem **ersten** oder dem **letzten Auftritt** des Suchparameters möglich

  ◇ Alle Suchfunktionen sind `const`-Funktionen, d.h. sie ändern das aktuelle Objekt nicht

  ◇ Alle Suchfunktionen geben eine Wert vom Typ `size_type` zurück.
  Bei **Such-Erfolg** handelt es sich um die **Position** (Index) des (Beginns des) gefundenen Auftritts.
  Bei **Misserfolg** (Suchparameter ist im aktuellen Objekt nicht vorhanden) wird **`npos`** als Kenzeichnung einer **ungültigen Position** zurückgegeben.

  ◇ Es existieren die folgenden Funktionsgruppen :

▻ **`size_type find(...) const`**	Suchen nach **erstem** Auftritt
▻ **`size_type rfind(...) const`**	Suchen nach **letztem** Auftritt
▻ **`size_type find_first_of(...) const`**	Suchen nach **erstem** Auftritt eines Zeichens aus einer **Zeichenmenge**
▻ **`size_type find_last_of(...) const`**	Suchen nach **letztem** Auftritt eines Zeichens aus einer **Zeichenmenge**
▻ **`size_type find_first_not_of(...) const`**	Suchen nach **erstem** Auftritt eines Zeichens, das in einer **Zeichenmenge nicht** enthalten ist
▻ **`size_type find_last_not_of(...) const`**	Suchen nach **letztem** Auftritt eines Zeichens, das in einer **Zeichenmenge nicht** enthalten ist

  ◇ **Einige ausgewählte Suchfunktionen** :

`size_type find(const string& str, size_type pos = 0) const;`	Suchen nach erstem Auftritt des Strings `str` ab der Position `pos` im aktuellen String
`size_type find(const char* s, size_type pos = 0) const;`	Suchen nach erstem Auftritt des C-Strings `s` ab der Position `pos` im aktuellen String
`size_type find(char c, size_type pos = 0) const;`	Suchen nach erstem Auftritt des Zeichens `c` ab der Position `pos` im aktuellen String
`size_type find(const char* s, size_type pos, size_type n) const;`	Suchen nach erstem Auftritt der ersten `n` Zeichen des C-Strings `s` ab der Position `pos` im aktuellen String
`size_type rfind(const string& str, size_type pos = 0) const;`	Suchen nach letztem Auftritt des Strings `str` ab der Position `pos` im aktuellen String
`size_type find_first_of(const string& str, size_type pos = 0) const;`	Suchen nach erstem Auftritt eines der im String `str` enthaltenen Zeichen ab der Position `pos` im aktuellen String
`size_type find_last_not_of(const string& str, size_type pos = 0) const;`	Suchen nach letztem Auftritt eines der im String `str` nicht enthaltenen Zeichen ab der Position `pos` im aktuellen String

ANSI/ISO-C++-Standardbibliothek : Freie Stringbearbeitungsfunktionen (1)

• Freie Operatorfunktionen zur String-Konkatenation

Für **alle Funktionen** gilt : Sie erzeugen einen **neuen String** (`string`-Objekt), den sie als **Funktionswert** zurückgeben
In ihrem Verhalten werden sie auf den Aufruf der Memberfunktion `append()` zurückgeführt
Das bedeutet, dass die Exception **`length_error`** geworfen wird, falls die Länge des neu zu erzeugenden Strings den maximal möglichen Wert für die Stringlänge übersteigt.

`string operator+(const string& str1, const string& str2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des Strings `str1` mit dem String `str2` besteht
`string operator+(const char* s1, const string& str2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des umgewandelten C-Strings `s1` mit dem String `str2` besteht
`string operator+(const string& str1, const char* s2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des Strings `str1` mit dem umgewandelten C-String `s2` besteht
`string operator+(char c1, const string& str2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des umgewandelten Zeichens `c1` mit dem String `str2` besteht
`string operator+(const string& str1, char c2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des Strings `str1` mit dem umgewandelten Zeichen `c2` besteht

• Freie Operatorfunktionen zum String-Vergleich

Für die **folgenden Funktionen** gilt : Das Symbol **`op`** steht für einen der Vergleichsoperatoren **`== != < <= > >=`**
Rückgabewert : `true`, falls der Vergleich erfüllt ist,
`false`, falls der Vergleich nicht erfüllt ist

`bool operator op(const string& str1, const string& str2);`	Vergleich des Strings `str1` mit dem String `str2`
`bool operator op(const char* s1, const string& str2);`	Vergleich des C-Strings `s1` mit dem String `str2`
`bool operator op(const string& str1, const char* s2);`	Vergleich des Strings `str1` mit dem C-String `s2`

• Freie Funktion zum Vertauschen zweier Strings

`void swap(string& str1, string& str2);`	Vertauschen des Inhalts Strings `str1` mit dem Inhalt des Strings `str2` Wirkungsgleich mit : **`str1.swap(str2);`**

ANSI/ISO-C++-Standardbibliothek : Freie Stringbearbeitungsfunktionen (2)

- **Freie Operatorfunktionen zur Stream-Ein-/Ausgabe**

istream& operator>>(istream& is, string& str);	Einlesen der nächsten Zeichen aus dem durch `is` referierten Eingabestream und Ablage dieser im String `str`. Falls eine Feldbreite mit `is.width(...)` explizit gesetzt worden ist, werden maximal `is.width()` Zeichen eingelesen, andernfalls werden maximal `str.max_size()` Zeichen eingelesen. Das Einlesen wird früher beendet, wenn das nächste einzulesende Zeichen ein *White-Space-Character* ist oder das Dateiende erreicht ist Rückgabewert : `is`
ostream& operator<<(ostream& is, string& str);	Ausgabe des Strings `str` in den durch `os` referierten Ausgabestream Rückgabewert : `os`

- **Freie Funktionen zum Einlesen von Textzeilen aus Eingabe-Streams**

istream& getline(istream& is, string& str, char delim);	Einlesen einer mit dem Zeichen `delim` abgeschlossenen Zeichenfolge aus dem durch `is` referierten Eingabestream und Ablage dieser im String `str`. Das Zeichen `delim` wird aus dem Stream entfernt aber nicht mit im String abgelegt. Das Einlesen wird vor Auftritt des Zeichens `delim` beendet, wenn bereits `str.max_size()` Zeichen gelesen wurden oder das Dateiende erreicht ist
istream& getline(istream& is, string& str);	Einlesen einer mit dem Zeichen `'\n'` abgeschlossenen Zeichenfolge aus dem durch `is` referierten Eingabestream und Ablage dieser im String `str`. Das Zeichen `'\n'` wird aus dem Stream entfernt aber nicht mit im String abgelegt. Das Einlesen wird vor Auftritt des Zeichens `'\n'` beendet, wenn bereits `str.max_size()` Zeichen gelesen wurden oder das Dateiende erreicht ist Der Aufruf dieser Funktion entspricht dem Aufruf von `getline(is, str, '\n');`

Demo-Programm 1 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (1)

- **C++-Quelldatei `stringdem_1.cpp` (`main()`-Funktion des Programms `stringdem1`)**

```
// C++-Quelldatei stringdem1_m.cpp
// Demonstrationsprogramm stringdem1 zur Bibliotheks-Klasse string

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


void stringInfo(ostream& out, const string& s)
{ out << s;
  out << "\nsize     : " << s.size();
  out << "\ncapacity : " << s.capacity();
  out << "\nmax_size : " << s.max_size();
}


int main()
{
  cout << "\nWert von npos : " << hex << string::npos << dec << endl;

  string s1;         // Leer-String
  cout << "\ns1 Leerstring";
  stringInfo(cout, s1);
  cout << endl;

  char* cp="Hallo !";
  string s2(cp);
  cout << "\ns2 aus C-String : ";
  stringInfo(cout, s2);
  cout << endl;

  string s3(s2);
  cout << "\ns3 mit Copy-Konstruktor : ";
  stringInfo(cout, s3);
  cout << endl;

  string s4(3, 'Z');
  cout << "\ns4 aus Einzelzeichen : ";
  stringInfo(cout, s4);
  cout << endl;

  string::size_type nLen=3;
  s2.resize(nLen);
  cout << "\ns2 nach resize(" << nLen << ") : ";
  stringInfo(cout, s2);
  cout << endl;

  nLen=10;
  s2.resize(nLen, 'o');
  cout << "\ns2 nach resize(" << nLen << ") mit Fuellzeichen : ";
  stringInfo(cout, s2);
  cout << endl;

  nLen=80;
  s2.reserve(nLen);
  cout << "\ns2 nach reserve(" << nLen << ") : ";
  stringInfo(cout, s2);
  cout << endl;

  return 0;
}
```

Demo-Programm 1 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (2)

- **Ausgabe des Programms `stringdem1`**

```
Wert von npos : ffffffff

s1 Leerstring
size     : 0
capacity : 0
max_size : 4294967293

s2 aus C-String : Hallo !
size     : 7
capacity : 31
max_size : 4294967293

s3 mit Copy-Konstruktor : Hallo !
size     : 7
capacity : 31
max_size : 4294967293

s4 aus Einzelzeichen : ZZZ
size     : 3
capacity : 31
max_size : 4294967293

s2 nach resize(3) : Hal
size     : 3
capacity : 31
max_size : 4294967293

s2 nach resize(10) mit Fuellzeichen : Haloooooo
size     : 10
capacity : 31
max_size : 4294967293

s2 nach reserve(80) : Haloooooo
size     : 10
capacity : 95
max_size : 4294967293
```

Demo-Programm 2 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string`

- **C++-Quelldatei `stringdem2_m.cpp` (`main()`-Funktion des Programms `stringdem2`)**

```
// C++-Quelldatei stringdem2_m.cpp
// Demonstrationsprogramm stringdem2 zur Bibliotheks-Klasse string

#include <string>
#include <iostream>
#include <exception>
#include <iomanip>
using namespace std;

int main()
{ string s1;
  string s2("Blaer bricht das Voelkerrecht");
  string s3, s4;
  try
  { s1="Busch";
    s4=s1;
    s1.append(" Monkey");
    cout << endl << s1.c_str() << endl;    // s1.c_str() statt s1 nur zu Demo-Zweck
    s3.assign(s2, 5, string::npos);
    cout << endl << s3 << endl;
    s4+=s3;
    cout << endl << s4 << endl;
    cout << endl << endl;
    for (unsigned i=6; i<20; i++)
      cout << hex << setw(2) << setfill('0') << (s1[i]&0xff) << ' ';
    cout << dec << setfill(' ') << endl;
    for (i=6; i<20; i++)
      cout << hex << setw(2) << setfill('0') << (s1.at(i)&0xff) << ' ';
    cout << dec << setfill(' ') << endl;
  }
  catch (exception& e)
  { cout << "\nexception : " << e.what() << endl;
  }
  try
  { cout << "\nreserve(" << hex << string::npos << dec << ") : ";
    s4.reserve(string::npos);
  }
  catch (exception& e)
  { cout << "\nexception : " << e.what() << endl;
  }
  return 0;
}
```

- **Ausgabe des Programms `stringdem2`**

```
Busch Monkey

 bricht das Voelkerrecht

Busch bricht das Voelkerrecht


4d 6f 6e 6b 65 79 00 00 00 00 00 00 00 00
4d 6f 6e 6b 65 79
exception : invalid string position

reserve(ffffffff) :
exception : string too long
```

Demo-Programm 3 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (1)

- **C++-Quelldatei `stringdem_3.cpp` (`main()`-Funktion des Programms `stringdem3`)**

```
// C++-Quelldatei stringdem3_m.cpp
// Demonstrationsprogramm stringdem3 zur Bibliotheks-Klasse string

#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char** argv)
{
  int iRet=0;
  string dname;
  if (argc>=2)
    dname=argv[1];
  else
  { cout << "\nName der Textdatei ? ";
    cin >> dname;
  }

  ifstream idat(dname.c_str());
  if (!idat)
  {
    cout << "\nDatei \"" << dname << "\" kann nicht geoeffnet werden !\n";
    iRet=1;
  }
  else
  {
    string text;
    string zeile;
    string grzeil;

    while (getline(idat, zeile))
    {
      if (zeile>grzeil)
        grzeil=zeile;
      text+=zeile;
      text+='\n';
    }
    cout << "\nInhalt der eingelesenen Textdatei :\n";
    cout << endl << text << endl;
    cout << "Gesamte Textlaenge : " << text.length() << endl;
    cout << "\n\"Groesste\" Zeile :\n";
    cout << grzeil << endl;

    string wort;
    cout << "\nSuchen nach ? ";
    cin >> wort;
    string::size_type pos=text.find(wort);
    while(pos!=string::npos)
    {
      cout << "enthalten ab Pos. : " << pos << endl;
      text.insert(pos, 1, '*');
      pos+=1+wort.length();
      text.insert(pos++, 1, '*');
      pos=text.find(wort, pos+1);
    }
    cout << "\nMarkierte Textdatei :\n";
    cout << endl << text << endl;
  }
  return iRet;
}
```

Demo-Programm 3 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (2)

- **Ausgabe des Programms `stringdem3` (Beispiel-Aufruf : `stringdem3 hausverbot.txt`)**

```
Inhalt der eingelesenen Textdatei :

Hausverbot fuer Bush und Blair in Geburtskirche
US-Praesident George Bush und der britische Premierminister Tony Blair haben
lebenslanges Hausverbot fuer die Geburtskirche in Bethlehem erhalten.
Dies betonte der palaestinensische Archimandrit Attalah Hanna vom
griechisch-orthodoxen Patriarchat in Jerusalem bei "Islam-online".
Auch US-Verteidigungsminister Donald Rumsfeld und der britische
Aussenminister Jack Straw duerften das Gotteshaus niemals mehr betreten.
Gleichzeitig sprach der Archimandrit, ranghoechstes Mitglied des
griechisch-orthodoxen Patriarchats, auch von einer "Exkommunikation".
Bush und Blair haetten sich selbst aus der Kirchengemeinschaft
ausgeschlossen, weil sie die Warnungen aller christlichen Kirchen vor
einem Irak-Krieg missachtet haetten, sagte der Geistliche.
2. April 2003 (SZ)

Gesamte Textlaenge : 817

"Groesste" Zeile :
lebenslanges Hausverbot fuer die Geburtskirche in Bethlehem erhalten.

Suchen nach ? und
enthalten ab Pos. : 21
enthalten ab Pos. : 76
enthalten ab Pos. : 379
enthalten ab Pos. : 615

Markierte Textdatei :

Hausverbot fuer Bush *und* Blair in Geburtskirche
US-Praesident George Bush *und* der britische Premierminister Tony Blair haben
lebenslanges Hausverbot fuer die Geburtskirche in Bethlehem erhalten.
Dies betonte der palaestinensische Archimandrit Attalah Hanna vom
griechisch-orthodoxen Patriarchat in Jerusalem bei "Islam-online".
Auch US-Verteidigungsminister Donald Rumsfeld *und* der britische
Aussenminister Jack Straw duerften das Gotteshaus niemals mehr betreten.
Gleichzeitig sprach der Archimandrit, ranghoechstes Mitglied des
griechisch-orthodoxen Patriarchats, auch von einer "Exkommunikation".
Bush *und* Blair haetten sich selbst aus der Kirchengemeinschaft
ausgeschlossen, weil sie die Warnungen aller christlichen Kirchen vor
einem Irak-Krieg missachtet haetten, sagte der Geistliche.
2. April 2003 (SZ)
```

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (1)

• Allgemeines

◇ **Funktionsobjekte** sind Objekte, für die der Funktionsaufruf-Operator **`operator()`** definiert ist.
Derartige Objekte lassen sich **wie Funktionen verwenden**.
Gegenüber normalen Funktionen können sie einen inneren Zustand (Datenkomponenten) besitzen.
Dadurch lassen sie sich flexibler und effizienter als diese verwenden.

◇ In der **Utility-Bibliothek** sind zahlreiche Klassen für Funktionsobjekte, die diverse Standard-Operationen implementieren, definiert.
Um eine von den Operanden-Typen der jeweiligen Operation unabhängige Formulierung zu ermöglichen, sind sie als **Klassen-Templates** (mittels `struct`) implementiert.
In erster Linie sind sie für die Verwendung mit den Algorithmen der STL vorgesehen.
Sie lassen sich aber auch unabhängig davon einsetzen.

◇ U.a. stellt die Bibliothek für sämtliche in der Sprache enthaltenen **arithmetischen**, **Vergleichs-** und **logischen Operatoren** (Standard-Operationen) entsprechende Funktionsobjekt-Klassen zur Verfügung.

◇ Ihre Definitionen (und Implementierungen) sind in der **Headerdatei `<functional>`** enthalten.

• Basisklassen für Funktionsobjekte

◇ Es ist je eine Basisklasse für "**einstellige Funktionsobjekte**" und für "**zweistellige Funktionsobjekte**" definiert.
Für "einstellige Funktionsobjekte" wird die Operatorfunktion `operator()` mit einem Parameter aufgerufen,
für "zweistellige Funktionsobjekte" analog mit zwei Parametern.

◇ Durch die Basisklassen werden im wesentlichen nur standardisierte **Namen** für die **Parameter-** und **Rückgabe-Typen** definiert.
Diese Typnamen werden für die Definition der – ebenfalls in der Utility-Bibliothek implementierten – **Binder-Klassen** benötigt.

◇ **Basisklasse für "einstellige Funktionsobjekte"**

```
template <class Arg, class Result>
  struct unary_function
  {
    typedef Arg    argument_type;
    typedef Result result_type;
  }
```

◇ **Basisklasse für "zweistellige Funktionsobjekte"**

```
template <class Arg1, class Arg2, class Result>
  struct binary_function
  {
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
  }
```

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (2)

- **Funktionsobjekt-Klassen für arithmetische Operationen**

  ◇ **Prinzipielle Definition** der Klassen-Templates :

```
template <class T>
  struct plus : binary_function<T, T, T>
  {
    T operator()(const T& x, const T& y) const
    { return (x + y);
    }
  }
```

  ◇ **Überblick**

Funktionsobjekt-Klasse	Parameter von `operator()`	Ergebnis von `operator()`
plus<T>	(const T& **x**, const T& **y**)	**x + y**
minus<T>	(const T& **x**, const T& **y**)	**x - y**
multiplies<T>	(const T& **x**, const T& **y**)	**x * y**
divides<T>	(const T& **x**, const T& **y**)	**x / y**
modulus<T>	(const T& **x**, const T& **y**)	**x % y**
negate<T>	(const T& **x**)	**- x**

- **Funktionsobjekt-Klassen für Vergleichs- und logische Operationen**

  ◇ Die Operatorfunktion `operator()` dieser Klassen liefern einen Funktionswert vom Typ `bool`. Funktionsobjekte mit dieser Eigenschaft werden **Prädikate** genannt.

  ◇ **Prinzipielle Definition** der Klassen-Templates :

```
template <class T>
  struct equal_to : binary_function<T, T, bool>
  {
    bool operator()(const T& x, const T& y) const
    { return (x == y);
    }
  }
```

  ◇ **Überblick**

Funktionsobjekt-Klasse	Parameter von `operator()`	Ergebnis von `operator()`
equal_to<T>	(const T& **x**, const T& **y**)	**x == y**
not_equal_to<T>	(const T& **x**, const T& **y**)	**x != y**
greater<T>	(const T& **x**, const T& **y**)	**x > y**
less<T>	(const T& **x**, const T& **y**)	**x < y**
greater_equal<T>	(const T& **x**, const T& **y**)	**x >= y**
less_equal<T>	(const T& **x**, const T& **y**)	**x <= y**
logical_and<T>	(const T& **x**, const T& **y**)	**x && y**
logical_or<T>	(const T& **x**, const T& **y**)	**x || y**
logical_not<T>	(const T& **x**)	**! x**

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (3)

- **Einfaches Demo-Programm**

```
// C++-Quelldatei bibfodem1_m.cpp
// Modul mit main()-Funktion fuer das Programm bibfuncobjdem1
// Einfaches Demo-Programm zu Funktionsobjekten der Standard-Bibliothek

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


void testfunc();

int main(void)
{
  plus<int> iadd;        // Definition des Funktionsobjekts iadd
  int i1, i2, i3, i4;
  i1=5;
  i2=6;
  i3=10;
  i4=iadd(i1, i2);       // Aufruf operator() für Objekt iadd

  bool gleich;
  gleich=equal_to<int>()(i4, i3); // Erzeugung eines namenlosen
                                  // temporären Funktionsobjektes
                                  // Aufruf operator() fuer dieses Objekt
  cout << boolalpha ;
  cout << "\nUngleichheit von (" << i1 << "+" << i2 << ") und ";
  cout << i3 << " : " << logical_not<bool>()(gleich) << endl;
  testfunc();
  return 0;
}

template<class T, class Func>
   T binop(const T& a, const T& b, Func f)
   { return f(a, b);
   }

double quadsum(double a, double b)
{ return a*a + b*b; }

void testfunc()
{
  cout << "\nAddition   : "
       << binop(3.5, 2.7, plus<double>());     // Uebergabe Funktionsobjekt
  cout << "\nDivision   : "
       << binop(37, 7, divides<int>());        // Uebergabe Funktionsobjekt
  cout << "\nQuad-Summe : "
       << binop(3.2, 4.3, quadsum) << endl;    // Uebergabe Funktionspointer
}
```

Ausgabe des Programms :

```
Ungleichheit von (5+6) und 10 : true

Addition   : 6.2
Division   : 5
Quad-Summe : 28.73
```

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (4)

• Funktions-Adapter

◇ In der Utility-Bibliothek sind – ebenfalls in der Headerdatei **`<functional>`** – auch so genannte **Funktions-Adapter** definiert.
Dies sind Klassen (genauer : Klassen-Templates), die es auf einfache Art und Weise ermöglichen, neue Funktions-Objekte durch Modifikation anderer Funktionsobjekte bzw durch Anpassung von Funktions-Pointern zu erzeugen.

◇ Es gibt vier **Gruppen von Funktions-Adaptern** :

- ▻ **Binder** (*binders*)
  Sie erzeugen aus einem zweistelligen Funktionsobjekt ein einstelliges Funktionsobjekt, indem ein Argument der Operatorfunktion `operator()` an einen bestimmten festen Wert gebunden wird.
- ▻ **Memberfunktionsadapter** (*adapters for pointers to members*)
  Sie ermöglichen es, dass eine Memberfunktion wie ein Funktionsobjekt als Argument eines Algorithmus verwendet werden kann.
- ▻ **Funktionszeigeradapter** (*adapters for pointers to functions*)
  Sie ermöglichen es, dass eine freie (oder statische Member-) Funktion wie ein Funktionsobjekt als Argument eines Algorithmus verwendet werden kann.
- ▻ **Negierer** (*negators*)
  Sie invertieren ein Prädikat.

◇ Funktions-Adapter sind auch miteinander **kombinierbar**.

◇ **Funktions-Adapter** sind als **Klassen-Templates** definiert.
Template-Parameter ist die Klasse des umzuwandelnden Funktionsobjektes.
Dem Konstruktor wird – gegebenenfalls zusammen mit weiteren Parametern – das umzuwandelnde Funktions-Objekt als Parameter übergeben. Er speichert dieses in einer Datenkomponente
Die Operator-Funktion `operator()` führt die modifizierte Operation unter Verwendung des gespeicherten Funktionsobjekts aus.
Zu jedem Adapter gehört eine geeignete **Umwandlungsfunktion** (definiert als Funktions-Template), die aus einem als Argument übergebenen Funktionsobjekt ein Funktionsobjekt vom Adapter-Typ erzeugt.
Die jeweilige Umwandlungsfunktion stellt die **Anwendungsschnittstelle** eines Funktions-Adapters dar.

• Überblick über die Funktions-Adapter

Funktionsdapter-Klasse	Umwandlungsfunktion	Gruppe
`binder2nd`	`bind2nd()`	Binder
`binder1st`	`bind1st()`	
`mem_fun_t`	`mem_fun()`	Memberfunktionsadapter
`const_mem_fun_t`	`mem_fun()`	
`mem_fun1_t`	`mem_fun()`	
`const_mem_fun1_t`	`mem_fun()`	
`mem_fun_ref_t`	`mem_fun_ref()`	
`const_mem_fun_ref_t`	`mem_fun_ref()`	
`mem_fun1_ref_t`	`mem_fun_ref()`	
`const_mem_fun1_ref_t`	`mem_fun_ref()`	
`pointer_to_unary_function`	`ptr_fun()`	Funktionszeigeradapter
`pointer_to_binary_function`	`ptr_fun()`	
`unary_negate`	`not1()`	Negierer
`binary_negate`	`not2()`	

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (5)

- **Beispiel für einen Binder : Umwandlungsfunktion `bind2nd` (Adapter `binder2nd`)**

  ◇ Erzeugt **aus** einem **zweistelligen Funktionsobjekt** und einem Wert `y` ein **einstelliges Funktionsobjekt**, in dem das zweite Argument (des zweistelligen Funktionsobjekts) an den Wert `y` gebunden wird

  ◇ **Definition des Klassen-Templates `binder2nd`**

```
template<class Operation>
  class binder2nd
    : public unary_function<typename Operation::first_argument_type,
                            typename Operation::result_type>
{ protected :
    Operation op;
    typename Operation::second_argument_type value;
  public :
    binder2nd(const Operation& binop,
              const typename Operation::second_argument_type& y)
      : op(binop), value(y) { }
    typename Operation::result_type  operator()
      (const typename Operation::first_argument_type& x) const
    { return op(x, value); }
};
```

  ◇ **Definition der Umwandlungsfunktion `bind2nd`**

```
template<class Operation, class T>
  binder2nd<Operation> bind2nd(const Operation& binop, const T& y)
  { return
      binder2nd<Operation>(binop, typename Operation::second_argument_type(y));
  }
```

  ◇ **Beispiel :**

  `less<int>()`
  ist ein – namenloses – **zweistelliges Funktionsobjekt** (expliziter Konstruktoraufruf) für den Vergleich zweier `int`-Werte.
  Seiner Operatorfunktion `operator()(a, b)` müssen die beiden zu vergleichenden Werte `a und b` als Parameter übergeben werden..

  mittels
  `bind2nd(less<int>(), 9)`
  wird hieraus ein **einstelliges Funktionsobjekt**, das einen `int`-Wert mit dem Wert `9` vergleicht.
  Der Operatorfunktion `operator()(a)` dieses Objekts ist nur der Wert `a` als einziger Parameter zu übergeben.

Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 11

11. Standard-Template-Library (STL)

Standard-Template-Library (STL) von C++ : Überblick (1)

• Einführung

◇ Die **Standard-Template-Library (STL)** ist ein wesentlicher Bestandteil der ANSI-C++-Standard-Bibliothek.
Sie stellt ein relativ mächtiges und effizientes Werkzeug zur Verarbeitung von **Datenmengen** unterschiedlichsten Typs zur Verfügung

◇ Es handelt sich um eine **generische Bibliothek** auf der Basis von **C++-Templates.**
Nicht der Typ der zu bearbeitenden Daten steht im Vordergrund, sondern die Art ihrer Verwaltung, der Zugriff zu ihnen und die auf sie anwendbaren Bearbeitungsoperationen.

◇ Die STL besteht im wesentlichen aus **drei** aufeinander abgestimmten **Teilen** :

- ▻ ***Containers Library*** : Definition von Container-Klassen.
Container-Objekte speichern andere Objekte (→ Datenmengen)
- ▻ ***Iterator Library*** : Definition von Iterator-Klassen.
Iterator-Objekte ermöglichen den Zugriff zu den in Container-Objekten gespeicherten Objekten (den Elementen einer Menge)
- ▻ ***Algorithm Library*** : Definition von Funktionen zur Bearbeitung von Datenmengen und Elementen von Datenmengen

• Grundkonzept

◇ Den Kern der STL bilden **7 Container-Klassen** :
`vector, list, deque, set, multiset, map` und **`multimap.`**
Alle Container-Klassen sind als **Klassen-Templates** definiert. Dabei ist der Typ der in einem Container zu speichernden Objekte (Elemente) Template-Parameter (→ generische Programmierung).
Ein bestimmter Container kann also immer nur Elemente eines bestimmten Typs speichern.
Die verschiedenen Container-Klassen spiegeln die unterschiedlichen Möglichkeiten zur Realisierung und Verwaltung einer Datenmenge wieder. Dementsprechend besitzen sie jeweils spezifische Vor- und Nachteile.

◇ Die für die verschiedenen Container-Klassen **implementierten Schnittstellen** sind so gehalten, dass wo immer möglich und sinnvoll **gleichnamige Memberfunktionen** (mit **gleichartiger Funktionalität**) existieren.
So stehen u.a. für alle Container-Klassen gleichnamige Funktionen zum Ermitteln der Iteratorgrenzwerte, sowie die Vergleichsoperatoren und der Zuweisungsoperator zur Verfügung.
→ Die verschiedenen Containerklassen lassen sich teilweise **gleichartig verwenden**.

◇ Zusätzlich zu den obigen fundamentalen Container-Klassen sind drei **Container-Adapter-**Klassen definiert :
`stack, queue,` und **`priority_queue.`**
Diese Container-Adapter passen die Container-Klassen `vector, deque` und `list` an spezielle Anforderungen an.

◇ Zu allen Container-Klassen (außer den Container-Adaptern) sind jeweils zugehörige **Iterator-Klassen** definiert.
Iteratoren erlauben einen **zeigerähnlichen Zugriff** zu den **Elementen eines Containers**. Dabei bieten sie unabhängig von der jeweiligen Container-Klasse die **gleiche Schnittstelle** für den Element-Zugriff. Zu den in einem Container gespeicherten Elementen kann damit gleichartig – unabhängig von der internen Implementierung – zugegriffen werden.
Da die Iteratoren containerklassen-spezifisch implementiert sind, berücksichtigen sie die im Container eingesetzte Datenorganisation.
Es gibt verschiedene **Iterator-Kategorien**, die – zusätzlich zu einer für alle Iteratoren gleichartigen Grund-Funktionalität – unterschiedliche Operationen zur Verfügung stellen.
Nicht alle Iterator-Kategorien sind für alle Container-Klassen anwendbar.

◇ Zahlreiche **Algorithmen** zur Bearbeitung von Mengen als Ganzes bzw Mengenelementen (z.B. Traversieren, Suchen, Finden, Sortieren, Kopieren usw) sind durch **freie Funktions-Templates** implementiert. Sie arbeiten mit **Iteratoren** (Template-Parameter, Funktions-Parameter).
Viele dieser Funktionen können auf alle Container-Arten angewendet werden, andere nur auf solche, die eine spezielle Iterator-Kategorie anbieten.
Häufig können einer Algorithmen-Funktion **Hilfsfunktionen als Parameter** (Funktions-Pointer oder Funktions-Objekte) übergeben werden, mit denen eine Anpassung des Algorithmus an spezielle Bedürfnisse erreicht wird.

◇ Diese Art der Implementierung – **Trennung von Daten und Operationen** – **widerspricht** zwar dem **objektorientierten Gedanken**. Sie hat aber gegenüber der Implementierung der Algorithmen als Memberfunktionen der Container den **Vorteil**, dass die **Algorithmen** jeweils **nur einmal** (und nicht pro Container-Klasse) zu **implementieren** sind.

Standard-Template-Library (STL) von C++ : Überblick (2)

• Anmerkungen zu Container-Elementen

◇ Alle Container der STL besitzen eine **Wert-Semantik**.
Das bedeutet, dass die in einen Container aufgenommenen Elemente als **Kopie** und nicht als Referenz (Pointer) **abgelegt** werden und auch **Kopien** der enthaltenen Elemente **zurückgeliefert** werden.
Vorteile : Probleme, wie Verweise auf nicht mehr existierende Elemente, können nicht auftreten
Nachteile : Das Kopieren der Elemente dauert i.a. länger als das Kopieren ihrer Adressen
Ein Element kann nicht gleichzeitig von mehreren Containern änderbar verwaltet werden.

◇ Die prinzipiell mögliche Verwendung von **Pointern als Container-Elemente** sollte i.a. **vermieden** werden.
Hierdurch lässt sich zwar indirekt eine Referenz-Semantik implementieren, die aber Probleme beinhaltet :
- So kann es vorkommen, dass als Elemente enthaltene Pointer auf Objekte zeigen, die gar nicht mehr existieren.
- Ausserdem arbeiten Vergleiche von Zeigern mit Adressen und nicht mit den durch die Zeiger referierten Objekten (Adress-Vergleich statt Werte-Vergleich).

◇ Die Container der STL können prinzipiell **Elemente beliebigen Typs** verwalten.
Allerdings müssen diese Typen die für das **Kopieren notwenigen Eigenschaften** implementieren :

▻ Es muss ein **Copy-Konstruktor** öffentlich verfügbar sein (ein selbstdefinierter, wenn der Default-Copy-Konstruktor nicht richtig arbeiten würde, andernfalls reicht dieser aus).
Er sollte ein gutes Zeitverhalten besitzen.

▻ Es muss ein **Zuweisungsoperator** öffentlich verfügbar sein (ein selbstdefinierter, wenn der Default-Zuweisungs-operator nicht richtig arbeiten würde, andernfalls reicht dieser aus).

▻ Der **Destruktor** muss öffentlich verfügbar sein, da ein Element beim Entfernen aus dem Container zerstört werden muss.

◇ Darüberhinaus müssen die Element-Typen gegebenenfalls **weitere Anforderungen** erfüllen :

▻ Für die Anwendung einiger Memberfunktionen bestimmter Container-Klassen muss der **Default-Konstruktor** definiert sein.

▻ Für Suchfunktionen muss der **Vergleichsoperator** für **Gleichheit** (`==`) definiert sein.

▻ Für Sortierfunktionen muss der **Vergleichsoperator** für "**kleiner als**" (`<`) definiert sein.

• Fehlerbehandlung in der STL

◇ In der STL finden praktisch **keinerlei Überprüfungen** auf **Fehler** wie
- Überschreitung von Bereichsgrenzen,
- Verwendung von falschen, fehlerhaften oder ungültigen Iteratoren,
- Zugriff zu nicht vorhandenen Elementen

statt.
Beim Auftritt derartiger Fehler ist das Verhalten der entsprechenden Bibliotheks-Komponente und damit des Programms undefiniert.

◇ Lediglich **zwei Memberfunktionen** der Container-Klasse **`vector`** (**`at()`** und **`reserve()`**) sowie **eine Memberfunktion** der Container-Klasse **`deque`** (**`at()`**) können eine **Exception auslösen**.

◇ Darüberhinaus können gegebenenfalls lediglich die von anderen – durch die STL benutzten – Bibliothekskomponenten oder sonstigen aufgerufenen Funktionen erzeugten Exceptions auftreten (z.B. `bad_alloc` bei Allokationsfehlern).

◇ Die STL behandelt (fängt) diese Exceptions aber nicht. Tritt eine Exception auf, ist der Zustand aller beteiligten STL-Objekte undefiniert. Wird z.B. beim Einfügen eines Elements in einen Container eine Exception generiert, so gerät der Container in einen undefinierten Zustand, der seine weitere sinnvolle Verwendung verhindert.

◇ Der Grund für die nicht vorhandene Fehlerüberprüfung liegt in dem **Grundgedanken**, die STL mit **optimalem Zeit-verhalten** zu implementieren.
Fehlerüberprüfungen kosten aber Zeit.

Standard-Template-Library (STL) von C++ : Container (Überblick) (1)

- **Container-Arten**

  ◇ **Sequentielle Container**
  Sie speichern die Elemente als **geordnete Mengen**, in denen jedes Element eine bestimmte **Position** besitzt, die durch den **Zeitpunkt** und **Ort** des **Einfügens** festgelegt ist.
  Die enthaltenen Elemente sind damit **linear angeordnet** und können über ihre jeweilige Position angesprochen werden.
  Typischerweise erfolgt die Speicherung der Elemente in dynamischen **Arrays** bzw **Listen**.

  ◇ **Assoziative Container**
  Sie speichern die Elemente als **sortierte Mengen**, in denen die **Position** eines Elementes durch ein **Sortierkriterium** bestimmt ist. → sehr gutes Zeitverhalten bei Suchen und Finden.
  Ein neues Element wird entsprechend dem Sortierkriterium **automatisch sortiert** eingefügt.
  Der Zugriff zu den Elementen erfolgt **assoziativ** über **Suchschlüssel** (*keys*).
  Typischerweise erfolgt die Speicherung der Elemente in einem balanzierten **Binärbaum**.

- **Sequentielle Container**

  ◇ Die Klassen-Templates für die sequentiellen Container besitzen **2 Template-Parameter** :
  `<T, Allocator = allocator<T> >`

  ▻ **`T`** ist der **Datentyp** der zu **verwaltenden Objekte** (Elemente).

  ▻ **`Allocator`** ist eine **Allokator-Klasse**, die das Speichermodell für die zu verwendende dynamische Speicherverwaltung definiert. Als **Default** ist die Standard-Allokator-Klasse **`allocator<T>`**, die `new` und `delete` zur Speicherallokation verwendet, festgelegt.

  ◇ **Vektor :**

```
template <class T, class Allocator = allocator<T> > class vector;
```

  Ein Vektor (-Container) verwaltet die Elemente in einem dynamischen Array. Er ermöglicht einen direkten wahlfreien Zugriff zu den einzelnen Elementen. Hierfür existieren der Indexoperator und Direktzugriffs-Iteratoren.
  Das Anhängen und Löschen von Elementen am Ende des Arrays erfolgt optimal schnell.
  Ein Einfügen oder Löschen von Elementen mitten im Array ist dagegen zeitaufwändig (Verschieben von Elementen !).
  Vektoren sind daher bevorzugt einzusetzen, wenn Einfüge- und Löschoperationen vor allem am Ende stattfinden.

  ◇ **Deque :**

```
template <class T, class Allocator = allocator<T> > class deque;
```

  Deque = ***double ended queue***
  Ein Deque (-Container) verwaltet die Elemente in einem nach beiden Seiten offenen (verlängerbaren) dynamischen Array. Damit ist das Einfügen und Löschen von Elementen nicht nur am Ende sondern auch am Anfang optimal schnell, aber etwas langsamer als bei einem Vektor. Auch hier ist das Einfügen oder Löschen in der Mitte zeitaufwändig.
  Deques gestatten ebenfalls einen direkten wahlfreien Zugriff zu den einzelnen Elementen über einen Index bzw mittels eines Direktzugriffs-Iterators.
  Sie sollten dann gewählt werden, wenn Einfüge- u./o. Löschoperationen häufig sowohl am Ende als auch am Anfang stattfinden.

  ◇ **Liste :**

```
template <class T, class Allocator = allocator<T> > class list;
```

  Ein Listen-Container (eine Liste) verwaltet seine Elemente in einer doppelt verketteten Liste.
  Ein direkter wahlfreier Zugriff zu den einzelnen Elementen ist nicht möglich, Indexoperator und Direktzugriffs-Iteratoren sind daher nicht implementiert. Das Einfügen und Löschen von Elementen erfolgt an allen Positionen gleich schnell.
  Listen sind immer dann bevorzugt einzusetzen, wenn viele Einfüge- u/o Löschoperationen insbesondere in der Mitte der gespeicherten Sequenz stattfinden.

Standard-Template-Library (STL) von C++ : Container (Überblick) (2)

- **Container-Adapter**

  ◇ Sie implementieren jeweils eine **spezielle Container-Funktionalität**, die sie auf einen **sequentiellen Container** abbilden.

  ◇ Für Container-Adapter existieren keine Iteratoren. Auf sie können daher auch nicht die Algorithmen der STL direkt angewendet werden.

  ◇ Die Klassen-Templates für die Container-Adapter besitzen **2** bzw **3 Template-Parameter** :
  `<T, Container, [Compare]>`

  ▻ **`T`** ist der **Datentyp** der zu **verwaltenden Objekte** (Elemente).

  ▻ **`Container`** ist die Container-Klasse, auf die die Abbildung erfolgt.
  Ein Objekt einer Container-Adapter-Klasse verwendet zur eigentlichen Speicherung seiner Elemente ein Objekt dieser Klasse als `protected`-Komponente

  ▻ **`Compare`** ist eine **Comparator-Klasse**, die eine Vergleichsfunktion für Objekte vom Typ `T` definiert.
  Eine Comparator-Klasse ist eine Klasse, für die der Funktionsaufruf-Operator (`operator()`) mit der Funktionalität einer Vergleichsfunktion überladen ist (→ Funktions-Objekte)
  Dieser Parameter wird nur für Priority-Queues verwendet.

  ◇ **Stack :**

```
template <class T, class Container = deque<T> > class stack;
```

  Implementiert die Funktionalität eines **Stacks** (Kellerspeicher, **LIFO**) unter Verwendung der Container-Klassen `deque` (Default), `vector` oder `list`.

  ◇ **Queue**

```
template <class T, class Container = deque<T> > class queue;
```

  Implementiert die Funktionalität eines **Pufferspeichers** (**FIFO**) unter Verwendung der Container-Klassen `deque` (Default) oder `list`.

  ◇ **Priority-Queue :**

```
template <class T, class Container = vector<T>
          class Compare = less<T> > class priority_queue;
```

  Implementiert einen **Pufferspeicher**, bei der die Elemente nach einer durch ein **Sortierkriterium** festgelegten "Priorität" wieder ausgelesen werden können. Es wird immer das Element mit der **höchsten Priorität** zurückgeliefert.
  Das Sortierkriterium kann als Template-Parameter übergeben werden. **Default** ist die Comparator-Klasse **`less<T>`**, d.h. die **höchste Priorität** hat das **grösste** Element.
  Priority-Queues verwenden zur Speicherung ihrer Elemente die Container-Klasse `vector` (Default) oder `deque`.

Standard-Template-Library (STL) von C++ : Container (Überblick) (3)

- **Assoziative Container**

  ◇ Die Klassen-Templates für die Container-Adapter besitzen **3** bzw **4 Template-Parameter** :
  `<Key, [T], Compare, Allocator>`

  ▻ **`Key`** ist der **Datentyp** des **Suchschlüssels**, nach dem die zu **verwaltenden Objekte** (Elemente) sortiert werden. Bei den Container-Klassen `set` und `multiset` sind Suchschlüssel und zu verwaltendes Objekt jeweils identisch. Dieser Typ muss sortierbar sein, d.h. es muss der **Vergleichsoperator** **`<`** für ihn anwendbar sein.

  ▻ **`T`** ist der **Datentyp** eines mit dem Suchschlüssel zu einem **Paar** verknüpften "**Werts**". Dieser Parameter existiert nur bei den Container-Klassen `map` und `multimap`, bei denen die mit dem Suchschlüssel verknüpften "Werte" die eigentlichen **verwalteten Objekte** sind.

  ▻ **`Compare`** ist eine **Comparator-Klasse**, die eine Vergleichsfunktion für Objekte vom Typ `Key` definiert. Eine Comparator-Klasse ist eine Klasse, für die der Funktionsaufruf-Operator (`operator()`) mit der Funktionalität einer Vergleichsfunktion überladen ist (→ Funktions-Objekte). Sie legt das **Sortierkriterium** fest.
  Als **Default** ist die Comparator-Klasse **`less<Key>`** vorgesehen.

  ▻ **`Allocator`** ist eine **Allokator-Klasse**, die das Speichermodell für die zu verwendende dynamische Speicherverwaltung definiert. Als **Default** ist die Standard-Allokator-Klasse **`allocator<...>`**, die `new` und `delete` zur Speicherallokation verwendet, festgelegt.

  ◇ **Set :**

```
template <class Key, class Compare = less<Key>
          class Allocator = allocator<Key> > class set
```

  Ein Set (-Container) verwaltet nach ihrem "Wert" sortierte Elemente, d.h. die Elemente selbst bilden auch den Suchschlüssel. Jedes Element darf nur einmal in einem Set vorkommen.
  Der Zugriff zu einem gesuchten Element erfolgt sehr schnell. Auch das Einfügen u/o. Löschen von Elementen besitzt an allen Stellen ein gutes Zeitverhalten. Die Elemente von Sets können auch sequentiell durchlaufen werden.
  Die Elemente können nicht direkt geändert werden. Eine Änderung könnte die Sortierung ungültig machen. Um ein Element zu ändern, muß es daher aus dem Set entfernt werden und nach der Änderung als neues Element wieder eingefügt werden.

  ◇ **Multiset :**

```
template <class Key, class Compare = less<Key>
          class Allocator = allocator<Key> > class multiset
```

  Ein Multiset (-Container) entspricht einem Set mit dem Unterschied, dass Elemente auch mehrfach enthalten sein können.

  ◇ **Map** ("Dictionary", "assoziatives Array") **:**

```
template <class Key, class T, class Compare = less<Key>
          class Allocator = allocator<pair<const Key,T> > > class map
```

  Ein(e) Map (-Container) speichert Schlüssel-/Werte-Paare als Elemente. Die Elemente sind nach dem (Such-)Schlüssel sortiert. Der Suchschlüssel dient zum Auffinden des mit ihm assoziierten Werts (das eigentliche verwaltete Objekt). Jeder Suchschlüssel darf nur einmal in einer Map vorkommen.
  Die Eigenschaften entsprechen denen eines Sets, mit dem Unterschied, dass die enthaltenen Elemente Paare sind. Der Schlüssel eines Elements darf nicht geändert werden, wohl aber sein Wert.

  ◇ **Multimap :**

```
template <class Key, class T, class Compare = less<Key>
          class Allocator = allocator<pair<const Key,T> > > class multimap
```

  Ein Multimap (-Container) entspricht einer Map mit dem Unterschied, dass mehrere Elemente mit dem gleichen Schlüssel enthalten sein können.

Standard-Template-Library (STL) von C++ : Container (Überblick) (4)

- **Für alle Container-Klassen definierte Memberfunktionen**

◇ **Anmerkungen :**

- ▻ In der folgenden Zusammenstellung steht **`Container`** für eine der Container-Klassen,
  z.B. bei einem **Vektor** für : **`vector<T, Allocator>`**
  z.B. bei einer **Map** für : **`map<Key, T, Compare, Allokator>`**
- ▻ **`size_type`** ist ein implementierungsabhängiger innerhalb der jeweiligen Containerklasse definierter vorzeichenloser Ganzzahl-Typ
- ▻ Die mit ✡ markierten Funktionen sind für **Container-Adapter nicht** (explizit) definiert

Default-Konstruktor	Konstruktor für Default-Initialisierung des Containers
Konstruktor(en) mit Parametern	Konstruktor(en) für unterschiedliche Initialisierungsmethoden
✡ **Copy-Konstruktor**	Initialisierung eines Containers mit Kopie eines existierenden Containers
`size_type` **`size()`** `const;`	Anzahl der aktuell im Container enthaltenen Elemente (Container-Größe)
✡ `size_type` **`max_size()`** `const;`	maximale Anzahl der Elemente, die ein Container aufnehmen kann
`bool` **`empty()`** `const;`	`true`, wenn Container leer, sonst `false`
✡ `void` **`swap`**`(Container& b);`	vertauscht den Inhalt des aktuellen Containers mit dem des Containers b
✡ `void` **`clear`**`();`	entfernt alle Elemente aus dem Container
✡ `Container&` **`operator=`**`(const Container& b);`	Zuweisung des Inhalts von Container `b` an akt. Container
`iterator` **`insert`**`(iterator pos, const value_type& val);`	fügt ein neues Element mit dem Wert `val` an der Position `pos` ein. `value_type` ist ein container-spezifischer Typ
✡ `iterator` **`erase`**`(iterator pos);` ✡ `iterator` **`erase`**`(iterator fi, iterator la);` **Anm.** : bei assoziativen Containern Rückgabetyp : `void`	löscht das Element an der Iterator-Pos. `pos` löscht alle Elemente zwischen `fi` (einschliesslich) und `la` (ausschliesslich) liefert Iterator, der auf das folgende Element zeigt
✡ `iterator` **`begin`**`();` ✡ `const_iterator` **`begin`**`() const;`	liefert Iterator , der auf das erste Element zeigt
✡ `iterator` **`end`**`();` ✡ `const_iterator` **`end`**`() const;`	liefert Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt
✡ `reverse_iterator` **`rbegin`**`();` ✡ `const_reverse_iterator` **`rbegin`**`() const;`	liefert einen Reverse-Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt
✡ `reverse_iterator` **`rend`**`();` ✡ `const_reverse_iterator` **`rend`**`() const;`	liefert einen Reverse-Iterator, der auf das erste Element zeigt

Standard-Template-Library (STL) von C++ : Container (Überblick) (5)

- **Für alle Container-Klassen definierte freie Funktionen**

  ◇ Hierbei handelt es sich – bis auf eine Ausnahme – um Operatorfunktionen für die **Vergleichsoperatoren.**

  ◇ **Anmerkungen :**

  ▻ Die als **Funktions-Templates** definierten Funktionen werden in der folgender Zusammenstellung in **vereinfachter Darstellung** angegeben.
  Dabei steht **`Container`** für eine der Container-Klassen (s. "Allen Containern gemeinsame Memberfunktionen")
  Statt z.B. die **nur** für **Vektoren** geltende Darstellung anzugeben

```
template <class T, class Allocator>
  bool operator==(const vector<T, Allocator>& x,
                  const vector<T, Allocator>& y);
```

  wird die **allgemeingültige Formulierung** verwendet :

```
  bool operator==(const Container& x, const Container& y);
```

  ▻ **Alle** hier aufgelisteten Funktionen sind **nicht** für den Container-Adapter **`priority_queue`** definiert.

  ▻ Die mit ✡ markierte Funktion ist für **Container-Adapter nicht** (explizit) definiert

`bool operator==(const Container& x, const Container& y);`	`true` wenn `x==y`, sonst `false` Zwei Container derselben Klasse sind gleich, wenn sie die gleichen Elemente in der gleichen Reihenfolge besitzen. Für den Vergleich wird für alle Elemente der Reihe nach jeweils `operator==()` aufgerufen.
`bool operator!=(const Container& x, const Container& y);`	`true` wenn `x!=y`, sonst `false`
`bool operator<(const Container& x, const Container& y);`	`true` wenn `x<y`, sonst `false`
`bool operator<=(const Container& x, const Container& y);`	`true` wenn `x<=y`, sonst `false`
`bool operator> (const Container& x, const Container& y);`	`true` wenn `x>y`, sonst `false`
`bool operator>=(const Container& x, const Container& y);`	`true` wenn `x>=y`, sonst `false`
✡ `void swap(Container& x, Container& y);`	vertauscht den Inhalt des Containers `x` mit dem Inhalt des Containers `y`

Standard-Template-Library (STL) von C++ : Klassen-Template `vector` (1)

- **Eigenschaften**

  ◇ Implementierung von **Vektoren**. Ein **Vektor** (-Container) verwaltet die Elemente in einem **dynamischen Array**.
  Die Elemente besitzen eine **definierte Reihenfolge** ("*ordered collection*")

  ◇ Zu den einzelnen Elementen kann **direkt wahlfrei zugegriffen** werden.
  Hierfür existieren der **Indexoperator** und **Direktzugriffs-Iteratoren**.

  ◇ Das **Anhängen** und **Löschen** von Elementen **am Ende** des Arrays erfolgt **optimal schnell**.
  Ein Einfügen oder Löschen von Elementen **mitten im Array** ist dagegen **zeitaufwändig** (Verschieben von Elementen !).
  → Vektoren sind daher **bevorzugt einzusetzen**, wenn **Einfüge- und Löschoperationen** vor allem **am Ende** stattfinden.

  ◇ Die **Definition** des Klassen-Templates `vector<>` befindet sich in der **Headerdatei `<vector>`**

```
template <class T, class Allocator = allocator<T> >
  class vector
  { // ...
  };
```

- **Konstruktoren**

`explicit vector(const Allocator& = Allocator());`	Erzeugung eines Vektors der Länge 0
`explicit vector(size_type n, const T& val = T(), const Allocator& = Allocator());`	Erzeugung eines Vektors der Länge n, Initialisierung der Elemente mit dem Wert `val`
`template <class InputIterator> vector(InputIterator first, InputIterator last, const Allocator& = Allocator());`	Erzeugung eines Vektors, dessen Elemente mit den zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elementen eines anderen Containers initialisiert werden
`vector(const vector<T, Allocator>& x);`	Copy-Konstruktor

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**

  ◇ **Methoden zum Ermitteln und Ändern der Kapazität sowie Änderung der aktuellen Größe**

  Kapazität = maximale Anzahl der Elemente, die der Vektor ohne Neuallokation enthalten kann.
  = Größe des aktuell allozierten Arrays

`size_type capacity() const;`	Rückgabe der aktuellen Kapazität des Vektors
`void reserve(size_type n);`	Vergrösserung der Kapazitaet auf einen Wert `>=n`, wenn `n >` akt. Kapazität Keine Wirkung, wenn `n<=` aktuelle Kapazität Falls `n > max_size()` ist, wird Exception `length_error` geworfen **Achtung** : Alle **Referenzen**, **Pointer** und **Iteratoren** auf Elemente des Vektors werden nach einer Kapazitätsvergrößerung (Neu-Allokation !) **ungültig**
`void resize(size_type_sz, T c= T());`	Veränderung der akt. Größe des Vektors auf den Wert `sz` (Vergrößerung oder Verkleinerung) bei Vergrößerung : Initialisierung der neuen Elemente mit dem Wert `c`

Standard-Template-Library (STL) von C++ : Klassen-Template `vector` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

  ◇ **Methoden zum Elementzugriff**

`T& operator[](size_type n);` `const T& operator[](size_type n) const;`	Rückgabe des Elements an der Position mit dem Index `n` Falls `n >= size()` ist das Verhalten undefiniert
`T& at(size_type n);` `const T& at(size_type n) const;`	Rückgabe des Elements an der Position mit dem Index `n` Falls `n >= size()` wird Exception `out_of_range` geworfen
`T& front();` `const T& front() const;`	Rückgabe des ersten Elements (Index `0`)
`T& back();` `const T& back() const;`	Rückgabe des letzten Elements (Index `size()-1`)

  ◇ **Methoden zum Einfügen und Löschen von Elementen**

  Achtung : Durch das Einfügen mittels **`insert()`** werden alle **Referenzen**, **Pointer** und **Iteratoren** auf Elemente ab der Einfügeposition **ungültig**. Bei erforderlicher Neu-Allokation gilt das auch für alle übrigen Positionen.

`void push_back(const T& x);`	Einfügen des Objekts (Kopie) `x` als neues letztes Element
`void pop_back();`	Löschen des letzten Elements
`iterator insert(iterator pos, const T& x);`	Einfügen des Objekts `x` als neues Element an der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um eine Position weitergeschoben) Rückgabe : die Einfügeposition `pos`
`void insert(iterator pos,` `size_type n, const T& x);`	Einfügen von `n` Kopien des Objekts `x` als neue Elemente ab der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um `n` Positionen weitergeschoben)

`template <class InputIterator>`
`void insert(iterator pos,`
`InputIterator first,`
`InputIterator last);`
	Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente ab der Position `pos.`
	(das bisherige Element an dieser Position und alle folgenden Elemente werden um entsprechend viele Positionen weitergeschoben)

Standard-Template-Library (STL) von C++ : Klassen-Template `vector` (3)

- **Einfaches Demonstrationsprogramm `vectordem`**

```
// C++-Quelldatei vectordem_m.cpp --> Programm vectordem
// Einfaches Demo-Programm zum Klassen-Template vector<> der STL

#include <vector>
#include <string>
#include <iostream>
using namespace std;

template <class T>
  void showSizeData(const vector<T>& vec, ostream& out)
  { out << "size()     : " << vec.size() << endl;
    out << "max_size() : " << vec.max_size() << endl;
    out << "capacity() : " << vec.capacity() << endl;
  }

template <class T>
  void showContent(const vector<T>& vec, ostream& out)
  { for (int i=0; i<vec.size(); i++)
      out << vec[i] << ' ';
    out << endl;
  }

int main(void)
{
  vector<string> satz;
  cout << "\nleerer string-Vector :\n";
  showSizeData(satz, cout);
  vector<string>::size_type nsz = 5
  satz.reserve(nsz);
  cout << "\nnach reserve(" << nsz << ") :\n";
  showSizeData(satz, cout);
  satz.push_back("Achtung,");
  satz.push_back("dies");
  satz.push_back("ist");
  satz.push_back("ein");
  satz.push_back("Satz");
  satz.push_back("als");
  satz.push_back("Beispiel");
  satz.push_back("!");
  cout << "\nstring-Vector enthaelt jetzt " << satz.size() << " Elemente :\n";
  showSizeData(satz, cout);
  cout << "\nInhalt :\n";
  showContent(satz, cout);
  swap(satz[1], satz[2]);
  string hilf=satz[satz.size()-2];
  satz.insert(satz.begin()+4, hilf);
  satz.erase(satz.end()-1);
  satz.pop_back();
  satz.back()="?";
  satz.insert(satz.begin()+5, "fuer");
  satz.insert(satz.begin()+6, "einen");
  satz.insert(satz.begin()+7, "wirklich");
  satz.insert(satz.begin()+8, "guten");
  cout << "\nstring-Vector nach Manipulation :\n";
  showSizeData(satz, cout);
  cout << "\nInhalt :\n";
  showContent(satz, cout);
  return 0;
}
```

Standard-Template-Library (STL) von C++ : Klassen-Template `vector` (4)

- **Ausgabe des Demonstrationsprogramms `vectordem`**

```
leerer string-Vector :
size()     : 0
max_size() : 268435455
capacity() : 0

nach reserve(5) :
size()     : 0
max_size() : 268435455
capacity() : 5

string-Vector enthaelt jetzt 8 Elemente :
size()     : 8
max_size() : 268435455
capacity() : 10

Inhalt :
Achtung, dies ist ein Satz als Beispiel !

string-Vector nach Manipulation :
size()     : 11
max_size() : 268435455
capacity() : 20

Inhalt :
Achtung, ist dies ein Beispiel fuer einen wirklich guten Satz ?
```

Standard-Template-Library (STL) von C++ : Klassen-Template `deque` (1)

- **Eigenschaften**

  ◇ Implementierung von **Deques** (Deque = ***double ended queue***)

  ◇ Ein Deque (-Container) verwaltet die Elemente in einem **nach beiden Seiten offenen** (verlängerbaren) **dynamischen Array**. Realisiert wird dieses typischerweise durch **mehrere Speicherblöcke**, wobei der erste Block logisch in die eine Richtung und der letzte in die andere Richtung wächst.
  Auch die Elemente eines Deques besitzen eine **definierte Reihenfolge** ("*ordered collection*")

  ◇ Zu den einzelnen Elementen kann ebenfalls **direkt wahlfrei zugegriffen** werden.
  Hierfür stehen der **Indexoperator** und **Direktzugriffs-Iteratoren** zur Verfügung.

  ◇ Das **Einfügen** und **Löschen** von Elementen ist nicht nur **am Ende** sondern auch **am Anfang** optimal **schnell**, aber etwas langsamer als bei einem Vektor. Auch hier ist das Einfügen oder Löschen in der Mitte zeitaufwändig. Deques sollten dann gewählt werden, wenn Einfüge- u./o. Löschoperationen häufig sowohl am Ende als auch am Anfang stattfinden.

  ◇ Die **Definition** des Klassen-Templates `deque<>` befindet sich in der **Headerdatei `<deque>`**

```
template <class T, class Allocator = allocator<T> >
  class deque
  { // ...
  };
```

- **Konstruktoren**

`explicit deque(const Allocator& = Allocator());`	Erzeugung eines Deques der Länge 0
`explicit deque(size_type n, const T& val = T(), const Allocator& = Allocator());`	Erzeugung eines Deques der Länge n, Initialisierung der Elemente mit dem Wert `val`
`template <class InputIterator> deque(InputIterator first, InputIterator last, const Allocator& = Allocator());`	Erzeugung eines Deques, dessen Elemente mit den zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elementen eines anderen Containers initialisiert werden
`deque(const deque<T, Allocator>& x);`	Copy-Konstruktor

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**

  ◇ **Methode zur Änderung der aktuellen Größe**

`void resize(size_type_sz, T c= T());`	Veränderung der akt. Größe des Deques auf den Wert `sz` (Vergrößerung oder Verkleinerung) bei Vergrößerung : Initialisierung der neuen Elemente mit dem Wert `c`

Standard-Template-Library (STL) von C++ : Klassen-Template `deque` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

  ◇ **Methoden zum Elementzugriff**

`T& operator[](size_type n);` `const T& operator[](size_type n) const;`	Rückgabe des Elements an der Position mit dem Index `n` Falls `n >= size()` ist das Verhalten undefiniert
`T& at(size_type n);` `const T& at(size_type n) const;`	Rückgabe des Elements an der Position mit dem Index `n` Falls `n >= size()` wird Exception `out_of_range` geworfen
`T& front();` `const T& front() const;`	Rückgabe des ersten Elements (Index `0`)
`T& back();` `const T& back() const;`	Rückgabe des letzten Elements (Index `size()-1`)

  ◇ **Methoden zum Einfügen und Löschen von Elementen**

  Achtung : Durch das Einfügen mittels **`insert()`**
  - in der **Mitte** des Deques werden alle **Referenzen**, **Pointer** und **Iteratoren** auf Elemente des Deques **ungültig**
  - an einem der beiden **Enden** werden nur die **Iteratoren** auf Deque-Elemente **ungültig**

`void push_back(const T& x);`	Einfügen des Objekts (Kopie) `x` als neues letztes Element
`void push_front(const T& x);`	Einfügen des Objekts (Kopie) `x` als neues erstes Element
`void pop_back();`	Löschen des letzten Elements
`void pop_front();`	Löschen des ersten Elements
`iterator insert(iterator pos, const T& x);`	Einfügen des Objekts `x` als neues Element an der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um eine Position weitergeschoben) Rückgabe : die Einfügeposition `pos`
`void insert(iterator pos,` `size_type n, const T& x);`	Einfügen von `n` Kopien des Objekts `x` als neue Elemente ab der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um `n` Positionen weitergeschoben)

`template <class InputIterator>`
`void insert(iterator pos,`
`InputIterator first,`
`InputIterator last);`
	Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente ab der Position `pos`.
	(das bisherige Element an dieser Position und alle folgenden Elemente werden um entsprechend viele Positionen weitergeschoben)

Standard-Template-Library (STL) von C++ : Klassen-Template `deque` (3)

- **Einfaches Demonstrationsprogramm `dequedem`**

```
// C++-Quelldatei dequedem_m.cpp --> Programm dequedem
// Einfaches Demo-Programm zum Klassen-Template deque<> der STL

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

template <class T>
  void showSizeData(const deque<T>& deq, ostream& out)
  { out << "size()     : " << deq.size() << endl;
    out << "max_size() : " << deq.max_size() << " (" << hex << deq.max_size()
        << ')' << dec << endl;
  }

template <class T>
  void showContent(const deque<T>& deq, ostream& out)
  { for (int i=0; i<deq.size(); i++)
      out << deq[i] << "  ";
    out << endl;
  }

int main(void)
{
  deque<float> fq;
  cout << "\nleerer float-Deque :\n";
  showSizeData(fq, cout);
  cout << "\nnach Einfuegen von Elementen :\n";
  fq.push_back(2.2f);
  fq.push_front(1.1f);
  fq.resize(4, 3.3f);
  showContent(fq, cout);
  cout << "\nnach Entfernung des ersten und letzten Elements :\n";
  fq.pop_front();
  fq.pop_back();
  showContent(fq, cout);
  cout << "\nnach weiterer Modifikation des Inhalts :\n";
  fq.push_front(5.5f);
  fq.push_front(6.6f);
  fq[2]=-7.7f;
  fq.pop_back();
  fq.push_back(8.8f);
  showContent(fq, cout);
  return 0;
}
```

- **Ausgabe des Demonstrationsprogramms `dequedem`**

```
leerer float-Deque :
size()     : 0
max_size() : 1073741823 (3fffffff)

nach Einfuegen von Elementen :
1.1  2.2  3.3  3.3

nach Entfernung des ersten und letzten Elements :
2.2  3.3

nach weiterer Modifikation des Inhalts :
6.6  5.5  -7.7  8.8
```

Standard-Template-Library (STL) von C++ : Klassen-Template `list` (1)

• Eigenschaften

◇ Implementierung von **Listen-Containern** (Listen)
Ein Listen-Container verwaltet seine Elemente in einer **doppelt verketteten Liste**.

◇ Die Elemente besitzen eine **definierte Reihenfolge**, ein direkter **wahlfreier Zugriff** zu einzelnen Elementen ist jedoch **nicht möglich**. Indexoperator und Direktzugriffs-Iteratoren sind daher nicht implementiert.

◇ Durch das **Einfügen** oder **Löschen** von Elementen werden **Verweise** auf andere Elemente **nicht ungültig**

◇ Das **Einfügen** und **Löschen** von Elementen erfolgt an **allen Positionen gleich schnell**.
Listen sind daher immer dann bevorzugt einzusetzen, wenn viele Einfüge- u/o Löschoperationen insbesondere in der Mitte der gespeicherten Sequenz stattfinden.

◇ Die **Definition** des Klassen-Templates `list<>` befindet sich in der **Headerdatei `<list>`**

```
template <class T, class Allocator = allocator<T> >
  class list
  { // ...
  };
```

• Konstruktoren

`explicit list(const Allocator& = Allocator());`	Erzeugung einer Liste der Länge 0
`explicit list(size_type n, const T& val = T(), const Allocator& = Allocator());`	Erzeugung einer Liste der Länge n, Initialisierung der Elemente mit dem Wert `val`
`template <class InputIterator> list(InputIterator first, InputIterator last, const Allocator& = Allocator());`	Erzeugung einer Liste, deren Elemente mit den zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elementen eines anderen Containers initialisiert werden
`list(const list<T, Allocator>& x);`	Copy-Konstruktor

• Zusätzliche spezifische Memberfunktionen (Auswahl)

◇ Methode zur Änderung der aktuellen Größe

`void resize(size_type_sz, T c= T());`	Veränderung der akt. Größe der Liste auf den Wert `sz` (Vergrößerung oder Verkleinerung) bei Vergrößerung : Initialisierung der neuen Elemente mit dem Wert `c`

◇ Methoden zum Elementzugriff

`T& front();` `const T& front() const;`	Rückgabe des ersten Elements
`T& back();` `const T& back() const;`	Rückgabe des letzten Elements

Standard-Template-Library (STL) von C++ : Klassen-Template `list` (2)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)**

  ◇ **Methoden zum Einfügen und Löschen von Elementen**

  Achtung : Durch das Einfügen oder Löschen werden **Referenzen**, **Pointer** und **Iteratoren** auf Elemente der Liste **nicht ungültig** (ausser Verweise auf die gelöschten Elemente)

`void` **`push_back`**`(const T& x);`	Einfügen des Objekts (Kopie) `x` als neues letztes Element
`void` **`push_front`**`(const T& x);`	Einfügen des Objekts (Kopie) `x` als neues erstes Element
`void` **`pop_back`**`();`	Löschen des letzten Elements
`boid` **`pop_front`**`();`	Löschen des ersten Elements
`iterator` **`insert`**`(iterator pos, const T& x);`	Einfügen des Objekts `x` als neues Element an der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um eine Position weitergeschoben) Rückgabe : die Einfügeposition `pos`
`void` **`insert`**`(iterator pos,` `size_type n, const T& x);`	Einfügen von `n` Kopien des Objekts `x` als neue Elemente ab der Position `pos` (das bisherige Element an dieser Position und alle folgenden Elemente werden um `n` Positionen weitergeschoben)

`template <class InputIterator>`
`void` **`insert`**`(iterator position,`
`InputIterator first,`
`InputIterator last);`
	Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente ab der Position `pos.`
	(das bisherige Element an dieser Position und alle folgenden Elemente werden um entsprechend viele Positionen weitergeschoben)

`void` **`remove`**`(const T& val);`	Löschen aller Elemente, die den Wert `val` besitzen
`template <class Predicate>` `void` **`remove_if`**`(Predicate pred);`	Löschen aller Elemente, für die das als Parameter übergebene Funktionsobjekt `pred` den Wert `true` liefert
`void` **`unique`**`();`	Löschen aller Elemente jeder Gruppe aufeinanderfolgender gleicher Elemente außer dem jeweils ersten.
`template <class BinaryPredicate>` `void` **`unique`**`(BinaryPredicate binpred);`	Löschen aller direkten Nachfolger-Elemente eines Elements, für die das als Parameter übergebene Funktionsobjekt `binpred` den Wert `true` liefert. `binpred` muß ein zweistelliges Funktionsobjekt sein, das auf jedes Element und sein jeweiliges direktes Nachfolger-Element angewendet wird.

Standard-Template-Library (STL) von C++ : Klassen-Template `list` (3)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, weitere Forts.)**

  ◇ **Weitere Methoden zur Modifikation von Listen**

`void splice(iterator pos, list<T, Allocator>& x);`	Einfügen aller Elemente der Liste `x` in die aktuelle Liste vor die Position `pos`. Aus der Liste `x` werden alle Elemente entfernt
`void splice(iterator pos, list<T, Allocator>& x, iterator i);`	Einfügen des durch den Iterator `i` referierten Elements der Liste `x` in die aktuelle Liste vor die Position `pos`. Das Element wird aus der Liste `x` entfernt Die Liste `x` darf mit der akt. Liste identisch sein.
`void splice(iterator pos, list<T, Allocator>& x, iterator first, iterator last);`	Einfügen der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente der Liste `x` in die aktuelle Liste vor die Position `pos`. Die eingefügten Elemente werden aus der Liste `x` entfernt. Die Liste `x` darf mit der akt. Liste identisch sein. Falls in diesem Fall `pos` zwischen `first` und `last` liegt, ist das Verhalten undefiniert
`void merge(list<T, Allocator>& x);`	Einfügen aller Elemente der Liste `x` in die akt. Liste unter Beibehaltung einer Sortierreihenfolge. Beide Listen müssen nach dieser Reihenfolge sortiert sein. Aus der Liste `x` werden alle Elemente entfernt
`template <class Compare>` `void merge(list<T, Allocator>& x, Compare comp);`	Einfügen aller Elemente der Liste `x` in die akt. Liste unter Beibehaltung der durch den Parameter `comp` bestimmten Sortierreihenfolge. Beide Listen müssen nach dieser Reihenfolge sortiert sein. Aus der Liste `x` werden alle Elemente entfernt
`void sort();`	Sortieren der akt. Liste unter Verwendung von `operator<()`. Diese Operatorfunktion muss für den Typ `T` definiert sein
`template <class Compare>` `void sort(Compare comp);`	Sortieren der akt. Liste unter Verwendung des Funktionsobjekts `comp`
`void reverse();`	Invertierung der Reihenfolge der Listenelemente

Standard-Template-Library (STL) von C++ : Klassen-Template `list` (4)

- **Einfaches Demonstrationsprogramm `listdem`**

```
// C++-Quelldatei listdem_m.cpp --> Programm listdem
// Einfaches Demo-Programm zum Klassen-Template list<> der STL

#include <list>
#include <iostream>
#include <cstdlib>
using namespace std;

template <class T>
  void showSizeData(const list<T>& lst, ostream& out)
  { out << "size()     : " << lst.size() << endl;
    out << "max_size() : " << lst.max_size() << endl;
  }

template <class T>
  void showContent(list<T>& lst, ostream& out)
  { list<T>::iterator it=lst.begin();
    if (it==lst.end()) out << "leer";
    else
      for (; it!=lst.end(); ++it)
        out << *it << "  ";
    out << endl;
  }

int main(void)
{
  list<int> il1, il2, il3;
  cout << "\nleere int-Liste :\n"; showSizeData(il1, cout);
  for (int i=0; i<10; i++)
  { il1.push_back(rand()%11+1);
    il2.push_front(i+1);
    il3.push_back(rand()%12);
  }
  cout << "\nInhalt Liste 1 :\n"; showContent(il1, cout);
  cout << "\nInhalt Liste 2 :\n"; showContent(il2, cout);
  cout << "\nInhalt Liste 3 :\n"; showContent(il3, cout);
  il1.splice(il1.begin(), il3);
  cout << "\nInhalt Liste 1 nach il1.splice(il1.begin(), il3) :\n";
  showContent(il1, cout);
  cout << "\nInhalt Liste 3 nach il1.splice(il1.begin(), il3) :\n";
  showContent(il3, cout);
  il1.sort();
  il2.reverse();
  cout << "\nInhalt Liste 1 nach sort() :\n";
  showContent(il1, cout);
  cout << "\nInhalt Liste 2 nach reverse() :\n";
  showContent(il2, cout);
  il1.merge(il2);
  cout << "\nInhalt Liste 1 nach il1.merge(il2) :\n";
  showContent(il1, cout);
  cout << "\nInhalt Liste 2 nach il1.merge(il2) :\n";
  showContent(il2, cout);
  il1.unique();
  cout << "\nInhalt Liste 1 nach unique() :\n";
  showContent(il1, cout);
  il1.remove_if(bind2nd(not_equal_to<int>(), 2));
  cout << "\nInhalt Liste 1 nach remove_if(...) :\n";
  showContent(il1, cout);
  return 0;
}
```

Standard-Template-Library (STL) von C++ : Klassen-Template `list` (5)

- **Ausgabe des Demonstrationsprogramms `listdem`**

```
leere int-Liste :
size()     : 0
max_size() : 1073741823 (3fffffff)

Inhalt Liste 1 :
9 10 8 6 2 8 6 7 4 10

Inhalt Liste 2 :
10 9 8 7 6 5 4 3 2 1

Inhalt Liste 3 :
11 4 4 6 8 5 3 11 2 0

Inhalt Liste 1 nach il1.splice(il1.begin(), il3) :
11 4 4 6 8 5 3 11 2 0 9 10 8 6 2 8 6 7 4 10

Inhalt Liste 3 nach il1.splice(il1.begin(), il3) :
leer

Inhalt Liste 1 nach sort() :
0 2 2 3 4 4 4 5 6 6 6 7 8 8 8 9 10 10 11 11

Inhalt Liste 2 nach reverse() :
1 2 3 4 5 6 7 8 9 10

Inhalt Liste 1 nach il1.merge(il2) :
0 1 2 2 2 3 3 4 4 4 4 5 5 6 6 6 6 7 7 8 8 8 8 9 9 10 10 10 11 11

Inhalt Liste 2 nach il1.merge(il2) :
leer

Inhalt Liste 1 nach unique() :
0 1 2 3 4 5 6 7 8 9 10 11

Inhalt Liste 1 nach remove_if(...) :
2
```

Standard-Template-Library (STL) von C++ : Container-Adapter (1)

- **Klassen-Template `stack<>`**

  ◇ Implementiert die Funktionalität eines **Stacks** (Kellerspeicher, **LIFO**) unter Verwendung der Container-Klassen `deque` (Default), `vector` oder `list`.

  ◇ Die **Definition** befindet sich in der Headerdatei **`<stack>`**

```
template <class T, class Container = deque<T> >
  class stack
  {
    public :
      typedef typename Container::value_type   value_type;
      typedef typename Container::size_type    size_type;
      typedef Container                        container_type;
    protected :
      Container c;
    public :
      explicit stack(const Container& co = Container()) : c(co) {}
      bool              empty() const             { return c.empty(); };
      size_type         size() const              { return c.size();  }
      value_type&       top()                     { return c.back();  }
      const value_type& top() const               { return c.back();  }
      void              push(const value_type& x) { c.push_back(x);   }
      void              pop()                     { c.pop_back();     }
  };
```

- **Klassen-Template `queue<>`**

  ◇ Implementiert die Funktionalität eines **Pufferspeichers** (**FIFO**) unter Verwendung der Container-Klassen `deque` (Default) oder `list`.

  ◇ Die **Definition** befindet sich in der Headerdatei **`<queue>`**

```
template <class T, class Container = deque<T> >
  class queue
  {
    public :
      typedef typename Container::value_type   value_type;
      typedef typename Container::size_type    size_type;
      typedef Container                        container_type;
    protected :
      Container c;
    public :
      explicit queue(const Container& co = Container()) : c(co) {}
      bool              empty() const             { return c.empty(); };
      size_type         size() const              { return c.size();  }
      value_type&       front()                   { return c.front(); }
      const value_type& front() const             { return c.front(); }
      value_type&       back()                    { return c.back();  }
      const value_type& back() const              { return c.back();  }
      void              push(const value_type& x) { c.push_back(x);   }
      void              pop()                     { c.pop_front();    }
  };
```

Standard-Template-Library (STL) von C++ : Container-Adapter (2)

- **Klassen-Template `priority_queue<>`**

  ◇ Implementiert einen **Pufferspeicher**, bei der die Elemente nach einer durch ein **Sortierkriterium** festgelegten "Priorität" wieder ausgelesen werden können. Es wird immer das Element mit der **höchsten Priorität** zurückgeliefert.

  ◇ Das Sortierkriterium kann als Template-Parameter übergeben werden. **Default** ist die Funktionsobjekt-Klasse **`less<T>`**, d.h. die **höchste Priorität** hat das **grösste** Element.

  ◇ Priority-Queues verwenden zur Speicherung ihrer Elemente die Container-Klasse `vector` (Default) oder `deque`.

  ◇ Für Priority-Queues sind **keine Vergleichsoperatoren** definiert.

  ◇ Die **Definition** befindet sich in der Headerdatei **`<queue>`**

```
template <class T, class Container = vector<T>,
          class Compare = less<typename Container::value_type> >
  class priority_queue
  {
    public :
      typedef typename Container::value_type   value_type;
      typedef typename Container::size_type    size_type;
      typedef Container                        container_type;
    protected :
      Container c;
      Compare comp;
    public :
      explicit priority_queue(const Compare& x = Compare(),
                              const Container& co = Container())
                             : c(co), comp(x) {}

      template <class InputIterator>
        priority_queue (InputIterator first, InputIterator last,
                        const Compare& x = Compare(),
                        const Container& co = Container())
                       : c(co), comp(x)
       { c.insert(c.end(),first,last);
         make_heap(c.begin(), c.end(), comp);
       }

      bool              empty() const             { return c.empty(); };
      size_type         size() const              { return c.size();  }
      const value_type& top() const               { return c.front(); }

      void push(const value_type& x)
      { c.push_back(x);
        push_heap(c.begin(), c.end(), comp);
      }

      void pop()
      { pop_heap(c.begin(), c.end(), comp);
        c.pop_back();
      }
  };
```

  ◇ Die zur **Implementierung** der Priority-Queue verwendeten freien Funktionen **`make_heap()`**, **`push_heap()`** und **`pop_heap()`** sind Funktionen aus der **Algorithmen-Bibliothek** der STL.
  Sie stellen sicher, dass der Container als "**Heap**" verwaltet wird, d.h. das sein erstes Element immer das Element mit der **"höchsten" Erfüllung** des **Vergleichskriteriums** ist.

Standard-Template-Library (STL) von C++ : Container-Adapter (3)

- **Einfaches Demonstrationsprogramm `priqueuedem` zum Container-Adapter `priority_queue`**

```
// C++-Quelldatei priqueuedem_m.cpp --> Programm priqueuedem
// Einfaches Demo-Programm zum Container-Adapter priority_queue<> der STL

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

int main(void)
{
  priority_queue<double> pqd;
  double w;
  cout << "\nleere double-Priority-Queue :\n";
  cout << "size() : " << pqd.size() << endl;
  cout << "\nAblage der folgenden Elemente :\n";
  pqd.push(w=27.33); cout << w << endl;
  pqd.push(w=63.12); cout << w << endl;
  pqd.push(w=12.84); cout << w << endl;
  pqd.push(w=21.37); cout << w << endl;
  cout << "\nEntfernen der beiden \"obersten\" Elemente :\n";
  cout << pqd.top() << endl;
  pqd.pop();
  cout << pqd.top() << endl;
  pqd.pop();
  cout << "\nAblage eines weiteren Elements :\n";
  pqd.push(w=99.88); cout << w << endl;
  cout << "\nEntfernen der beiden \"obersten\" Elemente :\n";
  cout << pqd.top() << endl;
  pqd.pop();
  cout << pqd.top() << endl;
  pqd.pop();
  cout << "\nverbleibende Anzahl von Elementen :\n";
  cout << "size() : " << pqd.size() << endl;
  return 0;
}
```

- **Ausgabe des Demonstrationsprogramm `priqueuedem`**

```
leere double-Priority-Queue :
size() : 0

Ablage der folgenden Elemente :
27.33
63.12
12.84
21.37

Entfernen der beiden "obersten" Elemente :
63.12
27.33

Ablage eines weiteren Elements :
99.88

Entfernen der beiden "obersten" Elemente :
99.88
21.37

verbleibende Anzahl von Elementen :
size() : 1
```

Standard-Template-Library (STL) von C++ : Klassen-Template `set` (1)

• Eigenschaften

◇ Implementierung von **Sets** (Set-Containern)
Ein Set (-Container) verwaltet nach ihrem "Wert" **sortierte Elemente** – typischerweise in einem balancierten Binärbaum. Das Sortierkriterium (Funktionsobjekt-Klasse) ist Template-Parameter (default : `less<>`)
Jedes Element darf **nur einmal** in einem Set vorkommen.

◇ Aufgrund der automatischen Sortierung in einem Binärbaum ermöglichen Sets ein **schnelles Suchen** und **Finden** von Elementen. Suchschlüssel sind die Elemente selbst.
Auch das **Einfügen** u/o. **Löschen** von Elementen besitzt an **allen Stellen** ein **gutes Zeitverhalten**.

◇ Die Elemente von Sets können auch **sequentiell durchlaufen** werden.
Ein direkter **wahlfreier Zugriff** zu einzelnen Elementen ist jedoch **nicht möglich**. Indexoperator und Direktzugriffs-Iteratoren sind daher nicht implementiert.
Beim Zugriff zu Elementen über Iteratoren werden die Elemente als Konstante betrachtet.

◇ Die Elemente können **nicht direkt geändert** werden. Eine Änderung könnte die Sortierung ungültig machen. Um ein Element zu ändern, muß es daher aus dem Set entfernt werden und nach der Änderung als neues Element wieder eingefügt werden.

◇ Die **Definition** des Klassen-Templates `set<>` befindet sich in der **Headerdatei `<set>`**

```
template <class Key, class Compare = less<Key>
          class Allocator = allocator<Key> >
  class set
  { // ...
  };
```

• Konstruktoren

`explicit set(const Compare& = Compare(), const Allocator& = Allocator());`	Erzeugung eines leeren Sets
`template <class InputIterator> set(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());`	Erzeugung eines zunächst leeren Sets, in das dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden
`set(const set<Key, Compare, Allocator>& x);`	Copy-Konstruktor

• Zusätzliche spezifische Memberfunktionen (Auswahl)

◇ Zusätzliche Methode zum Löschen eines Elements

`size_type erase(const Key& x);`	Löschen des Elements `x` Rückgabewert = - 1, falls gelöscht wurde (Element war enthalten) - 0, falls nicht gelöscht wurde (Element war nicht enthalten)

Standard-Template-Library (STL) von C++ : Klassen-Template `set` (2)

• Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)

◇ Methoden zum Einfügen von Elementen

`pair<iterator, bool> insert(const Key& x);`	Einfügen des Objekts `x` als neues Element, falls es noch nicht im Set vorhanden ist. **Rückgabewert** : Wertepaar mit Einfügeposition (Komponente `first`) und Einfügeerfolg (Komponente `second`) Einfügeerfolg = `true`, wenn eingefügt wurde
`iterator insert(iterator pos,` `  const Key& x);`	Einfügen des Objekts `x` als neues Element, falls es noch nicht im Set vorhanden ist. Der Parameter `pos` dient als ein Hinweis, wo im Set mit der Suche nach der Einfügeposition begonnen werden sollte. **Rückgabewert** : Einfügeposition bzw Position, an der sich das Element bereits befindet.

`template <class InputIterator>`
`  void insert(InputIterator first,`
`  InputIterator last);`
	Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers befindlichen Elemente.
	Ein Element wird nur eingefügt, wenn es noch nicht im Set vorhanden ist.

◇ Methoden zum Suchen

`iterator find(const Key& x) const;`	Suchen des Elements, das gleich dem Objekt `x` ist. **Rückgabewert** : - Position von `x`, falls vorhanden - `end()`, falls nicht vorhanden
`size_type count(const Key& x) const;`	Ermittlung, ob das Objekt `x` im Set enthalten ist **Rückgabewert** : - `1`, falls vorhanden - `0`, falls nicht vorhanden
`iterator lower_bound(const Key& x) const;`	Rückgabe der Position des ersten Elements im Set, das nicht kleiner als `x` (d.h. größer gleich `x`) ist
`iterator upper_bound(const Key& x) const;`	Rückgabe der Position des ersten Elements im Set, das größer als `x` ist
`pair<iterator, iterator>` `  equal_range(const Key& x) const;`	Rückgabe eines `pair`-Objekts aus `lower_bound(x)` und `upper_bound(x)`. `lower_bound(x)` und `upper_bound(x)` sind nur dann verschieden, wenn das Objekt `x` im Set enthalten ist

Standard-Template-Library (STL) von C++ : Klassen-Template `set` (3)

- **Einfaches Demonstrationsprogramm `setdem`**

```
// C++-Quelldatei setdem_m.cpp --> Programm setdem
// Einfaches Demo-Programm zum Klassen-Template set<> der STL

#include <set>
#include <iostream>
#include <string>
using namespace std;

typedef set<string, greater<string> > StringDownSet;
typedef set<string>                   StringSet;

template <class SetType>
void einfuegen(SetType& sds, const string& str)
{ pair<SetType::iterator, bool> success = sds.insert(str);
  cout << "\"" << str << "\"";
  if (success.second) cout << " erfolgreich eingefuegt\n";
  else cout << " bereits vorhanden\n";
}

ostream& operator<<(ostream& out, StringSet& menge)
{ StringSet::iterator loc;
  for (loc=menge.begin(); loc!=menge.end(); ++loc)
    out << *loc << ' ';
  return out << endl;
}


int main(void)
{ StringDownSet wmenge1;
  wmenge1.insert("man");
  wmenge1.insert("politikern");
  wmenge1.insert("kann");
  wmenge1.insert("tauben");
  wmenge1.insert("zutrauen");
  wmenge1.insert("sowie");
  einfuegen(wmenge1, "kann");
  einfuegen(wmenge1, "wesentliches");
  cout << endl;
  StringDownSet::iterator pos;
  for (pos=wmenge1.begin(); pos!=wmenge1.end(); ++pos)
    cout << *pos << ' ';
  cout << endl;

  #ifndef _MSC_VER              // fuer ANSI-C++ kompatible Compiler
  StringSet wmenge2(wmenge1.begin(), wmenge1.end());
  #else                         // Alternative fuer Visual-C++
  StringSet wmenge2;
  for (pos=wmenge1.begin(); pos!=wmenge1.end(); ++pos)
    wmenge2.insert(*pos);
  #endif

  cout << wmenge2;
  wmenge2.erase(wmenge2.find("sowie"), wmenge2.find("wesentliches"));
  cout << wmenge2;
  wmenge2.erase(wmenge2.begin(), wmenge2.find("politikern"));
  cout << wmenge2 << endl;
  einfuegen(wmenge2, "niemals");
  cout << endl;
  cout << wmenge2;
  return 0;
}
```

Standard-Template-Library (STL) von C++ : Klassen-Template `set` (4)

- **Ausgabe des Demonstrationsprogramms `setdem`**

```
"kann" bereits vorhanden
"wesentliches" erfolgreich eingefuegt

zutrauen wesentliches tauben sowie politikern man kann
kann man politikern sowie tauben wesentliches zutrauen
kann man politikern wesentliches zutrauen
politikern wesentliches zutrauen

"niemals" erfolgreich eingefuegt

niemals politikern wesentliches zutrauen
```

Standard-Template-Library (STL) von C++ : Klassen-Template `multiset`

- **Eigenschaften**

  ◇ Implementierung von **Multisets** (Multiset-Containern)
  Ein Multiset (-Container) entspricht einem Set mit dem Unterschied, dass Elemente auch mehrfach enthalten sein können.

  ◇ Die **Definition** des Klassen-Templates `multiset<>` befindet sich ebenfalls in der **Headerdatei `<set>`**

```
template <class Key, class Compare = less<Key>
          class Allocator = allocator<Key> >
  class multiset
  { // ...
  };
```

- **Konstruktoren**

`explicit multiset(const Compare& = Compare(), const Allocator& = Allocator());`	Erzeugung eines leeren Multisets
`template <class InputIterator> multiset(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());`	Erzeugung eines zunächst leeren Multisets, in das dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden
`multiset(const multiset<Key, Compare, Allocator>& x);`	Copy-Konstruktor

- **Anmerkungen zu den zusätzlichen spezifischen Memberfunktionen**

  ◇ Das Klassen-Template `multiset` besitzt die **gleichem Memberfunktionen** wie das Klassen-Template **`set`.**
  Sie **unterscheiden** sich lediglich durch die **Auswirkungen** des möglichen **Mehrfachenthaltenseins** von Elementen.

  ◇ Im wesentlichen bestehen die **folgenden Unterschiede** :

  ▻ **`erase`**`(x)` Löschen aller Elemente, die gleich dem Objekt `x` sind.

  ▻ **`insert`**`(x)` Der Rückgabewert ist vom Typ `iterator` (statt `pair<...>`)
  Einfügen des Objekts `x` als neues Element auch dann, wenn es bereits im Multiset enthalten ist. Es wird immer die Einfügeposition zurückgegeben.

  ▻ **`insert`**`(pos, x)` Einfügen des Objekts `x` als neues Element auch dann, wenn es bereits im Multiset enthalten ist. Der Rückgabewert ist immer die Einfügeposition

  ▻ **`insert`**`(first, last)` Einfügen immer aller Elemente aus dem Bereich `first` (einschl.) bis `last` (ausschl.), auch die, die bereits vorhanden sind.

  ▻ **`count`**`(x)` Rückgabe der Anzahl Elemente, die gleich dem Objekt `x` sind

  ▻ **`find`**`(x)` Suchen des ersten Elements, das gleich dem Objekt `x` ist.

Standard-Template-Library (STL) von C++ : Klassen-Template `map` (1)

- **Eigenschaften**

  ◇ Implementierung von **Maps** (Map-Containern)
  Ein(e) Map (-Container) speichert **Schlüssel-/Werte-Paare** als Elemente (Klasse **`pair<const Key, T>`**).
  Die Elemente sind nach dem **(Such-)Schlüssel** (Klasse `Key`) **sortiert**. Der Suchschlüssel dient zum Auffinden des mit ihm assoziierten Werts (das eigentliche verwaltete Objekt, Klasse `T`). Jeder **Suchschlüssel** darf **nur einmal** in einer Map vorkommen, d.h. mit einem Suchschlüssel kann nur ein einziger Wert assoziiert sein ("***one-to-one-mapping***")

  ◇ Auch eine Map verwaltet ihre Elemente typischerweise in einem balancierten Binärbaum. Das Sortierkriterium (Funktionsobjekt-Klasse) ist ebenfalls Template-Parameter (default : `less<>`)

  ◇ Der **Schlüssel** eines Elements darf **nicht geändert** werden, wohl **aber** sein **Wert**.

  ◇ Die übrigen Eigenschaften (bezüglich Zeitverhalten bei Suchen/Finden und Einfügen/Löschen sowie den Zugriffsmöglichkeiten zu den Elementen) entsprechen denen eines Sets, mit dem Unterschied, dass die enthaltenen Elemente Paare sind.

  ◇ Die **Definition** des Klassen-Templates `map<>` befindet sich in der **Headerdatei `<map>`**

```
template <class Key, class T,
          class Compare = less<Key>
          class Allocator = allocator<pair<const Key,T> > >
  class map
  { // ...
  };
```

- **Konstruktoren**

`explicit map(const Compare& = Compare(), const Allocator& = Allocator());`	Erzeugung einer leeren Map
`template <class InputIterator> map(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());`	Erzeugung einer zunächst leeren Map, in die dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden Diese Elemente müssen vom Typ `pair<const Key, T>` sein.
`map(const map<Key, T, Compare, Allocator>& x);`	Copy-Konstruktor

- **Zusätzliche spezifische Memberfunktionen (Auswahl)**

  ◇ **Zusätzliche Methode zum Löschen eines Elements**

`size_type erase(const Key& k);`	Löschen des Elements (Schlüssel-/Wert-Paar) mit dem Schlüssel `k` Rückgabewert = - 1, falls gelöscht wurde (Element mit Schlüssel `k` war enthalten) - 0, falls nicht gelöscht wurde (kein Element mit Schlüssel `k` enthalten.)

Standard-Template-Library (STL) von C++ : Klassen-Template `map` (2)

• Zusätzliche spezifische Memberfunktionen (Auswahl, Forts.)

◇ Methoden zum Einfügen von Elementen

`pair<iterator, bool>` `  insert(pair<const Key, T>& x);`	Einfügen des Schlüssel-/Werte-Paares `x` als neues Element, falls noch kein Element mit dem Schlüssel `x.first` in der Map vorhanden ist. **Rückgabewert** : Wertepaar mit Einfügeposition (Komponente `first`) und Einfügeerfolg (Komponente `second`). Einfügeerfolg = `true`, wenn eingefügt wurde
`iterator insert(iterator pos,` `  pair<const Key, T>& x);`	Einfügen des Schlüssel-/Werte-Paares `x` als neues Element, falls noch kein Element mit dem Schlüssel `x.first` in der Map vorhanden ist. Der Parameter `pos` dient als Hinweis, wo in der Map mit der Suche nach der Einfügeposition begonnen werden sollte. **Rückgabewert** : Einfügeposition bzw Position, an der sich ein Element mit dem Schlüssel bereits befindet.

`template <class InputIterator>`
`  void insert(InputIterator first,`
`  InputIterator last);`
	Einfügen von Kopien der zwischen den Positionen `first` (einschl.) und `last` (ausschl.) eines anderen Containers (i.a. einer Map oder Multimap) befindlichen Elemente.
	Ein Element wird nur eingefügt, wenn es noch kein Element mit seinem Schlüssel in der Map gibt.

◇ Methoden zum Suchen

`iterator find(const Key& k);` `const_iterator find(const Key& k) const;`	Suchen des Elements, das den Schlüssel `k` hat. **Rückgabewert** : - Position des Elements, falls vorhanden - `end()`, falls nicht vorhanden
`size_type count(const Key& k) const;`	Ermittlung, ob ein Element mit dem Schlüssel `k` in der Map enthalten ist **Rückgabewert** : - `1`, falls vorhanden - `0`, falls nicht vorhanden
`iterator lower_bound(const Key& k);` `const_iterator lower_bound(const Key& k) const;`	Rückgabe der Position des ersten Elements in der Map, dessen Schlüssel nicht kleiner als `k` (d.h. größer gleich `k`) ist
`iterator upper_bound(const Key& k);` `const_iterator upper_bound(const Key& k) const;`	Rückgabe der Position des ersten Elements in der Map, dessen Schlüssel größer als `k` ist

`pair<iterator, iterator>`
`  equal_range(const Key& k);`
 	
`pair<const_iterator, const_iterator>`
`  equal_range(const Key& k) const;`
	Rückgabe eines `pair`-Objekts aus `lower_bound(k)` und `upper_bound(k)`. `lower_bound(k)` und `upper_bound(k)` sind nur dann verschieden, wenn ein Element mit dem Schlüssel `k` in der Map enthalten ist

Standard-Template-Library (STL) von C++ : Klassen-Template `map` (3)

- **Zusätzliche spezifische Memberfunktionen (Auswahl, weitere Forts.)**

  ◇ **Methode zum direkten Elementzugriff (Indexoperator)**

`T& operator[](const Key& k);`	Ermittlung der Wert-Komponente des Elements mit dem Schlüssel `k`. Falls kein Element mit dem Schlüssel `k` existiert, wird ein neues Element angelegt (dessen Wert-Komponente mit ihrem Default-Konstruktor initialisiert wird).

Diese Methode macht eine Map zur Implementierung eines **assoziativen Arrays**.
Die Auswahl eines "Array"-Elements erfolgt über einen Teil seines Inhalts (hier den Schlüssel `k`).
Der Auswahl-"Index" kann damit von einem beliebigen (und nicht nur einem ganzzahligen) Typ sein.

Interessant ist, dass es **keinen unerlaubten** "Wert" für den **Index** gibt.
Existiert kein Element für den Index, wird ein neues Element erzeugt.

Beispiel :

```
// ...
map<string, float> preisliste;   // leere Map
preisliste["Uhr"]=24.50;
// ...
```

→ Es wird ein neues Element mit dem Schlüssel `"Uhr"` angelegt.
Eine Referenz auf die Wert-Komponente diese Elements, die zunächst undefiniert ist, wird von der Index-Operatorfunktion zurückgegeben.
Anschliessend wird dieser Komponente der Wert `24.50` zugewiesen.

- **Einige innere Datentypen**

  ◇ Innerhalb des Klassen-Templates `map` sind – wie in allen Container-Klassen – mehrere Datentypen definiert.
  Es handelt sich bei ihnen um `public`-Komponenten

  ◇ Einige dieser Datentypen sind :

```
template <class Key, class T,
          class Compare = less<Key>
          class Allocator = allocator<pair<const Key,T> > >
  class map
  { public :
      typedef Key                    key_type;
      typedef T                      mapped_type;
      typedef pair<const Key, T>     value_type;
      // ...
  };
```

- **Möglichkeiten zur Erzeugung von Schlüsel-/Werte-Paaren (z.B. als Parameter für `insert(...)`)**

  ◇ mit **`pair<>`** : z.B. **`pair<const string, double>("Fahrrad", 259.00);`**
  Vereinfachung : Definition eines eigenen Typnamens für den `pair`-Typ.
  z.B. `typedef pair<const string, double>` **`StrDoubPair;`**
  `StrDoubPair("Fahrrad", 259.00);`

  ◇ mit **`value_type`** : z.B. **`map<string, double>::value_type("Fahrrad", 259.00);`**

  ◇ mit **`make_pair()`** : z.B. **`make_pair(string("Fahrrad"), 259.00);`**
  Achtung : Die Typen von `pair` müssen eindeutig aus den aktuellen Parametern von `make_pair()` erkennbar sein

Standard-Template-Library (STL) von C++ : Klassen-Template `multimap (1)`

- **Eigenschaften**

  ◇ Implementierung von **Multimaps** (Multimap-Containern)
  Ein Multimap (-Container) entspricht einer Map mit dem Unterschied, dass mehrere Elemente mit dem gleichen Schlüssel enthalten sein können ("***one-to- many mapping***")

  ◇ Die **Definition** des Klassen-Templates `multimap<>` befindet sich ebenfalls in der **Headerdatei `<map>`**

```
template <class Key, class T,
          class Compare = less<Key>
          class Allocator = allocator<pair<const Key,T> > >
  class multimap
  { // ...
  };
```

- **Konstruktoren**

`explicit multimap(const Compare& = Compare(), const Allocator& = Allocator());`	Erzeugung einer leeren Multimap
`template <class InputIterator> multimap(InputIterator first, InputIterator last, const Compare& =Compare(), const Allocator& = Allocator());`	Erzeugung einer zunächst leeren Multimap, in die dann die zwischen den Positionen `first` (einschl.) und `last` (ausschl.) befindlichen Elemente eines anderen Containers eingefügt werden. Diese Elemente müssen vom Typ `pair<const Key, T>` sein.
`multimap(const multimap<Key, T, Compare, Allocator>& x);`	Copy-Konstruktor

- **Anmerkungen zu den zusätzlichen spezifischen Memberfunktionen**

  ◇ Die Indexoperatorfunktion **`operator[]()`** ist für das Klassen-Template `multimap` **nicht definiert**.
  Da mehrere Elemente mit dem gleichen Schlüssel vorkommen können, kann dieser nicht zur Elementauswahl verwendet werden. → Multimaps eignen sich nicht als assoziative Arrays.

  ◇ Im übrigen besitzt das Klassen-Template `multimap` die **gleichen Memberfunktionen** wie **`map`.**
  Sie **unterscheiden** sich lediglich durch die **Auswirkungen** des möglichen **Mehrfachenthaltenseins** von Schlüsseln.

  ◇ Im wesentlichen bestehen die **folgenden Unterschiede** :

  ▻ **`erase`**`(k)` Löschen aller Elemente, die dem Schlüssel `k` besitzen.

  ▻ **`insert`**`(x)` Der Rückgabewert ist vom Typ `iterator` (statt `pair<...>`)
  Einfügen des Schlüssel-/Wert-Paares `x` als neues Element auch dann, wenn bereits ein Element mit dem Schlüssel `x.first` in der Multimap enthalten ist.
  Es wird immer die Einfügeposition zurückgegeben.

  ▻ **`insert`**`(pos, x)` Einfügen des Schlüssel-/Wert-Paares `x` als neues Element auch dann, wenn bereits ein Element mit dem Schlüssel `x.first` in der Multimap enthalten ist.
  Der Rückgabewert ist immer die Einfügeposition

  ▻ **`insert`**`(first, last)` Einfügen immer aller Elemente aus dem Bereich `first` (einschl.) bis `last` (ausschl.), auch die, deren Schlüssel bereits vorhanden sind.

  ▻ **`count`**`(k)` Rückgabe der Anzahl Elemente, die den Schlüssel `k` besitzen.

  ▻ **`find`**`(k)` Suchen des ersten Elements, dessen Schlüssel gleich `k` ist.

Standard-Template-Library (STL) von C++ : Klassen-Template `multimap (2)`

- **Einfaches Demonstrationsprogramm `multimapdem`**

```
// C++-Quelldatei multimapdem_m.cpp --> Programm multimapdem
// Einfaches Demo-Programm zum Klassen-Template multimap<> der STL

#include <map>
#include <string>
#include <iostream>
#include <utility>
#include <iomanip>
using namespace std;

typedef multimap<string, string>   StrStrMMap;
typedef pair<const string, string> StringPaar;


int main(void)
{
  StrStrMMap dict;
  dict.insert(make_pair(string("clever"), string("klug")));
  dict.insert(pair<const string, string>("clever", "gewandt"));
  dict.insert(StrStrMMap::value_type("clever", "raffiniert"));
  dict.insert(StringPaar("smart", "klug"));
  dict.insert(StringPaar("smart", "gewandt"));
  dict.insert(StringPaar("smart", "elegant"));
  dict.insert(StringPaar("wise", "klug"));
  dict.insert(StringPaar("wise", "erfahren"));
  dict.insert(StringPaar("strange", "fremd"));
  dict.insert(StringPaar("strange", "seltsam"));
  dict.insert(StringPaar("odd", "seltsam"));
  dict.insert(StringPaar("odd", "ungerade"));
  dict.insert(StringPaar("quick", "schnell"));
  dict.insert(StringPaar("quick", "gewandt"));
  dict.insert(StringPaar("car", "Auto"));
  dict.insert(StringPaar("again", "nochmals"));
  dict.insert(StringPaar("ship", "Schiff"));

  // Ausgabe aller Eintraege (Schluessel-Werte-Paare)
  cout << "Enthalten sind " << dict.size() << " Eintraege\n" << left << endl;
  StrStrMMap::iterator pos;
  for (pos=dict.begin(); pos!=dict.end(); ++pos)
    cout << "english : " << setw(15) << pos->first
         << "deutsch : " << setw(15) << pos->second << endl;
  cout << right << endl;;

  // Ausgabe aller Werte zu einenm bestimmten Schluessel
  string wort("clever");
  cout << wort << " : " << dict.count(wort) << " Eintraege" << endl;
  for (pos=dict.lower_bound(wort);
       pos!=dict.upper_bound(wort) ; ++pos)
    cout << "   " << pos->second << endl;
  cout << endl;

  // Ausgabe aller Schluessel zu einem bestimmten Wert
  wort="klug";
  cout << wort << " : " <<endl;
  for (pos=dict.begin(); pos!=dict.end(); ++pos)
    if (pos->second==wort)
      cout << "   " << pos->first << endl;
  cout << endl;

  return 0;
}
```

Standard-Template-Library (STL) von C++ : Klassen-Template `multimap (3)`

- **Ausgabe des Demonstrationsprogramms `multimapdem`**

```
Enthalten sind 17 Eintraege

english : again          deutsch : nochmals
english : car            deutsch : Auto
english : clever         deutsch : klug
english : clever         deutsch : gewandt
english : clever         deutsch : raffiniert
english : odd            deutsch : seltsam
english : odd            deutsch : ungerade
english : quick          deutsch : schnell
english : quick          deutsch : gewandt
english : ship           deutsch : Schiff
english : smart          deutsch : klug
english : smart          deutsch : gewandt
english : smart          deutsch : elegant
english : strange        deutsch : fremd
english : strange        deutsch : seltsam
english : wise           deutsch : klug
english : wise           deutsch : erfahren

clever : 3 Eintraege
   klug
   gewandt
   raffiniert

klug :
   clever
   smart
   wise
```

Standard-Template-Library (STL) von C++ : Iteratoren (1)

• Einführung

◇ **Iteratoren** sind Objekte, die einen Mechanismus zum **Durchlaufen** von **Container-Objekten** zur Verfügung stellen.

◇ Die in der STL implementierten Iteratoren ermöglichen einen **zeigerähnlichen Zugriff** zu den einzelnen **Elementen** eines Containers. Analog zu einem Zeiger, der die Position eines Array-Elements angibt, **repräsentiert** ein derartiges **Iterator-Objekt** die **Position** eines **Container-Elements**.
Solche Iteratoren können somit als **Verallgemeinerung von Zeigern** aufgefasst werden.

◇ Da Iteratoren **Zustandsinformationen** des jeweiligen **Containers**, auf dem sie arbeiten, auswerten und die **Organisationsform** seiner Elemente berücksichtigen müssen, ist eine **Iterator-Klasse** immer an eine **bestimmte Container-Klasse gebunden**.
D.h. zu jeder Container-Klasse gehört jeweils eine **spezifische** passende **Iterator-Klasse**. Typischerweise ist diese als **innere Klasse** der Containerklasse definiert. Bei den Iteratoren der STL ist das immer der Fall.

◇ Alle **Iterator-Klassen** der STL stellen – unabhängig von der Container-Klasse, an die sie jeweils gebunden sind und unabhängig von ihrer eigenen Implementierung – eine **einheitliche Schnittstelle** zur Verfügung.
Diese wird durch **Operator-Funktionen** gebildet, die die für Zeiger anwendbaren Operationen implementieren (wie Dereferenzierung, Incrementierung usw).

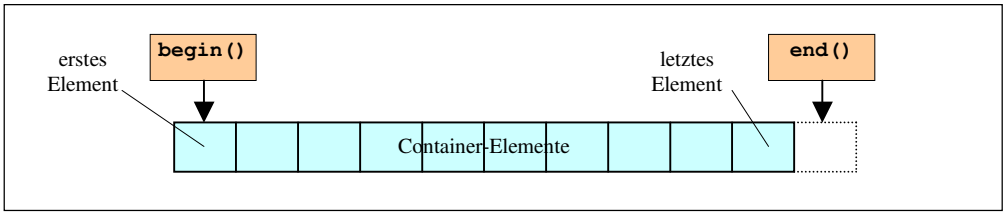
◇ Die **Container-Klassen** ihrerseits stellen eine **einheitliche Schnittstelle** zur **Bereitstellung** und **Verwendung** der **Iteratoren** bereit :

▻ In **jeder Container-Klasse** – ausser den Container-Adaptern – sind die implementierungsabhängigen `public`-Typen **`iterator`** und **`const_iterator`** definiert.
Container-Elemente, die von Iterator-Objekten des Typs `const_iterator` referiert werden, können nicht geändert werden.

```
template < ... >
  class ...
  {
    public :
       // ...
       typedef implementation defined iterator;
       typedef implementation defined const_iterator;
       // ...
  };
```

▻ **Jede Container-Klasse** – ausser den Container-Adaptern – stellt **Memberfunktionen** zur Ermittlung des **Iterator-bereichs** eines Container-Objekts zu Verfügung. :

- **`begin()`** liefert den Iterator, der auf das erste Element im Container zeigt
- **`end()`** liefert einen Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt.



Der Bereich **`[ begin(), end() )`** bildet also ein **halboffenes Intervall**, über das durch alle Container-Elemente iteriert werden kann.
Ist `begin()==end()`, ist das Intervall leer, d.h. der Container enthält keine Elemente.

Beide Funktionen existieren jeweils in **zwei überladenen Formen**, die eine liefert ein Iterator-Objekt vom Typ `iterator`, die andere ein Iterator-Objekt vom Typ `const_iterator`.

◇ Diese einheitlichen Schnittstellen ermöglichen es, dass die **Objekte** der **unterschiedlichen Container-Klassen**, die **unterschiedliche Datenstrukturen** implementieren, **gleichartig bearbeitet** werden können.

Standard-Template-Library (STL) von C++ : Iteratoren (2)

• Überblick über die Iterator-Bibliothek

◇ Die in der Iterator-Bibliothek enthaltenen Definitionen **iterator-spezifischer Datentypen** und **Funktionen** der STL sind in der Headerdatei **`<iterator>`** zusammengefasst.

◇ Im wesentlich handelt es sich hierbei um

- ▻ **grundlegende Datentypen**, die im Container-Teil der STL für die Definition der container-spezifischen Iterator-Klassen verwendet werden, sowie zur Definition eigener Iterator-Klassen eingesetzt werden können.
  U.a. ist hier auch das als **Iterator-Basisklasse** dienende Klassen-Template **`iterator`** definiert
- ▻ **freie Iterator-Funktionen** (Funktions-Templates), die bestimmte Operationen auf Iterator-Objekte ausführen.
- ▻ **Iterator-Adapter**, durch die Iterator-Klassen mit spezifischen Eigenschaften vordefiniert werden.
  Sie erweitern die Anwendungsmöglichkeiten der Algorithmen der STL erheblich.

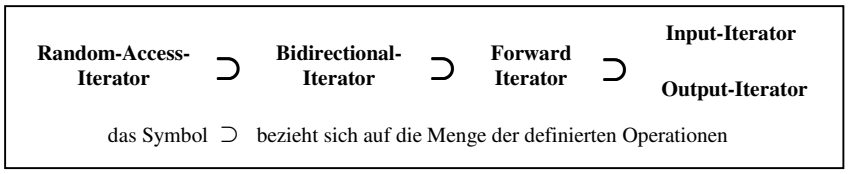
  Hierbei handelt es sich um
  - **Reverse-Iteratoren**, mit denen die Durchlaufrichtung von Container umgekehrt wird.
  - **Insert-Iteratoren**, mit denen sich die Kopier-Algorithmen der STL zum Einfügen (statt zum normalerweise stattfindenden Überschreiben) einsetzen lassen.
  - **Stream-Iteratoren** und **Streambuffer-Iteratoren**, mit denen zu Eingabe- bzw Ausgabe-Streams wie zu Containern zugegriffen werden kann.
    Das Lesen und Schreiben von Daten lässt sich damit unter Verwendung von Iteratoren durchführen.

◇ Zur Arbeit mit Iteratoren muss die Headerdatei `<iterator>` in der Regel aber nicht explizit eingebunden werden, da sie von allen Headerdateien für Container und der Headerdatei für Algorithmen bereits eingebunden wird.

• Iterator-Kategorien

◇ In Abhängigkeit von den auf Iteratoren **anwendbaren Operationen** werden **fünf Iterator-Kategorien** unterschieden :

- ▻ Input-Iteratoren
- ▻ Output-Iteratoren
- ▻ Forward-Iteratoren
- ▻ Bidirectional-Iteratoren
- ▻ Random-Access-Iteratoren



◇ **Input-Iteratoren** (InpIter, *input iterators*)
Sie erlauben lediglich einen **lesenden Zugriff** auf das durch sie referierte Element des Containers, sowie ein Durchlaufen des Containers (der Sequenz) in Vorwärtsrichtung (Implementierung des Increment-Operators). Dabei ist mit einem Input-Iterator nur ein einmaliger Durchlauf möglich, d. h. er kann nur für *one-pass algorithms* eingesetzt werden
Ausserdem lassen sich auf Input-Iteratoren der Zuweisungs-Operator, sowie der Gleichheits- und der Ungleichheits-Operator anwenden.

◇ **Output-Iteratoren** (OutpIter, *output iterators*)
Sie erlauben lediglich einen **schreibenden Zugriff** auf das durch sie referierte Container-Element, sowie ein Durchlaufen des Containers (der Sequenz) in Vorwärtsrichtung. Auch ein Output-Operator gestattet nur einen einmaligen Durchlauf.
Der Zuweisungs-Operator lässt sich auf sie anwenden, nicht jedoch der Gleichheits- und der Ungleichheits-Operator

◇ **Forward-Iteratoren** (ForwIter, *forward iterators*)
Sie kombinieren die Fähigkeiten von Input- und Output-Operatoren. Zusätzlich kann mit dem gleichen Iterator eine Elemente-Menge auch mehrfach durchlaufen werden, d.h. diese Iteratoren unterstützen *multi-pass algorithms*.

◇ **Bidirectional-Iteratoren** (BidirIter, *bidirectional iterators*)
Sie besitzen alle Fähigkeiten der Vorwärts-Iteratoren. Zusätzlich ermöglichen sie ein Durchlaufen des Containers in Rückwärtsrichtung (Implementierung des Decrement-Operators)

◇ **Random-Access-Iteratoren** (RanAccIter, *random access iterators*)
Sie besitzen alle Fähigkieten der Bidirectional-Iteratoren. Zusätzlich erlauben sie einen direkten wahlfreien Zugriff zu jedem Element des Containers. Hierzu implementieren sie den Index-Operator sowie eine "Adress-Arithmetik".
Ausserdem können auf diese Iteratoren auch die Vergleichs-Operationen `>, >=, <, <=` angewendet werden.
Damit stellen sie die **volle gleiche Funktionalität** wie normale typgebundene **Pointer** zu Verfügung.

Standard-Template-Library (STL) von C++ : Iteratoren (3)

• Überblick über die Iterator-Operationen

◇ In der folgenden Übersicht
- sind `p` und `q` Iteratoren
- bedeutet `X` : die Operation steht für die jeweilige Iterator-Kategorie zur Verfügung

Operation	Bemerkung	InpIter	OutpIter	ForwIter	BidirIter	RanAccIter
Kopieren						
`p=q`	Zuweisung	X	X	X	X	X
Dereferenzieren						
`x=*p`	Lesen (Rvalue)	X		X	X	X
`x=p->k`	`x=(*p).k`	X		X	X	X
`*p=x`	Schreiben (Lvalue)		X	X	X	X
Incrementieren						
`++p`	Ergebnis= neuer Wert	X	X	X	X	X
`p++`	Ergebnis=alter Wert	X	X	X	X	X
Decrementieren						
`--p`	Ergebnis=neuer Wert				X	X
`p--`	Ergebnis=alter Wert				X	X
Vergleichen						
`p==q`	gleich	X		X	X	X
`p!=q`	ungleich	X		X	X	X
`p<q`	kleiner					X
`p<=q`	kleiner gleich					X
`p>q`	grösser					X
`p>=q`	grösser gleich					X
Direktzugriff						
`p=p+n`	`n` Positionen nach `p`					X
`p+=n`	`n` Positionen nach `p`					X
`p=p-n`	`n` Positionen vor `p`					X
`p-=n`	`n` Positionen vor `p`					X
`x=p[n]`	`x=*(p+n)`					X
`p[n]=x`	`*(p+n)=x`					X
Iteratordistanz						
`n=p-q`	Entfernung in Anz. Positionen					X

◇ **Anmerkung zum Incrementieren/Decrementieren** :
Da der **Pre-Increment**- und der **Pre-Decrement-Operator** den aktuellen – neuen – Wert (und nicht den alten Wert) zurückliefern, lassen sie sich wesentlich **effizienter** als die entsprechenden Post-Operatoren **implementieren**.
Man sollte sie daher gegenüber diesen bevorzugt einsetzen : **`++p`** ist **effizienter** als `p++`.

• Iterator-Kategorien der STL-Container-Klassen

◇ Die STL-Containerklassen **`vector`** und **`deque`** unterstützen **Random-Acces-Iteratoren**

◇ Die STL-Containerklassen **`list`**, **`set`**, **`multiset`**, **`map`** und **`multimap`** unterstützen **Bidirectional-Iteratoren**

Standard-Template-Library (STL) von C++ : Iteratoren (4)

- **Einfaches Demonstrationsprogramm zu Random-Access-Iteratoren `randiterdem`**

```
// C++-Quelldatei randiterdem_m.cpp --> Programm randiterdem
// Einfaches Demo-Programm zu Random-Access-Iteratoren der STL

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


int main(void)
{
  vector<int> zahlen;
  int i;
  vector<int>::iterator pos;

  for (i=1; i<=10; i++)
    zahlen.push_back(i);

  cout << "\nAnzahl : " << zahlen.end()- zahlen.begin() << endl;

  cout << "Inhalt :\n";
  cout << "Verwendung *-Operator\n";
  for (pos=zahlen.begin(); pos<zahlen.end(); ++pos)
    cout <<*pos << "  ";
  cout << endl;

  cout << "Zugriff ueber Index-Operator\n";
  for (i=0; i<zahlen.size(); i++)
    cout << zahlen.begin()[i] << "  ";;
  cout << endl;

  cout << "jedes 2. Element\n";
  for (pos=zahlen.begin(); pos<zahlen.end(); pos+=2)
    cout <<*pos << "  ";
  cout << endl;


  return 0;
}
```

- **Ausgabe des Demonstrationsprogramms `randiterdem`**

```
Anzahl : 10
Inhalt :
Verwendung *-Operator
1  2  3  4  5  6  7  8  9  10
Zugriff ueber Index-Operator
1  2  3  4  5  6  7  8  9  10
jedes 2. Element
1  3  5  7  9
```

Standard-Template-Library (STL) von C++ : Iteratoren (5)

• Freie Iterator-Funktionen

◇ Die STL stellt **zwei freie Funktionen** zur Verfügung, mit denen einige nur für Random-Access-Iteratoren definierte Operationen auch für Input-Iteratoren (und damit auch für Bidirectional- und Forward-Iteratoren) ermöglicht werden.

◇ Die beiden als **Funktions-Templates** in der Headerdatei **<iterator>** definierten Funktionen werden nachfolgend in **vereinfachter Darstellung** deklariert.

◇ **`void advance(InputIterator& pos, Dist n);`**

- ▻ **Weitersetzen** des (Input-)**Iterators** `pos` um `n` Elemente
- ▻ Für Bidirectional- und Random-Access-Iteratoren darf `n` auch negativ sein.
- ▻ Die Typen `InputIterator` und `Dist` sind tatsächlich Template-Parameter.
  Da alle Iteratoren, ausser den Output-Iteratoren, die Funktionalität von Input-Iteratoren implementieren, kann die aktuelle Iteratorklasse eine Input-, Forward-, Bidirectional- oder Random-Access-Iterator-Klasse sein.
  Der den formalen Typ-Parameter `Dist` ersetzende aktuelle Typ muss ein für die jeweilige aktuelle Iteratorklasse anwendbarer Ganzzahltyp zur Darstellung von "Iteratorabständen" sein.

◇ **`Dist distance(InputIterator first, InputIterator last);`**

- ▻ **Ermittlung des Abstands** zwischen den (Input-)Iteratoren `first` und `last`.
  = Anzahl der Increments bzw Decrements, um von `first` nach `last` zu gelangen.
- ▻ Beide Iteratoren müssen zum gleichen Container ghören.
- ▻ Der Iterator `last` muss vom Iterator `first` aus erreichbar sein.
  Das bedeutet, dass bei allen Iteratoren, die keine Random-Access-Iteratoren sind, `last` hinter `first` liegen muss (d.h. über Increments erreichbar sein muss).
- ▻ Der Typ `InputIterator` ist tatsächlich Template-Parameter.
  Aktuell kann es sich um eine Iteratorklasse der Kategorien Input-Iterator, Forward-Iterator, Bidirectional-Iterator oder Random-Access-Iterator handeln, da alle Iteratoren dieser Klassen die Funktionalität von Input-Iteratoren implementieren.
- ▻ Der Rückgabetyp `Dist` ist der für die jeweilige aktuelle Iteratorklasse definierte Ganzzahltyp zur Darstellung von "Iteratorabständen" ("Abstands-Typ" der Iteratorklasse).

◇ Eine **weitere freie Iterator-Funktion** ist in der Algorithmen-Bibliothek (Headerdatei **<algorithm>**) definiert.
Es handelt sich um eine Funktion, mit der die von zwei Iteratoren referierten **Elemente vertauscht** werden können.
Auch diese Funktion ist als Funktions-Template definiert.
Ihre Deklaration wird nachfolgend ebenfalls in vereinfachter Darstellung angegeben.

◇ **`void iter_swap(ForwardIterator1 a, ForwardIterator2 b);`**

- ▻ **Vertauschen** der durch die Iteratoren `a` und `b` **referierten Elemente**.
- ▻ Die Typen `ForwardIterator1` und `ForwardIterator2` sind tatsächlich Template-Parameter.
  Aktuell kann für sie jede Iteratorklasse der Kategorien Forward-Iterator, Bididrectional-Iterator oder Random-Access-Iterator verwendet werden.
- ▻ Die beiden Iteratorklassen müssen nicht gleich sein. Die referierten Elemente müssen sich lediglich gegenseitig zuweisen lassen.

Standard-Template-Library (STL) von C++ : Reverse-Iteratoren (1)

- **Allgemeines**

  ◇ In der STL ist – in der Headerdatei **`<iterator>`** – der **Iterator-Adapter `reverse_iterator`** als Klassen-Template definiert. Template-Parameter ist eine Iterator-Klasse.
  Dieser Iterator-Adapter erzeugt **Reverse-Iteratoren**

  ◇ Reverse-Iteratoren sind Iteratoren, bei denen die **Durchlaufrichtung** durch einen Container **umgekehrt** ist, d.h. ein **Incrementieren** bewegt den Iterator zum **vorhergehenden Element**.
  Mit ihrer Hilfe kann bei allen Algorithmen die Verarbeitungsreihenfolge der Elemente umgekehrt werden, ohne dass entsprechende neue Algorithmen implementiert werden müssen.

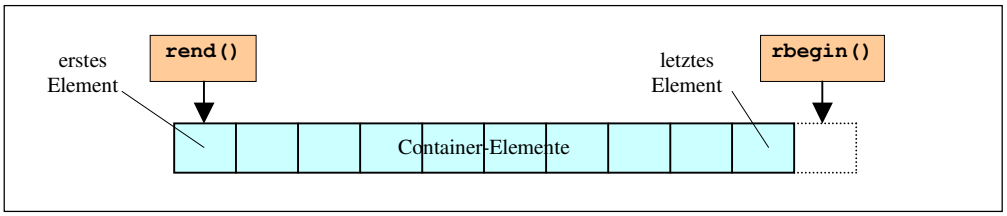
  ◇ Reverse-Iteratoren können nur zu Bidirectional- (und damit auch zu Random-Access-) Iteratoren gebildet werden.
  Da **alle in der STL definierten Container-Klassen** Iteratoren dieser Kategorien zu unterstützen, lassen sich für sie auch Reverse-Iteratoren erzeugen.

  ▻ In **jeder Container-Klasse** – ausser den Container-Adaptern – sind deshalb auch die implementierungsabhängigen `public`-Typen **`reverse_iterator`** und **`const_reverse_iterator`** definiert.
  Container-Elemente, die von Reverse-Iterator-Objekten des Typs `const_reverse_iterator` referiert werden, können nicht geändert werden.

```
template < ... >
  class ...
  {
    public :
       // ...
       typedef std::reverse_iterator<iterator> reverse_iterator;
       typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
       // ...
  };
```

  ▻ Ausserdem stellt **jede Container-Klasse** – ausser den Container-Adaptern – auch **Memberfunktionen** zur Ermittlung des **Reverse-Iteratorbereichs** eines Container-Objekts zu Verfügung. :

  - **`rbegin()`** liefert den Reverse-Iterator, der auf die Position unmittelbar nach dem letzten Element zeigt
    Dies ist der in einen Reverse-Iterator umgewandelte Funktionswert von `end()`.
  - **`rend()`** liefert einen Reverse-Iterator, der auf das erste Element im Container zeigt.
    Dies ist der in einen Reverse-Iterator umgewandelte Funktionswert von `begin()`.



  Auch diese **beiden Funktionen** existieren jeweils in **zwei überladenen Formen**, die eine liefert ein Reverse-Iterator-Objekt vom Typ `reverse_iterator,` die andere ein Reverse-Iterator-Objekt vom Typ `const_reverse_iterator.`

  ◇ Bei der **Erzeugung eines Reverse-Iterators** aus einem – normalen – Iterator **bleibt** die **Iterator-Position gleich**.
  Um beim Durchlaufen von Containers-Bereichen mit Reverse-Iteratoren die **gleiche Semantik** wie mit – normalen – Iteratoren verwenden zu können (halboffenes Intervall, letzter Iterator zeigt auf nicht mehr vorhandenes Element), wird bei der **Dereferenzierung eines Reverse-Iterators** das **Element** zurückgeliefert, das sich an der **Position unmittelbar vor der eigentlich referierten Position** befindet.

Standard-Template-Library (STL) von C++ : Reverse-Iteratoren (2)

- **Definition des Klassen-Templates `reverse_iterator` (unvollständiger Auszug)**

```
template <class Iterator>
  class reverse_iterator : public iterator< ... >
{
  protected :
    Iterator current;
  public :
    // Typ-Definitionen

    reverse_iterator();
    explicit reverse_iterator(Iterator x);
    template <class U> reverse_iterator(const reverse_iterator<U>& u);

    Iterator base() const;

    // Operatorfunktionen zum Element-Zugriff und zur Iterator-Aenderung

};

// freie Vergleichs-Operatorfunktionen
```

- **Konstruktoren**
  - ▻ Konstruktor zur Erzeugung eines **nichtinitialisierten** Reverse-Iterator-Objekts.
  - ▻ Konstruktor zur Erzeugung eines Reverse-Iterator-Objekts, das **mit einem** – normalen – **Iterator-Objekt initialisiert** wird
  - ▻ Copy-Konstruktor

- **Operationen mit Reverse-Iteratoren**
  - ◇ Für Reverse-Iteratoren sind die **gleichen Operationen** wie für die entsprechenden – normalen – Iteratoren definiert.
  - ◇ Allerdings sind **einige Wirkungs-Unterschiede** zu beachten :
    - ▻ Ein **Incrementieren** des Reverse-Iterators wird in ein **Decrementieren umgesetzt** (statt `++p` → `--p`) und umgekehrt (`--p` → `++p`).
    - ▻ Eine Veränderung eines Reverse-Iterators um einen **positiven Abstand** wird in eine Veränderung um einen **negativen Abstand umgesetzt** und umgekehrt.
    - ▻ Die **Dereferenzierung** eines Inverse-Iterators führt zur Rückgabe des **unmittelbar davor befindlichen Elements**
    - ▻ Bei **Vergleichsoperationen** werden die **Reihenfolgen der Operanden vertauscht** :
      `x>y` bzw **`x>=y`** führt zur Auswertung von **`y.current>x.current`** bzw **`y.current>=x.current`**
      `x<y` bzw **`x<=y`** führt zur Auswertung von **`y.current<x.current`** bzw **`y.current<=x.current`**
  - ◇ Zur **Rückwandlung eines Reverse-Iterators in einen – normalen – Iterator** existiert die Memberfunktion

    `Iterator base() const;`

    Sie liefert den – normalen – Iterator, der auf die gleiche Position wie der Reverse-Operator verweist.

    Wenn **`rpos`** ein Reverse-Iterator ist, kann somit mittels **`*rpos.base()`** auf das **tatsächlich referierte** (und nicht auf das davor befindliche) **Element zugegriffen** werden.

Standard-Template-Library (STL) von C++ : Stream-Iteratoren (1)

• Allgemeines

◇ **Iteratoren** setzen voraus, dass die **Elemente** der von ihnen durchlaufbaren Datenmenge in einer **Reihenfolge** (Sequenz) **angeordnet** sind

◇ **Stream-Iteratoren** sind **Iterator-Adapter**, die es ermöglichen, dass zu Streams, die ja auch aus einer Folge von Daten bestehen, mit der **gleichen Schnittstelle** wie zu Containern zugegriffen werden kann.
Dadurch lassen sich zahlreiche **Algorithmen der STL** (z.B. zum Kopieren) auch sinnvoll **direkt auf Streams** anwenden, d.h. Algorithmen können Eingabedaten statt aus Containern aus Streams beziehen und Ausgabedaten statt in Container in Streams ausgeben.

◇ **Ostream-Iteratoren** dienen zum **Ausgeben** von Elementen in Streams.
Istream-Iteratoren dienen zum **Einlesen** von Elementen aus Streams.

◇ Neben Stream-Iteratoren gibt es auch **Streambuffer-Iteratoren**. Diese ermöglichen einen **zeichenweisen** (und nicht wertweisen) Zugriff zu den von Streams verwendeten Streambuffern.

• Ostream-Iteratoren

◇ Ostream-Iteratoren gehören zur Kategorie der **Output-Iteratoren**.
Statt an ein Container-Objekt sind sie an ein **Ausgabestream-Objekt gebunden**.
Dieses muss bei der Erzeugung eines Ostream-Iterator-Objekts bereitgestellt werden.
Ein Ostream-Iterator referiert immer die nächste Ausgabeposition im Stream.
Die **wesentliche Operation** – schreibender Zugriff zum referierten Element – führt zur **Ausgabe** des dem Element **zugewiesenen Werts** in den **Ausgabestream**.

◇ Zur Erzeugung von Ostream-Iteratorklassen definiert die STL das **Klassen-Template `ostream_iterator`**
Template-Parameter sind : – der **Typ `T`** der referierten (d.h. auszugebenden) **Daten**.
– Zeichentyp **`charT`** (Default : `char`)
– Zeicheneigenschaften-Typ `traits` (Default : `char_traits<charT>`).

◇ **Konstruktoren** (Vereinfachte Darstellung für die Default-Template-Parameter):

`ostream_iterator``(ostream& s);`	Erzeugung eines Ostream-Iterators für das `ostream`-Objekt `s`
`ostream_iterator``(ostream& s, const char* del);`	Erzeugung eines Ostream-Iterators für das `ostream`-Objekt `s`, der nach jeder Ausgabe eines Elements den C-String `del` ausgibt
`ostream_iterator``(const ostream_iterator<T>& x);`	Copy-Konstruktor

◇ **Zuweisungs-Operatorfunktion** (Vereinfachte Darstellung für die Default-Template-Parameter):

`ostream_iterator<T>&` **`operator=`**`(const T& val);`	Ausgabe des Werts `val` in den Ausgabestream

Statt durch eine Zuweisung an den dereferenzierten Iterator kann ein Wert auch durch direkte Zuweisung an den Iterator ausgegeben werden, d.h. **`*p=val`** `und` **`p=val`** sind **wirkungsgleich**. (`p` sei ein Ostream-Iterator)

◇ **Weitere Operationen**

▻ Die beiden **Increment-Operatorfunktionen** sind zwar definiert, sie bewirken aber nichts und geben lediglich eine Referenz auf das aktuelle Ostream-Iterator-Objekt zurück.

▻ Genaugenommen gibt auch die **Dereferenzierungs-Operatorfunktion** lediglich eine Referenz auf das aktuelle Objekt zurück. Die Anwendung des Zuweisungs-Operators auf dieses führt dann erst zur Ausgabe eines Werts.

Standard-Template-Library (STL) von C++ : Stream-Iteratoren (2)

• Istream-Iteratoren

◇ Istream-Iteratoren sind **Input-Iteratoren**.
Statt an ein Container-Objekt sind sie an ein **Eingabestream-Objekt** gebunden.
Dieses muss bei der Erzeugung eines Istream-Iterator-Objekts dem Konstruktor übergeben werden.
Ein Istream-Iterator **referier**t immer die **aktuelle Eingabeposition im Stream**. Beim **Erzeugen** eines Istream-Iterators sowie bei jedem Incrementieren des Iterators wird ein **Element** aus dem Stream **gelesen**.
Jeder **dereferenzierende Zugriff zum Iterator**, der **nur lesend** zulässig ist, liefert das **zuletzt eingelesene Element**.

◇ Die STL definiert das **Klassen-Template `istream_iterator`** zur Erzeugung von Istream-Iteratorklassen
Template-Parameter sind :
- der **Typ `T`** der referierten (d.h. auszugebenden) **Daten**.
- Zeichentyp **`charT`** (Default : `char`)
- Zeicheneigenschaften-Typ **`traits`** (Default : `char_traits<charT>`).
- Abstands-Typ **`Dist`** (Default : `ptrdiff_t`, ein vorzeichenloser Ganzzahltyp)

◇ **Konstruktoren** (Vereinfachte Darstellung für die Default-Template-Parameter):

`istream_iterator``();`	Erzeugung eines End-of-Stream-Iterators (referiert `EOF`)
`istream_iterator``(istream& s);`	Erzeugung eines Istream-Iterators für das `istream`-Objekt `s`, Einlesen des ersten Elements aus dem Stream
`istream_iterator``(const istream_iterator<T>& x);`	Copy-Konstruktor

◇ **Member-Operatorfunktionen** (Vereinfachte Darstellung für die Default-Template-Parameter):

`const T&` **`operator*`**`() const;`	Rückgabe des zuletzt gelesenen Elements
`const T*` **`operator->`**`() const;`	Rückgabe eines Pointers auf das zuletzt gelesene Element, über diesen kann ein Zugriff zu seinen Komponenten erfolgen
`istream_iterator<T>&` **`operator++`**`();`	Einlesen des nächsten Elements aus dem Stream, Rückgabe des fortgeschalteten neuen Iterators
`istream_iterator<T>` **`operator++`**`(int);`	Einlesen des nächsten Elements aus dem Stream, Rückgabe des alten Iterators

Falls beim **Incrementieren** des Istream-Iterators das **Stream-Objekt** in einen **Fehlerzustand** gelangt (z.B. auch bei Erreichen von `EOF`), wird der fortgeschaltete neue Iterator zu einem **End-of-Stream-Iterator**.

◇ **freie Vergleichsfunktionen**

▻ Die Operatorfunktionen für den **Vergleich auf Gleichheit** (**`operator==()`**) und **Ungleichheit** (**`operator!=()`**) sind als **freie Funktions-Templates** definiert

▻ Zwei Istream-Iteratoren sind dann **gleich**,
- wenn sie beide kein gültiges Element mehr referieren (d.h. End-of-Stream-Iteratoren sind)
- oder wenn beide keine End-of-Stream-Iteratoren sind und zum gleichen `istream`-Objekt gehören

▻ **Anmerkung zur Überprüfung auf Streamende (`EOF`)** :
Nach jedem Incrementieren eines Istream-Iterators sollte durch Vergleich des Iterators mit einem End-of-Stream-Iterator der Fehlerzustand des Streams überprüft (und damit auch auf `EOF` geprüft) werden.

Standard-Template-Library (STL) von C++ : Stream-Iteratoren (3)

- **Einfaches Demonstrationsprogramm zu Stream-Iteratoren `streamiterdem`**

```
// C++-Quelldatei streamiterdem_m.cpp --> Programm streamiterdem
// Einfaches Demo-Programm zu Stream-Iteratoren der STL

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


int main(void)
{
  cout << "\nGeben Sie Integer-Werte ein : ";

  istream_iterator<int> input(cin); // Erzeugung Istream-Iterator
                                    // erster int-Wert wird aus Stream cin gelesen
  istream_iterator<int> eof;        // Erzeugung End-of-Stream-Iterator
  int i1, cnt=0, sum=0;

  while (input!=eof)
  {
    i1=*input;                  // Rückgabe letzter gelesener int-Wert
    cnt++;
    sum+=i1;
    ++input;                    // naechster int-Wert wird aus Stream cin gelesen
  }

  ostream_iterator<int> output(cout, "\n");
  cout << "\nAnz. eingegebener Werte : ";
  *output=cnt;                  // Ausgabe Anzahl Werte, alternativ : output=cnt;
  cout << "Summe aller Werte       : ";
  *output=sum;                  // Ausgabe Summe,        alternativ : output=sum;

  return 0;
}
```

- **Ein- und Ausgabe des Demonstrationsprogramms `streamiterdem` (Beispiel)**

```
Geben Sie Integer-Werte ein : 3 4 6 7 9 10
^Z
^Z

Anz. eingegebener Werte : 6
Summe aller Werte       : 39
```

Standard-Template-Library (STL) von C++ : Insert-Iteratoren (1)

• Allgemeines

◇ Bei "**normalen**" **Iteratoren** bewirkt eine **Zuweisung** an das durch den Iterator referierte Element, dass dieses **Element überschrieben** wird (d.h. einen neuen Wert bekommt).

Beispiel :
```
// first, last und res seien normale Iteratoren
while (first != last) *res++ = *first++;
```

Obige Schleife bewirkt ein Kopieren der Elemente des Bereichs `[first, last)` (Quellbereich) in die Elemente eines Bereichs der mit `res` beginnt (Zielbereich). Die Elemente des Zielbereichs müssen hierzu existieren.

◇ Insert-Iteratoren sind **Iterator-Adapter**, die eine **Zuweisung** an den Iterator bzw an das referierte Element in ein **Einfügen** umsetzen.
Wenn `res` in obigem Beispiel ein Insert-Iterator ist, werden Kopien der Elemente aus dem Quellereich als **neue** zusätzliche **Elemente** in den Zielbereich **eingefügt**.

• Operationen

◇ Insert-Iteratoren gehören zur Kategorie der **Output-Iteratoren**.
Das bedeutet, dass zu einem dereferenzierten Iterator (also dem referierten Element) nur **schreibend** zugegriffen werden kann.

◇ Analog zu Ostream-Iteratoren kann ein derartiger schreibender Zugriff (**Zuweisung)** auch **direkt an** den **Iterator** erfolgen.
D.h. auch für einen Insert-Iterator `p` gilt : **`*p=val`** und **`p=val`** sind **wirkungsgleich**

Dies wird dadurch ermöglicht, dass die **Dereferenzierungs-**Operatorfunktion **`operator*()`** nicht das referierte Element sondern das Iterator-Objekt selbst `(*this)` zurückgibt, d.h. für den Insert-Iterator `p` gilt **`*p==p.`**

◇ Die **Zuweisungs-**Operatorfunktion **`operator=()`** ist so definiert, dass sie eine Kopie des ihr als Parameter übergebenen Elements als **neues Element** in den **Container** des Zielbereichs **einfügt**.
Das Einfügen erfolgt **unmittelbar vor** der Position, auf die der Iterator zeigt.

◇ Die beiden **Increment**-Operatorfunktionen sind zwar auch definiert, aber wie bei Ostream-Iteratoren **bewirken** sie **nichts,** sondern geben lediglich das aktuelle Iterator-Objekt bzw eine Referenz hierauf zurück.
Insert-Iteratoren können also ihre Position nicht verändern.

• Insert-Iterator-Arten

◇ In Abhängigkeit von der Position, an der sie ein Einfügen bewirken, werden **drei Arten** von Insert-Iteratoren unterschieden :

▻ **Front-Insert-Iteratoren (Front-Inserter)**
Sie bewirken ein Einfügen am **Anfang** eines Containers (vor der ersten Position)
Zum Einfügen rufen sie die Container-Funktion **`push_front(val)`** auf.
Sie sind nur bei Containern, die diese Funktion zur Verfügung stellen, möglich : **Deques** und **Listen**

▻ **Back-Insert-Iteratoren (Back-Inserter)**
Sie bewirken ein Einfügen am **Ende** eines Containers (hinter der letzten Position).
Zum Einfügen rufen sie die Container-Funktion **`push_back(val)`** auf.
Da nur **Vektoren**, **Deques** und **Listen** diese Funktion zur Verfügung stellen, sind sie nur bei diesen Container-Arten möglich.

▻ (positionierbare) **Insert-Iteratoren (Inserter)**
Sie bewirken ein Einfügen vor einer **beliebigen gültigen** Container-**Position**.
Die Einfügeposition ist bei der Erzeugung eines Insert-Iterators anzugeben.
Zum Einfügen rufen sie die Container-Funktion **`insert(pos, val)`** auf.
Da diese Funktion für alle STL-Container-Arten implementiert ist, lassen sich diese Insert-Iteratoren bei **jedem STL-Container** verwenden.
Anmerkung : Bei assoziativen Containern hat die anzugebende Einfügeposition `pos` nur Hinweischarakter.

Standard-Template-Library (STL) von C++ : Insert-Iteratoren (2)

• Definition der Insert-Iterator-Klassen

◇ Für die drei Iterator-Arten sind in der Headerdatei **<iterator>** **Klassen-Templates** definiert und implementiert :

▻ `template <class Container> class` **`front_insert_iterator;`**

▻ `template <class Container> class` **`back_insert_iterator;`**

▻ `template <class Container> class` **`insert_iterator;`**

◇ **Definition des Klassen-Templates `front_insert_iterator`**

```
template <class Container>
  class front_insert_iterator : public iterator<...>
  {
    protected :
      Container* container;
    public :
      typedef Container container_type;
      explicit front_insert_iterator(Container&);
      front_insert_iterator<Container>&
        operator=(typename Container::const_reference val);
      front_insert_iterator<Container>& operator*();
      front_insert_iterator<Container>& operator++();
      front_insert_iterator<Container>  operator++(int);
  };
```

◇ Die **Definition** der beiden **anderen Klassen-Templates** erfolgt **analog**.
Lediglich beim Klassen-Template **`insert_iterator`** besteht ein wesentlicher **Unterschied** :
Dem Konstruktor ist die Einfügeposition als zusätzlicher Parameter (Typ : `Container::iterator`) zu übergeben.
Diese wird in einer zusätzlichen `protected`-Datenkomponente gespeichert.

• Erzeugung von Insert-Iteratoren

◇ Mittels des jeweiligen **Konstruktors.**
Diesem ist das **Container-Objekt**, für den der Insert-Iterator erzeugt werden soll, als **Parameter** zu übergeben.
Der Konstruktor für einen **positionierbaren Insert-Iterator** (Klassen-Template **`insert_iterator`**) benötigt als **zusätzlichen Parameter** einen Iterator, der die **Einfügeposition** angibt.

Beispiel :
```
vector<int> imeng;
back_insert_iterator<vector<int> > backint(imeng);
insert_iterator<vector<int> > posint(imeng, imeng.begin()+4);
```

◇ Als **Alternative** steht für jede Insert-Iterator-Art eine **freie Erzeugungsfunktion** in Form eines **Funktions-Templates** zur Verfügung. Diese erzeugen jeweils ein neues Insert-Iterator-Objekt und geben es als Funktionswert zurück.
Diese Funktions-Templates sind ebenfalls in der Headerdatei **<iterator>** definiert.

```
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);

template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template <class Container, class Iterator>
  insert_iterator<Container> inserter(Container& x, Iterator i);
```

Beispiel : **`back_inserter(imeng)`**`=23;` `//` Einfügen neues Element mit Wert `23` am Ende von `imeng`

Standard-Template-Library (STL) von C++ : Insert-Iteratoren (3)

- **Einfaches Demonstrationsprogramm zu Insert-Iteratoren `insiterdem`**

```
// C++-Quelldatei insiterdem_m.cpp --> Programm insiterdem
// Einfaches Demo-Programm zu Insert-Iteratoren der STL

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

template <class Container>
  void ausgabe(Container& cont)
  { Container::iterator di;
    ostream_iterator<Container::value_type> output(cout, "  ");
    cout << endl;
    for (di=cont.begin(); di!=cont.end(); ++di)
      *output=*di;
    cout << endl;
  }

int main(void)
{ deque<int> imeng;
  deque<int>::iterator di;
  int i;

  for (i=0; i<8; ++i)
  { imeng.push_back(i+1);
    imeng.push_front(i+11);
  }
  cout << "\nInhalt imeng :";
  ausgabe(imeng);

  back_insert_iterator<deque<int> > backint(imeng);
  for (i=0; i<4; i++)
    *backint++=i+20;                // increment ++ ist wirkungslos und uberfluessig
  cout << "\nnach back_insert :";
  ausgabe(imeng);

  di=imeng.begin()+8;
  *di=30;
  cout << "\nnach Zuweisung ueber normalen Iterator (Element Nr. 9):";
  ausgabe(imeng);

  *inserter(imeng, di) = 0;         // gleichbedeutend : inserter(imeng, di) = 0;
  cout << "\nnach Zuweisung ueber Insert-Iterator (Element Nr. 9):";
  ausgabe(imeng);
  return 0;
}
```

- **Ausgabe des Demonstrationsprogramms `insiterdem`**

```
Inhalt imeng :
18  17  16  15  14  13  12  11  1  2  3  4  5  6  7  8

nach back_insert :
18  17  16  15  14  13  12  11  1  2  3  4  5  6  7  8  20  21  22  23

nach Zuweisung ueber normalen Iterator (Element Nr. 9):
18  17  16  15  14  13  12  11  30  2  3  4  5  6  7  8  20  21  22  23

nach Zuweisung ueber Insert-Iterator (Element Nr. 9):
18  17  16  15  14  13  12  11  0  30  2  3  4  5  6  7  8  20  21  22  23
```

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (1)

• Allgemeines

◇ Die STL stellt in der Algorithmen-Bibliothek ca **70 Algorithmen** zur Bearbeitung von Containern und den in ihnen enthaltenen Elementen zur Verfügung.

◇ Diese Algorithmen sind generisch als **freie Funktions-Templates** implementiert.
Sie greifen zu den Elementen der Container nur **indirekt über Iteratoren** zu, d.h. sind unabhängig von speziellen Implementierungs-Details der verschiedenen Container-Klassen.
Diese **Trennung** der **Algorithmen** von den **Containern** widerspricht zwar objektorientierten Grundkonzepten, hat aber den Vorteil einer **grösseren Flexibilität**, **effizienteren Implementierung** (Algorithmus muss nicht für jede Container-Klasse gesondert als Memberfunktion realisiert werden) und **einfacheren Erweiterbarkeit** (Anwendung auch für selbstdefinierte Container- und Iterator-Klassen)

◇ **Alle Algorithmen** bearbeiten einen oder mehrere **Bereiche** (*ranges*) von **Elementen**.
Ein derartiger Bereich kann alle Elemente eines Containers umfassen oder nur aus einer Teilmenge derselben bestehen.
Der zu bearbeitende Bereich wird durch **zwei** als **Funktionsparameter** zu übergebende **Iteratoren** festgelegt.
Dabei referiert der erste Iterator das erste zu bearbeitende Element und der zweite Iterator das Element, das auf das letzte zu bearbeitende unmittelbar folgt.
Die zu bearbeitende Menge wird also immer als eine **halboffene Menge** angegeben : Sie besteht aus allen Elementen zwischen dem ersten (einschliesslich) und dem letzten (ausschliesslich) Element.
Dabei führen die Algorithmen **keinerlei Bereichs- und Gültigkeitsüberprüfungen** aus.
Insbesondere muss der Anwender eines Algorithmus darauf achten, dass das übergebene **Ende** eines Bereiches **vom Anfang** aus **erreichbar** sein muss. Das bedeutet, dass beide übergebenen Iteratoren zu demselben Container-Objekt gehören müssen und die Ende-Position sich logisch nicht vor der Anfangsposition befinden darf.

◇ Bei den meisten Algorithmen, die mit **mehreren Bereichen** arbeiten, wird **nur der erste** Bereich direkt durch **Anfangs- und End-Iterator** festgelegt. Für die **anderen Bereiche** wird **nur** ein **Anfangs-Iterator** angegeben. Das Ende ergibt sich dann aus der Funktion und der Arbeitsweise des Algorithmus.
Bei Algorithmen, die auf einen derartigen Bereich **schreibend** zugreifen (z.B. durch Kopieren in einen Zielbereich), muss **sichergestellt** werden, dass dieser **Bereich** eine **ausreichende Groesse** besitzt.
Als **Alternative** können einfügende Iteratoren (**Insert-Iteratoren**) für den Zielbereich verwendet werden.

◇ Bei **mehreren Bereichen** können diese in **Containern unterschiedlichen Typs** liegen.

◇ Sehr viele Algorithmen liefern einen **Iterator** als **Rückgabewert**.
Dabei wird der **End-Iterator** des übergebenen Bereichs zur **Kennzeichnung** eines **Misserfolgs** (z.B. "nicht gefunden") oder einer sonstigen Fehlfunktion verwendet.

◇ **Nicht alle Algorithmen** können auf **alle Container**-Klassen **angewendet** werden.
U.a. bestimmt die vom Algorithmus verwendete **Iterator-Katogerie** die Anwendbarkeit. So lassen sich z.B. Algorithmen, die mit Random-Access-Iteratoren arbeiten, nicht auf assoziative Container anwenden.
Weiterhin können Algorithmen, die Elemente verändern, nicht auf Container angewendet werden, deren **Elemente** als **konstant** betrachtet werden (z.B. assoziative Container).

◇ Die **Anwendung** der Algorithmen ist **nicht auf die Container-Klassen** der STL **beschränkt**. Sie lassen sich vielmehr für **alle Sequenzen** von Daten, für die **Iteratoren** zum Durchlaufen existieren, einsetzen. Insbesondere arbeiten sie auch

- ▻ mit **Ein- bzw. Ausgabe-Streams**,
- ▻ den **String-Klassen** der Standardbibliothek (es existieren String-Memberfunktionen zur Erzeugung von Anfangs- und End-Iteratoren)
- ▻ sowie **normalen C-Arrays** (als Iteratoren dienen Pointer auf die Array-Elemente)

◇ Die **Deklarationen** (und **Implementierungen**) der Funktions-Templates der STL-Algorithmen befinden sich in der **Headerdatei `<algorithm>`.**

◇ Eine besondere – sehr kleine – Gruppe von **numerischen Algorithmen** ist nicht in der STL sondern in der **numerischen Bibliothek** enthalten. Sie sind in der **Headerdatei `<numeric>`** definiert.

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (2)

- **Bedeutung der Template-Parameter-Namen**

  ◇ **Template-Parameter** können sein :

  - ▻ **Iteratorklassen**
    Jedes Funktions-Template ist mit wenigstens einem Iteratorklassen-Typ parameterisiert
  - ▻ **Funktionsobjekt-Klassen**
    Eine Reihe von Algorithmen können durch einen derartigen Template-Parameter konfiguriert werden. Dadurch wird ihre Funktionalität modifiziert und erweitert.
  - ▻ **Typ** der zu bearbeitenden **Daten**
    Einen derartigen Template-Parameter besitzen einige Algorithmen
  - ▻ **Datentyp** zur Darstellung **ganzer Zahlen** (Grössen-Angaben)
    Einen derartigen Template-Parameter besitzen einige wenige Algorithmen

  ◇ Die **Namen** der **formalen Template-Parameter** sind so gewählt, dass sie die **Anforderungen** ausdrücken, die an die aktuellen Typ-Parameter gestellt werden.

  ◇ So bedeuten z.B. die formalen Typangaben **`InputIterator`**, **`InputIterator1`** oder **`InputIterator2`**, dass der entsprechende aktuelle Typ die funktionellen Anforderungen eines **Input-Iterators** erfüllen muss.
  Diese werden von allen Iterator-Kategorien ausser den Output-Iteratoren bereitgestellt, d.h. aktueller Iterator-Typ kann dann auch ein Forward-Iterator, ein Bidirectional-Iterator oder ein Random-Access-Iterator sein.

  ◇ Analog bedeutet die formale Typangabe **`Predicate`**, dass der aktuelle Typ einstellige Funktionsobjekte beschreiben muss, die – auf einen dereferenzierten Iterator angewendet – einen auf `true` prüfbaren Wert liefern.

  In den in der ANSI-Norm enthaltenen Funktions-Deklarationen werden ausführliche und damit relativ lange formale Typnamen verwendet, die den beiden o.a. Beispielen entsprechen.
  Zur Erhöhung der Übersichtlichkeit und besseren Lesbarkeit werden diese hier durch die folgenden **abgekürzten Typ-Namen** ersetzt.

  - ▻ **`InpIt`** für `InputIterator`
  - ▻ **`OutpIt`** für `OutputIterator`
  - ▻ **`ForwIt`** für `ForwardIterator`
  - ▻ **`BidirIt`** für `BidirectionalIterator`
  - ▻ **`RandIt`** für `RandomAccessIterator`
  - ▻ **`Pred`** für `Predicate` (Klasse für einstellige Prädikate)
  - ▻ **`BinPred`** für `BinaryPredicate` (Klasse für zweistellige Prädikate)
  - ▻ **`Comp`** für `Compare` (Klasse für Vergleich-Funktionsobjekte)
  - ▻ **`UnOp`** für `UnaryOperation` (Klasse für einstellige Funktionsobjekte)
  - ▻ **`BinOp`** für `BinaryOperation` (Klasse für zweistellige Funktionsobjekte)
  - ▻ **`Func`** für `Function` (allgemeine Funktionsobjekt-Klasse)

  ◇ **Anmerkung zu Funktionsobjekt-Parametern** :
  Für Funktionsobjekt-Parameter können den Algorithmen aktuell statt Funktionsobjekten auch **Funktions-Pointer** entsprechender Funktionalität übergeben werden.

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (3)

• Überblick über die Algorithmen

◇ Nichtmodifizierende Algorithmen

Sie führen nur Lesezugriffe zu Elementbereichen aus und ändern damit weder den Wert noch die Reihenfolge von Elementen des jeweiligen Bereichs.

- Suchen nach Elementen und Elementfolgen (diverse verschiedene Suchkriterien)
- Zählen von Elementen
- Vergleich von Elementbereichen
- Ausführen einer – nicht modifizierenden – Funktion für alle Elemente

◇ Modifizierende Algorithmen

Sie führen auch Schreibzugriffe zu Elementbereichen aus und ändern damit den Wert und/oder die Reihenfolge von Elementen

▻ Wert-ändernde Algorithmen

Sie verändern den Wert von Elementen bzw fügen Elemente hinzu (in einem Quellbereich und/oder Zielbereich)

- Kopieren und ersetzendes Kopieren von Elementbereichen
- Vertauschen der Elemente von zwei Bereichen
- Transformation von Elementen
- Ersetzen von Elementwerten
- Füllen von Elementbereichen
- Zusammenfassung der Elemente zweier sortierter Bereiche zu einem neuen sortierten Bereich

▻ Löschende Algorithmen

Sie entfernen Elemente aus einem Bereich. Hierzu zählen auch Algorithmen, die einen Quellbereich unverändert lassen und aus diesem nur einen Teil der Elemente in einen Zielbereich kopieren.

- Entfernen von Elementen
- Teil-Löschendes Kopieren von Elementbereichen

▻ Mutierende Algorithmen

Sie verändern lediglich die Reihenfolge von Elementen, nicht aber ihren Wert.

- Vertauschen von zwei Elementen
- Umkehrung der Elementreihenfolge
- Rotation von Elementen
- Permutieren der Elemente eines Bereichs
- Umverteilung ("*shuffle*") der Elemente eines Bereichs
- Verschiebung von Element-Teilbereichen

▻ Sortierende Algorithmen

Sie verändern die Reihenfolge der Elemente eines Bereichs so, dass sie anschliessend wenigstens teilweise sortiert sind. Diese Algorithmen existieren alle in zwei Versionen : Die eine Version verwendet zum Sortieren den <−Operator (Default-Sortierkriterium), die andere verwendet ein als Funktionsobjekt übergebenes Sortierkriterium.

- Sortieren aller Elemente eines Bereichs
- Sortieren der ersten n Elemente eines Bereichs
- Sortieren hinsichtlich eines Vergleichselements

▻ Heap-Operationen

Ein Heap ist ein spezieller binärer Baum, bei dem das kleinste (bzw größte) Element Wurzelelement (1. Element) ist.

- Erzeugen, Verändern (Element-Einfügen, -Entfernen) und Sortieren (=Zerstören) eines Heaps

▻ Mengen-Operationen

Diese Algorithmen arbeiten mit sortierten Bereichen (Mengen)

- Bildung der Vereinigungs-, Schnitt-, Differenz- und Komplementär-Menge zweier Mengen (→ neuer Bereich)

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (4)

• Anmerkung zur Darstellung der Algorithmen

◇ Zur Vereinfachung und Erhöhung der Übersichtlichkeit werden im folgenden die Funktions-Deklarationen der Algorithmen nicht in der originalen Template-Form sondern in einer vereinfachten Darstellung wiedergegeben :
Der Template-Zusatz wird weggelassen.
Er kann aber leicht rekonstruiert werden, da die Typen aller Funktionsparameter auch Template-Parameter sind.

◇ **Beispiel** :

statt :
```
template <class ForwIt, class Size, class T, class BinPred>
  ForwIt search_n(ForwIt first, ForwIt last, Size count,
                  const T& val, BinPred pred);
```

wird formuliert :
```
ForwIt search_n(ForwIt first, ForwIt last, Size count,
                const T& val, BinPred pred);
```

• Einige nichtmodifizierende Algorithmen

◇ Die Algorithmen dieser Gruppe arbeiten mit Input- und Forward-Iteratoren und sind daher prinzipiell auf alle Container-Klassen anwendbar. Allerdings setzen einige von ihnen sortierte Bereiche voraus.

`InpIt find(InpIt first, InpIt last, const T& val);`	Suchen nach dem ersten Element mit dem Wert `val` Rückgabe : Position des Elements, falls gefunden, sonst `last`
`InpIt find_if(InpIt first, InpIt last, Pred prd);`	Suchen nach dem ersten Element für das das Prädikat `prd` erfüllt ist Rückgabe : Position des Elements, falls gefunden, sonst `last`
`ForwIt search(ForwIt1 first1, ForwIt1 last1, ForwIt2 first2, ForwIt2 last2);`	Suchen nach erstem Vorkommen des 2. Bereichs im 1. Bereich, Rückgabe : Pos. des 1.Elements des gefundenen Teilbereichs, bzw `last1`
`ForwIt lower_bound(ForwIt first, ForwIt last, const T& val);`	Rückgabe : Position des ersten Elements dessen Wert `>= val` ist (Bereich muss sortiert sein)
`const T& max(const T& a, const T& b);`	Rückgabe : Referenz auf grösseres Element
`ForwIt max_element(ForwIt first, FormIt last);`	Rückgabe : erste Position des grössten Elements
`ForwIt max_element(ForwIt first, FormIt last, Comp cmp);`	Rückgabe : erste Position des grössten Elements Vergleich erfolgt mittels `cmp`
`InpIt::difference_type count(InpIt first, InpIt last, const t& val);`	Suchen nach Elementen mit dem Wert val Rückgabe : Anzahl der gefundenen Elemente
`InpIt::difference_type count_if(InpIt first, InpIt last, Pred prd);`	Suchen nach Elementen, für die `prd` erfüllt ist Rückgabe : Anzahl der gefundenen Elemente
`bool equal(InpIt1 first1, InpIt1 last1, InpIt2 first2);`	Überprüfung zweier Bereiche auf Gleichheit Rückgabe : `true`, wenn gleich, sonst `false`
`UnOp for_each(InpIt first, InpIt last, UnOp f);`	Aufruf von `f(elem)` für alle Elemente `elem` des Bereichs `[first, last)`, Rückgabe : `f`

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (5)

• Einige wert-ändernde Algorithmen

◇ Da die Elemente von assoziativen Containern als konstant betrachtet werden, lassen sich diese Algorithmen nicht anwenden, wenn der Bereich, in dem Elemente geändert werden, in einem assoziativen Container liegt.

`OutpIt` **`copy`**`(InpIt first, InpIt last, OutpIt res);`	Kopieren der Elemente aus dem Bereich `[first,last)` in den bei `res` beginnenden Bereich `res` darf nicht im Bereich `[first,last)` liegen Rückgabe : `res + (last - first)`
`BidirIt` **`copy_backward`**`(BidirIt1 first, BidirIt1 last, BidirIt2 res);`	Kopieren der Elemente aus dem Bereich `[first,last)` in umgekehrter Reihenfolge in den Bereich ab `res` `res` darf nicht im Bereich `[first,last)` liegen Rückgabe : `res + (last - first)`
`OutpIt` **`replace_copy`**`(InpIt first, InpIt last, OutpIt res, const T& alt, const T& neu);`	Kopieren der Elemente aus dem Bereich `[first,last)` in den bei `res` beginnenden Bereich, wobei der Wert `alt` durch den Wert `neu` ersetzt wird. `res` darf nicht im Bereich `[first,last)` liegen Rückgabe : `res + (last - first)`
`ForwIt2` **`swap_ranges`**`(ForwIt1 first1, ForwIt1 last1, ForwIt2 first2);`	Vertauschen der Elemente des Bereiches `[first1, last1)` mit den Elementen des Bereichs der mit `first2` beginnt. Rückgabe : `first2 + (last1 - first1)` Die beiden Bereiche dürfen sich nicht überlappen
`OutpIt` **`transform`**`(InpIt first, InpIt last, OutpIt res, UnOp op);`	Jedes Element `*pos` aus dem Bereich `[first,last)` wird mittels `op(*pos)` transformiert und als neues Element in den Bereich, der bei `res` beginnt, kopiert. `res` darf mit `first` übereinstimmen Rückgabe : `res + (last - first)`
`void` **`replace`**`(ForwIt first, ForwIt last, const T& alt, const T& neu);`	Ersetzen des Werts `alt` der Elemente aus dem Bereich `[first,last)` durch den Wert `neu`.
`void` **`fill`**`(ForwIt first, ForwIt last, const T& val);`	Zuweisen des Werts `val` an jedes Element aus dem Bereich `[first,last)`
`void` **`generate`**`(ForwIt first, ForwIt last, Generator gen);`	Aufruf von `gen()` für jedes Element aus dem Bereich `[first,last)` und Zuweisen des dadurch erzeugten Werts an das jeweilige Element
`void` **`generate_n`**`(OutpIt first, Size n, Generator gen);`	Aufruf von `gen()` für die ersten `n` Elemente des mit `first` beginnenden Bereichs und Zuweisen des dadurch erzeugten Werts an das jeweilige Element
`OutpIt` **`merge`**`(InpIt1 first1, InpIt1 last1, InpIt2 first2, InpIt2 last2, OutpIt res);`	Zusammenfassen der Elemente der beiden sortierten Bereiche `[first1, last1)` und `[first2, last2)` in einen bei `res` beinnenden ebenfalls sortierten Bereich Quell- und Zielbereiche dürfen sich nicht überlappen Rückgabe : Position hinter dem letzten Element im Zielber.

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (6)

• Einige löschende Algorithmen

◇ Da die Elemente von assoziativen Containern als konstant betrachtet werden, lassen sich diese Algorithmen nicht anwenden, wenn der Bereich, in dem Elemente geändert werden, in einem assoziativen Container liegt.

`ForwIt` **`remove`**`(ForwIt first, ForwIt last, const T& val);`	Löschen aller Elemente aus dem Bereich `[first,last)`, die den Wert `val` haben Rückgabe : Position hinter dem letzten Element im modifizierten Bereich
`ForwIt` **`remove_if`**`(ForwIt first, ForwIt last, Pred prd);`	Löschen aller Elemente aus dem Bereich `[first,last)` für die das Prädikat `pred` erfüllt(`true`) ist Rückgabe : Pos. hinter dem letzten Element im mod. Bereich
`OutpIt` **`remove_copy`** `(InpIt first, InpIt last, OutpIt res); const T& val);`	Kopieren der Elemente aus dem Bereich `[first,last)`, die nicht den Wert `val` haben, in den bei `res` beginnenden Bereich Rückgabe : Position hinter dem letzten Element im Zielber.
`OutpIt` **`remove_copy_if`** `(InpIt first, InpIt last, OutpIt res); Pred prd);`	Kopieren der Elemente aus dem Bereich `[first,last)`, für die das Prädikat `prd` nicht erfüllt ist, in den bei `res` beginnenden Bereich Rückgabe : Position hinter dem letzten Element im Zielber.
`ForwIt` **`unique`**`(ForwIt first, ForwIt last);`	Löschen aller Elemente aus dem Bereich `[first,last)`, die einem Element mit gleichem Wert unmittelbar folgen Rückgabe : Pos. hinter dem letzten Element im mod. Bereich
`OutpIt` **`unique_copy`**`(InpIt first, InpIt last, OutpIt res);`	Kopieren der Elemente aus dem Bereich `[first,last)`, die nicht einem Element mit gleichem Wert unmittelbar folgen, in den bei `res` beginnenden Bereich,. `res` darf nicht im Bereich `[first,last)` liegen Rückgabe : Position hinter dem letzten Element im Zielber.

• Einige sortierende Algorithmen (nur für Container mit Random-Access-Iteratoren)

`void` **`sort`**`(RandIt first, RandIt last);`	Sortieren der Elemente im Bereich `[first,last)`, nach aufsteigender Reihenfolge (mit `operator<`)
`void` **`sort`**`(RandIt first, RandIt last, Comp cmp);`	Sortieren der Elemente im Bereich `[first,last)` mittels des durch `cmp` festgelegten Vergleichskriteriums
`void` **`partial_sort`**`(RandIt first, RandIt middle, RandIt last);`	Sortieren der Elemente im Bereich `[first,last)` so, dass anschliessend die ersten `(middle-first)` sortierten Elemente im Bereich `[first, middle)` stehen. Sortierung nach aufsteig. Reihenfolge (mit `operator<`) Die restlichen Elemente sind unsortiert.
`void` **`nth_element`**`(RandIt first, RandIt nth, RandIt last);`	Platzierung des Elements aus dem Bereich `[first,last)` das bei vollständiger Sortierung an der Position `nth` stehen würde, an diese Position (Vergleichselement). Im Teilbereich `[first, nth)` befinden sich (unsortiert) nur Elemente, die kleiner gleich dem Vergleichselement sind, im Teilbereich `[nth+1, last)` die restlichen Elemente

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (7)

• Einige mutierende Algorithmen

◇ Da die Elemente von assoziativen Containern als konstant betrachtet werden, lassen sich diese Algorithmen nicht anwenden, wenn der Bereich, in dem Elemente geändert werden, in einem assoziativen Container liegt.

`void swap(T& a, T& b);`	Vertauschen der Elemente `a` und `b`
`void reverse(BidirIt first, BidirIt last);`	Umkehren der Reihenfolge der Elemente des Bereichs `[first,last)`
`void rotate(ForwIt first, ForwIt middle, ForwIt last);`	Rotieren der Elemente aus dem Bereich `[first,last)` um `(middle - first)` Positionen nach links, so dass sich anschliessend das ehemalige Element an der Position `middle` an der Position `first` befindet `middle` muss im Bereich `[first,last)` liegen
`OutpIt rotate_copy(ForwIt first, ForwIt middle, ForwIt last, OutpIt res);`	Kopieren der Elemente aus dem Bereich `[first,last)` in den bei `res` beginnenden Bereich, wobei die Elemente um `(middle - first)` Positionen nach links rotiert werden, so dass sich anschliessend das ehemalige Element an der Position `middle` an der Position `res` im Zielbereich befindet. Quell- und Zielbereich dürfen sich nicht überlappen Rückgabe : `res + (last - first)`
`void random_shuffle(RandIt first, RandIt last);`	Änderung der Reihenfolge der Elemente des Bereiches `[first, last)` nach einer gleichmässigen Verteilung.
`void random_shuffle(RandIt first, RandIt last, RandNumGen& rand);`	Änderung der Reihenfolge der Elemente des Bereiches `[first, last)` nach einer durch den Zufallszahlen-. generator `rand` festgelegten Verteilung
`BidirIt partition(BidirIt first, , BidirIt last, Pred prd);`	Verschieben aller Elemente aus dem Bereich `[first,last)`, für die das Prädikat `prd` erfüllt ist, vor alle Elemente, für die es nicht erfüllt ist.. Rückgabe : Position des ersten Elements, für die das Prädikat `prd` nicht erfüllt ist

• Einige Mengen-Operationen

`OutpIt set_intersection (InpIt1 first1, InpIt1 last1, InpIt2 first2, InpIt2 last2, OutpIt res);`	Bildung einer sortierten Menge aus den Elementen, die sowohl in dem sortierten Bereich `[first1,last1)` als auch in `[first2, last2)` enthalten sind und Kopieren derselben in den bei `res` beginnenden Bereich. (Bildung der Schnittmenge) Rückgabe : Position hinter dem letzten Element im Zielber.
`OutpIt set_union(/* Parameter wie oben */);`	Bildung der Vereinigungsmenge
`OutpIt set_difference(/* Par. wie oben */);`	Bildung der Differenzmenge (Menge1 – Menge2)

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (8)

- **Einfaches Demonstrationsprogramm (1) zu Algorithmen `algodem1`**

```
// C++-Quelldatei algo1dem1_m.cpp --> Programm algo1dem1
// Einfaches Demo-Programm zu Algorithmen der STL

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
using namespace std;

template <class T>
  class ValOut
  { public: void operator()(const T& x) { cout << x << "  "; }
  };

template <class T>
  void checkFor(list<T>& lis, const T& val)
  { list<T>::iterator p;
    p=find(lis.begin(), lis.end(), val);
    cout << "gesucht : " << val ;
    if (p==lis.end()) cout << " und nicht gefunden" << endl;
    else              cout << " und gefunden : " << *p << endl;
  }

bool mod3(int x) { return x%3==0; }

int main(void)
{ list<int> li1;
  list<int>::difference_type cnt;
  list<int>::iterator ilp;
  li1.push_back(2);   li1.push_back(7);
  li1.push_back(9);   li1.push_back(3);
  li1.push_back(6);   li1.push_back(5);
  li1.push_back(15);  li1.push_back(11);
  ValOut<int> ivout;                          // Funktionsobjekt
  cout << "\nInhalt der Liste :\n";
  for_each(li1.begin(), li1.end(), ivout);
  cout << endl << endl;;
  checkFor(li1, 7);
  checkFor(li1, 8);
  cnt=count_if(li1.begin(), li1.end(),mod3);
  cout << "\nAnzahl durch 3 teilbarer Zahlen : " << cnt << endl;
  ilp=find_if(li1.begin(), li1.end(),mod3);
  cout << "Erste durch 3 teilbare Zahl     : " << *ilp << endl;
  ilp=max_element(li1.begin(), li1.end());
  cout << "Groesste enthaltene Zahl        : " << *ilp << endl;
  vector<int> vi1(li1.size());
  vector<int>::iterator ivp;
  copy(li1.begin(), li1.end(), vi1.begin());
  cout << "\nInhalt des Vektors :\n";
  for_each(vi1.begin(), vi1.end(), ivout);
  cout << endl << endl;
  int i=8;
  ivp=lower_bound(vi1.begin(), vi1.end(), i);
  cout << "Erste Zahl >= " << i << " (unsortiert) : " << *ivp << endl;
  sort(vi1.begin(), vi1.end());
  cout << "\nInhalt des Vectors nach Sortierung :\n";
  copy (vi1.begin(), vi1.end(), ostream_iterator<int>(cout, "  "));
  cout << endl << endl;
  ivp=lower_bound(vi1.begin(), vi1.end(), i);
  cout << "Erste Zahl >= " << i << " (sortiert)   : " << *ivp << endl;
  return 0;
}
```

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (8)

- **Ausgabe des Demonstrationsprogramms `algodem1`**

```
Inhalt der Liste :
2  7  9  3  6  5  15  11

gesucht : 7 und gefunden : 7
gesucht : 8 und nicht gefunden

Anzahl durch 3 teilbarer Zahlen : 4
Erste durch 3 teilbare Zahl     : 9
Groesste enthaltene Zahl        : 15

Inhalt des Vektors :
2  7  9  3  6  5  15  11

Erste Zahl >= 8 (unsortiert) : 15

Inhalt des Vectors nach Sortierung :
2  3  5  6  7  9  11  15

Erste Zahl >= 8 (sortiert)   : 9
```

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (9)

- **Einfaches Demonstrationsprogramm (2) zu Algorithmen `algodem2`**

```
// C++-Quelldatei algo1dem2_m.cpp --> Programm algo1dem2
// Einfaches Demo-Programm zu Algorithmen der STL

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include <set>
#include <cstdlib>
#include <functional>
using namespace std;

class GenInt
{ public :
    GenInt (unsigned s=1) { srand(s);}
    int operator()(void) { return (int)(rand()%100);}
};


int main(void)
{ vector<int> vi1;
  vi1.push_back(2);    vi1.push_back(7);    vi1.push_back(9);
  vi1.push_back(3);    vi1.push_back(24);   vi1.push_back(6);
  vi1.push_back(15);   vi1.push_back(5);    vi1.push_back(11);
  ostream_iterator<int> iaus(cout, "  ");
  cout << "\nInhalt des Vektors :\n";
  copy (vi1.begin(), vi1.end(), iaus);   cout << endl << endl;
  set<int> si1;
  copy(vi1.begin(), vi1.end(), inserter(si1, si1.begin()));
  cout << "\nInhalt des Sets :\n";
  copy (si1.begin(), si1.end(), iaus);   cout << endl << endl;
  GenInt rnum(1111);
  vi1.resize(6);
  generate(vi1.begin(), vi1.end(), rnum);
  cout << "\nInhalt des Vektors nach Zufallszahlen-Zuweisung :\n";
  copy (vi1.begin(), vi1.end(), iaus);   cout << endl << endl;
  generate_n(back_inserter(vi1), 5, rnum);
  cout << "\nInhalt des Vektors nach Verlaengerung um Zufallszahlen :\n";
  copy (vi1.begin(), vi1.end(), iaus);   cout << endl << endl;
  sort(vi1.begin(), vi1.end());
  list<int> li1(si1.size()+vi1.size());
  merge(si1.begin(), si1.end(), vi1.begin(), vi1.end(), li1.begin());
  cout << "\nInhalt der Zusammenfassung des Sets und des sortierten Vektors "
          "(--> Liste) :\n";
  copy (li1.begin(), li1.end(), iaus);   cout << endl << endl;;
  list<int>::iterator ilp = unique(li1.begin(), li1.end());
  cout << "\nInhalt der Liste nach Entfernung aufeinanderfolgender Duplikate :\n";
  copy (li1.begin(), li1.end(), iaus);   cout << endl << endl;;
  li1.erase(ilp, li1.end());
  cout << "\nInhalt der Liste nach Entfernung uberfluessiger End-Elemente :\n";
  copy (li1.begin(), li1.end(), iaus);   cout << endl << endl;;
  list<int> li2;
  remove_copy_if(li1.begin(), li1.end(), back_inserter(li2),
                 not1(bind2nd(modulus<int>(), 2)));
  cout << "\nInhalt der 2. Liste nach entfernenden Kopieren :\n";
  copy (li2.begin(), li2.end(), iaus);   cout << endl << endl;;
  return 0;
}
```

Standard-Template-Library (STL) von C++ : Algorithmen-Bibliothek (10)

- **Ausgabe des Demonstrationsprogramms `algodem2`**

```
Inhalt des Vektors :
2  7  9  3  24  6  5  15  11


Inhalt des Sets :
2  3  5  6  7  9  11  15  24


Inhalt des Vektors nach Zufallszahlen-Zuweisung :
66  47  24  58  15  50


Inhalt des Vektors nach Verlaengerung um Zufallszahlen :
66  47  24  58  15  50  58  15  47  53  69


Inhalt der Zusammenfassung des Sets und des sortierten Vektors (--> Liste) :
2  3  5  6  7  9  11  15  15  15  24  24  47  47  50  53  58  58  66  69


Inhalt der Liste nach Entfernung aufeinanderfolgender Duplikate :
2  3  5  6  7  9  11  15  24  47  50  53  58  66  69  53  58  58  66  69


Inhalt der Liste nach Entfernung ueberfluessiger End-Elemente :
2  3  5  6  7  9  11  15  24  47  50  53  58  66  69


Inhalt der 2. Liste nach entfernenden Kopieren :
3  5  7  9  11  15  47  53  69
```