Inhalt

Einführung in C++, III

- statics
- this pointer
- const-Objekte und const-Methoden
- friends
- Operatorüberladung
- element pointer

statics

Mit **`static`** deklarierte **Attribute** (static data members, Klassenvariablen):

- sind nicht an die Existenz von Objekten gebunden
- existieren nur einmal pro Klasse - nicht pro Objekt
- werden als globale Elemente für die Objekte einer Klasse verwendet
- unterliegen den üblichen Zugriffsrechten der Klasse
- werden in einer Klassendeklaration mit **`static`** deklariert
- werden in der Datei der Methodendefinitionen initialisiert

Hinweis: Konstruktoren sind hier nicht hilfreich, da statische Attribute auch ohne Objekte existieren - Konstruktoren jedoch nicht.

Mit **`static`** deklarierte **Methoden** (Klassenmethoden):

- sind nicht an die Existenz von Objekten gebunden
  ➔ können ohne Objekt aufgerufen werden
- haben keinen this pointer und können daher nur statische Attribute und Methoden verwenden

Zugriff auf statische Mitglieder:

- über ein Objekt - falls verfügbar - mit der Punktnotation

```
objektname.member
```

- **oder** über die Klasse mit dem Bereichsoperator

```
klassenname::member
```

Beispiel zu statics

```
class X
{
   static int i;
public:
   static const string str;

   static void seti(int i)
   {
      X::i = i;
   }

   static int  geti() { return i; }

   void inci() { i++; }
};
```

```
// statics initialisieren:
int X::i = 41;
const string X::str =
                  "Moin Flensburg!\n";
```

```
int main()
{
   // ohne Objekt:
   cout << X::str;
   cout << "i = " << X::geti() << endl;

   // mit Objekten:
   X obj1, obj2;
   obj1.inci();   // inci nicht statisch

   cout << obj1.str;
   cout << "i = "
        << obj2.geti() << endl;
}
```

```
Ausgabe:
Moin Flensburg!
i = 41
Moin Flensburg!
i = 42
```

`this` pointer

- Der **`this`** pointer wird allen nicht statischen Methoden als hidden argument übergeben und ist selbst konstant. Methoden haben also ein weiteres Argument der Form:

  `Klasse* const this`

  (konstanter Zeiger auf Objekt der eigenen Klasse)

- Das hidden argument kann optional angegeben werden, es ist immer das erste Argument.
  Z.B. für den Destruktor einer Klasse **`A`**:

```
~A( A* const this );
```

const-Objekte und const-Methoden

- Objekte können, genau wie Variablen auch, als konstant deklariert werden, z.B.:

```
const A a(42);
```

← **const**-Objekt

- Nun sind alle „normalen“ Methoden des Objekts **a** gesperrt, denn selbst wenn alle Methodenparameter als **const** deklariert wären, könnten die Methoden noch mit **this** schreibend auf die Attribute zugreifen.
- Methoden, die auch auf const-Objekte anwendbar sind (nur lesender Zugriff), heißen const-Methoden. Sie werden durch **const** hinter der Parameterliste gekennzeichnet:

```
string gibHer(int i) const;
```

← **const**-Methode

- Hierdurch ändert sich der Typ des hidden Arguments zu:
  const *Klasse const this**
  (konstanter Zeiger auf konstantes Objekt)

→ const-Methoden können den Objektzustand nicht ändern

friends

Eine Klasse kann zum Freund (friend) erklären:

- globale Funktionen
- Methoden anderer Klassen oder
- ganze Klassen (alle Methoden einer anderen Klasse).

- friends haben Zugriff auf alle members der Klasse.
- friends sind selbst keine class members und haben daher keinen this pointer.
  Der Bezug zum Objekt muss mit einem Parameter vom Typ *freundklasse* (oder Referenz/Pointer auf *freundklasse*) hergestellt werden.
- friends können mit mehreren Klassen befreundet sein, die Freundschaft kann aber nicht 'weitergereicht' werden.

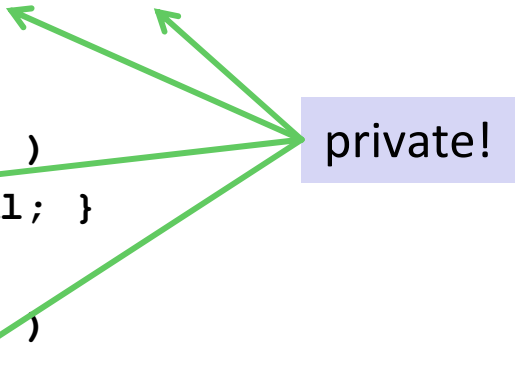
Die positive Lesart ist:

- friends stellen eine 'kontrollierte' Schnittstelle einer Klasse dar
- die Freundschaft wird immer von einer Klasse ausdrücklich erteilt

Beispiel friends

```
class A
{
   int i;
public:
   A( int i ): i(i) {}
   friend void duDarfstA( A& a );
   friend class C;
};

class B
{
   double d;
   double getd() { return d; }
public:
   B( double d ): d(d) {}
   friend void duDarfstB( B& b );
   friend class C;
};
```

```
class C
{
public:
   void freundAusC( A& a, B& b )
   { cout << int(a.i + b.d) << endl; }
};

void duDarfstA( A& a )
{ cout << a.i << endl; }

void duDarfstB( B& b )
{ cout << b.getd() << endl; }

int main()
{
   A objA(19); B objB(23); C objC;

   duDarfstA( objA );
   duDarfstB( objB );

   objC.freundAusC( objA, objB );
}
```




```
Ausgabe:
19
23
42
```

Operatorüberladung

Operatoren können überladen werden, allerdings gelten folgende Einschränkungen:

- Es können keine neuen Operatoren eingeführt werden, nur die bestehenden Operatoren können überladen werden
- Die Anzahl der Operanden kann nicht verändert werden (z.B.: unäres **++** bleibt immer unär).
- Die Rangfolge der Operatoren kann nicht verändert werden.
- Der Typ mindestens eines Operanden muss benutzerdefiniert sein. Dies verhindert, dass die Standardoperatoren überschrieben werden.
- Die Operatoren `.` `.*` `::` `?:` können nicht überladen werden.

- Operatoren können implementiert werden als:
  - globale Funktionen (non-member functions)
  - Methoden oder
  - friends
- Die Operatoren `=` `()` `[]` `->` können nur als member function realisiert werden.
- Die Syntax der Operatorfunktionen ist identisch zur Syntax 'normaler' Funktionen, allein der Funktionsbezeichner lautet:

operator*op*

- Z.B. die „Verknüpfung" von Comicfiguren mit dem Operator `*` als globale Funktion:

```
Comic operator*( Comic links, Comic rechts )
{
    ...
}
```

Der Aufruf dieser Funktion erfolgt alternativ in Funktions- oder Operatorschreibweise
(**Tom** und **Jerry** seien Objekte der Klasse **Comic**):

```
Tom = operator*(Tom, Jerry);
Tom = Tom * Jerry;
```

Weiterhin sind zu unterscheiden:

- Methoden
- binäre oder
- unäre (pre- oder postfix) Funktionen

Als Methode hätte die Multiplikation nur einen Parameter
(der linke Parameter ist das hidden argument):

```
Comic Comic::operator*(Comic rechts)
{
    ...
}
```

mögliche Aufrufe:

```
Tom = Tom.operator*(Jerry);
Tom = Tom * Jerry;
```

Unäre Operatoren

Alle unären Operatoren sind prefix-Operatoren, nur **++** und **--** gibt es als pre- und postfix Variante. Es gibt so vier Möglichkeiten:

- **Methode, prefix**:

```
void Comic::operator++() { ... }
```

Aufrufe:
```
Tom.operator++();
++Tom;
```

zur Vereinfachung wurde hier void als return-type gewählt

- **Methode, postfix**:

```
void Comic::operator++( int dummy ) { ... }
```

Aufrufe:
```
Tom.operator++(42);
Tom++;
```

- **globale Funktion, prefix**:

```
void operator++(Comic c ) { ... }
```

Aufrufe:
```
operator++(Tom);
++Tom;
```

- **globale Funktion, postfix**:

```
void operator++(Comic c, int dummy) { ... }
```

Aufrufe:
```
operator++(Tom, 42);
Tom++;
```

int-Argument zur Unterscheidung von prefix/postfix

Beispiel: globaler Operator +, Operanden mit public members

```
class MeinTyp
{
public:
   int i;
   string s;
   MeinTyp() {}
   MeinTyp(int i)
      : i(i), s("") {}
   MeinTyp(int i, string s)
      : i(i), s(s) {}
   void zeigDich() { cout << i << s; }
};

MeinTyp operator+(const MeinTyp& links,
                  const MeinTyp& rechts)
{
   MeinTyp tmp;
   tmp.i = links.i + rechts.i;
   tmp.s = links.s + rechts.s;

   return tmp;
}
```

notwendig für `tmp`-Objekt

Typkonverter `int` → `MeinTyp`

public

```
int main()
{
   MeinTyp x( 30, " Moin ");
   MeinTyp y(  9, "Flensburg!\n");

   x = x + y;
   x = x + 1;  // Typwandlung 1 -> X
   x = 2 + x;  // Typwandlung 2 -> X

   x.zeigDich();
}
```

Operator verhält sich symmetrisch

Ausgabe:
```
42 Moin Flensburg!
```

Hinweise zum Beispiel S.13

- die Operatorfunktion ist global definiert
- die Attribute der Klasse sind public
- der default-Konstruktor muss erstellt werden, er wird in der Operatorfunktion benötigt (Objekt `tmp` instanziieren)
- die + Operation funktioniert auch mit gemischten Typen, wenn der Compiler eine Konvertierung zu **`MeinTyp`** findet.
  Hier werden die Integerliterale **`1`** und **`2`** mit dem Konstruktor **`MeinTyp(int)`** umgewandelt.

Eigene Typwandler können sein:

- conversion functions: Konstruktoren mit einem Argument, wobei der Typ des Arguments nicht der eigene Typ ist
  (wandelt z.B. von **`int -> MeinTyp`**)
- casting operators: Funktionsbezeichner ist ***`operator typ`*** , kann auch überladen werden, Aufruf z.B.: **`int(ObjektMeinesTyps)`**
  (wandelt von **`MeinTyp -> int`**)

Variante A: + Operator Methode, Operanden mit private members

```
class MeinTyp
{
   int i;
   string s;
public:
   MeinTyp() {}
   MeinTyp(int i)
      : i(i), s("") {}
   MeinTyp(int i, string s)
      : i(i), s(s) {}
   void zeigDich() { cout << i << s; }
   MeinTyp operator+(
                const MeinTyp& rechts);
};

MeinTyp MeinTyp::operator+(
                 const MeinTyp& rechts)
{
   MeinTyp tmp;
   tmp.i = i + rechts.i;
   tmp.s = s + rechts.s;

   return tmp;
}
```

nur ein Parameter

private

```
int main()
{
   MeinTyp x( 30, " Moin ");
   MeinTyp y(  9, "Flensburg!\n");

   x = x + y;
   x = x + 1;  // Typwandlung 1 -> X
   x = 2 + x;  // Typwandlung 2 -> X

   x.zeigDich();
}
```

nicht möglich, Operator verhält sich asymmetrisch

Hinweise zum Beispiel S.15

- Nicht statische Methoden verhalten sich asymmetrisch, da für das hidden argument keine benutzerdefinierten Konvertierungen vorgenommen werden.
  ➔ Das hidden argument muss immer den Typ *Zeiger auf Objekt der eigenen Klasse* haben.
- Notiert man die beiden Operatoraufrufe in Funktionsschreibweise, wird das Problem deutlich:

```
x.operator+(1);
2.operator+(x);
```

ok, 1 wird mit Konstruktor konvertiert

nok, 2 ist kein Objekt der Klasse `MeinTyp`

- Sind symmetrisches Verhalten und private Daten gefordert, muss der Operator als friend-Funktion implementiert werden.
  (Nur wenn **`this`** als Operand nicht verwendet werden kann, sind friends überhaupt erforderlich.)

Das Problem des asymmetrischen Verhaltens tritt bei unären Operatoren nicht auf. Sie können immer als Methoden implementiert werden.

Variante B: + Operator friend, Operanden mit private members

```
class MeinTyp
{
   int i;
   string s;
public:
   // Konstruktoren wie bisher
        ...
   void zeigDich() { cout << i << s; }
   friend MeinTyp operator+(
                const MeinTyp& links,
                const MeinTyp& rechts);
};

MeinTyp operator+(
                const MeinTyp& links,
                const MeinTyp& rechts)
{
   MeinTyp tmp;
   tmp.i = links.i + rechts.i;
   tmp.s = links.s + rechts.s;

   return tmp;
}
```

zwei Parameter

private, dennoch Zugriff

```
int main()
{
   MeinTyp x( 30, " Moin ");
   MeinTyp y(  9, "Flensburg!\n");

   x = x + y;
   x = x + 1;   // Typwandlung 1 -> X
   x = 2 + x;   // Typwandlung 2 -> X

   x.zeigDich();
}
```

ok, erster Parameter ist kein hidden argument

Beispiel template-Klasse **MeinArray**

```
template <typename T>
class MeinArray
{
   int laenge;
   T* feld;
   void check_index(int index) const;
public:
   MeinArray(int laenge);
   ~MeinArray();

   MeinArray(const MeinArray& arr);

   void operator=(const MeinArray& arr);

   T& operator[](int i);

   const T& operator[](int i) const;

   void zeigDich();
};
```

copy-Konstruktor

operator=

operator[]:

const operator[]:

```
template <typename T>
MeinArray<T>::MeinArray(int laenge)
{
   if( laenge <= 0 )
   {
      cout << "Konstruktor MeinArray: "
           << "Laenge <= 0\n";
      exit(EXIT_FAILURE);
   }

   this->laenge = laenge;
   feld = new T[laenge];

   if(feld == NULL)
   {
      cout << "Konstruktor MeinArray: "
           << "heap ist voll.\n";
      exit(EXIT_FAILURE);
   }
}
template <typename T>
MeinArray<T>::~MeinArray()
{ delete[] feld; }
```

Fortsetzung1 Beispiel template-Klasse **MeinArray**

```
// copy-Konstruktor, tiefe Kopie:
template <typename T>
MeinArray<T>::MeinArray(
                    const MeinArray& arr)
{
   laenge = arr.laenge;

   feld   = new T[laenge];
   if( feld == NULL )
   {
      cout << "copy-Konstruktor "
           << "MeinArray meldet: "
           << "heap ist voll.\n";
      exit(EXIT_FAILURE);
   }

   for(int i = 0; i < laenge; i++)
      feld[i] = arr[i];
}
```

const operator[] dieser Klasse

```
// operator=, tiefe Kopie:
template <typename T>
void MeinArray<T>::operator=(
                          const MeinArray& arr)
{
   laenge = arr.laenge;

   delete[] feld;

   feld   = new T[laenge];
   if( feld == NULL )
   {
      cout << "MeinArray::operator= "
           << "meldet:  heap ist voll.\n";
      exit(EXIT_FAILURE);
   }

   for(int i = 0; i < laenge; i++)
      feld[i] = arr[i];
}
```

const operator[] dieser Klasse

Fortsetzung2 Beispiel template-Klasse **MeinArray**

```
// operator[]
template <typename T>
T& MeinArray<T>::operator[](int i)
{
   check_index(i);
   return feld[i];
}
```

vorhandener Operator [] für Zeiger **feld**

```
// const operator[]
template <typename T>
const T& MeinArray<T>::operator[](int i)
                                  const
{
   check_index(i);
   return feld[i];
}
```

vorhandener Operator []

```
template <typename T>
void MeinArray<T>::check_index(int index)
                                  const
{
   if( index < 0 || index >= laenge )
   {
      cout << "check_index: Index nicht "
           << "im Wertebereich\n";
      exit(EXIT_FAILURE);
   }
}
```

```
template <typename T>
void MeinArray<T>::zeigDich()
{
   for(int i = 0; i < laenge; i++)
      cout << i << ": " << feld[i] << endl;

   cout << endl;
}
```

Bemerkungen zur Klasse **`MeinArray`**

- Die Zuweisung wird überladen, da die default-Zuweisung keine tiefe Kopie erstellt.
- Als linker Operand der Zuweisung soll nur ein Objekt dieser Klasse erlaubt sein. Das Symmetrieproblem tritt daher nicht auf.

  ➔ Implementation als Methode, kein friend erforderlich.
- Die Implementation ist *fast* identisch zum copy-Konstruktor, aber es ist eine Deallokation erforderlich.
  Anders als der copy-Konstruktor instanziiert die Zuweisung kein neues Objekt. Der heap des „alten linken Objekts“ muss freigegeben werden.
- Der const operator[] ist erforderlich, weil er im copy-Konstruktor und in der Zuweisung für eine const-Objekt aufgerufen wird.

element pointer

Auf nicht statische member kann über sog. element pointer zugegriffen werden. Dies wird zwar relativ selten verwendet, soll hier aber dennoch gezeigt werden, da die etwas exotische Syntax erklärungsbedürftig ist.

Beispiel:

```
class X
{
public:
    int a;
    void f1(int i);
    ...
};
```

Folgende element pointer können dann deklariert (nicht definiert) werden:

```
int X::*zXint;
void (X::*zXvoid)(int);
```

- **`zXint`** ist ein Zeiger, der auf alle **`int`** member der Klasse **`X`** zeigen kann.
- **`zXvoid`** ist ein Zeiger, der auf alle **`void`** Methoden der Klasse **`X`** mit einem **`int`**-Argument zeigen kann.

Den Zeigern können jetzt „Memberadressen“ zugewiesen werden (*an Member binden*), obwohl evtl. noch kein Objekt existiert:

```
zXint  = &X::a;
zXvoid = &X::f1;
```

Zur Dereferenzierung und Bindung dieser Zeiger an ein Objekt werden die speziellen Operatoren **`.*`** und **`->*`** eingesetzt (**`->*`** falls der linke Operand ein Zeiger auf ein Objekt ist).

Beispiel element pointer

```
class X
{
public:
   int a;
   int b;

   X(): a(0), b(0) {}

   void f1(int i)
   {
      cout << "a + i = "
           << a + i << endl;
   }

   void f2( int i )
   {
      cout << "b + i = "
           << b + i << endl;
   }
};
```

Ausgabe:

```
a + i = 52
b + i = 42
b + i = 84
```

```
int main()
{
   int X::*zXint;
   void (X::*zXvoid)(int);

   zXint  = &X::a;
   zXvoid = &X::f1;

   X obj;

   obj.*zXint = 42;      // obj.a=42;
   (obj.*zXvoid)(10);    // obj.f1(10);

   zXint  = &X::b;
   zXvoid = &X::f2;

   obj.*zXint = 22;      // obj.b=22;
   (obj.*zXvoid)(20);    // obj.f2(20);

   X* zobj = new X;
   zobj->*zXint = 54;    // zobj->b=54;
   (zobj->*zXvoid)(30); // zobj->f2(30);
}
```

Bindung an **a** und **f1**

Bindung an **b** und **f2**

neues Objekt, Bindung bleibt