Vorlesung

Objektorientiertes Programmieren in C++

Teil 7 - WS 2024/25

Detlef Wilkening
www.wilkening-online.de

14 Klassen

14.1 Motivation

In der Praxis benötigt man häufig mehrere Variablen, die logisch zusammenhängen, um zu beschreiben, was man darstellen möchte.

Beispiele:

- Bruch
  - int Nenner
  - int Zähler
- Complexe Zahl
  - double Realteil
  - double Imaginärteil
- Datum
  - int Jahr
  - int Monat
  - int Tag
- Person
  - string Vorname
  - string Nachname
  - string Straße
  - string Ort
  - vector<string> Telefonnummern

Hierfür kennen alle ernstzunehmenden Programmiersprachen irgendwelche Sprachmittel. Sie werden z.B. Strukturen, Records oder Verbundtypen genannt. Bei ihnen bündelt man bekannte Typen zu einem neuen Typ. Variablen diese neuen Typs enthalten alle inneren Teile, und es kann einfach auf sie zugegriffen werden.

```
// Achtung - dies ist zwar korrekter C++ Code, aber sowas machen wir nicht.
// Der direkte Zugriff auf die Attribute ist boese - wir werden gleich sehen,
// wie das besser geht.

struct date
{
   int year;
   int month;
   int day;
};

int main()
{
   date d;
   d.year = 2004;
   d.month = 11;
   d.day = 29;

   d.month = 56;
}
```

In der Praxis haben Strukturen mehrere Probleme:

- Häufig sind die Attribute einer Struktur voneinander abhängig. Da jeder auf die Attribute zugreifen kann, ist die Gefahr groß, dass die Attribute ungültige bzw. inkonsistente Werte bekommen. Z.B. sind die erlaubten Tag-Werte in einer Datums-Struktur von Monat und Jahr abhängig.
- Es kann uninitialisierte Strukturen geben, da entweder elementare Datentypen nicht initialisiert werden und zufällige Startwerte haben, oder die Default-Werte der Attribute nicht zusammenpassen.
- Strukturen werden möglicherweise nicht sauber abgebaut - geben z.B. Ressourcen wie Speicher oder Dateien nicht frei.
- Es kann Probleme beim Kopieren oder Zuweisen von Strukturen geben.
- Strukturen alleine (rein die Daten) sind häufig wertlos. Erst durch sie verarbeitende Funktionen werden sie echt leistungsfähig. Z.B. eine Strukur für Datums-Objekte ist für sich nicht besonders hilfreich, sondern wird es erst dann wenn man damit z.B. rechnen kann.
- Freie Funktionen sind hier nicht die beste Wahl für verarbeitende Funktionen auf Strukturen - z.B. wegen Zugriff, Zuordnung, u.a.
- Die interne Repräsentation der Daten läßt sich nicht so einfach ändern – z.B. complexe Zahlen mit Winkel phi und Radius r statt Real- und Imaginärteil, wenn jeder die Repräsentation kennt und nutzt.

Darum hat man in der Objektorientierung das Sprachmittel von Klassen eingefügt, das neben der Adressierung dieser Probleme auch noch viele weitere Möglichkeiten enthält, z.B. Vererbung und Polymorphie.

14.2 Klassen-Definition

Eine Klasse ist in C++ ein benutzerdefinierter Typ, und daher gilt für sie alles, was wir für Typen kennengelernt haben.

Eine Klasse muss in C++ **immer** definiert werden, bevor ihre Elemente (Klassen-Variablen, Element-Funktionen, Konstruktoren,...) implementiert werden können, oder die Klasse benutzt werden kann.

Erstmal kann eine Klasse für uns nur zwei Dinge enthalten:

- Funktionen, sogenannte **Element-Funktionen**, oder auch Memberfunctions
  Im folgenden Bsp. sind das die beiden Element-Funktionen init und print.
  Achtung - dies sind - wie man sofort sieht - ganz normale Funktions-Deklarationen.
- und Daten, sogenannte **Element-Variablen**, **Attribute**, oder Properties
  Im Bsp. sind das die drei „int“ Attribute für Jahr, Monat und Tag.

```
class date
{
public:
   void init(int, int, int);
   void print();

private:
   int day_;
   int month_;
   int year_;
};
```

14.2.1 Attribute

Die Attribute eine Klasse sind ganz normale Variablen, die einfach zu einem neuen Typ zusammengefasst werden.

Für die Namen von Attributen findet sich häufig eine der folgenden Konventionen:

- Präfix „m_“, klein beginnend und kapitalisiert geschrieben.
  Bsp.: „m_year“, „m_networkPort“
- Präfix „m“, groß beginnend und kapitalisiert geschrieben.
  Bsp.: „mYear“, „mNetworkPort“
- klein beginnend, kapitalisiert geschrieben, und mit Postfix „_“.
  Bsp.: „year_“, „networkPort_“
- klein beginnend, mit „_“ getrennt, und mit Postfix „_“.
  Bsp.: „year_“, „network_port_“

Hinweis – keine dieser Konventionen ist von der Sprache her vorgeschrieben, oder auch nur empfohlen. In der Praxis hat es sich aber bewährt, die Attribute im Namen als Attribute zu kennzeichnen, da sie eine besondere Rolle spielen. Und die obigen vier Konventionen finden sich halt häufig hierfür wieder - das Skript folgt der vierten Konvention. Z.B. Boost, Google oder Geosoft's verwenden den Postfix „_“, Microsoft verwendet den Präfix „m_“ mit Kapitalisierung (z.B. zu finden in der MFC), oder Possibility das Präfix „m“.

14.2.2 Element-Funktionen

Element-Funktionen sind im Prinzip ganz normale Funktionen. Es gelten daher alle bekannten Features z.B. bzgl. Deklaration, Überladen, Default-Argumente, Parameterliste, usw. Und im Beispiel sehen die Element-Funktions-Deklarationen ja auch wie ganz normale Deklarationen aus - und sind es auch.

14.2.3 Zugriffsspezifizierer

Mit den Schlüsselwörtern **public** und **private** werden Zugriffsrechte vergeben. Auf alle Elemente (Attribute, Funktionen,...)

- im **public**-Bereich kann von ausserhalb und innerhalb der Klasse,
- im **private**-Bereich kann nur von innerhalb der Klasse

zugegriffen werden.

14.2.4 Element-Funktions-Definitionen

Die Element-Funktionen einer Klasse müssen - als normale Funktionen - natürlich definiert werden. Der offensichtlichste Unterschied gegenüber der Definition von freien Funktionen ist ein aufwändigerer Funktions-Name, da sich dieser aus Klassen-Name, Scope-Resolution-Operator und dem eigentlichen Funktions-Namen zusammensetzt.

Syntax:
Rückgabetyp Klassen-Name :: Funktionsname (Parameterliste) { Anweisungen }

```
// Achtung - die Definition der Klasse 'date' muss dem Compiler bekannt sein!
// Daher die Klassen-Definition muss vorher im Quelltext stehen.

void date::init(int d, int m, int y)
{
   day_ = d;
   month_ = m;
   year_ = y;
}

void date::print()
{
   std::cout << day_ << '.' << month_ << '.' << year_;
}
```

Element-Funktionen sind fest an eine Klasse und ihren Kontext gebunden. Von daher muss die Klasse definiert worden sein, bevor eine Element-Funktion definiert werden kann, damit der Kontext (der Aufbau der Klasse) bekannt ist.

Element-Funktion einer Klasse können auf die **private** Attribute der Klasse zugegreifen, da sie Teil der Klasse sind und damit auch Zugriff auf den **private**-Bereiche haben.

Element-Funktionen sind in den Kontext einer Klasse eingebunden, d.h. können sie direkt über den Attribut-Namen ohne Klassenbezug auf die Attribute der Klasse zugreifen.

14.2.5 Klassen-Benutzung

Mit der Definition von „date“ ist „date“ zu einem benutzerdefinierten Datentyp geworden. Daher können von ihm **Objekte** angelegt werden - Syntax: „typ variablenname;“.

```
// Achtung - die Definition der Klasse 'date' muss dem Compiler bekannt sein!
// Achtung - die Element-Funktions Definitionen muessen fuer den Linker vorhanden sein!

int main()
{
   date d1;                 // ein Datums-Objekt wird erzeugt
   date d2, d3;             // zwei Datums-Objekte werden erzeugt

   d1.init(29, 11, 2004);   // d1 wird mit dem 29.11.2004 initialisiert
   d2.init(10,  5, 1999);   // d2 wird mit dem 10.05.1999 initialisiert
   d3.init( 5,  1, 2005);   // d3 wird mit dem 05.01.2005 initialisiert

   d1.print();              // => 29.11.2004
   d2.print();              // => 10.05.1999
   d3.print();              // => 05.01.2005
}
```

Auf die Objekt-Komponenten kann über das Objekt mit dem Punkt-Operator zugegriffen werden - hier im Bsp. sieht man dies für die Element-Funktionen „init“ und „print“. Die Element-Funktionen greifen hierbei auf die Daten des Objekts zu, mit dem sie aufgerufen wurden, d. h. auf die Daten des aktuellen Objekts. **Element-Funktionen haben immer einen Objektbezug.**

14.3 Zugriffsbereiche

Wie schon in angedeutet, werden mit den Zugriffsspezifizierern **public** und **private** die Zugriffsrechte auf Klassen-Elemente vergeben.

Auf alle Elemente (Attribute, Funktionen,...)

- im **public**-Bereich kann von ausserhalb und innerhalb der Klasse,
- im **private**-Bereich kann nur von innerhalb der Klasse

zugegriffen werden.

Der Default-Bereich in der Klassen-Definiton ist **private**.

Ein Zugriffsbereich darf leer sein.

Die Zugriffsbereiche dürfen mehrfach in beliebiger Reihenfolge vorkommen.

```
class A
{
   void f1();
   long l1;
public:
   void f2();
   long l2;
private:
   void f3();
   long l3;
private:
   void f4();
   long l4;
private:
public:
```

```
   void f5();
   long l5;
};

int main()
{
   A a;

   a.f1();        // Compiler-Fehler -> kein Zugriff von aussen, da private
   a.l1=7;        // "               -> "    "       "   "     , "  "

   a.f2();        // okay, Zugriff von aussen erlaubt, da public
   a.l2=7;        // "   , "       "   "      "      , "  "

   a.f3();        // Compiler-Fehler -> kein Zugriff von aussen, da private
   a.l3=7;        // "               -> "    "       "   "     , "  "

   a.f4();        // Compiler-Fehler -> kein Zugriff von aussen, da private
   a.l4=7;        // "               -> "    "       "   "     , "  "

   a.f5();        // okay, Zugriff von aussen erlaubt, da public
   a.l5=7;        // "   , "       "   "      "      , "  "
}
```

Bezeichnungen

- Der **public**-Bereich wird oft als Interface bezeichnet, da er die Schnittstelle der Klasse nach aussen darstellt.
- Der **private**-Bereich wird oft als Implementationsbereich bezeichnet, da er nur für den internen Gebrauch der Klasse zur Verfügung steht - der Implementierung.

Information Hiding

- **Alle Attribute sollten private sein.**
- **Zugriff auf die Attribute nur über Element-Funktionen.**

Im Sinne einer sauberen und sicheren Programmierung sollten alle Attribute in den private-Bereich einer Klasse liegen und alle Zugriffe über Element-Funktionen abgewickelt werden.

Die Zugriffsspezifizierer beziehen sich auf **alle** Symbole im zugehörigen Bereich, egal ob Attribute, Element-Funktionen, Operatoren, Konstruktoren, Destruktoren, usw., und egal ob normal, **virtual**, **static** oder **inline**.

Hinweis – neben public und private gibt es in C++ noch den Zugriffsberich **protected**, der von seiner Wirkung zwischen den beiden liegt. Erst mit Vererbung bekommt er einen Sinn, und wird auch erst dann vorgestellt.

Hinweis – bei der Anordnung der Zugriffsbereiche haben sich zwei Konventionen eingebürgert:

- Konvention 1 nutzt aus, dass der Default-Bereich „private“ ist, d.h. beginnt direkt ohne Zugriffsspezifizierer mit dem Private-Bereich, dann folgt der protected Bereich, und zum Schluß der public Bereich.
- Konvention 2 ordnet die Bereiche nach ihrer Wichtigkeit für den Benutzer der Klasse. Da dieser nur das Interface der Klasse, d.h. den public Bereich nutzen kann, kommt dieser als erstes. Dann folgt der protected Bereich, der nur noch für manche Benutzer interessant ist. Und zum Schluß der private Bereich, da dieser nur für den Entwickler der Klasse selber wichtig ist.

Das Skript folgt hier der zweiten Konvention.

14.4 Klassen sind benutzerdefinierte Typen

Klassen sind benutzerdefinierte Typen, und verhalten sich wie wir von Typen erwarten:
1. Referenzen (auch const) auf Objekte sind möglich.
2. Kopieren und Zuweisen ist möglich
3. Aufruf an Funktionen mit cbv (default) und cbr möglich.

```
void f_cbv(date d)          // call-by-value, d.h. Kopie wird angelegt
{
   d.print();               // => 29.11.2004
   d.init(24, 12, 2005);
   d.print();               // => 24.12.2005
}

void f_cbr(date& d)         // call-by-reference
{
   d.print();               // => 29.11.2004
   d.init(24, 12, 2005);
   d.print();               // => 24.12.2005
}

int main()
{
   date d, d2;
   d.init(29, 11, 2004);
   d.print();               // => 29.11.2004
   f_cbv(d);
   d.print();               // => 29.11.2004
   f_cbr(d);
   d.print();               // => 24.12.2005
   d2 = d;
   d2.print();              // => 24.12.2005
}
```

14.5 Objekt-Orientierung

Einer der Hauptgedanken der Objekt-Orientierung ist die Idee, abgeschlossene gekapselte Einheiten programmieren und anbieten zu können, bei denen der Benutzer sich keine Gedanken mehr über das Innenleben machen muss, sondern einfach die Objekte über deren Schnittstelle benutzt.

Wir haben solche Objekte schon kennen gelernt, z.B.:
- Streams
- Strings
- Container (vector, list, map, set,...)
- Iteratoren

Diese Idee hat viele bestechende Vorteile:
- Information-Hiding
- Daten versteckt (gegenseitige Abhängigkeiten, unterschiedliche Repräsentationen,...)
- Klare Schnittstelle

14.6 Erweiterung

Wunsch – ein Datums Objekt soll mit dem aktuellen Datum initialisiert werden können.

Lösung – z.B. zweite Element-Funktion „*init*“ ohne Parameter (d.h. Überladen).

```
#include <ctime>

class date
{
public:
   void init();                       // Neue Funktion - Rest wie bisher
   void init(int, int, int);
   void print();

private:
   int day_;
   int month_;
   int year_;
};

// Initialisiert das date-Objekt mit dem aktuellem Datum.
void date::init()
{
   std::time_t timer = std::time(0);
   std::tm* tblock = std::localtime(&timer);
   day_ = tblock->tm_mday;
   month  = tblock->tm mon+1;
   year_ = tblock->tm_year+1900;
}
```

```
date d;
d.init();
d.print();        // Ausgabe aktuelles Datum
```

Hinweis – man sollte in der Realität die Element-Funktion vielleicht aussagekräftiger „set_to_now“ oder „today“ oder so nennen. Aber ich wollte dies auch gleich als Beispiel nutzen, dass sich natürlich auch Element-Funktionen überladen lassen.

14.7 Objekt-Zustand

Problem – es kann passieren, dass ein Datums-Objekt ein Datum repräsentiert, das es nicht gibt, z. B. den 789.-2.0

Lösung – um dies zu verhindern, bauen wir eine *private* Testfunktion ein, die nach jeder Änderung des inneren Zustands aufgerufen wird, und diesen auf Korrektheit überprüft. Im Falle eines Problems wird eine Fehlermeldung ausgegeben und das Programm hart mit der Funktion „std::exit(int)“ aus „cstdlib“ beendet.

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

class date
{
public:
   void init();
   void init(int, int, int);
   void print();
```

```
private:
   void test();                        // Neue Funktion - Rest wie bisher

   int day ;
   int month_;
   int year_;
};

void date::init(int d, int m, int y)
{
   day_ = d;
   month_ = m;
   year_ = y;
   test();              // Und spaeter auch an allen anderen relevanten Stellen
}

// Testet, ob das Datums Objekt okay ist, und beendet im Fehlerfall mit
// einer Meldung das Programm - die Implementierung ist dem Praktikum
// ueberlassen.
void date::test()
{
   if ( ??? )
   {
      cout << "Datums-Objekt ";
      print();
      cout << " ist nicht korrekt\n";
      exit(1);
   }
}
```

```
int main()
{
   date d;
   d.init(29, 11, 2004);     // okay
   d.init(789, 1, 1999);     // Programm-Abbruch
}
```

Ein wichtiger Grundsatz in C++ ist, dass ein Objekt **immer** einen sauberen wohldefinierten Zustand haben sollte. Sobald Sie zulassen, dass ein Objekt einen unsauberen Objekt-Zustand erhalten kann, stehen Problemen und Fehlern „*Haus und Hof*“ offen. Denn das hiesse, dass vor jeder Benutzung eines Objekts sein Zustand abgefragt werden müsste. Und das ist einfach nicht praktikabel, würde die Benutzung unnötig erschweren, und die Akzeptanz untergraben. In der Praxis würde die Klasse dann entweder nicht oder nur mit einer „*es wird schon nicht schiefgehen*“ Mentalität benutzt werden. Treten dann fehlerhafte Zustände auf, so vermehren sie sich im Programm bis irgendwann komische Effekte auftreten. Wenn sie Glück haben, passiert nichts wildes, wenn sie aber Pech haben stehen mittlerweile komplett falsche Daten in der Datenbank und sonst was. Das zweite schlimme an der Situation ist, dass solche Fehler ja nicht sofort auffallen, sondern erst viel später – und dann wird die Fehlersuche oft sehr schwierig und mühsam.

Probleme sollten an der Wurzel bekämpft werden, damit sie nicht wachsen können, darum gewährleisten Sie, dass Objekte Ihrer Klassen **immer** einen sauberen wohldefinierten Zustand haben.

14.8 Konstruktoren

Wir haben gelernt, dass lokale Variablen einiger Typen bei der Definition rein zufällige Startwerte bekommen – z.B. alle elementaren Datentype.. Dem gegenüber ist z.B. ein String immer ein Leerstring, wenn er ohne Argumente erzeugt wird:

```
int main()
{
   int i;                                          // zufaelliger Startwert
   string s;                                       // genau definiert -> Leerstring
   cout << '"' << s "\" - " << i << '\n';
}
```

Dies gilt auch, wenn diese Typen in Klassen liegen, und ein Objekt der Klasse als lokale Variable erzeugt wird:

```
class A
{
public:
   int i;
   string s;
};

int main()
{
   A a;                                                // a.s ist Leerstring, a.i ist zufaellig
   cout << '"' << a.s << "\" - " << a.i << '\n';
}
```

Da ein Objekt **niemals** einen instabilen Zustand haben sollte, ist dieses Verhalten schlecht. Darum ist es möglich, ein Objekt direkt bei der Erstellung sauber zu initialisieren. Hierfür gibt es in C++ spezielle Element-Funktionen, die **Konstruktoren**:

- Konstruktoren tragen **immer** den Namen der Klasse.
- Sie haben **keinen** Rückgabewert, auch **nicht** „void“.
- Wird ein Objekt erzeugt, so wird **immer** automatisch der entsprechende Konstruktor aufgerufen – dies gilt ohne Ausnahme.
- Der entsprechende Konstruktor ergibt sich aus den Argumenten beim Aufruf – hier gelten die normalen Funktions-Überladen-Regeln.

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

class A
{
public:
   A();                          // <= Deklaration Konstruktor (1)
   A(int);                       // <= Deklaration Konstruktor (2)
   A(double);                    // <= Deklaration Konstruktor (3)
   A(int, double);               // <= Deklaration Konstruktor (4)

   void print();

private:
   int n_;
   double d_;
};

A::A()                           // <= Definition Konstruktor (1)
{
   cout << "A::A()\n";
   n_ = 0;
   d_ = 0.0;
}

A::A(int n)                      // <= Definition Konstruktor (2)
{
   cout << "A::A(int " << n << ")\n";
   n_ = n;
   d_ = 0.0;
}

A::A(double d)                   // <= Definition Konstruktor (3)
```

```
{
   cout << "A::A(double " << d << ")\n";
   n_ = 0;
   d  = d;
}

A::A(int n, double d)           // <= Definition Konstruktor (4)
{
   cout << "A::A(int " << n << ", double "  << d << ")\n";
   n  = n;
   d_ = d;
}

void A::print()
{
   cout << "=> n:" << n  << " - d:" << d  << '\n';
}

int main()
{
   A a1;                        // <= Nutzung Konstruktor (1)
   a1.print();                  // => n:0 - d:0

   A a2(4);                     // <= Nutzung Konstruktor (2)
   a2.print();                  // => n:4 - d:0

   A a3(2.7);                   // <= Nutzung Konstruktor (3)
   a3.print();                  // => n:0 - d:2.7

   A a4(6, 3.1);                // <= Nutzung Konstruktor (4)
   a4.print();                  // => n:6 - d:3.1
}
```

Ausgabe

```
A::A()
=> n:0 - d:0
A::A(int 4)
=> n:4 - d:0
A::A(double 2.7)
=> n:0 - d:2.7
A::A(int 6, double 3.1)
=> n:6 - d:3.1
```

Für Konstruktoren gilt:

- Konstruktoren **sollen** das Objekt sauber konstruieren.
- Sie dürfen überladen werden.
- Es dürfen Default-Argumente benutzt werden.
- Ihr vorzeitiges Ende kann mit return (ohne Ausdruck) erreicht werden.
- Bis auf ihren speziellen Verwendungszweck, ihrem fehlenden Rückgabetyp und der fehlenden Adresse sind sie ganz *normale* Element-Funktionen.

Übertragen auf unsere Klasse „date“ bedeutet dies, dass wir zwei Konstruktoren zur Verfügung stellen sollten:

- Einen Konstruktor ohne Parameter für das aktuelle Datum
- Einen Konstruktor mit drei Int-Parametern für die Übergabe von Tag, Monat und Jahr.

```
class date
{
public:
   date();                          // <= Deklaration Konstruktor (1)
   date(int, int, int);             // <= Deklaration Konstruktor (2)
   // Rest wie bisher
};

date::date()                        // <= Definition Konstruktor (1)
```

```
{
   init();
}

date::date(int d, int m, int y)     // <= Definition Konstruktor (2)
{
   init(d, m, y);
}

int main()
{
   date d1;                         // <= Nutzung Konstruktor (1)
   d1.print();                      // => <aktuelles Datum>

   date d2(18, 10, 2001);           // <= Nutzung Konstruktor (2)
   d2.print();                      // => 18.10.2001
}
```

Im Rahmen unseres bisherigen Wissens können wir also sagen, dass bei einer Objekt-Erstellung erst Speicher bereitgestellt, und dann **immer** der Konstruktor aufgerufen wird, der die Attribute des Objekts sauber initialisiert (sauber initialisierung sollte!).

Es gibt mehrere spezielle Konstruktoren bzw. Konstruktor-Familien, die wir in den nächsten Kapiteln näher besprechen werden:

- Standard-Konstruktor – siehe Kapitel 14.8.1
- Konvertierungs-Konstruktoren – siehe Kapitel 14.8.2
- Kopier-Konstruktor(en) – siehe Kapitel 14.8.4
- Move-Konstruktor – siehe Kapitel 14.8.5
- Sequenz-Konstruktor – siehe Kapitel 14.8.6

Hinweis – einige der spezielle Konstruktoren (Standard-, Kopier- und Move-Konstruktor) werden unter gewissen Umständen automatisch vom Compiler erzeugt – siehe die folgenden Kapitel 14.8.1 bis 14.8.5. Man nennt diese automatisch erzeugten Konstruktoren „implizite Konstruktoren“ oder auch „automatische Konstruktoren“ („impliziter Standard-Konstruktor“, „impliziter Kopier-Konstruktor“, …). Alle diese Konstruktoren kann man auch selber schreiben. In diesem Fall nennt man sie „explizite Konstruktoren“.

14.8.1 Standard-Konstruktor

Bislang konnte die „date“-Klasse genutzt werden, obwohl sie keinen Konstruktor enthielt. Dabei hieß es aber eben doch, dass bei **jeder** Objekterzeugung der entsprechende Konstruktor aufgerufen wird. Wie funktioniert das denn, wo die Klasse „date“ doch gar keinen Konstruktor hatte?

Dies war kein Problem, denn: Wenn Sie in der Klassen-Defintion **keinen einzigen** Konstruktor deklarieren, erzeugt der Compiler **automatisch** einen public Standard-Konstruktor, der für alle Datenelemente (inkl. Basis-Klassen) wiederum deren Standard-Konstruktoren aufruft. Dieser Konstruktor heißt:

- Impliziter Standard-Konstruktor, oder
- Automatischer Standard-Konstruktor

```
class A           // Klasse hat keinen user-deklarierten Konstruktor
```

```
{                  // => Compiler erzeugt impliziten Standard-Konstruktor
public:
   void fct();
};

int main()
{
   A a;            // okay - impliziter Standard-Konstruktor wird genutzt
   a.fct();
}
```

Deklarieren Sie dagegen **mindestens einen** beliebigen Konstruktor in der Klassen-Definition, so erzeugt der Compiler **keinen** Standard-Konstruktor. Benötigen Sie ihn trotzdem, so müssen Sie ihn dann selber erzeugen.

```
class A
{
public:
   A(int);       // User-deklarierte Konstruktor => kein impliziter Standard-Konstruktor
   void fct();
};

int main()
{
   A a1(1);      // Okay
   a1.fct();

   A a2;         // Compiler-Fehler -> kein passender Konstruktor (Standard-Konstruktor)
   a2.fct();
}
```

```
class A
{
public:
   A();          // Expliziter Standard-Konstruktor
   A(int);       // User-deklarierte Konstruktor => kein impliziter Standard-Konstruktor
   void fct();
};

int main()
{
   A a1(1);      // Okay
   a1.fct();

   A a2;         // Jetzt auch okay, nutzt den expliziten Standard-Konstruktor
   a2.fct();
}
```

Hinweis – selbst wenn diese Beispiele den oder die Konstruktor(en) nicht definiert haben – es sollte Ihnen klar sein, dass die Konstruktor-Definitionen natürlich für ein komplettes Programm notwendig sind.

Wenn Sie einen Standard-Konstruktor benötigen, der Compiler aber keinen für Sie erzeugt (da es mindestens einen user-deklarierten Konstruktor gibt), so müssen Sie ihn selbst deklarieren und definieren (s.o.). Wenn Ihnen der implizite Standard-Konstruktor ausgereicht hätte (daher der, den der Compiler eigentlich für Sie erzeugt hätte), dann gibt es in C++ eine einfache Lösung: Sie deklarieren den Standard-Konstruktor mit „ = default“ – dann erzeugt der Compiler für Sie explizit den impliziten Standard-Konstruktor:

```
class A
{
public:
   A() = default;    // Compiler erzeugt Standard-Konstruktor => keine Impl. notwendig
   A(int);           // Ihr eigener Konstruktor - Implementierung notwendig
   void fct();
```

```
};

A::A(int)            // Eigene Implementierung vom Int-Konstruktor
{
}

int main()
{
   A a1(1);     // okay
   a1.fct();

   A a2;        // okay
   a2.fct();
}
```

Umgekehrt kann man in C++ den impliziten Standard-Konstruktor auch verbieten, so daß der Compiler ihn niemals erzeugt. Dazu muss die Deklaration des Standard-Konstruktors mit „=delete“ abgeschlossen werden:

```
class A
{
public:
   A() = delete;    // Compiler verbietet den Standard-Konstruktor
   void fct();
};

int main()
{
   A a;     // Compiler-Fehler - kein Standard-Konstruktor vorhanden
}           //                   Von "A" kann kein Objekt erzeugt werden
```

Hinweis – das explizite Verbieten des Standard-Konstruktors ist eher ein Spezialfall. In der Praxis findet man das Verbieten von Konstruktoren mit „=delete“ eher bei den Kopier- und Move-Konstruktoren.

In C++ ist der Standard-Konstruktor nicht über die leere Parameterliste, sondern über den Aufruf definiert: **Der Konstruktor, der ohne Argumente aufgerufen werden kann, ist der Standard-Konstruktor oder auch Default-Konstruktor.**

Ein Standard-Konstruktor ist also:

- entweder ein Konstruktor ohne Parameter, bzw.
- einer, bei dem sämtliche Parameter mit Default-Argumenten belegt sind.

```
class A
{
public:
   A(int = 42);      // Dies ist auch ein Standard-Konstruktor
   void fct();
};

A::A(int n)
{
   cout << "A(" << n << ")\n";
}

int main()
{
   A a1;             // Aufruf des Standard-Konstruktors mit "42" - Default-Argument
   a1.fct();

   A a2(6);          // Aufruf des Standard-Konstruktors mit "6"
   a2.fct();
}
```

Ausgabe

```
A(42)
A(6)
```

Hinweis – in C++ ist der Standard-Konstruktor ein relativ wichtiger Konstruktor. Immer wenn Objekte erzeugt werden, ohne das der Benutzer einen speziellen Konstruktor angibt, dann werden sie automatisch mit dem Standard-Konstruktor erzeugt. Da in C++ häufig wertbasiert programmiert wird, passiert es relativ häufig, dass Objekte im Hintergrund einfach so erzeugt werden. Fast immer kann man die Erzeugung steuern – defaultmäßig wird aber immer der Standard-Konstruktor genommen.

Achtung – es gibt in C++ in Verbindung mit dem Standard-Konstruktor eine kleine Falle: Wollen Sie ein Objekt mit dem Standard-Konstruktor initialisieren, so dürfen Sie keine runden Klammern verwenden.

```
class A
{
public:
   A();
   A(int);
   void fct();
};

int main()
{
   A a1(6);
   A a2();      // Hier liegt der eigentliche Fehler, die Zeile ist aber syntaktisch okay

   a1.fct();
   a2.fct();    // Compiler-Fehler mit komischer Fehlermeldung vom Compiler
}
```

Lösung – „A a2()“ ist keine Objektdefinition, sondern eine Funktions-Deklaration der Funktion „a2“, die keine Parameter erwartet und ein A-Objekt per Kopie zurückgibt.

Tipp – der Fehler tritt nicht bei der *falschen* Objektdefinition auf, da diese eine syntaktisch korrekte Funktions-Deklaration ist, sondern erst bei der Verwendung des vermeintlichen Objekts. Schauen Sie sich bei einem solch unerklärlichen Fehler also ruhig mal Ihre Objekt-Definition an.

14.8.2 Temporäre Objekte

Wir können in C++ Konstruktoren explizit aufrufen und damit temporäre Objekte erzeugen. Temporäre Objekte sind Objekte, die keinen Namen haben (d.h. z.B. keine Variablen sind) und am Ende der Anweisung automatisch wieder zerstört werden.

Nehmen wir als Beispiel eine Klasse „A“, deren Objekte man mit 2 Int-Argumenten erzeugen kann und eine Funktion „fct“, die man mit einem A-Objekt aufrufen kann.

```
class A
{
public:
   A(int, int);
};

void fct(const A&);
```

Sind jetzt 2 Int-Variablen (z.B. „x1“ und „x2“) vorhanden, die zusammen ein A-Objekt darstellen (vielleicht Zähler und Nenner für einen Bruch oder so), so kann man explizit ein A-Objekt erzeugen und damit die Funktion „fct“ aufrufen:

```
int x1 = ..., x2 = ...;

A temp(x1, x2);
fct(temp);
```

Dies kann man in C++ auch kürzer schreiben:

```
fct(A(x1, x2));
```

Der explizite Konstruktor-Aufruf von „A“ erzeugt ein temporäres Objekt von „A“, das keinen Namen hat, und am Ende der Anweisung (also quasi beim Semikolon) automatisch zerstört wird. Damit kann das temporäre Objekt problemlos in der Funktion „fct“ genutzt werden, da die Anweisung erst nach Rückkehr aus der Funktion beendet ist und erst dann das Objekt zerstört wird.

In gewisser Weise ist dies eine explizite Konvertierung, denn die Objekte „x1“ und „x2“ werden zu einem A-Objekt gewandelt. Um zu zeigen, dass ein expliziter Konstruktor-Aufruf semantisch nur eine Konvertierung ist und sich auch entsprechend verhält, erweitern wir das Beispiel etwas. Wir fügen der Klasse „A“ noch einen Konstruktor mit nur einem Int-Parameter hinzu (damit wir u.a. „static_cast“ nutzen können), und vervollständigen das Beispiel noch mit einigen Ausgaben:

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

class A
{
public:
   A(int);              // Konstruktor mit einem "int" - geht auch mit z.B. "static_cast"
   A(int, int);         // Konstruktor mit zwei "int" - geht nur im funktionalen Stil

   void print();

private:
   int n1_, n2_;
};

A::A(int n1)
{
   n1_ = n1;
   n2_ = 0;
}

A::A(int n1, int n2)
{
   n1_ = n1;
   n2_ = n2;
}

void A::print()
{
   cout << "A mit n1:" << n1_ << " - n2:" << n2_ << '\n';
}

void fct(A a)           // Achtung - nun als Kopie - eigentlich schlechter - siehe Text (*)
{
   cout << "fct(A)\n-> ";
   a.print();
}
```

```
int main()
{
   fct(A(1, 2));               // Explizites temporaeres A-Objekt, funktionaler Stil

   fct(A(3));                  // Explizites temporaeres A-Objekt, funktionaler Stil
   fct((A)4);                  // Explizites temporaeres A-Objekt, alter C-Stil
   fct(static_cast<A>(5));     // Explizites temporaeres A-Objekt, mit "static_cast"
}
```

Ausgabe

```
fct(A)
-> A mit n1:1 - n2:2
fct(A)
-> A mit n1:3 - n2:0
fct(A)
-> A mit n1:4 - n2:0
fct(A)
-> A mit n1:5 - n2:0
```

Im Prinzip ist das Beispiel selbsterklärend, da es keine Neuigkeiten enthält, sondern nur bekannte Features wiederholt und zusammenfaßt. Einzige Besonderheit ist, dass die Funktion „fct“ in Zeile "(*)" das A-Objekt nicht mehr als Const-Referenz sondern als Kopie bekommt. Dies ist eigentlich eine Verschlechterung, denn wir haben ja gelernt, dass die Const-Referenz Übergabe bei Objekten zu bevorzugen ist. Hier musste ich auf die schlechtere Lösung mit der Kopie zurückfallen, da wir noch keine Const-Element-Funktionen kennen (siehe Kapitel 14.10), und ohne die die Funktion mit Const-Referenz nicht compilieren würde.

14.8.3 Konvertierungs-Konstruktoren

In den Kapiteln über implizite Konvertierungen und die Konvertierungs-Hierarchien wurde schon erwähnt, dass man in C++ auch benutzer-definierte Konvertierungen definieren kann, die der Compiler auch für implizite Konvertierungen nutzen darf. Diese benutzer-definierten Konvertierungen definiert man entweder mit Konvertierungs-Konstruktoren oder Konvertierungs-Operatoren (die in diesem Tutorial leider nicht besprochen werden).

Im Prinzip ist jeder Konstruktor, den man mit einem Argument aufrufen kann, ein Konvertierungs-Konstruktor:

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

class A
{
public:
   A(int);                    // Konvertierungs-Konstruktor
   A(const string&);          // Konvertierungs-Konstruktor
   A(bool, double = 3.14);    // Konvertierungs-Konstruktor - dank Default-Argument
};

A::A(int n)
{
   cout << "A(int: " << n << ")\n";
}

A::A(const string& s)
{
   cout << "A(string: " << s << ")\n";
}
```

```
A::A(bool b, double d)
{
   cout << "A(bool: " << b << ", double: " << d << ")\n";
}

void fct(const A&)
{
}

int main()
{
   cout << boolalpha;

   string str("C++");

   fct(1);          // Erzeugt mit Konvertierungs-Konstruktor "A(int)" temporaeres Objekt
   fct(str);        // Dito mit Konvertierungs-Konstruktor "A(const string&)"
   fct(true);       // Dito mit Konvertierungs-Konstruktor "A(bool, double=3.14)"
}
```

Ausgabe

```
A(int: 1)
A(string: C++)
A(bool: true, double: 3.14)
```

Möchte man nicht, dass ein Ein-Parameter-Konstruktor als Konvertierungs-Konstruktor zur Verfügung steht – z.B. um Mehrdeutigkeiten und Fehler zu vermeiden – so kann man ihn „explicit“ machen.

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

class A
{
public:
   explicit A(int);                  // <= Kein Konvertierungs-Konstruktor mehr: "explicit"
};

void fct(const A&)
{
}

int main()
{
   fct(2);               // Compiler-Fehler - kein Konvertierungs-Konstruktor vorhanden
   fct(A(3));            // Explizite Konvertierung geht natuerlich weiterhin
}
```

14.8.3.1 Konvertierungs-Konstruktoren in C++11

In C++11 wurde der Begriff der Konvertierungs-Konstruktoren noch erweitert. Jetzt können im Prinzip **alle** Konstruktoren Konvertierungs-Konstruktoren sein – nicht nur die, die mit einem Argument aufrufbar sind. Auch die, die mit keinem oder mehreren Argumenten aufrufbar sind. Damit beim Aufruf klar ist, welche Argumente zusammen ein Objekt bilden sollen, müssen diese dann in geschweifte Klammern gesetzt werden.

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

class A
{
public:
   A(int, int);                  // In C++03 KEIN Konvertierungs-Konstruktor
};                               // In C++11 mit {} als solcher nutzbar

A::A(int n1, int n2)
{
   cout << "A(n1: " << n1 << ", n2: " << n2 << ")\n";
```

```
}

void fct(const A&)
{
}

int main()
{
   fct( { 1, 2 } );     // Implizite Konvertierung mit geschweiften Klammern in C++11
}
```

Ausgabe
```
A(n1: 1, n2: 2)
```

Und die implizite Konvertierung mit „{}“ funktioniert auch für einen Standard-Konstruktor:

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

class A
{
public:
   A();
};

A::A()
{
   cout << "A()\n";
}

void fct(const A&)
{
}

int main()
{
   fct({});       // Aufruf von "A()" durch die geschweiften Klammern "{}"
}
```

Ausgabe
```
A()
```

Auch hier kann man die implizite Konvertierung wieder mit dem Schlüsselwort „explicit“ verhindern:

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

class A
{
public:
   explicit A(int, int);         // Auch in C++11 kein Konvertierungs-Konstruktor mehr
};

void fct(const A&)
{
}

int main()
{
   fct( { 1, 2 } );     // Compiler-Fehler - da Konstruktor "explicit"
   fct(A(3, 4));        // Explizite Konvertierung natuerlich weiterhin moeglich
}
```

Diese implizite Konvertierung mit den geschweiften Klammern funktioniert natürlich nicht nur bei Funktions-Aufrufen, sondern überall – also auch z.B. bei Funktions-Rückgaben. Beide Situationen haben wir auch schon kennen gelernt:

- Bei Funktions-Aufrufen haben wir sie schon bei der Nutzung der Element-Funktion „insert“

bei Maps gesehen.
- Bei Funktions-Rückgaben wurden sie auch schon vorgestellt.

14.8.3.2 Namens-Konvention

In diesem Tutorial verwende ich folgende Namens-Konvention:
- **Primäre Konvertierungs-Konstruktoren** sind Konvertierungs-Konstruktoren, die mit einem Argument aufgerufen werden können, d.h. sie können für implizite Konvertierungen ohne die geschweiften Klammern genutzt werden.
- **Sekundäre Konvertierungs-Konstruktoren** sind Konvertierungs-Konstruktoren, die **nicht** mit einem Argument aufgerufen werden können, und daher für implizite Konvertierungen die geschweiften Klammern benötigen. Sekundäre Konvertierungs-Konstruktoren gibt es daher nur in C++11.

Achtung – dies ist meine private Namens-Konvention. Ich kenne keine offizielle Namens-Konvention um die Konvertierungs-Konstruktoren zu unterscheiden.

14.8.4 Kopier-Konstruktoren

Immer wenn eine Kopie eines Objektes erzeugt wird, wird ein **Kopier-Konstruktor** (oder auch „copy-constructor“) der Klasse des Objekts aufgerufen. Kopien werden z.B. erzeugt, wenn eine Funktion einen Parameter „call-by-value“ erwartet, eine Funktion ein Objekt als Kopie zurückgibt, oder einfach ein Objekt aus einem anderen erzeugt wird:

```
void f(std::string);          // Parameter-Uebergabe "call-by-value"
std::string g();              // Funktions-Rueckgabe als Kopie

std::string s1;
std::string s2(s1);           // Kopie eines Objekts anlegen
```

Welchen Konstruktoren sind denn jetzt Kopier-Konstruktoren?

Jeder Konstruktor einer Klasse, der mit einem einzelnen Objekt der Klasse aufgerufen werden kann, ist ein Kopier-Konstruktor.

- Ein Kopier-Konstruktor muss das erste Argument per Referenz bekommen (sowohl const als auch non-const – normal ist die Const-Referenz) – ansonsten würde eine Endlos-Rekursion erzeugt werden.
- Ein Kopier-Konstruktor erwartet daher ein Klassen-Objekt als erstes Argument und kann beliebig viele weitere Parameter haben, die dann aber mit Default-Argumenten belegt sein müssen.
- Er wird benötigt, um ein neues Objekt aus einem bestehenden Objekt zu konstruieren, z.B. bei einer Objekt-Definition, einem Funktionsaufruf mit *call-by-value*-Parametern, oder der Rückgabe eines Objektes bei einer Funktion.

```
class A
{
public:
   A();             // Standard-Konstruktor
   A(const A&);     // Kopier-Konstruktor
};
```

```
A::A()
{
   cout << "Standard-Konstruktor\n";
}

A::A(const A&)
{
   cout << "Kopier-Konstruktor\n";
}

void f(A) { }     // freie Funktion, die eine Kopie erwartet

int main()
{
   cout << "Erzeuge a1\n";
   A a1;

   cout << "Erzeuge a2\n";
   A a2(a1);                      // Kopier-Konstruktor

   cout << "Rufe f auf\n";
   f(a1);                         // Kopier-Konstruktor wegen call-by-value
}
```

Ausgabe

```
Erzeuge a1
Standard-Konstruktor
Erzeuge a2
Kopier-Konstruktor
Rufe f auf
Kopier-Konstruktor
```

Hinweis – statt einer Const-Referenz könnte der Kopier-Konstruktor auch mit einer Non-Const Referenz implementiert werden. Im Normallfall wollen wir bei einer Kopie das Original aber nicht verändern – ein Kopier-Konstruktor mit Non-Const Referenz ist daher extrem selten.

14.8.4.1 Automatischer Kopier-Konstruktor

Genauso wenig, wie die Klasse „*date*“ bislang einen Standard-Konstruktor hatte, hatte sie auch keinen Kopier-Konstruktor. Trotzdem konnten wir Date-Objekte aus anderen Date-Objekte erzeugen, bzw. Date-Objekte an Funktionen übergeben – siehe Kapitel 14.4.

```
// Auch bisher war das Kopieren von Date-Objekten kein Problem

void f(date d)
{
   d.print();              // => 6.2.2004
}

int main()
{
   date d1(6, 2, 2004);
   date d2(d1);            // Kopier-Konstruktor
   f(d2);                  // Kopier-Konstruktor - wegen call-by-value
}
```

Der Grund dafür ist ein ähnlicher wie beim Standard-Konstruktor – der Compiler generiert in vielen Fällen einen automatischen (oder „impliziten“) Kopier-Konstruktor. Der Compiler erzeugt den automatischen Kopier-Konstruktor, wenn es keinen user-deklarierten Kopier-Konstruktor, Kopier-Zuweisungs-Operator, Move-Konstruktor, Move-Zuweisungs-Operator oder Destruktor gibt.

Der automatische (bzw. implizite) Kopier-Konstruktor:

- ist **public**,
- nimmt das Original-Objekt als const-Referenz an, und
- ruft für jedes einzelne Element innerhalb der Klasse den jeweiligen Kopier-Konstruktor auf und erzeugt so das neue Objekt.

Aber wenn der Compiler für Klassen einen Kopier-Konstruktor automatisch erzeugen kann, wozu dann einen eigenen schreiben? Nun, es gibt Situationen, in denen eine elementweise Kopie nicht möglich ist, bzw. instabile oder fehlerhafte Zustände liefert – wir werden solche Konstellationen noch kennen lernen. In solchen Fällen müssen Sie den Kopier-Konstruktor entweder selber implementieren oder ihn verbieten – siehe Kapitel 14.8.4.3.

Empfehlung – machen Sie sich beim Design und der Entwicklung von Klassen **immer** Gedanken darüber, ob der automatische Kopier-Konstruktor ausreichend ist und fehlerfrei arbeitet. Wenn nicht, **müssen** Sie selber einen sinnvollen Kopier-Konstruktor entwerfen und implementieren, oder den Kopier-Konstruktor verbieten (siehe Kapitel 14.8.4.3). Im Normallfall bezieht sich diese Überlegung nicht nur auf den Kopier-Konstruktor, sondern auch den Move-Konstruktor (siehe Kapitel 14.8.5), den Destruktor (siehe Kapitel 14.9), den Kopier-Zuweisungs-Operator und den Move-Zuweisungs-Operator – und mündet dann in der „Regel der 3, 4, 5, 6, 0“, die aus Zeitmangel nicht besprochen werden.

Die „Regel-der-Drei, -Vier, -Fünf, -Sechs, oder –Null“ sagt aus, dass man entweder alle diese fünf speziellen Element-Funktionen plus die Swap-Funktion selber implementiert oder verbietet – oder bei allen die impliziten Varianten nutzt. Entweder kümmert man sich um alle, oder um gar keine. Alles andere macht in 99,99999 % der Fälle keinen Sinn.

14.8.4.2 Alternative Kopier-Konstruktor Syntax

Was steht semantisch in der zweiten Zeile?

```
string s1;
string s2 = s1;       // Was ist das hier semantisch?
```

Falsch, es ist **keine** Zuweisung! Bitte bedenken Sie, hier wird ein neues Objekt „s2“ erstellt, also **muss** es eine Objekt-Konstruktion sein, d.h. der Aufruf eines Konstruktors: „Immer wenn ein Objekt erstellt wird, wird der entsprechende Konstruktor aufgerufen!“.

Nur welcher Konstruktor ist das hier? Diese Syntax mit dem Operator „=“ ist eine **alternative Syntax für den Kopier-Konstruktor.** Und falls auf der rechten Seite ein Objekt vom gleichen Typ steht, wie das was konstruiert werden soll – dann ist das ja auch kein Problem. Wie im obigen Beispiel: auf der rechten Seite vom Operator „=“ steht das Objekt „s1“ vom Typ „string“, und links soll das Objekt s2„“ vom Typ „string“ als Kopie von „s1“ konstruiert werden. Alles easy, alles okay.

Aber Vorsicht, das ist nicht immer so.

```
class A
{
public:
   A(int);
```

```
   A(const A&);
private:
   int n_;
};

A::A(int n)
{
   n_ = n;
   cout << "A(int) : " << n_ << '\n';
}

A::A(const A& a)
{
   n_ = a.n_;
   cout << "A(const A&) : " << n_ << '\n';
}

int main()
{
   A a1(2);      // A(int) Konstruktor
   A a2(a1);     // Kopier-Konstruktor
   A a3 = 5;     // sematisch A(int) und Kopier-Konstruktor (*)
}
```

Ausgabe

```
A(int) : 2
A(const A&) : 2
A(int) : 5
```

Zeile (*) ist zwar die alternative Syntax für den Kopier-Konstruktor, aber auf der rechten Seite steht kein Objekt vom Typ „A“, d.h. es kann kein Kopier-Konstruktor benutzt werden. Statt dessen muss der Compiler das Argument auf der rechten Seite in ein „A“ Objekt konvertieren (dafür nimmt er natürlich den „A(int)“ Konstruktor als Konvertierungs-Konstruktor – siehe Kapitel 14.8.3), und nutzt dann dessen Ergebnis via Kopier-Konstruktor zur Objekt-Initialisierung. Wir sehen: es wird ein eigentlich überflüssiges Objekt erzeugt und dann wieder zerstört – und das kostet unnötige Performance. Zum Glück muss der Compiler das seit C++17 optimieren, von daher sehen wir auch keine entsprechenden Ausgaben.

Trotzdem muss der Compiler semantisch die Benutzung beider Konstruktoren checken muss. Daher wenn er den notwendigen Konstruktor nicht zur Typkonvertierung benutzen darf, bzw. kein Kopier-Konstruktor vorhanden ist, geht diese Syntax schief (Compiler-Fehler) – während die normale Syntax weiter funktionieren würde.

14.8.4.3 Kopier-Konstruktor verbieten

Wie verbietet man den Kopier-Konstruktor einer Klassen? Und damit implizit das Kopieren von Objekten eines Typs?

Die einfache Lösung in C++ ist die Benutzung von „=delete“, die wir schon für den Standard-Konstruktor in Kapitel 14.8.1 kennen gelernt haben.

```
class A
{
public:
   A();
   A(const A&) = delete;   // Kopier-Konstruktor verboten
};

A::A()
{
```

```
}

int main()
{
   A a1;
   A a2(a1);      // Compiler-Fehler, da Kopier-Konstruktor verboten
}
```

Wird der Kopier-Konstruktor verboten, so sollte eigentlich immer auch der Move-Konstruktor (siehe Kapitel 14.8.5), der Destruktor (siehe Kapitel 14.9), der Kopier-Zuweisungs-Operator und der Move-Zuweisungs-Operator verboten werden. Alle diese fünf speziellen Element-Funktionen gehören zusammen – und dies findet sich dann in der „Regel der 3,4,5,6 oder 0“ wieder.

Die Nicht-Implementierung des Kopier-Konstruktor (und später auch des Kopier-Zuweisungs-Operators) hat oft noch einen weiteren Hintergrund – häufig verbietet man bei einer Klasse auch das Kopieren, wenn es semantisch keinen Sinn macht:

- Nehmen Sie z.B. an, sie hätten eine Klasse, die in einer grafischen Anwendung den Maus-Cursor repräsentiert. Was sollte hier passieren, wenn Sie das Maus-Cursor Objekt kopieren? Bekommen Sie nun einen zweiten Maus-Cursor auf dem Bildschirm?
- Wenn man semantisch nicht beschreiben kann, was eine Funktion machen soll – wie will man sie denn dann implementieren? Die Nicht-Implementierung bewahrt einen also vor der unlösbaren Aufgabe, etwas nicht spezifierbares umsetzen zu müssen.
- Andere Beispiele sind z.B. die Streams, die sich nicht kopieren lassen. Auch hier ist nicht klar, was passieren sollte, wenn Sie ein File-Stream-Objekt kopieren könnten – sollte dann die Datei kopiert werden?

14.8.5 Move-Konstruktor

Der Move-Konstruktor soll an dieser Stelle nur grob erwähnt werden. Der Hintergrund für die sogenannte Move-Semantik sind Objekte, deren Kopien relativ *„teuer“* sind (bzgl. Performance und Speicher-Verbrauch), die sich aber relativ *„billig“* verschieben lassen. Beispiele für solche Klassen sind die String-Klasse oder die meisten Container-Klassen.

Deklariert bzw. definiert wird der Move-Konstruktor mit einer Non-Const R-Value Referenz auf ein Objekt der Klasse. Analog zum Move-Konstruktor gibt es auch noch einen Move-Zuweisungs-Operator.

```
class A
{
public:
   A(A&&);                 // Deklaration Move-Konstruktor
   A& operator=(A&&);      // Deklaration Move-Zuweisungs Operator
};

A::A(A&&)                  // Definition Move-Konstruktor
{
   // Wie auch immer eine sinnvolle Implementierung aussieht...
}
```

Auch der Move-Konstruktor wird vom Compiler automatisch erzeugt, wenn:

- kein benutzer-deklarierter Kopier-Konstruktor,

- kein benutzer-deklarierter Kopier-Zuweisungs-Operator,
- kein benutzer-deklarierter Move-Konstruktor,
- kein benutzer-deklarierter Kopier-Zuweisungs-Operator, und
- kein benutzer-deklarierter Destruktor

vorliegt.

Hinweis – während es Standard-, Kopier- und Konvertierungs-Konstruktoren schon in C++98 gab, ist der Move-Konstruktor eine Neuigkeit von C++11.

14.8.6 Sequenz-Konstruktor

Auch den Sequenz-Konstruktor will ich nur kurz erwähnen, und nicht im Detail vorstellen. Er ist wie der Move-Konstruktor eine Neuigkeit von C++11. Er ist ein sehr spezieller Konstruktor, der nur selten benötigt wird. Er ist dann notwendig, wenn man ein Objekt mit einer beliebig großen Menge von Werten eines Typs initialisieren möchte. Wir kennen dies z.B. von den Containern wie dem Vektor, den wir mit Werten vorbelegen wollen:

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

int main()
{
   vector<int> v = { 1, 2, 3, 5, 7 };              // Vorbelegung mit einer Menge von Werten

   for (int x : v)
   {
     cout << x << " - ";
   }
   cout << '\n';
}
```

```
Ausgabe
1 - 2 - 3 - 5 - 7 -
```

Möchten wir eine vergleichbare Semantik für unsere eigenen Klasse haben – d.h. die Initialisierung mit den geschweiften Klammern und einer beliebigen Menge von Werten eines Typs – dann ist der Sequenz-Konstruktor unser Freund.

14.9 Destruktoren

Analog zu den Konstruktoren gibt es eine spezielle Funktion zum Zerstören eines Objekts - den **Destruktor**. Er wird **immer** automatisch aufgerufen, wenn ein vollständig konstruiertes Objekt zerstört wird.

- Ein Destruktor hat keinen Rückgabewert (auch nicht **void**).
- Er hat keine Parameter.
- Sein Name ist der Klassen-Name mit führender Tilde '~'.
- Eine Klasse hat immer genau einen Destruktor.
- Wird er nicht explizit deklariert, so erzeugt der Compiler einen impliziten Destruktor.

```
class A
{
```

```
public:
   A(int);
   ~A();
private:
   int i;
};

A::A(int n)
{
   i=n;
   cout << "Konstruktor " << i << '\n';
}

A::~A()
{
   cout << "Destruktor " << i << '\n';
}

int main()
{
   cout << "Start\n";
   A a1(7);

   {
      cout << "Start neuer Block\n";
      A a2(3);
      cout << "Ende neuer Block\n";
   }                                          // <- Destruktoraufruf fuer a2

   cout << "Ende\n";
}                                             // <- Destruktoraufruf fuer a1
```

Ausgabe

```
Start
Konstruktor 7
Start neuer Block
Konstruktor 3
Ende neuer Block
Destruktor 3
Ende
Destruktor 7
```

- Die Aufgabe eines Destruktors ist es, das Objekt sauber abzubauen.
- Wird ein Objekt zerstört, so wird zuerst der Destruktor aufgerufen und dann der Speicherplatz freigegeben – genau umgekehrt zu den Konstruktoren.
- Ein impliziter Destruktor ist immer public und ruft für alle Attribute und Basis-Klassen seinerseits die Destruktoren auf.

Bemerkung – der implizite Destruktor für die Klasse *'date'* macht nichts, da sie nur int-Variablen enthält, die – wie alle elementaren Datentypen – *leere* Destruktoren haben.

Empfehlung – machen Sie sich beim Design und der Entwicklung von Klassen **immer** Gedanken darüber, ob der implizite Destruktor ausreichend ist und fehlerfrei arbeitet. Wenn nicht, **müssen** Sie selber einen sinnvollen Destruktor entwerfen und implementieren.

14.10 Const-Element-Funktionen

Nach dem bisherigen Wissen wäre folgendes richtig, liefert aber einen Compiler-Fehler.

```
int main()
{
```

```
   const date d;
   d.print();           // Compiler-Fehler
}
```

Warum aber gibt der Compiler einen Fehler aus? Wir können doch:

- ein konstantes Datum definieren
- und *print()* lief bislang problemlos

Problem des Compilers

- d ist konstantes date Objekt
- Aber es könnte sein, dass *print()* das Objekt verändert

Lösung

Wir wissen, das *'print()'* das Objekt nicht ändert, der Compiler aber nicht. Darum müssen wir dies dem Compiler mittteilen. Dafür wird das Schlüsselwort **const** sowohl hinter die Element-Funktions-Deklaration, als auch hinter den Kopf der Element-Funktions-Definition geschrieben.

```
class date
{
   ...
   void print() const;              // hier ein const
   ...
};

void date::print() const            // hier auch ein const
{
   ...
}

int main()
{
   const date d;
   d.print();                       // jetzt okay
}
```

Das Schlüsselwort **const** hinter einer Element-Funktion besagt, dass diese Element-Funktion das Objekt nicht ändert. Denken Sie daran, dass const nach links bindet, und links steht *quasi* das Objekt.

Versucht eine Const-Element-Funktion ein Objekt zu ändern, gibt der Compiler natürlich einen Fehler aus.

```
void date::print() const
{
   ++year_;                 // Compiler-Fehler
   ...
}
```

In einer **const**-Element-Funktion können wiederum auch nur Element-Funktionen aufgerufen werden, die selbst als **const** deklariert sind.

```
class A
{
public:
   void fct() const;

   void fct_is_const() const;
```

```
   void fct_is_not_const();
};

void A::fct() const
{
   fct_is_const();          // okay, da eine const Element-Funktion
   fct_is_not_const();      // Compiler-Fehler, da nicht const
}
```

14.10.1 const gehört zum Funktions-Namen

Das Schlüsselwort **const** gehört wie die Signatur (Name + Parameterliste) zum Funktionsnamen.

Konsequenz – es kann zwei bis auf **const** vom Funktions-Namen und der Parameterliste her identische Element-Funktionen geben. Die Entscheidung, welche Funktion vom Compiler aufgerufen wird, trifft er anhand von Überladenregeln bezogen auf das aktuelle Objekt. Für const Objekte wird die const Element-Funktion, für non-const Objekte die non-const Element-Funktion aufgerufen.

```
class A
{
public:
   void f();
   void f() const;
};

void A::f()                              // Definition der 'normalen' Version
{
   cout << "normale Version\n";
}

void A::f() const                        // Definition der const-Version
{
   cout << "const Version\n";
}

int main()
{
   A a;
   const A ca;
   a.f();           // ruft die 'normale' Version auf
   ca.f();          // ruft die const Version auf
}
```

```
Ausgabe
normale Version
const Version
```

Hinweise:

- Eine const Element-Funktion kann natürlich auch für non-const Objekte aufgerufen werden, und wird es auch, wenn keine const-Funktion exisitert.
- Zwei bis auf **const** vom Funktions-Namen und der Parameterliste her identische Element-Funktionen dürfen unterschiedliche Rückgabe-Typen haben, da es zwei gänzlich unabhängige Funktionen sind.

Empfehlung – machen Sie jede Element-Funktionen **const**, bei der das möglich ist. Sie schränken sonst die Benutzung ihrer Klassen unnötig ein – z.B. bei der typischen Übergabe eines Objekts an eine Funktion mit *„const type&“* können nur const-Element-Funktionen für das Objekt aufgerufen werden.

Die andere Lösung wäre natürlich, einfach konsequent im gesamten Programm auf const zu verzichten. Dann verschenken Sie aber viel Sicherheit – viel Spass bei der Fehlersuche.

14.11 this

In jeder Element-Funktion ist automatisch ein Zeiger auf das aktuelle Objekt definiert, repräsentiert durch das Schlüsselwort **this**. Da wir Zeiger noch nicht kennen, nehmen wir das erstmal so hin. Merken sie sich aber, dass - ähnlich zu Iteratoren - das dereferenzierte „**this**“, d.h. „***this**“ immer das aktuelle Objekt selber ist, d.h. das Objekt für das die Element-Funktion aufgerufen wurde.

```
class A
{
public:
   A(int);

   A& f1();
   void f2();

private:
   int n_;
};

A::A(int n)
{
   n_ = n;
}

A& A::f1()
{
   cout << "f1:" << n_++ << '\n';
   return *this;
}

void A::f2()
{
   cout << "f2:" << n_ << '\n';
}

int main()
{
   A a(4);
   a.f2();
   a.f1().f2();
}
```

Ausgabe

```
f2:4
f1:4
f2:5
```

Der this-Zeiger wird in der Praxis benutzt um z.B.:

- die Adresse des aktuellen Objekts zu ermitteln – z.B. bei Objektvergleichen,
- um das aktuelle Objekt selber zurückzugeben – z.B. um Funktionsaufrufe zu verketten,
- um das aktuelle Objekt an andere Funktionen übergeben zu können.

14.12 Klassen verwenden Klassen

Klassen können natürlich selber wieder als Attribute eingesetzt werden:

```
class person
{
public:
   person(const date&);

private:
   date birthday_;
};

person:: person(const date& birthday)
{
   birthday_ = birthday;
}
```

Wird ein Objekt erzeugt, so wird defaultmäßig:
1. Speicher reserviert,
2. die Standard-Konstruktoren der Attribute **in der Reihenfolge der Deklarationen**, d.h. ihrem Vorkommen in der Klassen-Definition aufgerufen, und
3. der Konstruktor der Klasse selber durchlaufen (Konstruktor-Rumpf).

Wird ein Objekt zerstört, ist die Reihenfolge genau umgekehrt, d. h.
1. wird der Destruktor der Klasse durchlaufen,
2. werden die Destruktoren der Attribute **in der umgekehrten Reihenfolge der Deklaration** aufgerufen, und
3. wird der Speicher freigegeben.

Mit dieser Strategie wird sichergestellt, dass Objekte „Ebene für Ebene“ konstruiert werden. Damit setzt die aktuelle Ebene immer nur auf vollständig fertige Ebenen auf, d.h. kann nur auf Objekte zugreifen, die einen stabilen Objektzustand erreicht haben.

14.13 Member-Initialisierungs-Listen

Probleme

Die Konstruktion eines Objekts wie im Beispiel in Kap. 14.12 ist nicht optimal, denn:
- Performance - erst wird das Attribut mit dem Standard-Konstruktor aufwändig initialisiert, direkt danach wird es auf einen neuen Wert gesetzt.
- Was, wenn Attribute keinen Standard-Konstruktor haben?
- Wie können const- oder Referenz- Attribute initialisiert werden?

Lösung

Member-Initialisierungs-Listen

Syntax:

Konstruktorkopf : Member-Initialisierungs-Liste { Konstruktorrumpf }

```
class A
{
public:
   A(int, const double&, const date&);

private:
   int i;
   double d1;
   double d2;
   date da1;
   date da2;
```

```
};

A::A(int v1, const double& v2, const date& v3)
 : i(v1), d2(2*v2), da1(v3), da2()
{
}
```

Für die in der Member-Initialisierungs-Liste aufgeführten Attribute werden die angegebenen Konstruktoren statt der Standard-Konstruktoren aufgerufen – es kann natürlich auch der Standard-Konstruktor angegeben werden, siehe im Beispiel das Attribut „da2“.

14.13.1 Attribute ohne Standard-Konstruktor

Attribute, die keinen Standard-Konstruktor haben, **müssen** in der Member-Initialisierungs-Liste aufgeführt werden – dies gilt auch für Basis-Klassen.

```
class A
{
public:
   A(int);
};

class B
{
public:
   B();
   B(int);

private:
   A a;
};

B::B()              // Compiler-Fehler - Attribut a kann nicht initialisiert werden
{
}

B::B(int arg)       // okay - explizite Angabe des int-Konstruktors von A
 : a(arg)
{
}
```

14.13.2 Objekt-Konstanten bzw. const Attribute

Const Attribute müssen in der Member-Initialisierungs-Liste aufgeführt werden, ausser sie können ohne expliziten Konstruktor-Aufruf erzeugt werden.

```
class A
{
public:
   A();
   A(int);

private:
   const int ci;
};

A::A()             // Compiler-Fehler - const Attribut ci wird nicht initialisiert
{
}

A::A(int arg)      // okay
 : ci(2*arg+7)
{
}
```

Bemerkung – verwechseln Sie nicht das Objekt und seine Attribute. Manche Leute argumentieren immer wieder, dass das „A“ Objekt doch erst nach Abarbeitung des kompletten Konstruktors vollständig erzeugt worden ist. Es sollte also doch möglich sein, z.B. im Konstruktor von „A“ das Attribut „ci“ zu setzen, ohne die Initialisierungsliste zu benutzen.

```
A::A()        // Compiler-Fehler - ci wird nicht initialisiert
{
   ci = 7;    // Compiler-Fehler - ci ist const
}
```

Aber diese Argumentation ist falsch, denn es werden zwei Ebenen durcheinander gewürfelt. Ein Objekt ist komplett fertig, sobald sein Konstruktor erfolgreich komplett abgearbeitet wurde. Dies gilt natürlich auch für Objekte in Objekten.

14.13.3 Referenz-Attribute

Referenz-Attribute müssen in der Member-Initialisierungs-Liste initialisiert werden.

```
class A
{
public:
   A();
   A(const date&);

private:
   const date& date_;
};

A::A()                    // Compiler-Fehler - Referenz wird nicht initialisiert
{
}

A::A(const date& d)       // okay
 : date_(d)
{
}
```

Für den Standard-Konstruktor „A()“ muss der Compiler einen Fehler melden, da die Referenz nicht initialisiert wird.

Achtung - wenn sie ein Referenz-Attribut benutzen, muss sichergestellt sein, dass das referenzierte Objekt mindestens solange lebt wie das erstellte Objekt. Ansonsten zeigt die Referenz irgendwann in Speicherbereiche, die dem Programm nicht mehr gehören, bzw. wieder anderweitig benutzt werden.

Dies passiert schnell, wenn z.B. dem Konstruktor selber schon temporäre oder lokale Objekte mitgegeben werden. Für den Aufrufer sieht alles okay aus, da er die Implementation der Klasse nicht kennt, bzw. auch nicht kennen soll. Der Implementierer der Klasse hat keine Chance festzustellen, dass das Objekt nur eine begrenzte Lebensdauer hat.

Empfehlung - verwenden sie Referenz-Attribute sehr vorsichtig. Und wenn, verwenden sie sie nur so, dass ein Benutzer der Klasse keine Probleme mit der Lebensdauer hat, bzw. weisen sie ihn explizit darauf hin.

14.13.4 Typischer Fehler

Das folgende Beispiel stellt eine Klasse für einen Kreis dar. Aus Performancegründen wird in dieser Klasse sowohl der Umfang als auch der Radius gespeichert, damit die Berechnung nur einmal erfolgen muss.

Diese Klasse enthält einen Fehler, welchen?

Was gibt die Element-Funktion „print() const“ für das Objekt „limit“ aus?

```
class circle
{
public:
   circle(double);
   void print() const;

private:
   double circumference_;
   double radius_;
};

circle::circle(double radius)
 : radius_(radius), circumference_(2*3.1415926*radius_)
{
}

void circle::print() const
{
   cout << "Kreis mit Radius " << radius_
        << " und Umfang " << circumference_
        << '\n';
}

int main()
{
   circle limit(4.0);
   limit.print();
}
```

Fehler – da die Attribute in der Reihenfolge der Deklaration konstruiert werden – wird „circumference_“ vor „radius_“ mit einem zu dem Zeitpunkt rein zufälligen Wert für den Radius erzeugt.

Achtung – die Anordnung in der Member-Initialisierungs-Liste spielt keine Rolle für die Reihenfolge der Konstruktor-Aufrufe der Attribute.

14.14 Deklarationen

Zwischen einzelnen Klassen können Ring-Abhängigkeiten herrschen:
A braucht B und *B braucht A*.

Da der Compiler nur bekannte Klassen verwenden kann, können Klassen mit dem Schlüsselwort **class** und dem Klassen-Namen **deklariert** werden.

```
class B;                // macht die Klasse B für den Compiler bekannt

class A
{
public:
  int fct(const B&);    // Benutzung der Klasse B als Referenz-Parameter
};

class B                 // Deklaration Klasse B
```

```
{
public:
   A a;
};
```

Die Deklaration funktioniert nur, solange der Compiler keine näheren Angaben über die vorwärts deklarierte Klasse benötigt, z.B. Größe oder internen Aufbau.

```
class B;

class A
{
public:
   A();
   void f(B*);      // okay
   void f(B&);      // okay
   void f(B);       // okay

   B* g();          // okay
   B& g();          // okay
   B g();           // okay

private:
   B* p;            // okay
   B& r;            // okay
   B b;             // Compiler-Fehler
};
```

Eine Deklaration reicht aus für:

- Funktions-Parameter und Funktions-Rückgaben in Deklarationen, da der Compiler hier keinen Code erzeugt, sondern hier nur eine Funktion deklariert wird.
- Zeiger- und Referenz-Attribute, da deren Grösse unabhängig vom Aufbau der referenzierten Klasse ist, und dem Compiler die Grösse bekannt ist.

Für Wert-Attribute muss die Deklaration der Attribut-Klasse bekannt sein, da der Compiler z.B. die Grösse der Klasse wissen muss.

14.15 Klassen-Elemente

Klassen-Elemente sind klassenspezifische Elemente, die keinem Objekt **sondern der Klasse** zugeordnet sind.

Es gibt:

- Klassen-Variablen, und
- Klassen-Funktionen.

14.15.1 Klassen-Variablen

Eine Klassen-Variablen ist eine der Klasse zugeordnete Variable, die:

- nur **einmal** im Programm existiert, unabhängig von der Anzahl instanziierter Objekte,
- und den normalen Zugriffsrechten der Klasse unterliegt.

Angesprochen wird sie:

- von innerhalb der Klasse ganz normal über ihren Namen, und
- von ausserhalb mit zusätzlichem Objekt- oder Klassenbezug.

Klassen-Variablen müssen in der Klasse deklariert, und einmal im Programm (ausserhalb der Klasse) definiert werden:

Syntax
Deklaration: static typ name;
Definition: typ klasse::name { „(Konstruktor-Argument-Aufrufliste)“ | „= Initialisierer“ };

```
class A
{
public:
   void fct();

   static int si;
};

int A::si = 8;               // Definition mit Initialisierung

void A::fct()
{
   cout << si << '\n';       // direkter Zugriff, da innerhalb der Klasse
}

int main()
{
   cout << A::si << '\n';         // Zugriff ueber den Klassen-Namen
   A a;
   cout << a.si << '\n';          // Zugriff ueber ein Objekt
   a.fct();
}
```

Bemerkung - im Prinzip ist eine Klassen-Variable eine globale Variable, die aber im Namensraum der Klasse liegt, und damit z.B. zugriffsmässig eingeschränkt werden kann.

Hinweis - ein am Anfang gern gemachter Fehler ist das Vergessen der Definition einer Klassen-Variablen. Dies führt zu einem eigentlich eindeutigen Linker-Fehler, aber aller Anfang fällt schwer - erst recht in C++.

14.15.2 Klassen-Funktionen

Analog zu Klassen-Variablen gibt es Klassen-Funktionen, die ebenfalls nicht einem Objekt, sondern der Klasse zugeordnet sind, und auch den normalen Zugriffsrechten der Klasse unterliegen.

Angesprochen werden sie:
- von innerhalb der Klasse ganz normal über ihren Namen, und
- von ausserhalb mit zusätzlichem Objekt- oder Klassenbezug.

Klassen-Funktionen müssen in der Klasse mit static deklariert werden. Die Definition erfolgt analog zu den normalen Element-Funktionen.

```
class A
{
public:
   static void fct();
};

void A::fct()
```

```
{
   cout << "static A::fct()\n";
}


int main()
{
   A::fct();
   A a;
   a.fct();
}
```

Bemerkung - im Prinzip ist eine Klassen-Funktion eine globale Funktion, die aber im Namensraum der Klasse liegt, und damit z.B. zugriffsmässig eingeschränkt werden kann, oder z.B. Zugriff auf private Elemente der Klasse hat.

Hinweis - Klassen-Funktion dürfen nicht den gleichen Namen und die gleiche Parameterliste wie eine Element-Funktion der Klasse haben. Da eine Klassen-Funktion auch mit Objektbezug aufrufbar ist, wäre der Aufruf nicht eindeutig.

14.15.2.1 Kein Objektbezug

Klassen-Funktionen haben keinen Objektbezug, selbst wenn sie mit Objektbezug aufgerufen werden. Darum können sie auch mit Klassenbezug aufgerufen werden.

Daraus ergeben sich einige Unterschiede gegenüber Element-Funktionen:

- In Klassen-Funktionen ist kein **this**-Zeiger definiert - da sie keinen Objektbezug haben.
- Klassen-Funktionen können nicht **const** sein - worauf sollte sich das const beziehen?
- In Klassen-Funktionen kann nur auf andere Klassen-Elemente zugegriffen werden, denn der Aufruf von Element-Funktionen oder der Zugriff auf Attribute benötigt einen Objektbezug.
- Klassen-Funktionen können nicht virtuell sein.

```
class A
{
public:
   static void f1();
   static void f2();
   static void f3() const;      // Compiler-Fehler - kein const bei static Funktionen

   void fct();

private:
   int i;
   static int si;
};

void A::f1()
{
   si = 10;                     // okay, da Klassen-Variable
   f2();                        // okay, da Klassen-Funktion

   void* vpThis = this;         // Compiler-Fehler - this nicht definiert
   fct();                       // Compiler-Fehler - kein Objektbezug
   i = 12;                      // Compiler-Fehler - kein Objektbezug
}
```

14.16 friend

Mit dem Schlüsselwort **friend** kann freien Funktionen und Klassen erlaubt werden auf alle Elemente einer anderen Klasse zuzugreifen - **auch die privaten**. Sie werden *quasi* zu *Freunden der Klasse*.

14.16.1 Freie friend-Funktionen

Damit eine freie Funktion auf alle Elemente einer Klasse zugreifen kann, muss sie innerhalb der Klasse mit **friend** deklariert, d. h. zum *Freund der Klasse* gemacht werden.

An der Deklaration mit dem Schlüsselwort **friend** erkennt der Compiler, dass es sich nicht um eine Element-Funktion, sondern um eine freie Funktion handelt.

```
class A
{
public:
   A(int i) : n(i) {}

   friend int fct(const A&);           // Achtung - keine Element-Funktion,
                                       // sondern eine freie Funktion
private:
   int n;
};

int fct(const A& a)                    // Definition der freien Funktion fct
{                                      // Da friend von A, darf sie auf alle
   return a.n;                         // Elemente von A zugreifen
}

int main()
{
   A a(17);
   cout << fct(a) << '\n';             // Ausgabe: 17
}
```

Hinweis - noch einmal: obwohl *„fct“* innerhalb der Klasse *„A“* deklariert wurde, ist *„fct“* keine Element-Funktion, sondern aufgrund von **friend** eine ganz normale freie Funktion, die eben nur zusätzlich auf alle Elemente der befreundeten Klasse zugreifen kann.

14.16.2 friend-Klassen

Um eine komplette Klasse zum Freund einer anderen zu machen, muss die Klasse als Vorwärts-Deklaration mit **friend** in der Klassendefinition aufgeführt werden. Damit darf innerhalb der befreundeten Klasse auf alle Elemente der Freund-Klasse zugegriffen werden.

```
class B;

class A
{
public:
   int f(const B&) const;
};

class B
{
   friend class A;                     // macht A zum friend von B
public:
   B(int i) : n(i) {}

private:
```

```
   int n;
};

int A::f(const B& b) const              // Definition der Element-Funktion A::f
{                                       // Da A friend von B ist, duerfen alle
   return b.n;                          // Funktionen von A auf alle Elemente
}                                       // von B zugreifen

int main()
{
   A a;
   B b(42);
   std::cout << a.f(b) << '\n';         // Ausgabe: 42
}
```

14.16.3 Weiteres

friend-Beziehungen sind nicht transitiv
*A **friend** von B* und *B **friend** von C* daraus folgt **nicht:** *A **friend** von C.*

```
class A
{
   friend class B;
   int i;
};

class B
{
   friend class C;
};

class C
{
public:
   void fct(A& a) { a.i++; }     // Compiler-Fehler - C ist kein Freund von A
};
```

Bemerkung - friend-Beziehungen werden auch nicht vererbt.

14.17 Klassenbezogene Typen

In Klassen können nicht nur Element-Funktionen, Konstruktoren, Destruktoren, Attribute, Klassen-Funktionen und Klassen-Variablen definiert werden, sondern auch Typen, z.B.:

- Typ-Aliase
- Aufzählungstypen mit enum
- Innere Klassen

Diese Typen unterliegen den normalen Zugriffsbereichen der Klasse, d.h. private Typen können nur innerhalb der Klasse benutzt werden, während public Typen überall benutzbar sind – siehe z.B. Kapitel 14.3.

Angesprochen werden die Typen:

- von innerhalb der Klasse ganz normal über ihren Namen, und
- von ausserhalb mit zusätzlichem Klassenbezug.

```
class paragraph
{
public:
   enum alignment { left, center, right };
```

```
   alignment get_alignment() const { return alignment_; }
   void set_alignment(alignment arg) { alignment_ = arg; }

private:
   using length = long;

   class word
   {
   public:
      const string& value() const;

   private:
      string value_;
   };

   alignment alignment_;
   length length_;
};

const string& paragraph::word::value() const
{
   return value_;
}

int main()
{
   paragraph para;
   para.set_alignment(paragraph::center);
   paragraph::alignment al = para.get_alignment();

   paragraph::alignment a;           // okay, da public
   paragraph::lenght l;              // Compiler-Fehler, da private
   paragraph::word w;                // Compiler-Fehler, da private
}
```

Hinweis – innere Klassen (auch „verschachtelte Klassen“, „eingebettete Klassen“ oder „nested classes“) werden eigentlich nur als Hilfsklassen eingesetzt.