Visual C++
Vorlesung

Dr. habil. Arno Formella
(Homo Habilis)
HTW des Saarlandes

13. Dezember 1999

1 Um was geht es in der Vorlesung?

Wir wollen eine “einfache” Billard-Animation mit grafischer Oberfläche mit Hilfe eines strukturierten C++-Programms und einer Microsoft Entwicklungsumgebung (MS Visual C 5.0) implementieren!

2 Was sind die Grundlagen?

- C Grundkenntnisse
- Grundkenntnisse Windows Betriebssystem
- Zeit und Wille sich durch die praktischen Teile der Vorlesung zu arbeiten

```
statement;  // comment explains previous content of line
```

Präprozessorkommentare

```
#ifdef NEVER
...
#endif
```

5 Wie schreiben wir Programme?

5.1 Namensgebung

“meine Namesregeln”:

Funktionsnamen bestehen aus aussagekräftigen Ausdrücken aus einer Folge von Worten, deren erste Buchstaben jeweils groß geschrieben sind,
z.B. `InsertObject`, `DrawRectangle`

Variablennamen/Objektnamen bestehen aus aussagekräftigen Worten, die komplett klein geschrieben sind, je länger der Name umso größer ist der Sichtbarkeitsbereich der Variable, es wird immer der gleiche Name für die gleiche “Sache” verwendet

Typnamen/Klassennamen bestehen aus aussagekräftigen Worten, nur erster Buchstabe ist groß geschrieben

Makronamen bestehen aus nur Großbuchstaben

5.2 Konventionen

Es gibt viele Konventionen, die bei der Namensauswahl Sinn machen, insbesondere verwendet man Präfixe und Postfixe: z.B.

- Membervariablen beginnen mit `m_`,
- Zeigervariablen beginnen mit `p`,
- Klassennamen beginnen mit `Cl` oder `C`, Klassennamen
- bestimmte Klassen in Bilbliotheken beginnen mit besonderen Präfixen, z.B. `MFC`, `Qt`.

Ich schreibe Programme vollständig in Englisch (bis auf Ausgaben, falls so erwünscht), die meisten Klassenbibliotheken sind in Englisch.

> **Regel:** Wir verwenden nur den ASCII-Zeichensatz mit seinen druckbaren Zeichen, insbesondere keine Umlaute.

5.3 Layout

1. wir heben die Blockstruktur eines C++-Programms durch entsprechendes Layout des Programmcodes hervor
2. wir rücken mit mindestens 2 und höchsten 4 Leerzeichen ein
3. wir halten uns immer an genau einen Stil

Beispiele:

```
Return_type FunctionName(
  Parameter param1,     // comment
  Parameter param2      // comment
) {
  ...
}

while(condition) {
  ...
}

if(condition) {
  ...
}
else {
  ...
}

switch(selector) {
  case FIRST:
    ...
    break;
  case SECOND:
    ...
    break;
  default:
    ...
    break;
}
```

Vernünftige Editoren unterstützen dieses oder ein verwandtes strukturiertes Layout: Lernen Sie Ihren Editor kennen!

Regel: Wir schreiben unsere Programme immer nach diesen Vorgaben.

6 Woraus besteht ein Programm?

Ansammlung von .cpp- und .h-Dateien (sogenannten Quellcode-Dateien) und je nach Entwicklungsumgebung einer Menge von weiteren Dateien.

Alternativen für .cpp-Dateien sind bei manchen Compilern .cc-, .C-, oder .CC-Dateien. Bei VisualC++ von Microsoft ist insbesondere die .dsw-Datei wichtig, mit ihr kann per Doppelklick das entsprechende Projekt in der Entwicklungsumbegung gestartet werden.

6.1 Aufbau einer .h-Datei

- Einbinden von anderen .h-Dateien, zuerst Systemdateien, dann eigene Dateien
- Definition von Typen und Klassen
- Deklaration der Prototypen
- Makro-Definitionen

Zum Vermeiden von Mehrfacheinbindungen durch geschachtelte `#include`-Direktive ist eine .h-Datei immer mit folgender Klammer umgeben:

```
#ifndef Name_of_header_file
#define Name_of_header_file

...

#endif
```

Dabei steht `Name_of_header_file` für den Namen der .h-Datei oder eine sonstwie geartete eindeutige Bezeichnung (z.B. `HELLO_H` oder `VECTOR_H` in den späteren Beispielen).

6.2 Aufbau einer .cpp-Datei

- Einbinden von anderen .h-Dateien, zuerst Systemdateien, dann eigene Dateien
- Definition von Konstanten
- Definition von Typen und Klassen, falls nicht besser in .h-Datei
- globale Variablen und Objekte
- lokale Variablen und Objekte
- Funktionen

Bei Funktionsdefinitionen gibt es zwei Methoden, die man sinnvollerweise nicht miteinander vermischt:

1. erst definieren, dann verwenden
2. erst verwenden, dann definieren (Prototyp ist ja immer vorher bekannt!)

> **Regel:** Wir gliedern .h- und .cpp-Dateien immer so und halten Ausnahmen vom Schema so gering wie möglich.

Bemerkung: Nicht strukturierte und nicht ordentlich angeordnete Programme lese ich nicht!

7 Übung: HelloWorld

Datei main.cpp:

```
#include "hello.h"

void main(
) {
  HelloWorld();
}
```

Datei hello.cpp:

```
#include "hello.h"

void HelloWorld(
) {
  cout << "Hello World" << endl;
}
```

Datei hello.h:

```
#ifndef HELLO_H
#define HELLO_H

void HelloWorld(void);
#endif
```

8 Wie finde ich Fehler?

1. vieles ist Erfahrungssache
2. Compiler-Fehlermeldungen interpretieren lernen
3. nicht verzweifeln
4. Fehler kann früher als angezeigt liegen (z.B. in .h-Datei)
5. konzentriert arbeiten
6. Möglichkeiten der Sprache nutzen
7. Debug- und Release-Versionen erzeugen
8. in Gruppen arbeiten und gegenseitig Programm erklären
9. Debugger nutzen

9 Was ist wichtig in der Entwicklungsumgebung?

Eine kleine Reise durch die Menüpunkte ...

10 Worin bestehen wesentliche Unterschiede zwischen C und C++?

10.1 Default-Parameter

Funktionsprototypen können Default-Werte für die Parameter angeben. Diese Default-Werte müssen von rechts beginnend in der Parameterliste angegeben sein.
Prototyp:

```
void HelloWorld(int a = 1);
```

Implementierung:

```
void HelloWorld(
  int a
) {
  cout << "Hello World number " << a << endl;
}
```

Verwendung:

```
  HelloWorld();
  HelloWorld(17);
```

10.2 Überladen von Funktionsnamen

Funktionsnamen können überladen werden, d.h. der gleiche Funktionsname kann benutzt werden, sofern verschiedene Parametertypen und/oder Parameteranzahlen verwendet werden.
Prototypen:

```
double abs(double);         // absolute value of double
int    abs(int);            // absolute value of int
double abs(double,double); // absolute value of "complex"
```

Implementierung:

```
double abs(
  double x
) {
  return(x>=0.0?x;-x);
}

int abs(
  int i
) {
  return(i>=0?i;-i);
}

double abs(
```

```
  double real,
  double imag
) {
  return(sqrt(real*real+imag*imag));
}
```

Vorsicht ist wegen impliziter Typ-Konvertierung in C/C++ geboten, insbesondere wegen des `NULL`-Zeigers, der in C++ durch die einfache `0` ausgedrückt wird.

10.3 Konstanten

`const` bedeutet, dass folgende Variablen oder Funktionen konstant sind, d.h. entweder nicht verändert werden können – mit anderen Worten: sie stehen u.a. nie links von einem Gleichheitszeichen – oder selbst nichts verändern.
Beispiel:

```
const double pi=3.14159;
const double pi_2=atan(1);

  /* konstante Zeichenkette */
const char *str = "abc";

  /* konstanter Zeiger */
char * const str = "abc";

  /* konstante Zeichenkette mit konstantem Zeiger */
const char * const str = "abc";
```

Konstante Funktionen werden später erläutert.

> **Regel:** Wir verwenden `const` wo immer möglich, um eventuelle Seiteneffekte auszuschließen.

10.4 Referenzen

In C gab es nur Zeiger:

```
int  i;
int *p;   // p is pointer to int

i = 1;
p = &i;

printf("i  = %d\n",i);
printf("*p = %d\n",*p);

*p = 2;

printf("i  = %d\n",i);
printf("*p = %d\n",*p);
```

In C++ gibt es zusätzlich Referenzen, d.h. man kann sich Aliase von Variablen erzeugen:

```
int  i;
int& j=i;   // j becomes a reference to (or an alias for) i

i=0;

cout << "i = " << i << endl;
cout << "j = " << j << endl;

j=2

cout << "i = " << i << endl;
cout << "j = " << j << endl;
```

Referenzen müssen initialisiert werden und können anschließend nicht mehr geändert werden.
Referenzen sind besonders bei der Parameterübergabe sinnvoll aber auch gefährlich:

```
void swap(
  int& i,
  int& j
) {
   int h=i; i=j; j=h;
}

...
  x=17;
  y=13;
  cout << "x = " << x << ", y = " << y << endl;
  swap(x,y);
  cout << "x = " << x << ", y = " << y << endl;
...
```

> **Regel:** Wir verwenden eine Übergabe mit Referenz-Parametern entweder mit `const` oder aber sehr spärlich und gut dokumentiert (Warnung vor dem Seiteneffekt).

11 Übung: Überladen und Referenzen

Implementation der Swap-Funktion und experimentieren mit `const`.

12 Was sind Klassen und Objekte?

12.1 Klasse: Vector

Betrachten wir uns als Beispiel eine Vektorklasse:

```
class CVector {
private:
  double *vector;
  int     dimension;
public:
           CVector(int dimension=3); // constructor
  virtual ~CVector();                // destructor
  void     Input();
  void     Output() const;
};
```

Konstruktor: ein Objekt der Klasse wird erzeugt und alle damit verbundenen Initialisierungen werden durchgeführt, der Konstruktor wird implizit bei der Variablendeklaration aufgerufen

Destruktor: ein Objekt der Klasse wird wieder zerstört, der Destruktor wird implizit aufgerufen, wenn der Sichtbarkeitsbereich eines Objektes durch den Programmfluss verlassen wird

Eingabe und Ausgabe: hier nur beispielhaft, damit wir an der Stelle irgendwas haben, wir werden sie so nicht implementieren, sondern es wird später entscheidend verbessert!

Die Komponenten der Klasse (auch Member) genannt, sind hier in zwei Gruppen eingeteilt:

private:
- alles was nach `private` steht, ist nach außen *nicht* sichtbar, d.h. nachdem ein Objekt dieser Klasse definiert wurde, kann nicht einfach mit dem `.`-Operator auf die Komponente zugegriffen werden
- auch Member-Funktionen können `private` implementiert werden, sie können dann nur von anderen Member-Funktionen aufgerufen werden
- `private` ist die default-Einstellung

public: alles was nach `public` steht, *ist* nach außen sichtbar, d.h. nachdem ein Objekt dieser Klasse definiert wurde, kann einfach mit dem `.`-Operator auf die Komponente zugegriffen werden

protected: dies eine dritte Möglichkeit, die später bei der Ableitung von Klassen näher erläutert wird

Bemerkung: `public`, `protected` und `private` können beliebig oft benutzt werden.

> **Regel:** Wir gruppieren alle Komponenten in nur drei Gruppen: zuerst `private`, dann `protected`, dann public.

12.2 Implementierung der Schnittstellen

Konstruktor:

```
CVector::CVector(
  int dimension
)
{
#ifdef _DEBUG
  if(dimension<=0) {
    cout << "negative dimension in constructor?\n";
    cout << "dimension = " << dimension << endl;
    exit(1);
  }
#endif
  vector = new double[dimension];
  CVector::dimension = dimension;
}
```

Destruktor:

```
CVector::~CVector(
) {
  delete [] vector;
}
```

13 Übung: Implementierung der Klasse

Wir beginnen die Klasse zu implementieren. Beachte: damit `exit` und `cout` bekannt sind, muss `iostream.h` sowie `stdlib.h` eingebunden werden.

14 Gültigkeitsoperator

`::` ist Gültigkeitsoperator zeigt zu welchem Namensraum (linker Operand) der Bezeichner (rechter Operand) gehört.
Beispiel: `Vector::Input` bedeutet, dass die Eingabefunktion (mit Namen `Input`) zur Klasse `Vector`, d.h. in den Namensraum der Klasse, gehört.

15 Wie verwalten wir Speicherplatz bzw. Objekte dynamisch?

15.1 Anfordern von Speicherplatz

```
Class_name *object_name = new Class_name;
Class_name *object_name = new Class_name[number];
```

Die erste Zeile erzeugt ein Objekt der Klasse `Class_name`, der Zeiger `object_name` zeigt auf das Objekt.
Die zweite Zeile erzeugt ein Feld der Länge `number` von Objekten der Klasse `Class_name`, der Zeiger `object_name` zeigt auf das erste Objekt.

15.2 Freigeben von Speicherplatz

```
delete  * object_name;
delete [] object_name;
```

Die erste Zeile gibt das Objekt, auf das der Zeiger `object_name` zeigt, wieder frei.
Die zweite Zeile gibt das Feld von Objekten, auf das der Zeiger `object_name` zeigt, wieder frei.

16 Wohin schreiben wir die Member-Funktionen?

zwei Möglichkeiten:

innerhalb der Klassendefinition: d.h. in die .h-Datei,
damit werden die Funktion auch `inline`, d.h. der Compiler ersetzt zur Effizienzsteigerung einen Funktionsaufruf durch den entsprechenden Programmcode

außerhalb der Klassendefinition: d.h. in die .cpp-Datei der entsprechenden Klasse,
damit sind die Funktionen nicht mehr `inline`

Bemerkung: Man kann eine Funktion auch außerhalb der Klassendefinition innerhalb der .h-Datei mit dem Schlüsselwort `inline` definieren.

> **Regel:** Wir benutzen das Schlüsselwort `inline` nicht, sondern unterscheiden entsprechend obiger Möglichkeiten.

17 Wie finden wir den Kollisionszeitpunkt zweier Kugeln?

Unser Ziel ist es eine einfache Billard-Animation zu programmieren.
Annahme: Kugeln bewegen sich zwischen zwei Kollisionen mit konstanter, ungedämpfter Geschwindigkeit auf geradliniger Bahn.
Zwei Kugeln treffen sich, wenn sie gerade soweit zusammen kommen, dass der Abstand der beiden Mittelpunkte gleich der Summe der Radien ist.
Gegeben:

t_0	Ausgangszeitpunkt
$p_0(t_0)$	Position Kugel 0 zum Ausgangszeitpunkt
v_0	Geschwindigkeit Kugel 0
r_0	Radius Kugel 0
$p_1(t_0)$	Position Kugel 1 zum Ausgangszeitpunkt
v_1	Geschwindigkeit Kugel 1
r_1	Radius Kugel 1

Bewegungsgleichungen (sind Vektorgleichungen):

$$\begin{aligned} p_0(t) &= p_0(t_0) + t \cdot v_0 \\ p_1(t) &= p_1(t_0) + t \cdot v_1 \end{aligned}$$

z.B. im 2-Dimensionalen für die erste Gleichung:

$$\begin{pmatrix} x_0(t) \\ y_0(t) \end{pmatrix} = \begin{pmatrix} x_0(t_0) \\ y_0(t_0) \end{pmatrix} + t \cdot \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

Kollisionsbedingungen:

$$\begin{aligned} |p_0(t_c) - p_1(t_c)| &= r_0 + r_1 \\ t_c &> t_0 \\ t_c & \text{ minimal} \end{aligned}$$

d.h. Zeitpunkt t_c liegt in der Zukunft und ist der erste Zeitpunkt, an dem sich beide Kugeln gerade berühren.
Wir kennen Zusammenhang zwischen Betrag und Skalarprodukt:

$$|v| = \sqrt{\langle v, v \rangle}$$

Einsetzen:

$$\begin{aligned} & |p_0(t_c) - p_1(t_c)| \\ &= |(p_0(t_0) + t_c \cdot v_0) - (p_1(t_0) + t_c \cdot v_1)| \\ &= \sqrt{\langle (p_0(t_0) - p_1(t_0)) + t_c \cdot (v_0 - v_1), (p_0(t_0) - p_1(t_0)) + t_c \cdot (v_0 - v_1) \rangle} \end{aligned}$$

Verwenden:

$$\begin{aligned} \Delta p &= p_0(t_0) - p_1(t_0) \\ \Delta v &= v_0 - v_1 \\ R &= r_0 + r_1 \end{aligned}$$

Lösen:

$$\begin{aligned} \sqrt{\langle \Delta p + t_c \Delta v, \Delta p + t_c \Delta v \rangle} &= R \\ \langle \Delta p + t_c \Delta v, \Delta p + t_c \Delta v \rangle &= R^2 \\ \langle \Delta p, \Delta p \rangle + 2 \cdot t_c \cdot \langle \Delta p, \Delta v \rangle + t_c^2 \cdot \langle \Delta v, \Delta v \rangle &= R^2 \\ t_c^2 \cdot \langle \Delta v, \Delta v \rangle + 2 \cdot t_c \cdot \langle \Delta p, \Delta v \rangle + (\langle \Delta p, \Delta p \rangle - R^2) &= 0 \end{aligned}$$

Verwenden:

$$\begin{aligned} a_0 &= \langle \Delta p, \Delta p \rangle - R^2 \\ a_1 &= 2 \cdot \langle \Delta p, \Delta v \rangle \\ a_2 &= \langle \Delta v, \Delta v \rangle \end{aligned}$$

Also Lösen der quadratischen Gleichung

$$a_2 \cdot t_c^2 + a_1 \cdot t_c + a_0 = 0$$

liefert bei existierender minimaler Lösung größer t_0 den Kollisionszeitpunkt.
Schreiben wir das Skalarprodukt mit einem einfachen Punkt statt mit der Klammernotation, erhalten wir die quadratische Gleichung in t_c:

$$t_c^2 \cdot \Delta v \cdot \Delta v + 2 \cdot t_c \cdot \Delta p \cdot \Delta v + (\Delta p \cdot \Delta p - R^2) = 0$$

18 Was benötigen wir also für Vektoren?

Skalarmultiplikation d.h. Multiplikation eines skalaren Wertes mit einem Vektor

Skalarprodukt d.h. Multiplikation zweier Vektoren zu einem Skalar

Vektoraddition Addition (bzw. Subtraktion) zweier Vektoren

19 Können wir die mathematische Schreibweise weiter benutzen?

Wir würden in einem Programm gerne folgendes schreiben:

```
double  c;
CVector p_0,p_1,d;

d = p_0 - p_1;   // difference of two vectors
d = c * d * d;   // scaling d with c, dotproduct with d
```

also die erste Multiplikation ist eine Skalarmultiplikation, die zweite ein Skalarprodukt.

19.1 Überladen von Operatoren

Es gibt in C++ eine Menge von Operatoren:

```
 +   -   *   /   %       // arithmetisch
 ~   &   |   ^  <<  >>   // bitweise
++  --                   // In/Dekrement
 +   -                   // Vorzeichen
 =  +=  -=  *=  /=  %=   // Zuweisung
&=  |=  ^= <<= >>=
==   <   >  <=  >=  !=   // Vergleiche
 !  &&  ||               // logisch
 *   &  ->   .           // Adressen
 ,  ()  []  ?:  ::       // spezielle
new delete sizeof
#  ##                    // Praeprozessor
```

1. alle C++-Operatoren außer `?:`, `.` , `*` (Dereferenzierung), `::`, #, ##, `sizeof` können überladen werden
2. überladen werden können insbesondere auch `new` und `delete`
3. es gibt unäre und binäre Operatoren, unäre haben genau einen Operanden, binäre haben genau drei Operanden, (in C/C++ gibt es auch einen trinären Operator: `?:`)
4. die Vorrangregeln bleiben erhalten

5. die Abarbeitungsreihenfolge bleibt erhalten
6. es können keinen neuen Operatoren erfunden werden
7. eingebaute C++-Operatoren für Standardtypen können nicht überladen/verändert werden

Klassendefinition:

```
class CVector {
  ...
  CVector operator+(
    const CVector& right_operand
  ) const;
  ...
}
```

d.h. wir deklarieren einen Operator + als binäre Operation, welche implizit (als Objekt der aufgerufenen Memberfunktion) den linken Operanden und explizit als Parameter den rechten Operanden erhält.
Vorstellungsmöglichkeit: Wir hätten auch eine Memberfunktion `VectorAdd` definieren können:

```
class CVector {
  ...
  Vector VectorAdd(const CVector& right_operand) const;
  ...
};
```

welche dann wie folgt benutzt worden wäre:

```
CVector a,b,c;

c = a.VectorAdd(b);      // c = a + b
```

Hier ist der linke Operand `a` das Objekt, dessen Memberfunktion `VectorAdd` mit Operanden `b` aufgerufen wird.
Mit der Operatorschreibweise gilt folgende Vorstellung: Der Compiler erkennt zuerst den linken Operanden, danach den Operator, und weiß dann, welche Klassegemeint ist (Typ des linken Operanden), und sucht dann eine Memberfunktion (hier `operator+`), deren Parameter mit dem Typ des rechten Operanden übereinstimmt.
Genau genommen, folgende beiden Zeilen sind äquivalent und können beide verwendet werden:

```
c = a + b;
c = a.operator+(b);
```

Implementierung:

```
CVector CVector::operator+(
  const CVector& right_operand
) const {
  CVector v;     // construction of a CVector

#ifdef _DEBUG
  if(dimension!=right_operand.dimension) {
    cout << "fatal error: dimensions of vectors differ\n";
    exit(1);
  }
#endif
  for(int i=0;i<dimension;i++)
    v.vector[i]=vector[i]+right_operand.vector[i];
  return v;
}               // destruction of the CVector v
```

Die Funktion ist `const` deklariert, d.h. sie kann keine Member-Variable ändern.

> **Regel:** Wir implementieren Member-Funktionen wo immer möglich als `const`.

20 Genügt wirklich Konstruktor und Destruktor?

Nein! Betrachten wir uns obige Operatorfunktion.
Wir wollen den Vektor `v` zurückgeben, dieser ist jedoch eine lokale Variable, d.h. mit der schließenden Klammer der Funktion (letzte Zeile) wird der Destruktor für `v` aufgerufen und damit der bei der Konstuktion allokierte Speicherplatz wieder freigegeben.
Der Compiler gibt zwar eine Kopie von `v`, nämlich die beiden Komponenten Zeiger `vector` und Integer `dimension`, zurück, der allokierte Speicherplatz ist aber verschwunden, und damit: Programmabsturz!

20.1 Kopier-Konstruktor

Wir stellen dem Compiler also einen Kopier-Konstruktor als Memberfunktion zur Verfügung, damit er bei der Parameterübergabe und bei der Zurückgabe von Funktionswerten korrekt arbeiten kann.
Deklaration:

```
class CVector {
  ...
  CVector(const CVector& v);  // copy-constructor
  ...
};
```

Implementierung:

```
CVector::CVector(
  const CVector& v
) {
```

```
  dimension = v.dimension;
  vector    = new double[dimension];
  for(int i=0; i<dimension; i++)
    vector[i] = v.vector[i];
}
```

20.2 Zuweisungsoperator

Ein ähnliches Problem tritt bei der Zuweisung von Objekten auf:

```
CVector v,w;

v=w;
```

Was erwarten wir? dass richtig kopiert wird. Der Compiler kopiert aber wieder nur die Membervariablen (also `vector` und `dimension`) und der allokierte Speicherplatz von `v`, der im Konstruktor allokiert wurde, geht "verloren", und beide Vektoren haben plötzlich Zeiger auf das gleiche Feld!
Wir stellen dem Compiler also einen Zuweisungsoperator als Memberfunktion zur Verfügung, damit er bei der Zuweisung korrekt arbeiten kann.
Deklaration:

```
class CVector {
  ...
  CVector& operator=(        // assignment-operator
    const CVector& v
  );
  ...
};
```

Implementierung (vorläufig!):

```
CVector& CVector::operator=(
  const CVector& v
) {
#ifdef _DEBUG
  if(dimension!=v.dimension) {
    cout << "fatal error: dimensions of vectors differ\n";
    exit(1);
  }
#endif
  dimension = v.dimension;
  vector    = new double[dimension];
  for(int i=0; i<dimension; i++)
    vector[i] = v.vector[i];
  return ????;
}
```

Problem: was geben wir zurück? Wir müssten eigentlich gerade das Objekt zurückgeben, mit dem wir arbeiten ...

21 Kommen wir an das Objekt heran, dessen Memberfunktion wir aufgerufen haben?

Ja! In jeder Memberfunktion steht ein Zeiger `this` zur Verfügung, der auf das Objekt zeigt, mit dem die Funktion aufgerufen wurde. Mit anderen Worten: `this` zeigt auf das Objekt, welches entweder linker Operand eines binären Operators, Operand eines unären Operators, oder das Objekt links vom `.` ist.
Implementierung des Zuweisungsoperators:

```
CVector& CVector::operator=(
  const CVector& v
) {
  if(&v == this) return(*this);   // self-assignment

#ifdef _DEBUG
  if(dimension!=v.dimension) {
    cout << "fatal error: dimensions of vectors differ\n";
    exit(1);
  }
#endif
  dimension = v.dimension;
  vector    = new double[dimension];
  for(int i=0; i<dimension; i++)
    vector[i] = v.vector[i];
  return(*this);
}
```

> **Regel:** Wir schreiben für eine Klasse stets ein vollständiges Set mit den 4 notwendigen Funktionen: Konstrukter, Destruktor, Kopier-Konstruktor, und Zuweisungsoperator!

22 Können wir auch binäre Operatoren mit anderen linken Operanden schreiben?

Ja! Da es häufig sinnvoll ist, solche Operatoren zur Verfügung zu haben (z.B. skalare Multiplikation von links) wurde das Schlüsselword `friend` in C++ aufgenommen. Als `friend` in der Klasse deklarierte Funktionen haben das Recht auf private Komponenten zuzugreifen!
Beispiel der Ein- und Ausgabe über die Standard-Ströme:
Deklaration:

```
class CVector {
  ...
  friend ostream& operator<<(          // output to stream
    ostream&       os,
    const CVector& v
```

```
  );
  friend istream& operator>>(          // input from stream
    istream& is,
    CVector& v
  );
  ...
};
```

Implementierung:

```
ostream& operator<<(
  ostream&       os,
  const CVector& v
) {
  os << "( ";
  for(int i=0;i<v.dimension;i++) {
    os << v.vector[i] << " ";
  }
  os << ") " << endl;
  return(os);
}

istream& operator>>(
  istream& is,
  CVector& v
) {
  cout << "vector(" << v.dimension << ") = ";
  for(int i=0;i<v.dimension;i++) {
    is >> v.vector[i];
  }
  return(is);
}
```

Man beachte:

- `friend`-Funktionen sind keine Member-Funktionen, sondern global verfügbare Funktionen, weshalb auch kein Gültigkeitsbereich bei der Implementierung angegeben wird.
- `friend`-Funktionen haben damit keinen `this`-Zeiger.
- Damit sind `friend`-Funktionen immer öffentlich, egal an welcher Stelle sie in der Klasse deklariert sind.
- Die Operatoren `=`, `[]`, `()` und `->` können nicht mit `friend`-Funktionen überlagert werden.
- Es können auch alle Member-Funktionen einer Klasse als Freunde für eine andere Klasse deklariert werden: `friend class Class_name;`.

- Klassen können gegenseitig als Freunde deklariert werden.

> **Regel:** Wir verwenden `friend`-Funktionen nur wohlüberlegt, da sie Zugriff auf private Komponenten einer Klasse haben.

23 Wie sieht nun ein Hauptprogramm aus?

```
void main(
) {
  CVector a,b,c;
  double  d;

  cin >> a;
  cin >> b;
  cout << a+b << endl;  // sum
  c = a+a;
  cout << c << endl;    // after assignment
  d = a*b;
  cout << d << endl;    // dotproduct
  if(!a.IsNullVector()) {
    c = !a;
    cout << c << endl;  // normalization
  }
}
```

24 Übung: Implementierung der Vektorklasse

Wir implementieren obige `CVector`-Klasse mit weiteren Operatoren:

- unäres Minus `-` als Vektorumkehr
- binäres Minus `-` als Vektordifferenz
- unäres Plus `+` als Vektoridentität
- unäres Ausrufezeichnen `!` als Normierung
- binäres Mal `*` als Skalarprodukt,
- binäres Mal `*` als Skalarmultiplikation, wobei letzteres in beiden Varianten

Bemerkung: insbesondere auch die Operatoren `++` und `- -` können überladen werden, den Unterschied zwischen Postfix und Präfix erzeugt man durch Angabe eines "Platzhalter" Parameters vom Typ `int` beim Postfix-Operator.

> **Regel:** Wir überladen Operatoren nur sinnvoll, d.h. es sollte leicht erkennbar sein, was der Operator tun soll.

Beispiel: Überladen von `[]`
Deklaration:

```
class CVector {
  ...
  double operator[](
    int i
  ) const;
  ...
};
```

Implementierung:

```
double CVector::operator[](
  int i
) const {
  if(i<0 || i>=dimension) {
    cout << "index " << i << " out of range "
         << "for dimension " << dimension << endl;
    exit(1);
  }
  return(vector[i]);
}
```

Damit können wir folgendes schreiben:

```
  CVector v;
  double  d;
  ...
  d=v[2];
  ...
```

aber folgendes nicht:

```
  ...
  v[2]=17.0;
  ...
```

Wenn wir jedoch den Rückgabetyp des Operators in eine Referenz umwandeln geht dies auch! und wir können die Komponenten des Vektors setzen. Deklaration:

```
class CVector {
  ...
  double& operator[](int i) const;
  ...
};
```

25 Übung: Fertigstellen der Vektorklasse

Führen Sie folgende Arbeiten durch:

- Ändern Sie das bisherige `length` in `dimension`. (wurde hier im Skript schon durchgeführt).

- Ändern Sie die Klasse so, dass auch Vektoren der Dimension 0 korrekt behandelt werden.
- Erweitern Sie die Klasse um die Member-Funktionen `bool IsNullVector()`, `double Length()` (Euklidsche Länge) und `int Dimension()`.
- Fügen Sie der Klasse eine Funktion `CVector SolvePolynom()` hinzu, deren impliziter Parameter den Koeffizienten-Vektor eines Polynoms darstellt und als Rückgabewert den Vektor der Nullstellen berechnet.

26 Was können die Klassen `cout` und `cin`?

Ein Streifzug durch die Hilfe des Compilers ...

27 Einer für alle, alle auf Einen?

Manchmal kann es sinnvoll eine Variable zu haben, die für alle Instanzen einer Klasse nur einmal existiert. Ähnlich wie in C, wo man Variablen innerhalb von Funktionen mithilfe von `static` deklarieren konnte. Der Sichtbarkeitsbereich war dann zwar auf die Funktion beschränkt, sie verhielten sich jedoch wie globale Variablen, d.h. sie wurden nur einmal initialisiert und behielten über Funktionsaufrufe hinweg ihren Wert.
In C++ können Member-Variablen einen ähnliches Verhalten zeigen.
Deklaration:

```
class CVector {
private:
  static int    instances;     // instance-counter
  static double epsilon;       // zero-vector tolerance
...
public:
  void GetInstances() const;   // returns instances
  void SetEpsilon(double eps); // sets new tolerance
...
}
```

- Die mit `static` deklarierten Variablen existieren nur einmal.
- Sie müssen außerhalb der Klasse (in der entsprechenden .cpp-Datei) definiert werden:

Implementierung:

```
int    CVector::instances=0;
double CVector::epsilon  =1E-15;

void CVector::GetInstances(
) const {
```

```
  return(instances);
}

void CVector::SetEpsilon(
  double eps
) {
  epsilon=fabs(eps);
}
```

Erhöhen wir nun in den Konstruktoren `instances`, und erniedrigen wir es im Destruktor, so können wir mitzählen, wieviele Objekte `CVector` z.Z. im Programm definiert sind und z.B. folgendes Programmsegment schreiben:

```
...
  CVector *v;

  v = new CVector[17];

  cout << (*v).GetInstances() << endl;
...
```

Im obigen Vektorfeld können wir übrigens die dritte Komponente des sechsten Vektors wie folgt setzen:

```
  v[5][2]=2.0;
```

28 Wie sieht eine schon brauchbare Vektorklasse aus?

Damit sieht unsere soweit fertige Vektorklasse wie folgt aus:

```
class CVector {
private:
  static double epsilon;
  static int    instances;
private:
  int     dimension;
  double *vector;
public:
  /* constructors and destructor */
   CVector(int dimension=3);
   CVector(const CVector& v);
  ~CVector();

  /* input and output */
  friend ostream& operator<<(
          ostream& os,
    const CVector& v
  );
```

```
  friend istream& operator>>(
    istream& is,
    CVector& v
  );

  /* some operators */
  CVector& operator=(const CVector& v);
  CVector  operator+(const CVector& v) const;
  CVector  operator-(const CVector& v) const;
  double   operator*(const CVector& v) const;
  CVector  operator+() const;
  CVector  operator-() const;
  CVector  operator!() const;
  CVector  operator*(double d) const;
  friend CVector operator*(
    const CVector& v,
          double   d
  );
  double&  operator[](int i) const;

  /* useful functions */
  int      Dimension() const;
  double   Length() const;
  bool     IsNullVector() const;
  CVector  SolvePolynom() const;

  /* control functions */
  void     SetEpsilon();
  void     GetEpsilon();
  void     GetInstances();
};
```

29 Was können wir mit unseren Objekten machen?

Komposition *hat eine* Beziehung

Ableitung *ist eine* Beziehung

Beispiel: ein Kreis hat einen Mittelpunkt, aber ein Kreis ist eine Flaeche.
Eigentlich ist eine Fläche ein abstrakter Begriff, er bekommt erst eine konkrete Bedeutung, wenn die Geometrie festliegt. Trotzdem kann eine Fläche bestimmte Eigenschaften (Schnittstellen nach außen) haben, die unabhängig von der zugrundeliegenden Geometrie für alle Flächen vorhanden sind.
Eine Fläche hat z.B. einen Flächeninhalt und eine Schwerpunkt.
Gehen wir Schritt für Schritt vor:
1. Deklaration eines Kreises:

```
class CCircle {
private:
  CVector center;
  double  radius;
public:
  CCircle(double radius=1.0);   // (default) constructor
  CCircle(const CVector& center, double radius=1.0);
  CCircle(const CCircle& circle);
  virtual ~CCircle();
  CCircle& operator=(const CCircle& circle);
  ...                           // more functions and operators
};
```

Probleme: damit ist noch nicht gesagt, dass ein Kreis eine Fläche sein soll, und der Mittelpunkt hat noch keine Dimension (default wäre 3, wir wollen aber nur im Zweidimensionalen arbeiten).
1. Deklaration einer Fläche:

```
class CArea {
private:
  CVector gravity_center;
  double  area;
public:
  CArea();                      // constructor
  virtual ~CArea();             // destructor
  CArea(const CArea& area);     // copy-constructor
  CArea& operator=(const CArea& area);
  ...                           // more functions and operators
};
```

Wie konstruieren wir nun die Fläche, so dass sie einen Schwerpunkt der Dimension 2 hat?
Konstruktor einer Fläche:

```
CArea::CArea(
) :
  gravity_center(2)         // constructing member-object !!
{
  gravity_center[0]= 0.0;   // setting members to default
  gravity_center[1]= 0.0;   // values
  area             =-1.0;   // invalid value  !!
}
```

Also: der Schwerpunkt wird am Anfang des Konstruktors der Fläche (sozusagen vor der öffnenden geschweiften Klammer) als `CVector` der Dimension 2 konstruiert.
Weitere Konstruktoren von Member-Objekten würden mit Komma getrennt an die gleiche Stelle geschrieben.
Alle Member-Objekte werden initialisiert, insbesondere, da eine Fläche an sich - ohne Geometrie - keinen Sinn macht, wird der Inhalt negativ initialisiert.

Leiten wir nun den Kreis von der Basisklasse Fläche ab:
2. Deklaration eines Kreises:

```
class CCircle : public CArea {  // circle is an area
private:
  CVector center;               // circle has a center
  double  radius;
public:
  CCircle(double radius=1.0);   // (default) constructor
  CCircle(const CVector& center, double radius=1.0);
  CCircle(const CCircle& circle);
  virtual ~CCircle();
  CCircle& operator=(const CCircle& circle);
  ...                           // more functions and operators
};
```

und entwickeln den Konstruktor:
1. Definition eines Kreis-Konstruktors:

```
CCircle::CCircle(
  double radius
) :
  CArea(),              // explicit construction of base class
  center(2)             // construction of member-object
{
  center[0]=0.0;
  center[1]=0.0;
  CCircle::radius=radius;
}
```

Wir rufen also beim Konstruieren eines Kreises zusätzlich den Konstruktor der Basisklasse auf.
Bemerkung: wird der Konstruktor der Basisklasse nicht explizit aufgerufen, so wird implizit der Standard-Konstruktor der Basisklasse verwendet, was in unserem Beispiel das gleiche wäre.
Problem: das Objekt der Basisklasse ist noch nicht initialisiert, d.h. der Schwerpunkt und der Inhalt sind nicht richtig gesetzt!
Wie kommt die abgeleitete Klasse an die privaten Member-Objekte der Basisklasse heran? Gar nicht!
Wir müssen die Member-Objekte `protected` deklarieren! Damit sind sie nach außen weiterhin “unsichtbar”, können jedoch in Member-Funktionen abgeleiteter Klassen verwendet werden.
2. Deklaration einer Fläche:

```
class CArea {
protected:
  CVector gravity_center;
  double  area;
public:
```

```
  CArea();                      // constructor
  virtual ~CArea();             // destructor
  CArea(const CArea& area);     // copy-constructor
  CArea& operator=(const CArea& area);
  ...                           // more functions and operators
};
```

Damit können wir den Konstruktor eines Kreises vervollständigen:
2. Definition eines Kreis-Konstruktors:

```
CCircle::CCircle(
  double radius
) :
  CArea(),              // explicit construction of base class
  center(2)             // construction of member-object
{
  center[0]=0.0;
  center[1]=0.0;
  CCircle::radius=radius;
  gravity_center=center;             // init of base class
  area          =pi*radius*radius;   // members
}
```

Bemerkung: wir könnten `pi` z.B. als statische Member-Variable eines Kreises deklarieren.
Fügen wir die fehlenden Member-Funktionen ein, können wir damit z.B. folgendes Programm schreiben:

```
main(
) {
  CCircle K1(6.0);
  CVector m(2);

  m[0]=1.0;
  m[1]=2.0;

  CCircle K2(m);

  cout << K1 << endl;
  cout << K2 << endl;
}
```

was folgende oder ähnliche Ausgabe erzeugen sollte:

```
Kreis:
  Mittelpunkt = (0.0,0.0)
  Radius      = 6.0
  Schwerpunkt = (0.0,0.0)
```

```
  Inhalt      = 113.097

Kreis:
  Mittelpunkt = (1.0,2.0)
  Radius      = 1.0
  Schwerpunkt = (1.0,2.0)
  Inhalt      = 3.14159
```

Wir beachten also bei Konstruktoren für abgeleitete Klassen, dass folgende Punkte korrekt abgearbeitet werden:

1. Konstruktion der Basisklasse (eventuell mehrere Konstruktoren)
2. Konstruktion der Member-Objekte
3. Initialisierung der Member-Objekte
4. Initialisierung der Member-Objekte der Basisklasse

30 Können wir auch anders als `public` ableiten?

Wir eine Basisklasse nicht `public` abgeleitet, so ändern sich die Zugriffsrechte auf Member-Objekte in der abgeleiteten Klasse nach folgender Tabelle:

Ableitung	Basisklasse	abgeleitete Klasse
public	private protected public	private protected public
protected	private protected public	private protected protected
private	private protected public	private private private