Einführung in die Programmierung für Physiker

Die Programmiersprache C++ – Basics an Hand von Beispielen

Marc Wagner

Institut für theoretische Physik
Johann Wolfgang Goethe-Universität Frankfurt am Main

WS 2017/18

C++, objektorientierte Programmiersprachen

- **C** ist eine **imperative Programmiersprache**.
- **C++** ist eine **objektorientierte Programmiersprache**.
- **Wiki → Programmiersprache:**
  - **Imperative Programmiersprachen:**
    Ein in einer imperativen Programmiersprache geschriebenes Programm besteht aus Anweisungen (latein imperare = befehlen), die beschreiben, wie das Programm seine Ergebnisse erzeugt (zum Beispiel Wenn-dann-Folgen, Schleifen, Multiplikationen et cetera).
  - **Objektorientierte Programmiersprachen:**
    Hier werden Daten und Befehle, die auf diese Daten angewendet werden können, in Objekten zusammengefasst. Objektorientierung wird im Rahmen der objektorientierten Programmierung verwendet, um die Komplexität der entstehenden Programme zu verringern.
    Die Bausteine, aus denen ein objektorientiertes Programm besteht, werden als Objekte bezeichnet. Die Konzeption dieser Objekte erfolgt dabei in der Regel auf Basis der folgenden Paradigmen:
    - **Polymorphismus**:
      Fähigkeit eines Bezeichners, abhängig von seiner Verwendung unterschiedliche Datentypen anzunehmen.
    - **Datenkapselung**:
      Als Datenkapselung bezeichnet man in der Programmierung das Verbergen von Implementierungsdetails.
    - **Vererbung**:
      Vererbung heißt vereinfacht, dass eine abgeleitete Klasse die Methoden und Attribute der Basisklasse ebenfalls besitzt, also erbt.
- **C++** ist eine Erweiterung von **C**, d.h. **C**-Funktionen, **C**-Programmteile, etc. können auch in **C++** verwendet werden und sollten mit **C++**-Compilern problemlos zu kompilieren sein.

Klassen und Objekte

- Die wesentlichen Bausteine eines **C++**-Programms sind **Klassen** und **Objekte**, die eine Erweiterung der Strukturen der Programmiersprache **C** darstellen.

Klassen

- Die Definition einer Klasse ähnelt der Definition einer Struktur in **C**.
- Im Gegensatz zu einer Struktur besitzt eine Klasse nicht nur Variablen (werden als **Attribute** bezeichnet), sondern auch Funktionen (werden als **Methoden**/**member functions** bezeichnet).

Objekte

- Ein Objekt ist eine Variable, deren Datentyp eine Klasse ist.
- Es kann viele Objekte derselben Klasse geben.

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. // ********************
5.
6. // Definition einer Klasse fuer Matrizen beliebiger Groesse.
7.
8. class Matrix
9. {
10. public:  // Wird spaeter erlaeutert.
11.
12.   // Methoden.
13.
14.   // Reserviert entsprechend Speicher fuer die Matrixelemente, initialisiert
15.   // die Attribute (eine m_ x n_ Matrix, alle Eintraege 0.0).
16.   void Init(int m_, int n_);
17.
18.   // Liefert den Wert des (i,j)-ten Matrixelements.
19.   double Get(int i, int j);
20.
21.   // Gibt die Matrix am Bildschirm aus.
22.   void Print();
23.
24.   // *****
25.
26.   // Attribute.
27.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
28.   double *elements;  // Die Matrixeintraege.
29. };
30.
31. // Im Folgenden werden die in
32. //   class Matrix { ... }
33. // deklarierten Methoden definiert.
34.
35. void Matrix::Init(int m_, int n_)
36. {
37.   int i1;
38.
39.   // Innerhalb der Methoden einer Klasse kann direkt auf deren Attribute
40.   // zugegriffen werden; der in C benoetigte "." bzw "->" ist nur bei Zugriff
41.   // von aussen (z.B. von der main-Funktion oder von den Methoden einer
42.   // anderen Klasse aus) notwendig.
43.   m = m_;
44.   n = n_;
45.
46.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
```

```
47.     {
48.       fprintf(stderr, "Fehler in void Matrix::Init(...\n");
49.       exit(0);
50.     }
51.
52.   for(i1 = 0; i1 < m*n; i1++)
53.     elements[i1] = 0.0;
54. }
55.
56. double Matrix::Get(int i, int j)
57. {
58.   if(i < 0 || i >= m || j < 0 || j >= n)
59.     {
60.       fprintf(stderr, "Fehler in double Matrix::Get(...\n");
61.       exit(0);
62.     }
63.
64.   return elements[i*n + j];
65. }
66.
67. void Matrix::Print()
68. {
69.   int i1, i2;
70.
71.   for(i1 = 0; i1 < m; i1++)
72.     {
73.       printf("| ");
74.
75.       for(i2 = 0; i2 < n; i2++)
76.         {
77.           printf("%+.3e ", Get(i1, i2));
78.         }
79.
80.       printf("|\n");
81.     }
82. }
83.
84. // ********************
85.
86. int main(void)
87. {
88.   Matrix A;  // Definition eines Objekts A der Klasse Matrix.
89.   A.Init(3, 2);
90.   A.Print();
91. }
```

```
| +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 |
```

- **Wesentliche Idee:** Jedes Objekt bringt seine eigenen Funktionen mit; dies soll eher der **menschlichen Denkweise** entsprechen, als eine Trennung von Funktionen und Variablen; z.B.
  - `A.Init(3, 2);` ... "**Matrix A, initialisiere Dich mit Größe 3, 2!**"
  - `A.Print();` ... "**Matrix A, drucke Dich!**"

Konstruktoren und Destruktoren

Konstruktoren

- Viele Fehler entstehen dadurch, dass man vergisst, Variablen geeignet zu initialisieren.
- **C++** ermöglicht in Klassen die Definition sogenannter **Konstruktoren**, Methoden, die automatisch aufgerufen werden, sobald ein neues Objekt dieser Klasse ins Leben gerufen wird.
- Der Methodenname des Konstruktors entspricht dem Klassennamen; die Parameterliste kann beliebig gewählt werden; es gibt keinen Rückgabewert (auch kein **void**).

```
1. ...
2.
3. // Definition einer Klasse fuer Matrizen beliebiger Groesse.
4.
5. class Matrix
6. {
7. public:  // Wird spaeter erlaeutert.
8.
9.   // Methoden.
10.
11.   // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
12.   // initialisiert die Attribute (eine m_ x n_ Matrix, allen Eintraege 0.0).
13.   Matrix(int m_, int n_);
14.
15.   // Liefert den Wert des (i,j)-ten Matrixelements.
16.   double Get(int i, int j);
17.
18.   // Gibt die Matrix am Bildschirm aus.
19.   void Print();
20.
21.   // *****
22.
23.   // Attribute.
24.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
25.   double *elements;  // Die Matrixeintraege.
26. };
27.
28. // Im Folgenden werden die in
29. //   class Matrix { ... }
30. // deklarierten Methoden definiert.
31.
32. Matrix::Matrix(int m_, int n_)
33. {
34.   fprintf(stderr, "Der Konstruktor Matrix::Matrix(... wurde aufgerufen.\n");
35.
36.   int i1;
37.
38.   m = m_;
39.   n = n_;
40.
41.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
42.     {
43.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
44.       exit(0);
45.     }
46.
47.   for(i1 = 0; i1 < m*n; i1++)
48.     elements[i1] = 0.0;
49. }
50.
51. ...
52.
```

```
53. int main(void)
54. {
55.   // Definition eines Objekts A der Klasse Matrix; es wird automatisch der oben
56.   // definierte Konstruktor aufgerufen.
57.   Matrix A(3, 2);
58.
59.   A.Print();
60. }
```

```
Der Konstruktor Matrix::Matrix(... wurde aufgerufen.
| +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 |
```

Destruktoren

- Im obigen Beispiel wurde "vergessen", den für die Matrix **A** reservierten Speicher am Ende wieder freizugeben.
- **C++** ermöglicht in Klassen die Definition sogenannter **Destruktoren**, Methoden, die automatisch aufgerufen werden, sobald die Lebensdauer eines Objekts endet.
- Der Funktionsname des Konstruktors entspricht *~Klassenname*; es gibt weder Parameter, noch einen Rückgabewert.

```
1. ...
2.
3. // Definition einer Klasse fuer Matrizen beliebiger Groesse.
4.
5. class Matrix
6. {
7. public:  // Wird spaeter erlaeutert.
8.
9.   // Methoden.
10.
11.   // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
12.   // initialisiert die Attribute (eine m_ x n_ Matrix, allen Eintraege 0.0).
13.   Matrix(int m_, int n_);
14.
15.   // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
16.   // frei.
17.   ~Matrix();
18.
19.   // Liefert den Wert des (i,j)-ten Matrixelements.
20.   double Get(int i, int j);
21.
22.   // Gibt die Matrix am Bildschirm aus.
23.   void Print();
24.
25.   // *****
26.
27.   // Attribute.
28.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
29.   double *elements;  // Die Matrixeintraege.
30. };
31.
32. // Im Folgenden werden die in
33. //   class Matrix { ... }
34. // deklarierten Methoden definiert.
35.
36. ...
37.
38. Matrix::~Matrix()
39. {
40.   fprintf(stderr, "Der Destruktor Matrix::~Matrix(... wurde aufgerufen.\n");
41.
42.   if(elements != NULL)
43.     free(elements);
```

```
44. }
45.
46. ...
47.
48. int main(void)
49. {
50.   // Definition eines Objekts A der Klasse Matrix; es wird automatisch der oben
51.   // definierte Konstruktor aufgerufen.
52.   Matrix A(3, 2);
53.
54.   A.Print();
55.
56.   // Die Lebensdauer von A endet; es wird automatisch der oben definierte
57.   // Destruktor aufgerufen.
58. }
```

```
Der Konstruktor Matrix::Matrix(... wurde aufgerufen.
| +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 |
Der Destruktor Matrix::~Matrix(... wurde aufgerufen.
```

public, private

- Attribute und Methoden einer Klasse können **`public`** oder **`private`** sein (oder auch **`protected`**, was in dieser Einführung nicht weiter erläutert wird).
- **`public`**:
  Solche Attribute und Methoden sind überall im Programm sichtbar; damit kann jede Funktion oder Methode (auch Methoden anderer Klassen) auf sie zugreifen bzw. sie aufrufen.
- **`private`**:
  Nur Methoden dieser Klasse können solche Attribute und Methoden sehen und damit auf sie zugreifen bzw. sie aufrufen.
- **Beispiel:** **`public`**- und **`private`**-Attribute und -Methoden ...

```
 1. #include<stdio.h>
 2.
 3. class MyClass
 4. {
 5. public:
 6.   int a;
 7.   void Set_b(int b_);
 8.   void Print_b(void);
 9.
10. private:
11.   int b;
12. };
13.
14. void MyClass::Set_b(int b_)
15. {
16.   b = b_;
17. }
18.
19. void MyClass::Print_b(void)
20. {
21.   printf("%d", b);
22. }
23.
24. int main(void)
25. {
26.   MyClass cl;
27.   cl.a = 5;
28.   printf("cl.a = %d\n", cl.a);
29. }
```

```
cl.a = 5
```

- **Beispiel:** Der Zugriff von außen auf das **`private`**-Attribut **`b`** liefert einen Fehler beim Kompilieren ...

```
 1. ...
 2.
 3. int main(void)
 4. {
 5.   MyClass cl;
 6.   cl.a = 5;
 7.   printf("cl.a = %d\n", cl.a);
 8.   cl.b = 5;
 9.   printf("cl.b = %d\n", cl.b);
10. }
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 8
-rw-rw-r-- 1 mwagner mwagner  25 Jan 29 14:53 Makefile
-rw-rw-r-- 1 mwagner mwagner 333 Jan 29 16:56 prog.C
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o prog prog.C
prog.C: In Funktion "int main()":
```

```
prog.C:11:7: Fehler: "int MyClass::b" ist privat
prog.C:29:6: Fehler: in diesem Zusammenhang
prog.C:11:7: Fehler: "int MyClass::b" ist privat
prog.C:30:28: Fehler: in diesem Zusammenhang
```

- **Beispiel:** Methoden der Klasse **`MyClass`** können auf deren **`private`**-Attribut **`b`** zugreifen; sind diese Methoden **`public`**, kann damit indirekt von außen auf das **`private`**-Attribut **`b`** zugegriffen werden ...

```
...

int main(void)
{
  MyClass cl;

  cl.a = 5;
  printf("cl.a = %d\n", cl.a);

  cl.Set_b(7);
  printf("cl.b = ");
  cl.Print_b();
  printf("\n");
}
```

```
cl.a = 5
cl.b = 7
```

- **Vorteil/Nutzen von `public` und `private`:** Die Daten eines Objekts sollen von außen nicht beliebig zugreifbar, insbesondere nicht modifizierbar sein; idealer Weise existieren nur solche **`public`**-Methoden, mit denen die Daten des Objekts im Sinn der zugrunde liegenden Klasse verändert werden können, sinnlose Veränderungen oder solche, die zu Fehlern führen, jedoch nicht möglich sind.
- **Beispiel:** Zurück zur **`Matrix`**-Klasse ... ein unwissender oder böswilliger Benutzer dieser Klasse könnte schlimme Fehler verursachen ...

```
#include<stdio.h>
#include<stdlib.h>

// ********************

// Definition einer Klasse fuer Matrizen beliebiger Groesse.

class Matrix
{
public:  // Alle Methoden und Attribute sind public ...

  // Methoden.

  // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
  // initialisiert die Attribute (eine m_ x n_ Matrix, allen Eintraege 0.0).
  Matrix(int m_, int n_);

  // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
  // frei.
  ~Matrix();

  // Liefert den Wert des (i,j)-ten Matrixelements.
  double Get(int i, int j);

  // Gibt die Matrix am Bildschirm aus.
  void Print();

  // *****

  // Attribute.
  int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
  double *elements;  // Die Matrixeintraege.
```

```
33. };
34.
35. // Im Folgenden werden die in
36. //   class Matrix { ... }
37. // deklarierten Methoden definiert.
38.
39. Matrix::Matrix(int m_, int n_)
40. {
41.   int i1;
42.
43.   m = m_;
44.   n = n_;
45.
46.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
47.     {
48.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
49.       exit(0);
50.     }
51.
52.   for(i1 = 0; i1 < m*n; i1++)
53.     elements[i1] = 0.0;
54. }
55.
56. Matrix::~Matrix()
57. {
58.   if(elements != NULL)
59.     free(elements);
60. }
61.
62. double Matrix::Get(int i, int j)
63. {
64.   if(i < 0 || i >= m || j < 0 || j >= n)
65.     {
66.       fprintf(stderr, "Fehler in double Matrix::Get(...\n");
67.       exit(0);
68.     }
69.
70.   return elements[i*n + j];
71. }
72.
73. void Matrix::Print()
74. {
75.   int i1, i2;
76.
77.   for(i1 = 0; i1 < m; i1++)
78.     {
79.       printf("| ");
80.
81.       for(i2 = 0; i2 < n; i2++)
82.         {
83.           printf("%+.3e ", Get(i1, i2));
84.         }
85.
86.       printf("|\n");
87.     }
88. }
89.
90. // ********************
91.
92. int main(void)
93. {
94.   Matrix A(3, 2);
95.
96.   A.elements = NULL;
97.   A.Print();
98. }
```

```
Speicherzugriffsfehler (Speicherabzug geschrieben)
```

```
...

int main(void)
{
  Matrix A(3, 2);

  A.m = 6;
  A.n = 6;
  A.Print();
}
```

```
| +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 |
| +0.000e+00 +6.675e-319 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00 |
```

- ... bei geeigneter Verwendung von **`public`** und **`private`** liefert eine derartige Verwendung der Klasse **`Matrix`** bereits beim Kompilieren einen Fehler.

```
...

// Definition einer Klasse fuer Matrizen beliebiger Groesse.

class Matrix
{
public:  // Von aussen sichtbar.

  // Methoden.

  // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
  // initialisiert die Attribute (eine m_ x n_ Matrix, allen Eintraege 0.0).
  Matrix(int m_, int n_);

  // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
  // frei.
  ~Matrix();

  // Liefert den Wert des (i,j)-ten Matrixelements.
  double Get(int i, int j);

  // Gibt die Matrix am Bildschirm aus.
  void Print();

  // *****

private:  // Von aussen nicht sichtbar.

  // Attribute.
  int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
  double *elements;  // Die Matrixeintraege.
};

...

int main(void)
{
  Matrix A(3, 2);

  A.m = 6;
  A.n = 6;
  A.Print();
}
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 8
-rw-rw-r-- 1 mwagner mwagner   25 Jan 29 14:53 Makefile
```

```
-rw-rw-r-- 1 mwagner mwagner 1710 Jan 29 17:42 prog.C
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o prog prog.C
prog.C: In Funktion "int main()":
prog.C:33:7: Fehler: "int Matrix::m" ist privat
prog.C:98:5: Fehler: in diesem Zusammenhang
prog.C:33:10: Fehler: "int Matrix::n" ist privat
prog.C:99:5: Fehler: in diesem Zusammenhang
```

- Hilfsattribute und Hilfsmethoden, d.h. solche die nur intern benötigt werden, sollten immer **`private`** sein.
- Für einen Benutzer einer fertig implementierten Klasse ist eigentlich nur der **`public`**-Anteil der Klassendefinition (also **`public`**-Attribute und Deklarationen von **`public`**-Methoden) von Interesse (wird als **Schnittstelle** bezeichnet); der Rest, insbesondere die Definition und damit die Implementierung von Methoden, sollte bzw. braucht ihn nicht zu interessieren. Dies ist die eingangs erwähnte **Datenkapselung** (→ weniger fehleranfällige und übersichtlichere Programme; klare Schnittstellen erlauben auch die einfache Wiederverwendung einzelner Klassen in anderen Programmen).
- Häufig legt man für jede Klasse zwei Programmcodedateien an, eine kurze **`.h`**-Datei, die die Klassendefinition enthält (interessant für einen Benutzer) und eine **`.C`**-Datei, die die Implementierung der Methoden enthält (uninteressant für einen Benutzer).
- **Beispiel:** Die **`Matrix`**-Klasse ... aufgeteilt auf mehrere Dateien ...
  - Datei **`Matrix.h`**:

```
1. // Eine Klasse fuer Matrizen beliebiger Groesse.
2.
3. class Matrix
4. {
5. public:  // Von aussen sichtbar.
6.
7.   // Methoden.
8.
9.   // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
10.  // initialisiert die Attribute (eine m_ x n_ Matrix, allen Eintraege 0.0).
11.  Matrix(int m_, int n_);
12.
13.  // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
14.  // frei.
15.  ~Matrix();
16.
17.  // Liefert den Wert des (i,j)-ten Matrixelements.
18.  double Get(int i, int j);
19.
20.  // Gibt die Matrix am Bildschirm aus.
21.  void Print();
22.
23.  // *****
24.
25. private:  // Von aussen nicht sichtbar.
26.
27.  // Attribute.
28.  int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
29.  double *elements;  // Die Matrixeintraege.
30. };
```

  - Datei **`Matrix.C`**:

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. #include"Matrix.h"
5.
6. // ********************
7.
8. Matrix::Matrix(int m_, int n_)
9. {
```

```
10.   int i1;
11.
12.   m = m_;
13.   n = n_;
14.
15.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
16.     {
17.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
18.       exit(0);
19.     }
20.
21.   for(i1 = 0; i1 < m*n; i1++)
22.     elements[i1] = 0.0;
23. }
24.
25. // ********************
26.
27. Matrix::~Matrix()
28. {
29.   if(elements != NULL)
30.     free(elements);
31. }
32.
33. // ********************
34.
35. double Matrix::Get(int i, int j)
36. {
37.   if(i < 0 || i >= m || j < 0 || j >= n)
38.     {
39.       fprintf(stderr, "Fehler in double Matrix::Get(...\n");
40.       exit(0);
41.     }
42.
43.   return elements[i*n + j];
44. }
45.
46. // ********************
47.
48. void Matrix::Print()
49. {
50.   int i1, i2;
51.
52.   for(i1 = 0; i1 < m; i1++)
53.     {
54.       printf("| ");
55.
56.       for(i2 = 0; i2 < n; i2++)
57.         {
58.           printf("%+.3e ", Get(i1, i2));
59.         }
60.
61.       printf("|\n");
62.     }
63. }
```

- Datei **prog.C**:

```
1. #include"Matrix.h"
2.
3. int main(void)
4. {
5.   Matrix A(3, 2);
6.   A.Print();
7. }
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 16
```

```
-rw-rw-r-- 1 mwagner mwagner  25 Jan 29 14:53 Makefile
-rw-rw-r-- 1 mwagner mwagner 889 Jan 29 18:08 Matrix.C
-rw-rw-r-- 1 mwagner mwagner 711 Jan 29 18:09 Matrix.h
-rw-rw-r-- 1 mwagner mwagner  70 Jan 29 18:09 prog.C
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -c -o Matrix.o Matrix.C
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 20
-rw-rw-r-- 1 mwagner mwagner   25 Jan 29 14:53 Makefile
-rw-rw-r-- 1 mwagner mwagner  889 Jan 29 18:08 Matrix.C
-rw-rw-r-- 1 mwagner mwagner  711 Jan 29 18:09 Matrix.h
-rw-rw-r-- 1 mwagner mwagner 3016 Jan 29 18:09 Matrix.o
-rw-rw-r-- 1 mwagner mwagner   70 Jan 29 18:09 prog.C
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o prog Matrix.o prog.C
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 32
-rw-rw-r-- 1 mwagner mwagner   25 Jan 29 14:53 Makefile
-rw-rw-r-- 1 mwagner mwagner  889 Jan 29 18:08 Matrix.C
-rw-rw-r-- 1 mwagner mwagner  711 Jan 29 18:09 Matrix.h
-rw-rw-r-- 1 mwagner mwagner 3016 Jan 29 18:09 Matrix.o
-rwxrwxr-x 1 mwagner mwagner 9506 Jan 29 18:10 prog
-rw-rw-r-- 1 mwagner mwagner   70 Jan 29 18:09 prog.C
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ./prog
| +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 |
| +0.000e+00 +0.000e+00 |
```

Referenzen

- **C++** ermöglicht die Definition und Verwendung von **Referenzen**; diese verweisen auf Variablen und haben damit starke Ähnlichkeit zu Zeigern.
- Mit Hilfe einer Referenz kann die Variable, auf die die Referenz verweist, geändert werden.
- Die Verwendung des Inhaltsoperators *, wie man ihn beim Zugriff auf den Wert einer Variable mit Hilfe eines Zeigers verwendet, entfällt bei Referenzen.
- Eine Referenz kann daher im Programmcode wie die Variable selbst behandelt und verwendet werden; Referenzen erleichtern damit unter Umständen die Programmierung.
- Der Datentyp einer **int**-Referenz ist **int &**, der Datentyp einer **double**-Referenz ist **double &**, ...
- Referenzen müssen bei ihrer Definition initialisiert werden; sie verweisen danach unveränderbar auf die Variable, mit der sie initialisiert wurden.
- Jede weitere Verwendung einer Referenz entspricht der Verwendung der Variable, auf die die Referenz verweist.
- **Beispiel:** ...

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int a = 3;
6.   int &b = a;  // b ist eine Referenz auf a.
7.   printf("a = %d , b = %d\n", a, b);
8.
9.   a = a + 2;
10.  printf("a = %d , b = %d\n", a, b);
11.
12.  b = b - 3;
13.  printf("a = %d , b = %d\n", a, b);
14. }
```

```
a = 3 , b = 3
a = 5 , b = 5
a = 2 , b = 2
```

- **Beispiel:** Nochmal die **swap**-Funktion, dieses Mal ohne den Umweg über Zeiger ...

```
1. #include<stdio.h>
2.
3. void swap(int &a, int &b)  // Es werden Referenzen uebergeben, nicht nur Werte.
4. {
5.   int tmp = a;
6.   a = b;
7.   b = tmp;
8. }
9.
10. int main(void)
11. {
12.   int i1 = 3;
13.   int i2 = 7;
14.
15.   printf("i1 = %d, i2 = %d\n", i1, i2);
16.   swap(i1, i2);
17.   printf("i1 = %d, i2 = %d\n", i1, i2);
18. }
```

```
i1 = 3, i2 = 7
i1 = 7, i2 = 3
```

Überladen von Funktionen/Methoden und Operatoren

Überladen von Funktionen/Methoden

- In **C++** können zwei Funktionen/Methoden denselben Namen besitzen, wenn sie sich in der Parameterliste unterscheiden (dies ist ein Aspekt des eingangs erwähnten **Polymorphismus**).
- Die mehrfache Verwendung derselben Funktions-/Methodennamen bezeichnet man als **Überladen von Funktionen/Methoden**.
- **Beispiel:** Mehrere Versionen einer Maximumsfunktion `max` ...

```c
#include<stdio.h>

int max(int a, int b)
{
  fprintf(stderr, "Verwende int max(int a, int b) ...\n");

  if(a > b)
    return a;

  return b;
}

int max(int a, int b, int c)
{
  fprintf(stderr, "Verwende int max(int a, int b, int c) ...\n");

  if(a > b && a > c)
    return a;

  if(b > c)
    return b;

  return c;
}

double max(double a, double b)
{
  fprintf(stderr, "Verwende double max(double a, double b) ...\n");

  if(a > b)
    return a;

  return b;
}

int main(void)
{
  printf("%d\n", max(1, 4));
  printf("%d\n", max(3, 7, 2));
  printf("%f\n", max(2.0, 3.0));
}
```

```
Verwende int max(int a, int b) ...
4
Verwende int max(int a, int b, int c) ...
7
Verwende double max(double a, double b) ...
3.000000
```

```c
...

int main(void)
{
  printf("%d\n", max(1, 4));
  printf("%d\n", max(3, 7, 2));
```

```
  printf("%f\n", max(2.0, 3.0));

  // Fehler beim Kompilieren, da eine solche Version von max nicht
  // implementiert wurde.
  printf("%d\n", max(1, 4.0));
}
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 4
-rw-rw-r-- 1 mwagner mwagner 703 Jan 31 12:16 prog.C
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o prog prog.C
prog.C: In Funktion "int main()":
prog.C:44:28: Fehler: Aufruf des überladenen "max(int, double)" ist nicht eindeutig
prog.C:44:28: Anmerkung: Kandidaten sind:
prog.C:3:5: Anmerkung: int max(int, int)
prog.C:26:8: Anmerkung: double max(double, double)
```

- **Beispiel:** Zurück zur Matrix-Klasse ... mehrere Versionen des Konstruktors **`Matrix`** (und eine Methode zum Verändern von Matrixelementen) ...

```
#include<stdio.h>
#include<stdlib.h>

// ********************

// Definition einer Klasse fuer Matrizen beliebiger Groesse.

class Matrix
{
public:  // Von aussen sichtbar.

  // Methoden.

  // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
  // initialisiert die Attribute (eine m_ x n_ Matrix, allen Eintraege 0.0).
  Matrix(int m_, int n_);

  // Konstruktor: Kopiert eine bestehende Matrix.
  Matrix(const Matrix &A);

  // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
  // frei.
  ~Matrix();

  // Liefert den Wert des (i,j)-ten Matrixelements.
  double Get(int i, int j);

  // Setzt den Wert des (i,j)-ten Matrixelements auf den Wert von a.
  void Set(int i, int j, double a);

  // Gibt die Matrix am Bildschirm aus.
  void Print();

  // *****

private:  // Von aussen nicht sichtbar.

  // Wandelt Zeilenindex i und Spaltenindex j in den Index des Arrays
  // elements um; eine Hilfsfunktion fuer Get und Set, daher private.
  int Index(int i, int j);

  // Attribute.
  int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
  double *elements;  // Die Matrixeintraege.
};

// Im Folgenden werden die in
//   class Matrix { ... }
// deklarierten Methoden definiert.

```

```
51. Matrix::Matrix(int m_, int n_)
52. {
53.   int i1;
54.
55.   m = m_;
56.   n = n_;
57.
58.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
59.     {
60.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
61.       exit(0);
62.     }
63.
64.   for(i1 = 0; i1 < m*n; i1++)
65.     elements[i1] = 0.0;
66. }
67.
68. Matrix::Matrix(const Matrix &A)
69. {
70.   int i1;
71.
72.   m = A.m;
73.   n = A.n;
74.
75.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
76.     {
77.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
78.       exit(0);
79.     }
80.
81.   for(i1 = 0; i1 < m*n; i1++)
82.     elements[i1] = A.elements[i1];
83. }
84.
85. Matrix::~Matrix()
86. {
87.   if(elements != NULL)
88.     free(elements);
89. }
90.
91.
92. int Matrix::Index(int i, int j)
93. {
94.   if(i < 0 || i >= m || j < 0 || j >= n)
95.     {
96.       fprintf(stderr, "Fehler in int Matrix::Index(...\n");
97.       exit(0);
98.     }
99.
100.  return i*n + j;
101. }
102.
103. double Matrix::Get(int i, int j)
104. {
105.   return elements[Index(i, j)];
106. }
107.
108. void Matrix::Set(int i, int j, double a)
109. {
110.   elements[Index(i, j)] = a;
111. }
112.
113. void Matrix::Print()
114. {
115.   int i1, i2;
116.
```

```
117.   for(i1 = 0; i1 < m; i1++)
118.     {
119.       printf("| ");
120.
121.       for(i2 = 0; i2 < n; i2++)
122.         {
123.           printf("%+.3e ", Get(i1, i2));
124.         }
125.
126.       printf("|\n");
127.     }
128. }
129.
130. // ********************
131.
132. int main(void)
133. {
134.   Matrix A(2, 2);
135.   A.Set(0, 0, 1.0);
136.   A.Set(1, 1, 1.0);
137.   printf("A =\n");
138.   A.Print();
139.
140.   printf("********************\n");
141.
142.   Matrix B(A);
143.   printf("B =\n");
144.   B.Print();
145.
146.   printf("********************\n");
147.
148.   A.Set(0, 0, 2.0);
149.   A.Set(1, 1, 3.0);
150.   printf("A =\n");
151.   A.Print();
152.
153.   printf("********************\n");
154.
155.   printf("B =\n");
156.   B.Print();
157. }
```

```
A =
| +1.000e+00 +0.000e+00 |
| +0.000e+00 +1.000e+00 |
********************
B =
| +1.000e+00 +0.000e+00 |
| +0.000e+00 +1.000e+00 |
********************
A =
| +2.000e+00 +0.000e+00 |
| +0.000e+00 +3.000e+00 |
********************
B =
| +1.000e+00 +0.000e+00 |
| +0.000e+00 +1.000e+00 |
```

Überladen von Operatoren

- In **C++** können auch Operatoren (z.B. **+**, **-**, **=**, etc.) überladen werden.
- **Beispiel:** Auch wenn der Multiplikationsoperator * bereits für **int**, **double**, etc. existiert, können weitere Versionen definiert werden, z.B. Multiplikation einer reellen Zahl mit einer Matrix (also **double** mit **Matrix**) oder Multiplikation von zwei Matrizen (also **Matrix** mit **Matrix**) ...

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. // ********************
5.
6. // Definition einer Klasse fuer Matrizen beliebiger Groesse.
```

```
 7.
 8. class Matrix
 9. {
10. public:  // Von aussen sichtbar.
11.
12.   // Methoden.
13.
14.   // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
15.   // initialisiert die Attribute (eine m_ x n_ Matrix, allen Eintraege 0.0).
16.   Matrix(int m_, int n_);
17.
18.   // Konstruktor: Kopiert eine bestehende Matrix.
19.   Matrix(const Matrix &A);
20.
21.   // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
22.   // frei.
23.   ~Matrix();
24.
25.   // Liefert den Wert des (i,j)-ten Matrixelements.
26.   double Get(int i, int j) const;
27.
28.   // Liefert den Wert des (i,j)-ten Matrixelements.
29.   void Set(int i, int j, double a);
30.
31.   // Gibt die Matrix am Bildschirm aus.
32.   void Print() const;
33.
34.   int Get_m(void) const { return m; }
35.   int Get_n(void) const { return n; }
36.
37.   // *****
38.
39. private:  // Von aussen nicht sichtbar.
40.
41.   // Wandelt Zeilenindex i und Spaltenindex j in den Index des Arrays
42.   // elements um; eine Hilfsfunktion fuer Get und Set, daher private.
43.   int Index(int i, int j) const;
44.
45.   // Attribute.
46.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
47.   double *elements;  // Die Matrixeintraege.
48. };
49.
50. // Im Folgenden werden die in
51. //   class Matrix { ... }
52. // deklarierten Methoden definiert.
53.
54. Matrix::Matrix(int m_, int n_)
55. {
56.   int i1;
57.
58.   m = m_;
59.   n = n_;
60.
61.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
62.     {
63.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
64.       exit(0);
65.     }
66.
67.   for(i1 = 0; i1 < m*n; i1++)
68.     elements[i1] = 0.0;
69. }
70.
71. Matrix::Matrix(const Matrix &A)
72. {
```

```
 73.   int i1;
 74.
 75.   m = A.m;
 76.   n = A.n;
 77.
 78.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
 79.     {
 80.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
 81.       exit(0);
 82.     }
 83.
 84.   for(i1 = 0; i1 < m*n; i1++)
 85.     elements[i1] = A.elements[i1];
 86. }
 87.
 88. Matrix::~Matrix()
 89. {
 90.   if(elements != NULL)
 91.     free(elements);
 92. }
 93.
 94.
 95. int Matrix::Index(int i, int j) const
 96. {
 97.   if(i < 0 || i >= m || j < 0 || j >= n)
 98.     {
 99.       fprintf(stderr, "Fehler in int Matrix::Index(...\n");
100.       exit(0);
101.     }
102.
103.  return i*n + j;
104. }
105.
106. double Matrix::Get(int i, int j) const
107. {
108.   return elements[Index(i, j)];
109. }
110.
111. void Matrix::Set(int i, int j, double a)
112. {
113.   elements[Index(i, j)] = a;
114. }
115.
116. void Matrix::Print() const
117. {
118.   int i1, i2;
119.
120.   for(i1 = 0; i1 < m; i1++)
121.     {
122.       printf("| ");
123.
124.       for(i2 = 0; i2 < n; i2++)
125.         {
126.           printf("%+.3e ", Get(i1, i2));
127.         }
128.
129.       printf("|\n");
130.     }
131. }
132.
133. // ********************
134.
135. // Reelle Zahl * Matrix, d.h. d * A.
136. const Matrix operator*(const double &d, const Matrix &A)
137. {
138.   int i1, i2;
```

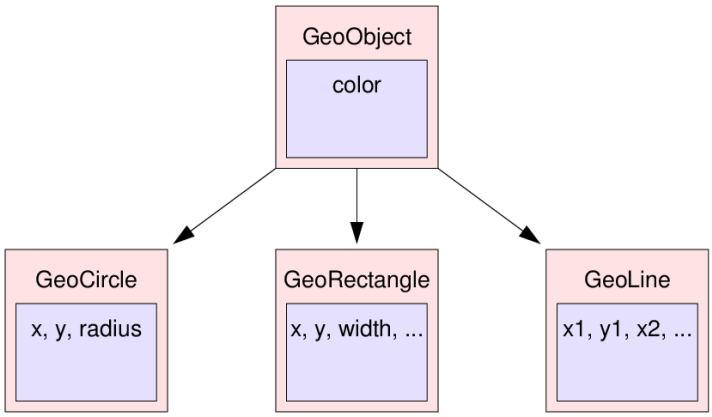
```
139.
140.   int m = A.Get_m();
141.   int n = A.Get_n();
142.
143.   Matrix B(m, n);
144.
145.   for(i1 = 0; i1 < m; i1++)
146.     {
147.       for(i2 = 0; i2 < n; i2++)
148.         B.Set(i1, i2, d * A.Get(i1, i2));
149.     }
150.
151.   return B;
152. }
153.
154. // Matrix * Matrix, d.h. A * B.
155. const Matrix operator*(const Matrix &A, const Matrix &B)
156. {
157.   double d1;
158.   int i1, i2, i3;
159.
160.   if(A.Get_n() != B.Get_m())
161.     {
162.       fprintf(stderr, "Fehler in const Matrix operator*(...\n");
163.       exit(0);
164.     }
165.
166.   int mA = A.Get_m();
167.   int nA = A.Get_n();
168.   int nB = B.Get_n();
169.
170.   Matrix C(mA, nB);
171.
172.   for(i1 = 0; i1 < mA; i1++)
173.     {
174.       for(i2 = 0; i2 < nB; i2++)
175.         {
176.           d1 = 0.0;
177.
178.           for(i3 = 0; i3 < nA; i3++)
179.             d1 += A.Get(i1, i3) * B.Get(i3, i2);
180.
181.           C.Set(i1, i2, d1);
182.         }
183.     }
184.
185.   return C;
186. }
187.
188. // ********************
189.
190. int main(void)
191. {
192.   Matrix A(2, 2);
193.   A.Set(0, 0, 1.0);
194.   A.Set(1, 1, 1.0);
195.   printf("A =\n");
196.   A.Print();
197.
198.   printf("********************\n");
199.
200.   Matrix B(5.0 * A);  // !!! Jetzt kann man einfach 5.0 * A schreiben. !!!
201.   printf("B =\n");
202.   B.Print();
203.
204.   printf("********************\n");
```

```
205.
206.    Matrix C(2, 1);
207.    C.Set(0, 0, 1.5);
208.    C.Set(1, 0, 0.5);
209.    printf("C =\n");
210.    C.Print();
211.
212.    printf("********************\n");
213.
214.    Matrix D(B * C);  // !!! Jetzt kann man einfach B * C schreiben. !!!
215.    printf("D =\n");
216.    D.Print();
217. }
```

```
A =
| +1.000e+00 +0.000e+00 |
| +0.000e+00 +1.000e+00 |
********************
B =
| +5.000e+00 +0.000e+00 |
| +0.000e+00 +5.000e+00 |
********************
C =
| +1.500e+00 |
| +5.000e-01 |
********************
D =
| +7.500e+00 |
| +2.500e+00 |
```

Vererbung

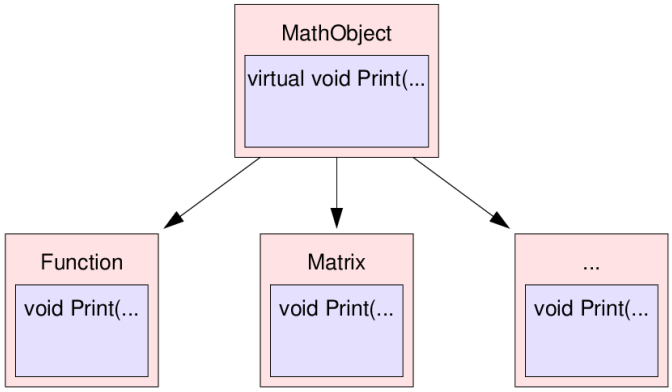
- **Grundidee der Vererbung (siehe auch erste Folie):** Klassen, die Gemeinsamkeiten aufweisen (z.B. gleiche Attribute oder Methoden oder zumindest Methodennamen), erben diese von einer gemeinsamen **Oberklasse**, in der diese Gemeinsamkeiten implementiert sind.
- **Beispiel:** Ein Zeichenprogramm speichert die gezeichneten Objekte (Kreise, Rechtecke, Linien) ... sie alle haben eine Farbe, das Attribut **`color`**, das in der gemeinsamen Oberklasse **`GeoObject`** definiert ist ... die Information über die Geometrie ist abhängig vom Objekt und damit in den **Unterklassen `GeoCircle`**, **`GeoRectangle`** und **`GeoLine`** enthalten ...




```
 1. class Color
 2. {
 3. public:
 4.
 5.   int r, g, b;
 6.
 7.   // ...
 8. };
 9.
10. class GeoObject
11. {
12. public:
13.
14.   Color color;
15.
16.   // ...
17. };
18.
19. class GeoCircle : public GeoObject
20. {
21. public:
22.
23.   double x, y, radius;
24.
25.   // ...
26. };
27.
28. class GeoRectangle : public GeoObject
29. {
30. public:
31.
32.   double x, y, width, height, angle;
33.
34.   // ...
35. };
```

```
36.
37. class GeoLine : public GeoObject
38. {
39. public:
40.
41.   double x1, y1, x2, y2;
42.
43.   // ...
44. };
45.
46. int main(void)
47. {
48.   GeoCircle circle;
49.
50.   // GeoCircle hat das Attribut color von GeoObject geerbt.
51.   circle.color.r = 255;
52.   circle.color.g = 0;
53.   circle.color.b = 0;
54.
55.   // Die Attribute x, y, z wurden wie gewohnt in GeoCircle definiert.
56.   circle.x = 1.5;
57.   circle.y = 2.5;
58.   circle.radius = 0.5;
59. }
```

- **Vorteile von Vererbung:**
  - Gemeinsamkeiten, werden nur einmal in der gemeinsamen Oberklasse implementiert, d.h. eine Code-Duplizierung ist nicht erforderlich.
  - Veränderungen in einer Oberklasse wirken sich sofort auf alle Unterklassen aus, d.h. der Code muss nur an einer zentralen Stelle modifiziert werden.
  - Polymorphismus (siehe unten).
- **Beispiel:**
  - Zurück zur Matrix-Klasse ...
  - ... Verallgemeinerung auf beliebige mathematische Objekte (Funktionen [Klasse **`Function`**], Matrizen [Klasse **`Matrix`**], ...) ...
  - ... alle können am Bildschirm ausgegeben werden, weshalb die Oberklasse **`MathObject`** eine Methode **`Print`** enthält ...
  - ... in den Unterklassen **`Function`** und **`Matrix`** wird diese Methode geeignet Überschrieben (was in der Oberklasse **`MathObject`** durch **`virtual`** gekennzeichnet wird) ...
  - ... hat man nun z.B. Zeiger auf mathematische Objekte (also vom Typ **`MathObject *`**), wie im Folgenden in der **`main`**-Funktion, und ruft darüber jeweils die Methode **`Print`** auf, werden entsprechend die in den Unterklassen **`Function`** oder **`Matrix`** implementierten Methoden **`Print`** ausgeführt (dies ist er eingangs erwähnte **Polymorphismus**) ...




```
#include<stdio.h>
#include<stdlib.h>

// ********************

// Definition einer abstrakten Klasse fuer beliebige mathematische Objekte
// (Funktionen, Matrizen, etc.).

class MathObject
{
public:  // Von aussen sichtbar.

  // Gibt das mathematische Objekt am Bildschirm aus.
  // Unterklassen, z.B. Function, Matrix, ... koennen Print ueberschreiben;
  // virtual zeigt an, dass bei Verwendung eines Zeigers vom Typ MathObject
  // auf ein Objekt einer Unterklasse von MathObject die ueberschriebenen
  // Versionen von Print verwendet werden (--> Polymorphismus); siehe das
  // Beispiel in der main-Funktion.
  virtual void Print() const { };
};

// ********************

// Definition einer Klasse fuer beliebige Funktionen.

// Function erbt von MathObject, ist also eine Unterklasse von MathObject.
class Function : public MathObject
{
public:  // Von aussen sichtbar.

  // Wertet die Funktion bei x aus.
  double Eval(double x) const;

  // Gibt die Funktion am Bildschirm aus.
  void Print() const;
};

double Function::Eval(double x) const
{
  // Momentan immer die 0-Funktion ... Klasse ist noch nicht vollstaendig
  // implementiert.
  return 0.0;
}

void Function::Print() const
{
  printf("function(x) = ... (Klasse ist noch nicht vollstaendig implementiert.)\n");
}

```

```
50. // ********************
51.
52. // Definition einer Klasse fuer Matrizen beliebiger Groesse.
53.
54. // Matrix erbt von MathObject, ist also eine Unterklasse von MathObject.
55. class Matrix : public MathObject
56. {
57. public:  // Von aussen sichtbar.
58.
59.   // Methoden.
60.
61.   // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
62.   // initialisiert die Attribute (eine m_ x n_ Matrix, allen Eintraege 0.0).
63.   Matrix(int m_, int n_);
64.
65.   // Konstruktor: Kopiert eine bestehende Matrix.
66.   Matrix(const Matrix &A);
67.
68.   // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
69.   // frei.
70.   ~Matrix();
71.
72.   // Liefert den Wert des (i,j)-ten Matrixelements.
73.   double Get(int i, int j) const;
74.
75.   // Liefert den Wert des (i,j)-ten Matrixelements.
76.   void Set(int i, int j, double a);
77.
78.   // Gibt die Matrix am Bildschirm aus.
79.   void Print() const;
80.
81.   int Get_m(void) const { return m; }
82.   int Get_n(void) const { return n; }
83.
84.   // *****
85.
86. private:  // Von aussen nicht sichtbar.
87.
88.   // Wandelt Zeilenindex i und Spaltenindex j in den Index des Arrays
89.   // elements um; eine Hilfsfunktion fuer Get und Set, daher private.
90.   int Index(int i, int j) const;
91.
92.   // Attribute.
93.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
94.   double *elements;  // Die Matrixeintraege.
95. };
96.
97. ...
98.
99. // ********************
100.
101. int main(void)
102. {
103.   int i1;
104.
105.   // *****
106.
107.   Matrix A(2, 2);
108.   A.Set(0, 0, 1.0);
109.   A.Set(1, 1, 2.0);
110.
111.   Matrix B(2, 1);
112.   B.Set(0, 0, 3.0);
113.   B.Set(1, 0, 4.0);
114.
115.   Function f;
```

```
116.
117.   // *****
118.
119.   MathObject *mo[3];
120.   mo[0] = &A;
121.   mo[1] = &f;
122.   mo[2] = &B;
123.
124.   // Das Array enthaelt jetzt drei mathematische Objekte, eine 2x2-Matrix,
125.   // eine Funktion, eine 2x1-Matrix.
126.
127.   for(i1 = 0; i1 < 3; i1++)
128.     // Beim Aufruf von Print werden aufgrund der virtual-Definition von Print
129.     // in MathObject die in den Unterklassen (Function, Matrix)
130.     // ueberschriebenen Versionen von Print verwendet (--> Polymorphismus).
131.     mo[i1]->Print();
132. }
```

```
| +1.000e+00 +0.000e+00 |
| +0.000e+00 +2.000e+00 |
function(x) = ... (Klasse ist noch nicht vollstaendig implementiert.)
| +3.000e+00 |
| +4.000e+00 |
```

- **Wiki → Virtuelle Methode:** Eine **virtuelle Methode** ist in der objektorientierten Programmierung eine Methode einer Klasse, deren Einsprungadresse erst zur Laufzeit ermittelt wird. Dieses sogenannte dynamische Binden ermöglicht es, Klassen von einer Oberklasse abzuleiten und dabei Funktionen zu überschreiben bzw. zu überladen. ...