Einführung in die Programmierung für Physiker

Die Programmiersprache C++ – Wesentliche Elemente

Marc Wagner

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

SeSe 2024

C++, objektorientierte Programmiersprachen

- **C** ist eine **imperative (= prozedurale) Programmiersprache**.
- **C++** ist eine **objektorientierte Programmiersprache**.
- **Wiki → Programmiersprache:**
  - **Imperative (= prozedurale) 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 ...

```
1. ...
2.
3. int main(void)
4. {
5.   MyClass cl;
6.
7.   cl.a = 5;
8.   printf("cl.a = %d\n", cl.a);
9.
10.  cl.Set_b(7);
11.  printf("cl.b = ");
12.  cl.Print_b();
13.  printf("\n");
14. }
```

```
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 ...

```
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:  // Alle Methoden und Attribute sind public ...
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.   // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
19.   // frei.
20.   ~Matrix();
21.
```

```
22.   // Liefert den Wert des (i,j)-ten Matrixelements.
23.   double Get(int i, int j);
24.
25.   // Gibt die Matrix am Bildschirm aus.
26.   void Print();
27.
28.   // *****
29.
30.   // Attribute.
31.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
32.   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)
```

```
1. ...
2.
3. int main(void)
4. {
5.   Matrix A(3, 2);
6.
7.   A.m = 6;
8.   A.n = 6;
9.   A.Print();
10. }
```

```
| +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.

```
1. ...
2.
3. // Definition einer Klasse fuer Matrizen beliebiger Groesse.
4.
5. class Matrix
6. {
7. public:  // Von aussen sichtbar.
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. private:  // Von aussen nicht sichtbar.
28.
29.   // Attribute.
30.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
31.   double *elements;  // Die Matrixeintraege.
32. };
33.
34. ...
35.
36. int main(void)
37. {
38.   Matrix A(3, 2);
39.
40.   A.m = 6;
41.   A.n = 6;
42.   A.Print();
43. }
```

```
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:** ...

```c
#include<stdio.h>

int main(void)
{
  int a = 3;
  int &b = a;  // b ist eine Referenz auf a.
  printf("a = %d , b = %d\n", a, b);

  a = a + 2;
  printf("a = %d , b = %d\n", a, b);

  b = b - 3;
  printf("a = %d , b = %d\n", a, b);
}
```

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

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

```c
#include<stdio.h>

void swap(int &a, int &b)  // Es werden Referenzen uebergeben, nicht nur Werte.
{
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void)
{
  int i1 = 3;
  int i2 = 7;

  printf("i1 = %d, i2 = %d\n", i1, i2);
  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
```

```
...

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 m_ Matrix, alle Eintraege 0.0).
  Matrix(int m_);

  // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
  // initialisiert die Attribute (eine m_ x n_ Matrix, alle 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();

```

```
38.   // *****
39.
40. private:  // Von aussen nicht sichtbar.
41.
42.   // Wandelt Zeilenindex i und Spaltenindex j in den Index des Arrays
43.   // elements um; eine Hilfsfunktion fuer Get und Set, daher private.
44.   int Index(int i, int j);
45.
46.   // Attribute.
47.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
48.   double *elements;  // Die Matrixeintraege.
49. };
50.
51. // Im Folgenden werden die in
52. //   class Matrix { ... }
53. // deklarierten Methoden definiert.
54.
55. Matrix::Matrix(int m_)
56. {
57.   int i1;
58.
59.   m = m_;
60.   n = m_;
61.
62.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
63.     {
64.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
65.       exit(0);
66.     }
67.
68.   for(i1 = 0; i1 < m*n; i1++)
69.     elements[i1] = 0.0;
70. }
71.
72. Matrix::Matrix(int m_, int n_)
73. {
74.   int i1;
75.
76.   m = m_;
77.   n = n_;
78.
79.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
80.     {
81.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
82.       exit(0);
83.     }
84.
85.   for(i1 = 0; i1 < m*n; i1++)
86.     elements[i1] = 0.0;
87. }
88.
89. Matrix::Matrix(const Matrix &A)
90. {
91.   int i1;
92.
93.   m = A.m;
94.   n = A.n;
95.
96.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
```

```
 97.     {
 98.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
 99.       exit(0);
100.     }
101.
102.   for(i1 = 0; i1 < m*n; i1++)
103.     elements[i1] = A.elements[i1];
104. }
105.
106. Matrix::~Matrix()
107. {
108.   if(elements != NULL)
109.     free(elements);
110. }
111.
112. int Matrix::Index(int i, int j)
113. {
114.   if(i < 0 || i >= m || j < 0 || j >= n)
115.     {
116.       fprintf(stderr, "Fehler in int Matrix::Index(...\n");
117.       exit(0);
118.     }
119.
120.  return i*n + j;
121. }
122.
123. double Matrix::Get(int i, int j)
124. {
125.   return elements[Index(i, j)];
126. }
127.
128. void Matrix::Set(int i, int j, double a)
129. {
130.   elements[Index(i, j)] = a;
131. }
132.
133. void Matrix::Print()
134. {
135.   int i1, i2;
136.
137.   for(i1 = 0; i1 < m; i1++)
138.     {
139.       printf("| ");
140.
141.       for(i2 = 0; i2 < n; i2++)
142.         {
143.           printf("%+.3e ", Get(i1, i2));
144.         }
145.
146.       printf("|\n");
147.     }
148. }
149.
150. // ********************
151.
152. int main(void)
153. {
154.   Matrix A(2);  // Matrix A(2, 2); wuerde ebenfalls funktionieren.
155.   A.Set(0, 0, 1.0);
```

```
156.   A.Set(1, 1, 1.0);
157.   printf("A =\n");
158.   A.Print();
159.
160.   printf("********************\n");
161.
162.   Matrix B(A);
163.   printf("B =\n");
164.   B.Print();
165.
166.   printf("********************\n");
167.
168.   A.Set(0, 0, 2.0);
169.   A.Set(1, 1, 3.0);
170.   printf("A =\n");
171.   A.Print();
172.
173.   printf("********************\n");
174.
175.   printf("B =\n");
176.   B.Print();
177. }
```

```
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 m_ Matrix, alle Eintraege 0.0).
16.   Matrix(int m_);
17.
18.   // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
19.   // initialisiert die Attribute (eine m_ x n_ Matrix, alle Eintraege 0.0).
```

```
20.   Matrix(int m_, int n_);
21.
22.   // Konstruktor: Kopiert eine bestehende Matrix.
23.   Matrix(const Matrix &A);
24.
25.   // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
26.   // frei.
27.   ~Matrix();
28.
29.   // Liefert den Wert des (i,j)-ten Matrixelements.
30.   double Get(int i, int j);
31.
32.   // Liefert den Wert des (i,j)-ten Matrixelements.
33.   void Set(int i, int j, double a);
34.
35.   // Gibt die Matrix am Bildschirm aus.
36.   void Print();
37.
38.   int Get_m(void) { return m; }
39.   int Get_n(void) { return n; }
40.
41.   // *****
42.
43. private:  // Von aussen nicht sichtbar.
44.
45.   // Wandelt Zeilenindex i und Spaltenindex j in den Index des Arrays
46.   // elements um; eine Hilfsfunktion fuer Get und Set, daher private.
47.   int Index(int i, int j);
48.
49.   // Attribute.
50.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
51.   double *elements;  // Die Matrixeintraege.
52. };
53.
54. // Im Folgenden werden die in
55. //   class Matrix { ... }
56. // deklarierten Methoden definiert.
57.
58. Matrix::Matrix(int m_)
59. {
60.   int i1;
61.
62.   m = m_;
63.   n = m_;
64.
65.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
66.     {
67.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
68.       exit(0);
69.     }
70.
71.   for(i1 = 0; i1 < m*n; i1++)
72.     elements[i1] = 0.0;
73. }
74.
75. Matrix::Matrix(int m_, int n_)
76. {
77.   int i1;
78.
```

```
 79.   m = m_;
 80.   n = n_;
 81.
 82.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
 83.     {
 84.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
 85.       exit(0);
 86.     }
 87.
 88.   for(i1 = 0; i1 < m*n; i1++)
 89.     elements[i1] = 0.0;
 90. }
 91.
 92. Matrix::Matrix(const Matrix &A)
 93. {
 94.   int i1;
 95.
 96.   m = A.m;
 97.   n = A.n;
 98.
 99.   if((elements = (double *)malloc(m*n*sizeof(double))) == NULL)
100.     {
101.       fprintf(stderr, "Fehler in Matrix::Matrix(...\n");
102.       exit(0);
103.     }
104.
105.   for(i1 = 0; i1 < m*n; i1++)
106.     elements[i1] = A.elements[i1];
107. }
108.
109. Matrix::~Matrix()
110. {
111.   if(elements != NULL)
112.     free(elements);
113. }
114.
115. int Matrix::Index(int i, int j)
116. {
117.   if(i < 0 || i >= m || j < 0 || j >= n)
118.     {
119.       fprintf(stderr, "Fehler in int Matrix::Index(...\n");
120.       exit(0);
121.     }
122.
123.  return i*n + j;
124. }
125.
126. double Matrix::Get(int i, int j)
127. {
128.   return elements[Index(i, j)];
129. }
130.
131. void Matrix::Set(int i, int j, double a)
132. {
133.   elements[Index(i, j)] = a;
134. }
135.
136. void Matrix::Print()
137. {
```

```
138.   int i1, i2;
139.
140.   for(i1 = 0; i1 < m; i1++)
141.     {
142.       printf("| ");
143.
144.       for(i2 = 0; i2 < n; i2++)
145.         {
146.           printf("%+.3e ", Get(i1, i2));
147.         }
148.
149.       printf("|\n");
150.     }
151. }
152.
153. // ********************
154.
155. // Reelle Zahl * Matrix, d.h. d * A.
156. const Matrix operator*(const double &d, Matrix &A)
157. {
158.   int i1, i2;
159.
160.   int m = A.Get_m();
161.   int n = A.Get_n();
162.
163.   Matrix B(m, n);
164.
165.   for(i1 = 0; i1 < m; i1++)
166.     {
167.       for(i2 = 0; i2 < n; i2++)
168.         B.Set(i1, i2, d * A.Get(i1, i2));
169.     }
170.
171.   return B;
172. }
173.
174. // Matrix * Matrix, d.h. A * B.
175. const Matrix operator*(Matrix &A, Matrix &B)
176. {
177.   double d1;
178.   int i1, i2, i3;
179.
180.   if(A.Get_n() != B.Get_m())
181.     {
182.       fprintf(stderr, "Fehler in const Matrix operator*(...\n");
183.       exit(0);
184.     }
185.
186.   int mA = A.Get_m();
187.   int nA = A.Get_n();
188.   int nB = B.Get_n();
189.
190.   Matrix C(mA, nB);
191.
192.   for(i1 = 0; i1 < mA; i1++)
193.     {
194.       for(i2 = 0; i2 < nB; i2++)
195.               {
196.                 d1 = 0.0;
```

```
197.
198.           for(i3 = 0; i3 < nA; i3++)
199.                   d1 += A.Get(i1, i3) * B.Get(i3, i2);
200.
201.                 C.Set(i1, i2, d1);
202.               }
203.     }
204.
205.   return C;
206. }
207.
208. // ********************
209.
210. int main(void)
211. {
212.   Matrix A(2);
213.   A.Set(0, 0, 1.0);
214.   A.Set(1, 1, 1.0);
215.   printf("A =\n");
216.   A.Print();
217.
218.   printf("********************\n");
219.
220.   Matrix B(5.0 * A);  // !!! Jetzt kann man einfach 5.0 * A schreiben. !!!
221.   printf("B =\n");
222.   B.Print();
223.
224.   printf("********************\n");
225.
226.   Matrix C(2, 1);
227.   C.Set(0, 0, 1.5);
228.   C.Set(1, 0, 0.5);
229.   printf("C =\n");
230.   C.Print();
231.
232.   printf("********************\n");
233.
234.   Matrix D(B * C);  // !!! Jetzt kann man einfach B * C schreiben. !!!
235.   printf("D =\n");
236.   D.Print();
237. }
```

```
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 |
```

const-Methoden (const member functions)

- **const-Methoden** dürfen keine Attribute der eigenen Klasse verändern. Außerdem können sie ausschließlich const-Methoden der eigenen Klasse aufrufen.
- Eine const Methode wird durch Anhängen des Schlüsselwortes **`const`** an die Parameterliste definiert.
  `type member_function_name``(`*`type1 para1, type2 para2, ...`*`)` **`const {...}`**.
- Ein konstantes Objekt darf ausschließlich const-Methoden der eigenen Klasse aufrufen, da Methoden, die nicht als const definiert sind, die Attribute dieses Objekts verändern könnten, was wiederum im Widerspruch zur "Konstanz" des Objekts steht.
- Da die Verwendung konstanter Objekte in vielen Fällen angebracht oder empfehlenswert ist, sollten Methoden wenn möglich (d.h. wenn sie keine Attribute der eigenen Klasse verändern) als const-Methoden definiert werden.
- **Beispiel:**

```
1. #include<stdio.h>
2.
3. class MyClass
4. {
5. public:
6.   double d;
7.   MyClass(double d_) { d = d_; };
8.   void Print() const { printf("%f\n", d); };
9. };
10.
11. int main(void)
12. {
13.   const MyClass cl(123.0);
14.   cl.Print();
15. }
```

```
123.000000
```

- Oder äquivalent bei Definition der const Methode **`Print`** außerhalb der Klasse **`MyClass`**:

```
1. #include<stdio.h>
2.
3. class MyClass
4. {
5. public:
6.   double d;
7.   MyClass(double d_) { d = d_; };
8.   void Print() const;
9. };
10.
11. void MyClass::Print() const
12. {
13.   printf("%f\n", d);
14. }
15.
16. int main(void)
17. {
18.   const MyClass cl(123.0);
19.   cl.Print();
20. }
```

- Wird die Methode **`Print`** nicht als const-Methode definiert, aber vom konstanten Objekt **`cl`** aufgerufen, liefert der Compiler einen Fehler.

```
 1. #include<stdio.h>
 2.
 3. class MyClass
 4. {
 5. public:
 6.   double d;
 7.   MyClass(double d_) { d = d_; };
 8.   void Print();
 9. };
10.
11. void MyClass::Print()
12. {
13.   printf("%f\n", d);
14. }
15.
16. int main(void)
17. {
18.   const MyClass cl(123.0);
19.   cl.Print();
20. }
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o tmp tmp.C
tmp.C: In function ‘int main()’:
tmp.C:19:11: error: passing ‘const MyClass’ as ‘this’ argument discards qualifiers [-fpermissive]
   19 |   cl.Print();
      |   ~~~~~~~~^~
tmp.C:11:6: note:   in call to ‘void MyClass::Print()’
   11 | void MyClass::Print()
      |      ^~~~~~~
```

- **Beispiel:** Zurück zur Matrix-Klasse (genau wie auf vorherigem Foliensatz) ... allerdings mit marginal veränderter **`main`**-Funktion (Definition des **`Matrix`**-Objekts **`B`** als konstant) ...

```
219. ...
220.   const Matrix B(5.0 * A);  // !!! Jetzt kann man einfach 5.0 * A schreiben. !!!
221.   printf("B =\n");
222.   B.Print();
223.
224.   printf("********************\n");
225.
226.   Matrix C(2, 1);
227.   C.Set(0, 0, 1.5);
228.   C.Set(1, 0, 0.5);
229.   printf("C =\n");
230.   C.Print();
231.
232.   printf("********************\n");
233.
234.   Matrix D(B * C);  // !!! Jetzt kann man einfach B * C schreiben. !!!
235. ...
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o tmp tmp.C
tmp.C: In function ‘int main()’:
tmp.C:222:10: error: passing ‘const Matrix’ as ‘this’ argument discards qualifiers [-fpermissive]
  222 |   B.Print();
      |   ~~~~~~~^~
tmp.C:136:6: note:   in call to ‘void Matrix::Print()’
  136 | void Matrix::Print()
      |      ^~~~~~
tmp.C:234:12: error: binding reference of type ‘Matrix&’ to ‘const Matrix’ discards qualifiers
  234 |   Matrix D(B * C);  // !!! Jetzt kann man einfach B * C schreiben. !!!
      |           ^
tmp.C:175:32: note:   initializing argument 1 of ‘const Matrix operator*(Matrix&, Matrix&)’
  175 | const Matrix operator*(Matrix &A, Matrix &B)
      |                        ~~~~~~~~^
```

- **Problem #1: Das konstante `Matrix`-Objekt `B` ruft die nicht-konstante Methode `Print` auf.**
- Problem #2: Das konstante `Matrix`-Objekt `B` wird mit dem überladenen Multiplikationsoperator verwendet, der für nicht-konstante `Matrix`-Objekte definiert wurde. (dieses Problem hat nichts mit const-Methoden zu tun, ist aber dennoch eng verwandt)
- Beide Probleme können leicht durch Verwendung von `const`-Schlüsselwörtern in der entsprechenden Methodendefinition bzw. Operatorüberladung beseitigt werden.

```
34. ...
35.   // Gibt die Matrix am Bildschirm aus.
36.   void Print() const;
37. ...
```

```
135. ...
136. void Matrix::Print() const
137. {
138. ...
```

```
173. ...
174. // Matrix * Matrix, d.h. A * B.
175. const Matrix operator*(const Matrix &A, const Matrix &B)
176. {
177. ...
```

- **Dies führt allerdings zu weiteren Problemen, da von dort aus "Get-Methoden" aufgerufen werden, die nicht als const-Methoden definiert wurden.**

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o tmp tmp.C
tmp.C: In member function ‘void Matrix::Print() const’:
tmp.C:146:31: error: passing ‘const Matrix’ as ‘this’ argument discards qualifiers [-fpermissive]
  146 |           printf("%+.3e ", Get(i1, i2));
      |                             ~~~^~~~~~~~
tmp.C:126:8: note:   in call to ‘double Matrix::Get(int, int)’
  126 | double Matrix::Get(int i, int j)
      |        ^~~~~~
tmp.C: In function ‘const Matrix operator*(const Matrix&, const Matrix&)’:
tmp.C:180:13: error: passing ‘const Matrix’ as ‘this’ argument discards qualifiers [-fpermissive]
  180 |   if(A.Get_n() != B.Get_m())
      |      ~~~~~~~^~
tmp.C:39:7: note:   in call to ‘int Matrix::Get_n()’
   39 |   int Get_n(void) { return n; }
      |       ^~~~~
tmp.C:180:26: error: passing ‘const Matrix’ as ‘this’ argument discards qualifiers [-fpermissive]
  180 |   if(A.Get_n() != B.Get_m())
      |                   ~~~~~~~^~
tmp.C:38:7: note:   in call to ‘int Matrix::Get_m()’
   38 |   int Get_m(void) { return m; }
      |       ^~~~~
tmp.C:186:19: error: passing ‘const Matrix’ as ‘this’ argument discards qualifiers [-fpermissive]
  186 |   int mA = A.Get_m();
      |            ~~~~~~~^~
tmp.C:38:7: note:   in call to ‘int Matrix::Get_m()’
   38 |   int Get_m(void) { return m; }
      |       ^~~~~
tmp.C:187:19: error: passing ‘const Matrix’ as ‘this’ argument discards qualifiers [-fpermissive]
  187 |   int nA = A.Get_n();
      |            ~~~~~~~^~
tmp.C:39:7: note:   in call to ‘int Matrix::Get_n()’
   39 |   int Get_n(void) { return n; }
      |       ^~~~~
tmp.C:188:19: error: passing ‘const Matrix’ as ‘this’ argument discards qualifiers [-fpermissive]
  188 |   int nB = B.Get_n();
      |            ~~~~~~~^~
tmp.C:39:7: note:   in call to ‘int Matrix::Get_n()’
   39 |   int Get_n(void) { return n; }
      |       ^~~~~
tmp.C:199:24: error: passing ‘const Matrix’ as ‘this’ argument discards qualifiers [-fpermissive]
  199 |             d1 += A.Get(i1, i3) * B.Get(i3, i2);
      |                    ~~~~~^~~~~~~
tmp.C:126:8: note:   in call to ‘double Matrix::Get(int, int)’
  126 | double Matrix::Get(int i, int j)
      |        ^~~~~~
tmp.C:199:40: error: passing ‘const Matrix’ as ‘this’ argument discards qualifiers [-fpermissive]
  199 |             d1 += A.Get(i1, i3) * B.Get(i3, i2);
      |                                    ~~~~~^~~~~~~
tmp.C:126:8: note:   in call to ‘double Matrix::Get(int, int)’
  126 | double Matrix::Get(int i, int j)
      |        ^~~~~~
```

- Nach konsequentem Definieren all derjenigen Methoden, die keine Attribute der **Matrix**-Klasse verändern, als const-Methoden, erhält man wieder einen einwandfrei kompilierbaren Code. Dieser ist nun leistungsstärker/universeller, da er sowohl für konstante als auch nicht-konstante **Matrix**-Objekte verwendet werden kann.

```
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 m_ Matrix, alle Eintraege 0.0).
16.   Matrix(int m_);
17.
18.   // Konstruktor: Reserviert entsprechend Speicher fuer die Matrixelemente,
19.   // initialisiert die Attribute (eine m_ x n_ Matrix, alle Eintraege 0.0).
20.   Matrix(int m_, int n_);
21.
22.   // Konstruktor: Kopiert eine bestehende Matrix.
23.   Matrix(const Matrix &A);
24.
25.   // Destruktor: Gibt den fuer die Matrixelemente reservierten Speicher wieder
26.   // frei.
27.   ~Matrix();
28.
29.   // Liefert den Wert des (i,j)-ten Matrixelements.
30.   double Get(int i, int j) const;  // !!!!! const Methode !!!!!
31.
32.   // Liefert den Wert des (i,j)-ten Matrixelements.
33.   void Set(int i, int j, double a);
34.
35.   // Gibt die Matrix am Bildschirm aus.
36.   void Print() const;  // !!!!! const Methode !!!!!
37.
38.   int Get_m(void) const { return m; }  // !!!!! const Methode !!!!!
39.   int Get_n(void) const { return n; }  // !!!!! const Methode !!!!!
40.
41.   // *****
42.
43. private:  // Von aussen nicht sichtbar.
44.
45.   // Wandelt Zeilenindex i und Spaltenindex j in den Index des Arrays
46.   // elements um; eine Hilfsfunktion fuer Get und Set, daher private.
47.   int Index(int i, int j) const;
48.
49.   // Attribute.
50.   int m, n;  // Eine Matrix mit m Zeilen und n Spalten.
51.   double *elements;  // Die Matrixeintraege.
52. };
53.
54. // Im Folgenden werden die in
```

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

```
114.
115. int Matrix::Index(int i, int j) const  // !!!!! const Methode !!!!!
116. {
117.   if(i < 0 || i >= m || j < 0 || j >= n)
118.     {
119.       fprintf(stderr, "Fehler in int Matrix::Index(...\n");
120.       exit(0);
121.     }
122.
123.  return i*n + j;
124. }
125.
126. double Matrix::Get(int i, int j) const  // !!!!! const Methode !!!!!
127. {
128.   return elements[Index(i, j)];
129. }
130.
131. void Matrix::Set(int i, int j, double a)
132. {
133.   elements[Index(i, j)] = a;
134. }
135.
136. void Matrix::Print() const  // !!!!! const Methode !!!!!
137. {
138.   int i1, i2;
139.
140.   for(i1 = 0; i1 < m; i1++)
141.     {
142.       printf("| ");
143.
144.       for(i2 = 0; i2 < n; i2++)
145.         {
146.           printf("%+.3e ", Get(i1, i2));
147.         }
148.
149.       printf("|\n");
150.     }
151. }
152.
153. // ********************
154.
155. // Reelle Zahl * Matrix, d.h. d * A.
156. const Matrix operator*(const double &d, const Matrix &A)  // !!!!! Parameter als const definiert !!!!!
157. {
158.   int i1, i2;
159.
160.   int m = A.Get_m();
161.   int n = A.Get_n();
162.
163.   Matrix B(m, n);
164.
165.   for(i1 = 0; i1 < m; i1++)
166.     {
167.       for(i2 = 0; i2 < n; i2++)
168.         B.Set(i1, i2, d * A.Get(i1, i2));
169.     }
170.
171.   return B;
172. }
```

```
173.
174. // Matrix * Matrix, d.h. A * B.
175. const Matrix operator*(const Matrix &A, const Matrix &B)  // !!!!! Parameter als const definiert !!!!!
176. {
177.   double d1;
178.   int i1, i2, i3;
179.
180.   if(A.Get_n() != B.Get_m())
181.     {
182.       fprintf(stderr, "Fehler in const Matrix operator*(...\n");
183.       exit(0);
184.     }
185.
186.   int mA = A.Get_m();
187.   int nA = A.Get_n();
188.   int nB = B.Get_n();
189.
190.   Matrix C(mA, nB);
191.
192.   for(i1 = 0; i1 < mA; i1++)
193.     {
194.       for(i2 = 0; i2 < nB; i2++)
195.         {
196.           d1 = 0.0;
197.
198.           for(i3 = 0; i3 < nA; i3++)
199.             d1 += A.Get(i1, i3) * B.Get(i3, i2);
200.
201.           C.Set(i1, i2, d1);
202.         }
203.     }
204.
205.   return C;
206. }
207.
208. // ********************
209.
210. int main(void)
211. {
212.   Matrix A(2);
213.   A.Set(0, 0, 1.0);
214.   A.Set(1, 1, 1.0);
215.   printf("A =\n");
216.   A.Print();
217.
218.   printf("********************\n");
219.
220.   const Matrix B(5.0 * A);  // !!! Jetzt kann man einfach 5.0 * A schreiben. !!!
221.   printf("B =\n");
222.   B.Print();
223.
224.   printf("********************\n");
225.
226.   Matrix C(2, 1);
227.   C.Set(0, 0, 1.5);
228.   C.Set(1, 0, 0.5);
229.   printf("C =\n");
230.   C.Print();
231.
```

```
232.   printf("********************\n");
233.
234.   Matrix D(B * C);  // !!! Jetzt kann man einfach B * C schreiben. !!!
235.   printf("D =\n");
236.   D.Print();
237. }
```

```
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 |
```

Templates

- **Templates** (Schablonen, Vorlagen) erlauben in **C++**, auch Datentypen als Parameter für Funktionen oder Klassen zu verwenden.

Funktionentemplates

- Das Überladen der Funktion **`max`** in Abschnitt "Überladen von Funktionen/Methoden und Operatoren", d.h. die Implementierung der nahezu identischen Funktionen
  - **`int max(int a, int b)`**
  - **`float max(float a, float b)`**
  - **`double max(double a, double b)`**
  - ...

ist zeitraubend und führt zu langen unübersichtlichen Programmcodes.

- Eleganter und praktischer ist die Verwendung bzw. Implementierung eines **Funktionentemplates**.
- **Beispiel:** Funktionentemplates für die Funktionen **`max`** und **`swap`**.

```
1. #include<stdio.h>
2.
3. template<typename T> T max(T a, T b)
4. {
5.   if(a > b)
6.     return a;
7.
8.   return b;
9. }
10.
11. template<typename T> void swap(T &a, T &b)
12. {
13.   T tmp = a;
14.   a = b;
15.   b = tmp;
16. }
17.
18. class Vector_2d
19. {
20. public:
21.
22.   Vector_2d(double x_, double y_)
23.   {
24.     x = x_;
25.     y = y_;
26.   }
27.
28.   void Print()
29.   {
30.     printf("(%+.3f , %+.3f)", x, y);
31.   }
32.
33.   double x;
34.   double y;
35. };
36.
```

```
37. int main(void)
38. {
39.   // Verwendung von max.
40.
41.   printf("%d\n", max(1, 4));  // Der Compiler ersetzt T durch int.
42.   printf("%f\n", max(2.0, 3.0));  // Der Compiler ersetzt T durch double.
43.
44.   // Verwendung von swap.
45.
46.   double x = 2.0;
47.   double y = 3.5;
48.   printf("x = %f, y = %f\n", x, y);
49.   swap(x, y);  // Der Compiler ersetzt T durch double.
50.   printf("x = %f, y = %f\n", x, y);
51.
52.   Vector_2d v1(1.0, 2.0);
53.   Vector_2d v2(3.0, 4.0);
54.   printf("v1 = ");
55.   v1.Print();
56.   printf(", v2 = ");
57.   v2.Print();
58.   printf("\n");
59.   swap(v1, v2);  // Der Compiler ersetzt T durch Vector_2d.
60.   printf("v1 = ");
61.   v1.Print();
62.   printf(", v2 = ");
63.   v2.Print();
64.   printf("\n");
65. }
```

```
4
3.000000
x = 2.000000, y = 3.500000
x = 3.500000, y = 2.000000
v1 = (+1.000 , +2.000), v2 = (+3.000 , +4.000)
v1 = (+3.000 , +4.000), v2 = (+1.000 , +2.000)
```

- Allgemein hat ein Funktionentemplate folgende Form: **`template<typename`** *`type_name`*`>` *`type0 function_name`*`(`*`type1 para1, type2 para2, ...`*`){ ... }`.
  - *type_name* steht für einen beliebigen Datentyp, kann also je nach Funktionsaufruf z.B. **`int`**, **`double`**, **`char **`**, ... sein.
  - Mindestens einer der Datentypen *type1*, *type2*, ... muss *type_name* sein, da der Kompiler beim Funktionsaufruf feststellen muss, um welchen Datentyp es sich bei *type_name* handelt.

Klassentemplates

- Gleiches Prinzip wie bei Funktionentemplates: Eine Klasse, die für verschiedene Datentypen benötigt wird oder eingesetzt werden kann, muss bei Verwenung eines **Klassentemplates** nur ein Mal implementiert werden.
- **Beispiel:** Klassentemplate für eine Klasse für 2-komponentige Vektoren; der Datentyp der Komponenten ist dabei nicht festgelegt, kann also z.B. **`float`**, **`double`**, **`int`**, ... sein.

```
1. #include<math.h>
2. #include<stdio.h>
3.
4. template<typename T> class Vector_2d
5. {
```

```
6. public:
7.
8.   Vector_2d(T x_, T y_)
9.   {
10.    x = x_;
11.    y = y_;
12.  }
13.
14.  T norm()
15.  {
16.    return sqrt(x*x + y*y);
17.  }
18.
19.  void Print();
20.
21.  T x;
22.  T y;
23. };
24.
25. template<typename T> void Vector_2d<T>::Print()
26. {
27.   printf("(%+.3f , %+.3f)", (double)x, (double)y);
28. }
29.
30. int main(void)
31. {
32.   Vector_2d<float> v1(1.0, 2.0);  // Der Compiler legt v1 als float-Vektor an.
33.   printf("v1 = ");
34.   v1.Print();
35.   printf("\n");
36.   printf("norm(v1) = %+.5f\n", v1.norm());
37.   printf("norm(v1) = %+.10f\n", v1.norm());
38.   printf("sizeof(v1) = %zu\n", sizeof(v1));
39.
40.   Vector_2d<double> v2(1.0, 2.0);  // Der Compiler legt v2 als double-Vektor an.
41.   printf("v2 = ");
42.   v2.Print();
43.   printf("\n");
44.   printf("norm(v2) = %+.5f\n", v2.norm());
45.   printf("norm(v2) = %+.10f\n", v2.norm());
46.   printf("sizeof(v2) = %zu\n", sizeof(v2));
47. }
```

```
v1 = (+1.000 , +2.000)
norm(v1) = +2.23607
norm(v1) = +2.2360680103
sizeof(v1) = 8
v2 = (+1.000 , +2.000)
norm(v2) = +2.23607
norm(v2) = +2.2360679775
sizeof(v2) = 16
```

- Allgemein hat ein Klassentemplate folgende Form:
  `template<typename` *`type_name`*`>` **`class`** *`class_name`* **`{ ... };`**.
  - *`type_name`* steht für einen beliebigen Datentyp, kann also je nach Objektdefinition z.B. **`int`**, **`double`**, **`char **`**, ... sein.
- Ein entsprechendes Objekt kann wie folgt definiert werden:
  `class_name<type_name> object_name;`
  bzw.
  `class_name<type_name> object_name(...);`.

Beispiel für ein häufig verwendetes Klassentemplate: `vector<...>`

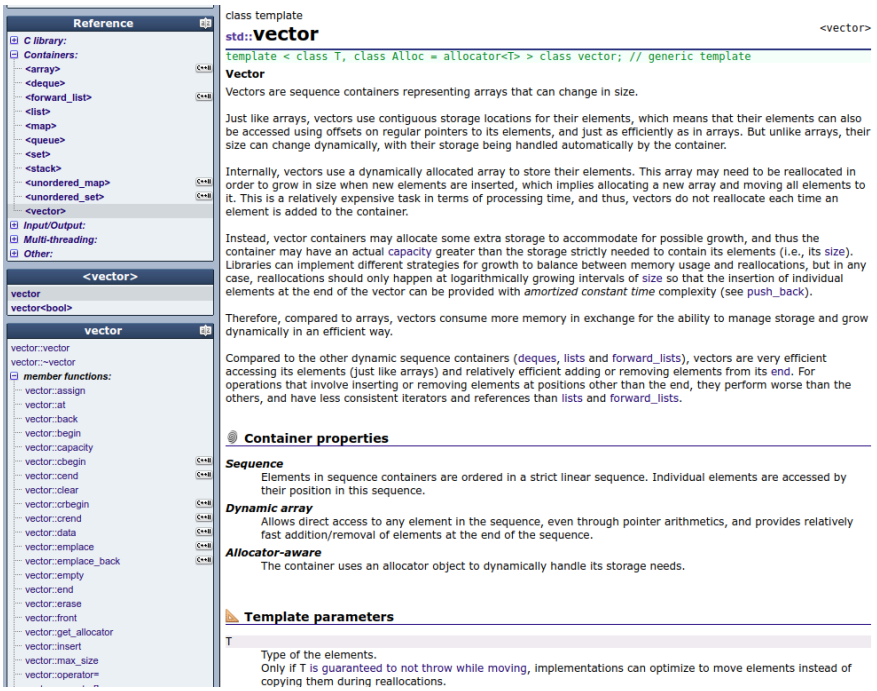
- Das Klassentemplate **`vector<...>`** stellt eine Art Array für nahezu beliebige Datentypen zur Verfügung, dessen Größe während der Programmlaufzeit verändert werden kann. Die Handhabung ist praktischer, übersichtlicher und weniger fehleranfällig als z.B. dynamisches Anfordern von Speicherplatz für Arrays mit **`malloc(...)`**. Das Klassentemplate **`vector<...>`** ist in der **Standard Template Library** enthalten.
- **Wiki → Standard Template Library:**
  The Standard Template Library (STL) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called algorithms, containers, functions, and iterators.
  The STL provides a set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). ...
  The STL achieves its results through the use of templates. ...
- **Beispiel:** Anlegen, Vergrößern und Verkleinern eines **`vector<int>`**.

```
#include<stdio.h>
#include<vector>  // Einbinden des Klassentemplates vector<...>.

using namespace std;  // Erlaubt die Verwendung von vector<...> an Stelle von std::vector<...>.

void print_int_vector(vector<int> &iv)
// void print_int_vector(std::vector<int> &iv)  // Falls ohne "using namespace std;".
{
  int i1;

  for(i1 = 0; i1 < iv.size(); i1++)  // Die member function size() liefert die Größe eines Vektors.
  {
    printf("%d", iv[i1]);

    if(i1 < iv.size() - 1)
      printf(", ");
    else
      printf("\n");
  }
}

int main(void)
{
  vector<int> A(3);  // Anlegen eines int-vectors mit 3 Elementen.
  // std::vector<int> A(3);  // Falls ohne "using namespace std;".
  A[0] = 1;  // Zugriff auf die Elemente eines Vektors mit [...] möglich (wie bei "normalem Array").
  A[1] = 2;
  A[2] = 5;
  print_int_vector(A);

  A.resize(5);  // Vergrößern des int-vectors auf 5 Elemente.
  A[4] = 9;
  print_int_vector(A);

  A.resize(2);  // Verkleinern des int-vectors auf 2 Elemente.
  print_int_vector(A);
```

```
37. }
```

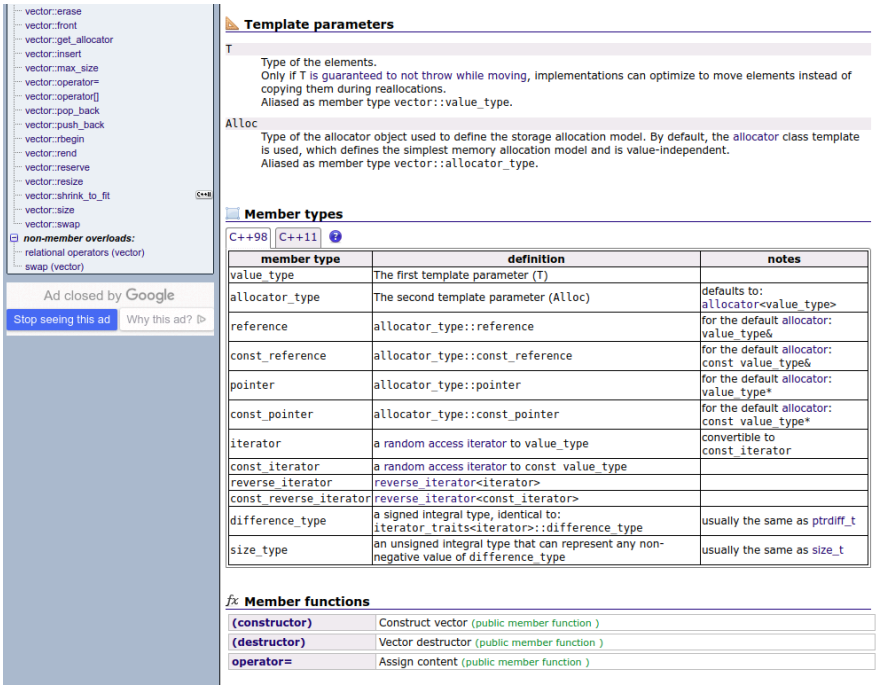
```
1, 2, 5
1, 2, 5, 0, 9
1, 2
```

- Es gibt zahlreiche Member-Funktionen und andere Sprachelemente, die einfaches und elegantes Arbeiten mit **`vector<...>`** ermöglichen. Siehe hierzu die Dokumentation von **C++** (z.B. **`http://www.cplusplus.com/reference/vector/vector/`**) oder entsprechende Lehrbücher.



Reference
C library:
Containers:
<array>
<deque>
<forward_list>
<list>
<map>
<queue>
<set>
<stack>
<unordered_map>
<unordered_set>
<vector>
Input/Output:
Multi-threading:
Other:
<vector>
vector
vector<bool>
vector
vector::vector
vector::~vector
member functions:
vector::assign
vector::at
vector::back
vector::begin
vector::capacity
vector::cbegin
vector::cend
vector::clear
vector::crbegin
vector::crend
vector::data
vector::emplace
vector::emplace_back
vector::empty
vector::end
vector::erase
vector::front
vector::get_allocator
vector::insert
vector::max_size
vector::operator=

class template
std::vector
<vector>
template < class T, class Alloc = allocator<T> > class vector; // generic template
Vector
Vectors are sequence containers representing arrays that can change in size.
Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.
Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see push_back).
Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.
Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists.
Container properties
Sequence
Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.
Dynamic array
Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.
Allocator-aware
The container uses an allocator object to dynamically handle its storage needs.
Template parameters
T
Type of the elements.
Only if T is guaranteed to not throw while moving, implementations can optimize to move elements instead of copying them during reallocations.



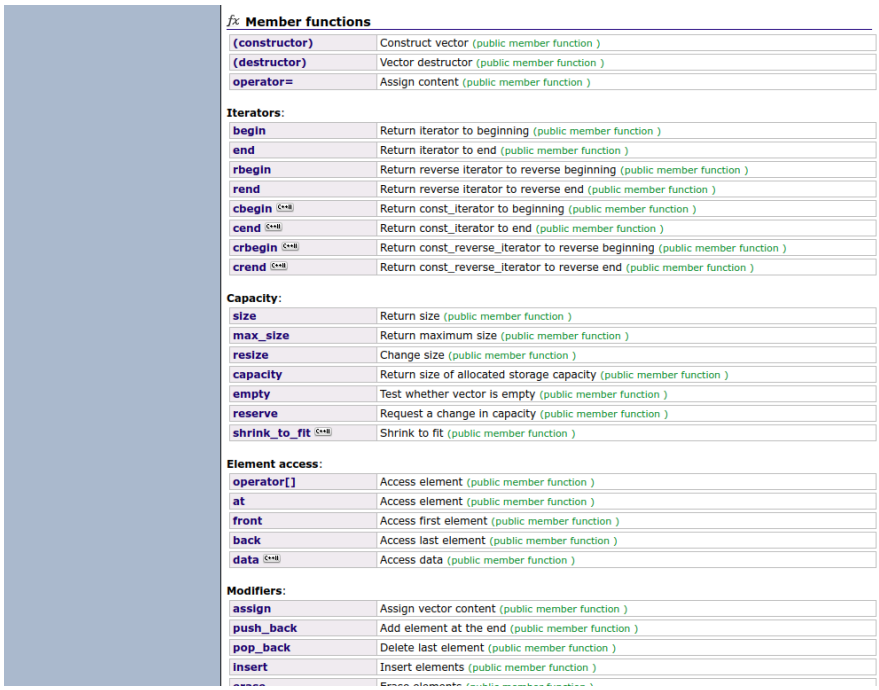
vector::erase
vector::front
vector::get_allocator
vector::insert
vector::max_size
vector::operator=
vector::operator[]
vector::pop_back
vector::push_back
vector::rbegin
vector::rend
vector::reserve
vector::resize
vector::shrink_to_fit
vector::size
vector::swap
non-member overloads:
relational operators (vector)
swap (vector)
Ad closed by Google
Stop seeing this ad
Why this ad?

Template parameters
T
Type of the elements.
Only if T is guaranteed to not throw while moving, implementations can optimize to move elements instead of copying them during reallocations.
Aliased as member type vector::value_type.
Alloc
Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent.
Aliased as member type vector::allocator_type.
Member types
C++98 C++11

member type	definition	notes
value_type	The first template parameter (T)	
allocator_type	The second template parameter (Alloc)	defaults to: allocator<value_type>
reference	allocator_type::reference	for the default allocator: value_type&
const_reference	allocator_type::const_reference	for the default allocator: const value_type&
pointer	allocator_type::pointer	for the default allocator: value_type*
const_pointer	allocator_type::const_pointer	for the default allocator: const value_type*
iterator	a random access iterator to value_type	convertible to const_iterator
const_iterator	a random access iterator to const value_type	
reverse_iterator	reverse_iterator<iterator>	
const_reverse_iterator	reverse_iterator<const_iterator>	
difference_type	a signed integral type, identical to: iterator_traits<iterator>::difference_type	usually the same as ptrdiff_t
size_type	an unsigned integral type that can represent any non-negative value of difference_type	usually the same as size_t

Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)



Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

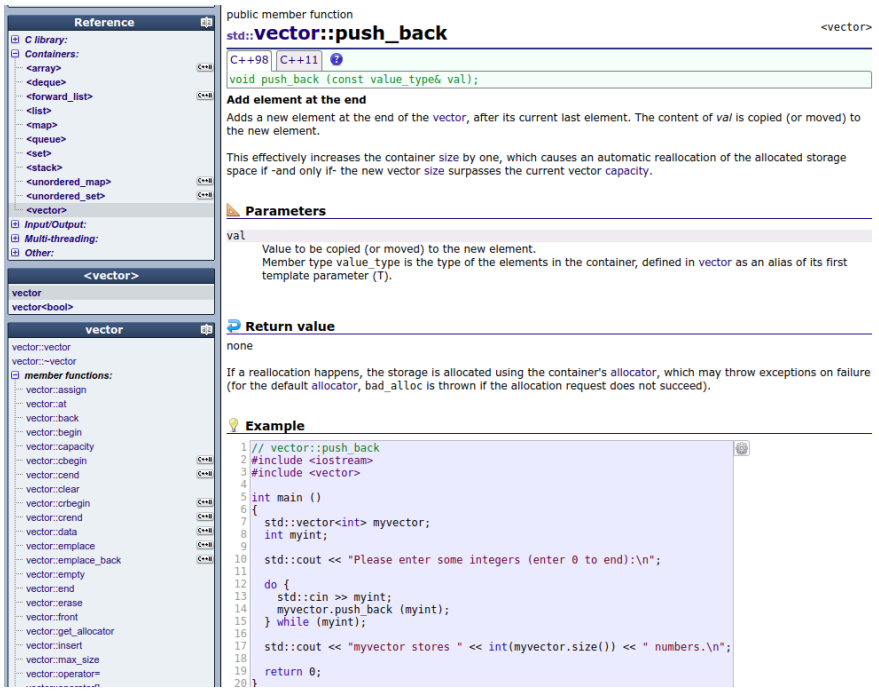
size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)



Reference
C library:
Containers:
<array> C++11
<deque>
<forward_list> C++11
<list>
<map>
<queue>
<set>
<stack>
<unordered_map> C++11
<unordered_set> C++11
<vector>
Input/Output:
Multi-threading:
Other:

<vector>
vector
vector<bool>

vector
vector::vector
vector::~vector
member functions:
vector::assign
vector::at
vector::back
vector::begin
vector::capacity
vector::cbegin C++11
vector::cend C++11
vector::clear
vector::crbegin C++11
vector::crend C++11
vector::data C++11
vector::emplace C++11
vector::emplace_back C++11
vector::empty
vector::end
vector::erase
vector::front
vector::get_allocator
vector::insert
vector::max_size
vector::operator=

public member function
<vector>

std::vector::push_back

C++98 C++11

```
void push_back (const value_type& val);
```

Add element at the end

Adds a new element at the end of the vector, after its current last element. The content of *val* is copied (or moved) to the new element.

This effectively increases the container size by one, which causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

Parameters

val
Value to be copied (or moved) to the new element.
Member type value_type is the type of the elements in the container, defined in vector as an alias of its first template parameter (T).

Return value

none

If a reallocation happens, the storage is allocated using the container's allocator, which may throw exceptions on failure (for the default allocator, bad_alloc is thrown if the allocation request does not succeed).

Example

```
// vector::push_back
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    myvector.push_back (myint);
  } while (myint);

  std::cout << "myvector stores " << int(myvector.size()) << " numbers.\n";

  return 0;
}
```

- **Beispiel:** Arbeiten mit einem **`vector<int>`** unter Verwendung solcher Member-Funktionen und anderen Sprachelemente.

```
#include<stdio.h>
#include<algorithm>
#include<vector>  // Einbinden des Klassentemplates vector<...>.

using namespace std;  // Erlaubt die Verwendung von vector<...> an Stelle von std::vector<...>.

void print_int_vector(vector<int> &iv)
{
  vector<int>::iterator it1;  // Ein Iterator; in diesem Fall ähnlich zu einem Zeiger vom Typ int *.

  for(it1 = iv.begin(); it1 != iv.end(); it1++)
  {
    printf("%d", *it1);

    if(it1+1 != iv.end())
      printf(", ");
```

```
17.     else
18.       printf("\n");
19.   }
20. }
21.
22. int main(void)
23. {
24.   vector<int> A;  // Anlegen eines int-vectors mit 0 Elementen.
25.   A.push_back(1);  // Anhängen des Elements 1 am Ende von A.
26.   A.push_back(2);  // Anhängen des Elements 2 am Ende von A.
27.   A.push_back(5);  // Anhängen des Elements 5 am Ende von A.
28.   print_int_vector(A);
29.
30.   vector<int>::iterator it1;  // Ein Iterator; in diesem Fall ähnlich zu einem Zeiger vom Typ int *.
31.   it1 = A.begin();  // Iterator it1 auf den Anfang von A setzen.
32.   A.insert(it1, 7);  // Einfügen des Elements 7 am Anfang von A.
33.   print_int_vector(A);
34.
35.   A.insert(A.begin() + 2, 8);  // Einfügen des Elements 8 als drittes Element von A.
36.   print_int_vector(A);
37.
38.   // Das Element 2 finden und löschen.
39.   if((it1 = find(A.begin(), A.end(), 2)) != A.end())
40.   {
41.     A.erase(it1);
42.     print_int_vector(A);
43.   }
44.   else
45.     printf("Element nicht gefunden.\n");
46.
47.         // Das nicht vorhandene Element 3 suchen.
48.   if((it1 = find(A.begin(), A.end(), 3)) != A.end())
49.   {
50.     A.erase(it1);
51.     print_int_vector(A);
52.   }
53.   else
54.     printf("Element nicht gefunden.\n");
55. }
```

```
1, 2, 5
7, 1, 2, 5
7, 1, 8, 2, 5
7, 1, 8, 5
Element nicht gefunden.
```

- Es gibt zahlreiche weitere **Container**-Klassentemplates in der Standard Template Library.

Containers library

The Containers library is a generic collection of class templates and algorithms that allow programmers to easily implement common data structures like queues, lists and stacks. There are three classes of containers -- sequence containers, associative containers, and unordered associative containers -- each of which is designed to support a different set of operations.

The container manages the storage space that is allocated for its elements and provides member functions to access them, either directly or through iterators (objects with properties similar to pointers).

Most containers have at least several member functions in common, and share functionalities. Which container is the best for the particular application depends not only on the offered functionality, but also on its efficiency for different workloads.

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

`array` (C++11)	static contiguous array (class template)
`vector`	dynamic contiguous array (class template)
`deque`	double-ended queue (class template)
`forward_list` (C++11)	singly-linked list (class template)
`list`	doubly-linked list (class template)

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

`set`	collection of unique keys, sorted by keys (class template)
`map`	collection of key-value pairs, sorted by keys, keys are unique (class template)
`multiset`	collection of keys, sorted by keys (class template)
`multimap`	collection of key-value pairs, sorted by keys (class template)

Unordered associative containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$

Ein- und Ausgabe mit Streams

- Die Objekte **`cout`**, **`cerr`** und **`cin`**, der **Ausgabeoperator `<<`** und der **Eingabeoperator `>>`** ermöglichen **stream-basierte** Bildschirmausgabe und Tastatureingabe (d.h. sind Alternativen z.B. zu **`printf(...)`**, **`fprintf(stderr, ...)`** und **`scanf(...)`**).
- Formatierte Ausgabe mit Hilfe von **Manipulatoren**, z.B. **`endl`**, **`setw(...)`**, **`fixed`**, ...
- Ob **`cout`**, ... oder **`printf(...)`**, ... besser, einfacher und/oder eleganter ist, ist Gegenstand immer wiederkehrender Diskussion.
- **Beispiel:**

```
1. #include<iomanip>
2. #include<iostream>
3.
4. using namespace std;  // Erlaubt die Verwendung von cout an Stelle von std::cout, ...
5.
6. int main(void)
7. {
8.   double a = 1.0;
9.   int b = 2;
10.  const char string[] = "abcd";
11.
12.  cout << "Bildschirmausgabe mit cout." << endl;  // endl = neue Zeile.
13.
14.  // Bildschirmausgabe von Variabeln bzw. Objekten mit cout.
15.  cout << a << " " << " " << b << " " << string << endl;
16.
17.  // Formatierte Bildschirmausgabe mit cout mit Hilfe von Manipulatoren.
18.  cout << setw(10) << a << endl;  // Breite 10, Kommazahl.
19.  cout << setw(10) << showpos << b << endl;  // Breite 10, positives Vorzeichen anzeigen.
20.  cout << setw(10) << setfill('.') << string << endl;  // Breite 10, Leerraum mit '.' füllen.
21. }
```

```
Bildschirmausgabe mit cout.
1  2 abcd
  1.000000
       +2
......abcd
```

- Der Ausgabeperator **`<<`** kann auch für selbst implementierte Klassen definiert, d.h. überladen werden.
- **Beispiel:** Das Klassentemplate **`Vector2d`**, diesmal mit überladenem Ausgabeperator **`<<`** (ein Funktionentemplate).

```
1. #include<math.h>
2. #include<stdio.h>
3. #include<iostream>
4.
5. using namespace std;
6.
7. // **********
8.
9. // Klassentemplate Vector_2d.
10.
11. template<typename T> class Vector_2d
12. {
13. public:
14.
```

```
15.   Vector_2d(T x_, T y_)
16.   {
17.     x = x_;
18.     y = y_;
19.   }
20.
21.   T norm()
22.   {
23.     return sqrt(x*x + y*y);
24.   }
25.
26.   void Print();
27.
28.   T x;
29.   T y;
30. };
31.
32. template<typename T> void Vector_2d<T>::Print()
33. {
34.   printf("(%+.3f , %+.3f)", (double)x, (double)y);
35. }
36.
37. // **********
38.
39. // Funktionentemplate zur streambasierten Ausgabe von Objekten vom Typ Vector_2d<T>.
40. // Der Ausgabeoperator << wird überladen.
41.
42. template<typename T> ostream &operator<<(ostream &s, const Vector_2d<T> &v)
43. {
44.   return s << "(" << v.x << " , " << v.y << ")";
45. }
46.
47. // **********
48.
49. int main(void)
50. {
51.   Vector_2d<int> v1(1, 2);
52.   printf("v1 = ");  // Ausgabe im C-Stil.
53.   v1.Print();
54.   printf("\n");
55.   cout << "v1 = " << v1 << endl;  // Stream-basierte Ausgabe im C++-Stil.
56.
57.   Vector_2d<double> v2(3.3, 4.4);
58.   printf("v2 = ");  // Ausgabe im C-Stil.
59.   v2.Print();
60.   printf("\n");
61.   cout << "v2 = " << v2 << endl;  // Stream-basierte Ausgabe im C++-Stil.
62. }
```

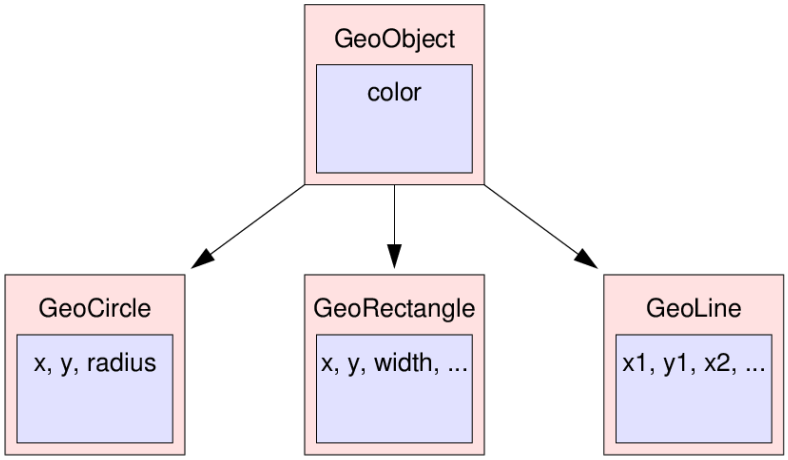
```
v1 = (+1.000 , +2.000)
v1 = (1 , 2)
v2 = (+3.300 , +4.400)
v2 = (3.3 , 4.4)
```

- Beachte:
  `template<typename T> ostream &operator<<(ostream &s, const Vector_2d<T> &v)`
  verwendet je nach Datentyp **`T`** die entsprechenden (d.h. für **`int`** und **`double`** definierten) Ausgabeoperatoren **`<<`**, wohingegen die nicht stream-basierte Ausgabe via
  `template<typename T> void Vector_2d<T>::Print()`

mit dem Formatstring `"(%+.3f , %+.3f)"` festgelegt ist. Die stream-basierte Ausgabe würde also z.B. auch Vektoren, deren Einträge komplexe Zahlen, Matrizen oder algebraische Ausdrücke sind, sinnvoll ausgeben, wenn deren Ausgabeoperatoren `<<` definiert sind.

Vererbung (1)

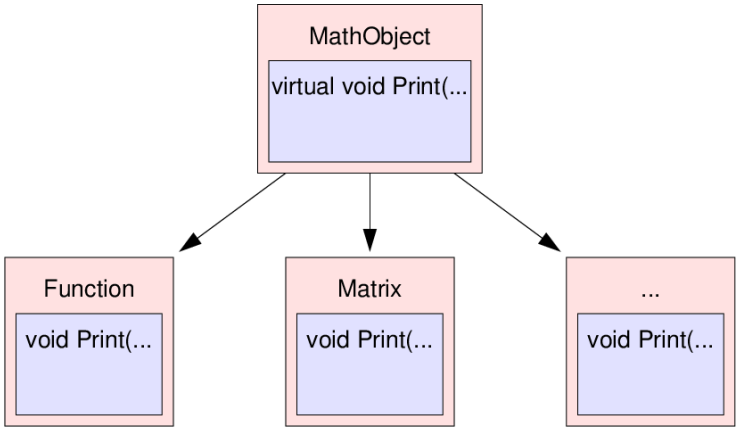
- **Grundidee der Vererbung (siehe auch erste Folie):** Klassen, die Gemeinsamkeiten aufweisen (z.B. gleiche Attribute oder Methoden oder zumindest Methodennamen), erben diese von einer gemeinsamen **Oberklasse**, in der diese Gemeinsamkeiten implementiert sind.
- **Beispiel:** Ein Zeichenprogramm speichert 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**) ...




```
 1. #include<stdio.h>
 2. #include<stdlib.h>
 3.
 4. // ********************
 5.
 6. // Definition einer abstrakten Klasse fuer beliebige mathematische Objekte
 7. // (Funktionen, Matrizen, etc.).
 8.
 9. class MathObject
10. {
11. public:  // Von aussen sichtbar.
12.
13.   // Gibt das mathematische Objekt am Bildschirm aus.
14.   // Unterklassen, z.B. Function, Matrix, ... koennen Print ueberschreiben;
15.   // virtual zeigt an, dass bei Verwendung eines Zeigers vom Typ MathObject
16.   // auf ein Objekt einer Unterklasse von MathObject die ueberschriebenen
17.   // Versionen von Print verwendet werden (--> Polymorphismus); siehe das
18.   // Beispiel in der main-Funktion.
19.   virtual void Print() const { };
20. };
21.
22. // ********************
23.
24. // Definition einer Klasse fuer beliebige Funktionen.
25.
26. // Function erbt von MathObject, ist also eine Unterklasse von MathObject.
27. class Function : public MathObject
28. {
29. public:  // Von aussen sichtbar.
30.
31.   // Wertet die Funktion bei x aus.
32.   double Eval(double x) const;
33.
34.   // Gibt die Funktion am Bildschirm aus.
35.   void Print() const;
36. };
37.
38. double Function::Eval(double x) const
39. {
40.   // Momentan immer die 0-Funktion ... Klasse ist noch nicht vollstaendig
41.   // implementiert.
42.   return 0.0;
43. }
44.
45. void Function::Print() const
```

```
46. {
47.   printf("function(x) = ... (Klasse ist noch nicht vollstaendig implementiert.)\n");
48. }
49.
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. ...

Vererbung (2), this-Zeiger

- Vererben bzw. Ableiten von vorgefertigten Klassen, z.B. aus der Standard Template Library, problemlos möglich.
- **Beispiel:** Definiere Klasse **`my_int_vector`** als Unterklasse von **`vector<int>`**; definiere einen Konstruktor
  `my_int_vector(int num_elements, int value_i)`,
  der den Vektor mit **`int num_elements`** Elementen initialisiert, **`value_i`**, **`value_i + 1`**, ...;
  `my_int_vector` erbt die volle Funktionalität von **`vector`**.

```
1. #include<stdio.h>
2. #include<vector>
3.
4. using namespace std;
5.
6. class my_int_vector : public vector<int>  // Leite Unterklasse my_int_vector von Oberklasse vector<int> ab.
7. {
8. public:
9.
10.   my_int_vector(int num_elements, int value_i) : vector<int>(num_elements)  // Hier wird der Konstruktor von vector<int> au
11.   {
12.     printf("size() = %lu\n", size());
13.
14.     for(int i1 = 0; i1 < num_elements; i1++)
15.       (*this)[i1] = value_i + i1;  // "this" ist ein Zeiger auf das Objekt (existiert in jedem Objekt).
16.   }
17. };
18.
19. int main(void)
20. {
21.   my_int_vector v(5, 123);  // Benutze Konstruktor der Unterklasse my_int_vector.
22.
23.   for(int i1 = 0; i1 < v.size(); i1++)
24.     printf("Position %d --> %d\n", i1, v[i1]);
25.
26.   v.push_back(17);  // Benutze Member-Function der Oberklasse vector<int>.
27.   v.push_back(18);  // Benutze Member-Function der Oberklasse vector<int>.
28.
29.   for(int i1 = 0; i1 < v.size(); i1++)
30.     printf("Position %d --> %d\n", i1, v[i1]);
31. }
```

```
size() = 5
Position 0 --> 123
Position 1 --> 124
Position 2 --> 125
Position 3 --> 126
Position 4 --> 127
Position 0 --> 123
Position 1 --> 124
Position 2 --> 125
Position 3 --> 126
Position 4 --> 127
Position 5 --> 17
Position 6 --> 18
```

- Beachte die Verwendung von **`this`** in Zeile 15. Der Zeiger **`this`** existiert in jedem Objekt und zeigt auf das Objekt. Oft ist ein solcher Zeiger erforderlich, wie z.B. im obigen Beispiel um mit **`[...]`** auf die Elemente des Vektors zuzugreifen.