Informatik III

von
Walter Krämer
überarbeitet von
Hans-Jürgen Buhl

Fachbereich Mathematik (7)
Institut für Angewandte Informatik
Bergische Universität – Gesamthochschule Wuppertal

Wuppertal 2000/2001, 2002

Inhaltsverzeichnis

Listings

Kapitel 1

Von C nach C++

1.1 Namensräume (namespaces):

Ein Namensraum entspricht einem Sichtbarkeitsbereich (`scope`). Wichtig ist dies bei Namenskonflikten im Zusammenhang mit dem gleichzeitigen Verwenden verschiedener Bibliotheken.
Zunächst sollte immer

Listing 1.1: namespace

```
using namespace std;
```

verwendet werden (genaueres später).
Dabei macht `using` die Namen aus einem Namensraum verfügbar. `std` ist dabei der Standardnamensraum bzw. der Namensraum der Standardbibliothek.

1.2 Vereinfachte Ein-/Ausgabe in C++

Die hierfür wichtige Headerdatei ist `iostream`: Sie ist die C++-Header-Datei mit den Definitionen der sogenannten **Ein-/Ausgabeströmen** (`streams`) und Deklarationen der EA-Funktionen.

Beispiel 1.1
Ein Beispiel für die Ausgabe in `C++` ist

Listing 1.2: Beispiel zur Nutzung der Ausgabeströme

```
#include <iostream>
using namespace std;
int main()
{
 cout << "Hallo!" << endl;
 return 0;
}
```

`cout` spezifiziert dabei die Standardausgabe (i.a. ist dies der Bildschirm), `<<` ist der sogenannte **Ausgabeoperator** und `endl` signalisiert das Zeilenende und leert den Ausgabepuffer.

1.3 Rest-der-Zeile-Kommentare

In `C++` gibt es durch `//` eine weitere Möglichkeit Kommentare zu kennzeichnen

```
int f(double); // Deklaration von f( )
                  Kommentar
int f(double x){ return 2*x;} // Definition der Funktion f()
```

Dabei leitet `//` einen Kommentar bis zum Zeilenende ein. Daraus ergibt sich nun eine Möglichkeit ein Testprogramm zu schreiben, welches zeigt, ob ein `C`- oder ein `C++`-Compiler verwendet wird:

Listing 1.3: Testprogramm, ob C oder C++ verwendet wird

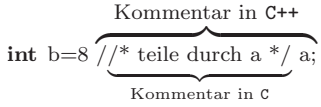
```
#include <stdio.h>

int main()
{
 int a=4;
 int b=8 //* teile durch a */ a;
 +a;
 printf("␣%d\n", b);
 return 0;
}
```

Dieses Programm liefert als Ausgabe:

in `C++` „`12`“

in `C` „`2`“

Dabei begründen sich die unterschiedlichen Ergebnisse darin, dass `//` in `C` keine Kommentare einleitet



1.4 Datenytyp `bool` für Wahrheitswerte

Listing 1.4: Datentyp bool

```
...
bool b = true;        // Definition mit Initialisierung
b=x==y ;              // == bindet staerker als =
cout << b << endl;    // Es wird 1 oder 0 asgegeben je nachdem ob
                      // x==y oder x!=y
```

Folgende logische Operatoren stehen zur Verfügung

1. `!` (Negation)
2. `&&` (logisches und)
3. `||` (logisches oder)

In `C++` findet zudem eine automatische Typkonvertierung von `int` nach `bool` statt

Listing 1.5: Typkonvertierung von int nach bool

```
bool b = 19; // true (!=0)
b=!b         // false (==0)
```

1.5 Semantikänderung bei Signaturen

In `C` bedeutet

```
int f();
```

dass f als eine Funktion mit beliebig langer Parameterliste definiert sein kann.
In `C++` ist obige Zeile äquivalent zu

```
int f(void);
```

D.h. eine solche Funktion darf in `C++` kein Argument besitzen.

1.6 Konstante Größen

In `C` werden Konstanten mittel `#define` definiert:

Listing 1.6: Konstanten in C

```
#define anz 100    /*Praeprozessor-Makro */

double feld[anz];
```

`anz` wird in diesem Fall vom Präprozessor textuell durch 100 ersetzt
In `C++` hingegen kann und sollte dies folgendermaßen geschehen

Listing 1.7: Verwendung von const

```
...
const int anz=100; // anz ist nicht aenderbare Integervariable
double feld[anz];
```

In `C++` sollten unveränderliche Größen stets als `const` deklariert werden!

Listing 1.8: Fehler bei const

```
const double pi=3.1415926;
...
pi=0.3; // liefert Fehlermeldung beim Uebersetzen
```

1.7 Lokale Variablen in Schleifen

`C++` bietet für Schleifen die Möglichkeit, lokal Laufparameter anzulegen:

Listing 1.9: Schleifenvariablen

```
... // hier sei i unbekannt
for (int i=0; i<10; i++)
{
  ...
  cout << i << endl;
  ...
}
// ab hier ist die Variable i nicht mehr verwendbar
// i ist nur im Schleifenrumpf bekannt
```

1.8 Mischen von Anweisungen und Definitionen

In `C++` ist das Mischen von Anweisungen und Definitionen erlaubt, wo in `C` noch alle Definitionen zu Beginn der Funktion angelegt werden mußten.

Listing 1.10: Mischen von Anweisungen und Definitionen

```
int main()
{
 double x= 7.2;
 x++;
 //...
 int k=3;
 x+=k;
 //...
}
```

1.9 Gültigkeitsbereiche

C++ erlaubt eine lokale Überdeckung von Variablen

Listing 1.11: Gültigkeitsbereiche

```
int main()
{
 int w=3;
 {
   int w(0);
   w=14;
   cout << w << endl;
 }
 cout << w << endl;
}
```

Dies ist aber schlechter Programmierstil, da es leicht zu Verwirrungen kommen kann, in welchem Gültigkeitsbereich man sich befindet.

1.10 Referenzparameter

Tauschen der Werte zweier Argumente

1. Falsche Version:

Listing 1.12: Tauschen zweier Argumente a

```
void tausche(int x, int y)
{
 int temp=x;
 x=y;
 y=temp;
 return;
}

int main()
{
int a=3, b=7;

tausche (a,b);
cout << a << " " << b << endl;
```

```
15
return 0;
}
```

Dies erzeugt die Ausgabe „`3 7`“. Die Parameter werden per Wertübergabe übergeben, d.h. die Werte der aktuellen Argumente werden in lokale (d.h. nur in `tausche` gültige Variablen) kopiert. Im Funktionsrumpf werden dann nur diese lokalen Variablen angesprochen.

2. Korrekt mit Zeigern:

Listing 1.13: Tauschen zweier Argumente b

```
void tausche(int* x, int* y)  // int* x kennzeichnet x als Zeiger auf int
{
 int temp=*x;  // * ist der Inhaltsoperator
 *x=*y;
5 *y=temp;
 return;
}

int main()
10 {
int a=3, b=7;

tausche (&a,&b);  // & ist der Adressoperator, d.h. es wird ein Zeiger auf den
                  // Speicherplatz der Variablen a und b uebergeben
15 cout << a << " " << b << endl;

return 0;
}
```

Dies erzeugt nun die Ausgabe „`7 3`“.

3. Vereinfachung durch Referenzparameter

Listing 1.14: Referenzparameter

```
void tausche (int &x, int &y) // & kennzeichnet hierbei die Variablen
   {                          // als Referenzparameter
 int temp=x;
 x=y;
5 y=temp;
 return;
}
```

Die übergebenen Variablen sollen als Referenzen behandelt werden, d.h. implizit werden Zeiger auf die beim Aufruf angeführten Variablen übergeben. Diese werden im Funktionsrumpf automatisch (implizit) dereferenziert.

```
...
int a=3, b=7;
```

```
tausche (a,b); // kein Adressoperator !
cout << a << " " << b << endl;
...
```

Diese Version ist im Prinzip eine notationelle Vereinfachung der Version 2.

Referenzparameter in `C++` entsprechen den `var`-Parametern in `Pascal`. Es werden keine lokalen Kopien der Originalparameter angelegt, vielmehr wird im Funktionsrumpf mit den Speicherbereichen, in denen die Werte der Originalargumente (des Aufrufs) abgelegt sind, gearbeitet.
Diese Speicherbereiche werden durch das automatische Dereferenzieren der Referenzen angesprochen. Die Referenzen selbst werden dabei nicht verändert!

Listing 1.15: Referenzen (allgemein)

```
#include <iostream>

using namespace std;

int main()
{
  int i=7, j;
  int &r1=i;   // Die Referenz r1 zeigt auf den Speicherplatz von i
               // der auf 'int &' folgende Variablenname bezeichnet eine Referenz
  int &r2=r1;  // Auch r2 bezieht sich auf den Speicherplatz von i
  cout << "i: " << i << " r1: " << r1 << " r2: " << r2 << endl;
  // r1 und r2 koennen als Alias fuer i aufgefasst werden
  i+=2;
  cout << "i: " << i << " r1: " << r1 << " r2: " << r2 << endl;
  // int &r1 == &i; ware ein Fehler. Beachte: r1 wird dereferenziert

  j = r1 ;      // Wert von i wird an j zugewisen
  cout << "j: " << j << endl;
  r1 ++;        // i wird um 1 erhoeht
  cout << "i: " << i << endl;

  return 0;
}
```

Die Ausgabe wäre hier

```
i: 7 r1: 7 r2: 7
i: 9 r1: 9 r2: 9
j: 9
i: 10
```

Referenzen müssen bei der Deklaration immer initialisiert werden. Sie können dann nicht mehr verändert werden. Sie sind also konstant!

Eine Referenz auf eine Referenz ist nicht möglich.

1.11 Ein-/Ausgabe von strings

Zur Behandlung von Zeichenketten stehen folgende Funktionen zur Verfügung

`cin`	Standardeingabe	(Tastatur)
`cout`	Standardausgabe	(Bildschirm)
`cerr`	Standardfehlerausgabe	(Bildschirm)
`clog`	Standardfehlerausgabe	(Bildschirm)

Der Eingabeoperator >>

Der Eingabeoperator wird folgendermaßen verwendet

Listing 1.16: Eingabeoperator

```
int zahl;
cin >> zahl; //liest eine Folge von Ziffern bis zu einem Nicht-Ziffernzeichen
             //und wandelt diese in die interne Darstellung einer int-Zahl um
```

Will man hingegen einzelne Zeichen (auch Leerzeichen) lesen, muss die Funktion `get()` verwendet werden:

Listing 1.17: get-Funktion

```
char c;
cin.get(c); // Aufruf einer Memberfunktion (Klassenkonzept wird spaeter erlaeutert)
```

Ein Programm zur Ein- und Ausgabe würde dann folgendermaßen aussehen:

Listing 1.18: Ein- und Ausgabe

```
#include <iostream>
#include <string>     // Header fuer string-Bibliothek
using namespace std;

int main()
{
 string Name;          // string als Datentyp der string-Bibliothek

 cin >> Name;
 cout << Name << endl;
 return 0;
}
```

Die Eingabe „`Tom Tykwer`“ würde dann bei diesem Programm „`Tom`“ als Ausgabe haben.
Es wird nur bis zu ersten `Whitespace`-Zeichen gelesen. Der Rest des Tastaturpuffers kann mit einem weiteren „`cin >>`“ gelesen werden.

Das Einlesen einer ganzen Zeile kann wie folgt geschehen:

Listing 1.19: ganze Zeile lesen

```
...
int main()
{
 string Name;
 getline(cin,Name);
// oder:
// char pName[MAX_LINE_LENGTH];
// cin.getline(pName, MAX_LINE_LENGTH);
 cout << Name << endl;
 return 0;
}
```

Die Eingabe „`Franka Potente`“ würde jetzt auch als Ausgabe „`Franka Potente`“ liefern.

Ausgabeoperator $<<$
Er wandelt die interne Darstellung in einen Text um.

`cin` und `cout` können auf Betriebssystemebene mittels `<` bzw. `>` umgeleitet werden. Ist z.B. `prog` ein ausführbares Programm und Bezeichnen `eingabe` und `ausgabe` zwei Dateien, dann kann der Aufruf so erfolgen:

```
prog < eingabe > ausgabe
```

1.12 Mehrere Übersetzungseinheiten in einem Projekt

Betrachten wir die beiden Funktionen

$$\begin{aligned} g(x) &:= x^2 \\ f(x) &:= 2 \cdot g(x) \end{aligned}$$

und die Aufgabe, $f(3.5) + g(17)$ zu berechnen.

Extrem: Jede Funktion in eine eigene Datei. Für jede Funktion auch jeweils eine eigene Headerdatei mit der Deklaration dieser Funktion

Listing 1.20: dat1.cpp (=Hauptprogramm)

```
#include <iostream>
#include "f.h"          // Deklaration der Funktion f( )
                        // Die Anfuehrungzeichen bewirkt eine Suche im Benutzerverzeichnis
#include "g.h"          // Deklaration der Funktion g( )
using namespace std;

int main()
{
 cout << f(3.5)+g(17) << endl;
 return 0;
}
```

Listing 1.21: g.cpp (=Definition von g())

```
#include "g.h"

double g(double x) {return x*x;}
```

Die zugehörige Headerdatei `g.h`:

Listing 1.22: g.h

```
double g(double);  // Deklaration von g()
```

Listing 1.23: f.cpp (=Definition von f())

```
#include "g.h"      // Deklaration von g() einbinden

double f(double x) {return 2*g(x);}
```

Die zugehörige Headerdatei `f.h`:

Listing 1.24: f.h

```
double f(double);  // Deklaration von f()
```

Jede Übersetzungseinheit (in diesem Fall jede Datei mit der Endung „.cpp“) ist getrennt übersetzbar:

`g++ -c f.cpp`	liefert Objektdatei `f.o`
`g++ -c g.cpp`	liefert Objektdatei `g.o`
`g++ -c dat1.cpp`	liefert Objektdatei `dat1.o`

`g++ dat1.o f.o g.o` liefert dann das ausführbare Programm `a.out`. Eine bessere Methode stellt allerdings ein `Makefile` zusammen mit `make` dar.

1.13 Mehrfachlesen von Headerdateien vermeiden

Betrachten wir z.B.

Listing 1.25: pi.h

```
const float pi =3.141593;
```

und

Listing 1.26: zweipi.h

```
#include "pi.h"
const float zweipi=2*pi;
```

und verwenden wir dann folgende Hauptprogramm

Listing 1.27: t.cpp

```
#include <iostream>
#include "pi.h"
#include "zweipi.h"    // ergibt Fehler: Redefinition von 'const float pi'

int main()
{
 cout << pi*zweipi << endl;
 return 0;
}
```

so ergibt sich die beschriebene Fehlermeldung. Dies begründet sich darin, dass die beiden Zeilen

```
#include "pi.h"
#include "zweipi.h"    // ergibt Fehler: Redefinition von 'const float pi'
```

vom Präprozessor durch

```
const float pi =3.141593;
const float pi =3.141593;
const float zweipi=2*pi;
```

ersetzt wird. Die erste Zeile ist der Inhalt von `pi.h` und die beiden folgenden von `zweipi.h`.

Dieser Fehler kan durch die Verwendung von sogenannten **Wächtern** vermieden werden. Diese zeigen an, ob eine Datei bereits vom Compiler eingebunden wurde (und somit kann eine wiederholte Einbindung unterdrückt werden). Im Falle der Datei `pi.h` könnte dies so aussehen

Listing 1.28: pi.h mit Wächter

```
#ifndef piHeaderBereitsGelesen // ifndef = if not defined
#define piHeaderBereitsGelesen
const float pi =3.141593;
#endif
```

Dabei werden die Zeilen 2 und 3 ignoriert, wenn der Wächter bereits definiert (also die Headerdatei schon eingebunden) ist.

1.14 Überladung von Funktionen

Funktionen mit gleichem Namen können mit verschiedenen Parameterlisten in derselben Übersetzungseinheit definiert werden:

Listing 1.29: Überladung von Funktionen

```
double sum(double x1, double x2) {return x1 + x2 ;}                    // (1)
double sum(double x1, double x2, double x3) {return x1 + x 2 + x3;} // (2)
int sum(int x1, int x2) {return x1+x2;}
```

Der Compiler erkennt beim Aurfuf von `sum(...)` an Hand der Anzahl der aktuelle Argumente oder deren Datentypen, welche Funktionsdefinition (am besten) passt. Diese wird dann ausgeführt:

```
cout <<          sum(2.0,5.0)          << "␣" << sum(1.0,2.0,3.0)  << endl;
          Definition (1) wird verwendet              hier Definition (2)

cout <<     sum(2,5)       << endl;
          hier Definition (3)
```

Beachte:

Der Ergebnistyp wird nicht zur Unterscheidung herangezogen. Die weitere Definition

```
double sum (int x1, int x2){...}
```

wäre oben nicht erlaubt (Konflikt mit (3)).

Was passiert beim Aufruf `sum(2,3.5)`? Ist in diesem Fall (1) oder (3) gemeint? Der Compiler meldet diese Unsicherheit. Man sollte explizite Typkonvertierung verwenden bzw. z.B. die Definition `double sum (int x1, double x2){...}` hinzufügen!
Allerdings würden für

```
bool a,b;
 ...
cout << sum(a,b) << endl;
```

automatisch nach `int` konvertiert, so dass die Definition (3) hier zum Einsatz käme.

Auch Operatoren (wie z.B. +, *, -) können (z.B. für selbstdefinierte Datentypen) überladen werden:

Listing 1.30: Überladen von Operatoren

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

int main()
{
 double x=3, y=5;
 string s1="Ich habe", s2=" Hunger";

 cout << x+y << " "<< s1+s2 << << endl;
 return 0;
}
```

Im ersten Fall ist + die arithmesche Addition, im zweiten Fall bewirkt + das Hintereinanderhängen (Konkatenation) von Zeichenketten (das heißt der +-Operator ist für Zeichenketten überladen).
Die Definition solcher Operatoren werden wir später im Zusammenhang mit selbstdefinierten Typen (Klassenkonzept) besprechen.

1.15 Default-Argumente bei Funktionen

Listing 1.31: Default-Argumente

```
float sum(float x1, float x2, float x3=0, float x4=0)
{
 return x1+x2+x3+x4;
}
```

Diese Definition ermöglicht nun Aufrufe wie

`sum(a,b)` (= `sum(a,b,0,0)`)
`sum(a,b,c)` (= `sum(a,b,c,0)`)
`sum(a,b,c,d)`

Kommt in einer formalen Argumentenliste ein Defaultargument vor, so müssen auch alle weitere rechts stehenden Argumente mit Defaultargumenten versehen werden!
Falsch wäre z.B.

```
int sum(int x, int y=0, int z ){...}
```

1.16 Beschaffung und Freigabe von Speicherplatz

Unter `C` existieren die beiden Funktionen `malloc` und `free` zur dynamischen Anforderung von Speicherplatz. Unter `C++` übernehmen nun `new` und `delete` diese Aufgabe. Die genaue Verwendung zeigt nun das folgende Programm

Listing 1.32: Berechnung der Fakultät

```
// Verwendung von new und delete

#include <iostream>
using namespace std;

int main()
{
  double *p= new double // Es wird ein Zeiger p definiert und gleichzeitig
                        // wird Speicherplatz beschafft, auf den p zeigt
  *p = 5;
  cout << *p << endl;
  delete p; // Freigabe des Speichers, auf den p zeigt
  // Jetzt waere die Verwendung von *p ein Fehler!

  int anz = 3;
  float* feld;
  feld = new flaot[anz];      // Komponenten feld[0] bis feld[2]
                              // feld ist ein Zeiger auf erstes Feldelement feld[0]
  for(int i=0;i<anz; i++)
    feld[i]=i*i;  // Feldelemente mit Werten belegen
  for(int i=0;i<anz; i++)
    cout << i << "_i*i=" << feld[i] << endl;
  delete[] feld;  // delete feld; wuerde nur feld[0] freigeben

  return 0;
}
```

Zu `new` gehört immer `delete`
Zu `new ...[]` gehört immer `delete[] ...`

Objekte, die mit `new` angelegt werden, liegen im sogenannten **Freispeicher** (auch **Heap** oder **dynamischer Speicher** genannt).

1.17 Funktionsschablonen (Funktionstemplates)

Zunächst ein Beispiel mit explizit überladenen Funktionen:

```
...
void print (char* p) { cout << *p << endl;}     // (1)
void print (int* p) { cout << *p << endl;}      // (2)
void print (double* p) { cout << *p << endl;} // (3)
//... Ausgabe fuer weitere ueber Zeiger angesprochene Groessen

int main()
{
 int k=3, *p=&k;
 print(p);      // (2)
 char c='x';
 print(&c);     // (1)
 double w=2.5;
 double *q;
```

```
 q=&w;
 print(q );      // (3)
 return 0;
}
```

Die explizite Definition aller obigen Funktionen kann durch das Verwenden einer sogenannten **Funktionsschablone** oder eines **Templates** vermieden werden. Der Datentyp des Arguments im Aufruf einer solchen Template-Funktion wird vom Compiler verwendet, um eine ausführbare Ausprägung zu realisieren. Es werden dabei nur Funktionen ausgeprägt, welche zu im Quelltext vorkommenden Aufrufen nötig sind.

Listing 1.33: Beispiel Templates

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

template <class T> void print (T p)
{
 cout << *p << endl;        // print ist eine Template-Funktion (Funktionsschablone),
                            // welche nach dem Typ T parametrisiert ist
 return;
}

int main ()
{
 char c='x';

 print (&c); // Es wird eine Auspaegung von print fuer den Datentyp
             // class T gleich char* veranlasst. Diese wird uebersetzt und
             // dann mit &c aufgerufen, da das Argument den Typ
             // "Pointer auf char" also "char*" hat
 return 0;
}
```

Dabei bedeutet

template	leitet eine Schablone ein
<**class** T>	Typ welche zur Parametrisierung dient; T heißt Templateparameter
void	Ergebnistyp
print	Name der Funktion
(T p)	T ist Typ des Parameters p und p das Argument

Der Aufruf `print(p);` würde, falls p gemäß `long int *p;` vereinbart worden wäre, eine weitere Ausprägung für den Datentyp `class T` gleich `long int*` veranlassen. D.h. `void print (T p)` in obiger Schablone würde zur Funktionsausprägung durch `void print (long int* p)` ersetzt.

Der Compiler generiert als bei einem Funktionsaufruf von `print(x)` den Code für die zum Argumententyp von x passende Funktion.

```
int k=3;
print(&k);
```

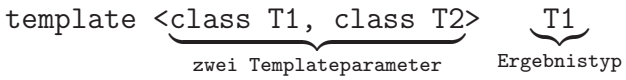
hätte also die Ausprägung `void print (int* p)` mit anschließnder Übersetzung und späterem Aufruf zur Folge.
Die schon bekannte Funktion `void swap(T& x, T& y)` bietet sich ebenfalls ausgezeichnet für eine Implementierung als Template an.

1.18 Mehrere Template-Parameter

Template-Funktionen können auch mehrere Template-Parameter haben

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

template <class T1, class T2> T1 max (T1 x, T2 y)
{
  return x>y ? x : y;
}

int main()
{
  cout << max(3.5, 9.2);
  cout << max(true, 4) << endl; // erzeugt die Auspraegung (bool) max(bool, int)
  return 0;
}
```

Dabei gilt



Für `max(true, 4)` ist der Ergebnistyp der Typ des ersten Arguments, also `bool` (d.h. die Ausgabe ist hier `0` oder `1`).
Die Ausgabe des Programms ist „`9.2 1`“ (Beachte die Identifikation von `false` mit 0 und `true` mit Werten ungleich 0).

1.19 Überladung von template-Funktionen

Eine Überladung ist auch für Template-Funktionen erlaubt.

Listing 1.34: Überladung von Template-Funktionen

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

// Ausgabe fuer die Variablen verschiedener Typen
template <class T> void print (T p)        // (1)
{
  cout << p;
  return;
}

// Ausgabe fuer ueber Pointer angesprochene Variablen
```

```
template <class T> void print (T* p)        // (2)
{
  cout << *p;
  return;
}

int main()
{
  char c='x';
  print(&c);  // Aufruf (2)
  print('x'); // Aufruf (1)
  print(c );   // Aufruf (1)
  // print(&('x')); waer Fehler, da kein Lvalue
  return 0;
}
```

Die Ausgabe wäre in diesem Fall „`xxx`“.

1.20 Auflösen von Mehrdeutigkeiten

`C++` bietet nun zum Auflösen von Mehrdeutigkeiten eine explizite Qualifizierung an

Listing 1.35: Auflösen von Mehrdeutigkeiten

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

template <class T> T max(T x, T y)
{
 return x>y ? x : y;
}

int main()
{
  // cout << max ('a',1) << endl; erzeugt den Fehler:
  // no matching function for call to ‘max (char, int)’,
  // weil beide Argumenttypen nicht uebereinstimmen

  cout << max<int>('a',1) << "␣";     // ist aequivalent zu:
                                      // max((int)'a', 1);
  int k='b'; // int-Wert der dem Buchstaben 'b' entspricht
  cout << max<char>('a',k) << endl; // ist aequivalent zu:
                                      // max('a', (char) k);

  return 0;
}
```

Durch die explizite Qualifizierung wie z.B. `max<int>` wird nun erreicht, dass der Compiler den Parameter `T=char` setzt. Die Ausgabe des obigen Programmes wäre „`97 b`“.

Analog kann dies natürlich auch für mehrere Template-Parameter geschehen. Dabei wird die Qualifizierung in der Reihenfolge der gewünschten Belegung der Template-Parameter angegeben, wie das folgende Beispiel zeigt:

Listing 1.36: Auflösen von Mehrdeutigkeiten (mehrere Template-Parameter)

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

template <class T1, class T2> T1 max(T1 x, T2 y)
{
 return x>y ? x : y;
}

int main()
  cout << max<bool, int>(1.5,4.3) << " ";
  // Auspraegung (bool) max( (bool) 1.5, (int ) 4.3) )
  return 0;
}
```

Folgendes Beispiel zeigt uns nun das Zusammenspiel von Template-Funktionen, explizit definierten Funktionen und spezialisierten Template-Funktionen.

Listing 1.37: Template- und explizit definierte Funktionen

```
// Zusammenspiel von
// allgemeiner Templatefunktion,  spezialisierter
// Templatefunktion und explizit  definierter  Funktion

#include <iostream>
using namespace std;

template <class T1, class T2> T1 max(T1 x, T2 y)
{
  cout << "Schablone wird verwendet" << endl;
  return x>y ? x : y;
}

bool max(bool a, int b) // explizit  definiert
{
  cout << "Es wurde max(bool, int) als " << endl;
  cout << "   explizit definierte Funktion aufgerufen" << endl;
  return (bool)max(int(a), b);
}

template<> bool max<bool, int>(bool a, int b) // spezialisiert
{
  cout << "Es wurde max(bool, int) als " << endl;
  cout << "   spezialisierte Template-Funktion aufgerufen" << endl;
  return (bool)max(int a, b);
}

int main()
{
  cout << max(true, 4) << endl << endl;
  cout << max(4, true) << endl << endl;
  cout << max<bool,int>(true, 4) << endl << endl; // qualifizierter Aufruf
  return 0;
}
```

Dabei wird beim Aufruf von `max(true, 4)` die explizit definierte Funktion einer Schablone vorgezogen; `max<bool, int>` hingegen erzwingt die Verwendung der spezialisierten Templatefunktion. Die Ausgabe des obigen

Programms ist

```
Es wurde max(bool, int) als
  explizit definierte Funtkion aufgerufen
Schablone wird verwendet
1

Schablone wird verwendet
4

Es wurde max(bool, int) als
  spezialisierte Templatefunktion aufgerufen
Schablone wird verwendet
1
```

1.21 Ganzzahlige Templateparameter

Die Templateparameter können auch Ganzzahlwerte sein:

Listing 1.38: ganzzahlige Templateparameter

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

template <bool k> void print ()
{
  if(k==true) // oder auch: if(k)
    cout << "true" << endl;
  else
    cout << "false" << endl;
  return;
}

int main()
{
  print<7>();
  print<0>();

  return 0;
}
```

Dies erzeugt die Ausgabe:

```
true
false
```

Zusätzlich steht nun auch hier wieder eine Möglichkeit zur Spezialisierung zur Verfügung

Listing 1.39: Spezialisierung bei Ganzzahlwerten

```
// Spezialisierung einer Template-Funktion

#include <iostream>
using namespace std;

// Ganzzahl-Werte als Template-Parameter
template <int k> void print ()
{
  cout << "Schablone ausgepraegt fuer k gleich " << k << endl;
  return;
}

// Spezialisierung fuer Ganzzahl-Wert k gleich 0
template<> void print<0> ()
{
  cout << "Fuer 0 wird eine andere Schablone benutzt!" << k << endl;
  return;
}


int main()
{
  print<7>();
  print<0>();

  return 0;
}
```

Dies ergibt

```
Schablone ausgepraegt fuer k gleich 7
Fuer 0 wird eine andere Schablone benutzt!
```

Diese kann man nun z.B. zur rekursiven Berechnung der Fakultät einer natürlichen Zahl benutzen:

Listing 1.40: Berechnung der Fakultät

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

// Ganzzahl-Werte als Template-Parameter
template <int k> long int faku ()
{
  return k*faku<k-1>();
}

// Spezialisierung fuer Ganzzahl-Wert k gleich 0
template<> long int faku<0>()
{
  return 1;
}

int main()
{
  cout << "5! = " << faku<5>() << ", ";
```

```
  const long int faku13=faku<13>();
  // Beachte: 13! wird bereits zur Compilezeit berechnet!
  cout << "13! = " << faku13 << endl;

  return 0;
}
```

Als Ausgabe ergibt sich „5! = 120, 13!= 1932053504“. Hier handelt es sich um abhängige oder berechnete Konstanten, die z.B. als Feldlänge benutzt werden können.

Template-Funktionen können sich auch nur im Ergebnistyp unterscheiden, dazu ist allerding ein qualifizierte Aufruf notwendig, wie uns das nächste Beispiel zeigt:

Listing 1.41: Template-Funktionen, die sich nur im Ergebnistyp unterscheiden

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

template <class T> T f(double x)
{
  return (T)x;
}

int main()
{
  cout << f<int>(3.9) << endl;
  cout << f<bool>(3.9) << endl;
  // cout << f(3.9) << endl; no matching function for call to ' f (double)'
  cout << (int)3.9 << " " << (bool)3.9 << endl;
  return 0;
}
```

Ergibt als Ausgabe

```
3
1
3 1
```

Bei der Verwendung von Templates spricht man von **generischer Programmierung**.
Templates erhöhen dabei die Compilerzeit, nicht aber die Laufzeit des Programms.

1.22 Fehlermeldungen zur Programmlaufzeit

Neben Fehlermeldungen der Art

Listing 1.42: Laufzeitfehlermeldung

```
if ( i > DIM ){
    cerr << endl << "*** Fehler: Index groesser als DIM" << endl;
    exit (1);
}
```

kann man mittels `<cassert>` abschaltbare Überprüfungen (Zusicherungen) in Programmtexte einbauen:

Listing 1.43: Zusicherung

```
#include <cassert>

// ...

    assert ( i <= DIM );

// ...
```

Will man dann in der fertig ausgetesteten Auslieferungsversion des Programms die Überprüfung der Zusicherungen aus Laufzeitgründen unterdrücken, so kann dies mittels

```
#define NDEBUG

// ...
```

in den Quellcodes oder besser durch Aufruf des Compilers mittels

```
g++ -DNDEBUG ...
```

geschehen.

1.23 Compiletime Assertions und Compilerfehlermeldungen bei rekursiven Templates

Das Makro

Listing 1.44: compile time assertion

```
#define CT_ASSERT(expr) \
   { struct xxxxxxyxyxyy { unsigned int bf : (expr); }; }
```

kann für Zusicherungen zur Übersetzungszeit eines Programmes benutzt werden. Wollen Sie etwa Ihr Programm nur auf solchen Compilern übersetzbar machen, auf denen der Typ `int` mindestens 16 Bit Genauigkeit anbietet, so kann das dann mittels

```
CT_ASSERT(sizeof(int) * CHAR_BIT >= 16);
```

geschehen.

Leider funktionieren solche Assertions innerhalb von rekursiv sich selbst aufrufenden Templates wie zum Beispiel `faku()` nicht. Deshalb muß man hier den Compiletime-Fehler schon zur Instantiierungszeit der Templatefunktionen etwa nach folgendem Muster erzwingen:

Listing 1.45: Compiletime-Fehlermeldung in rekursiven Templates

```
template<int k>
long int faku(){
   return faku<(1/(k>=0)) * (k-1)>() * k;
}

template<> long int faku<0>(){
   return 1;
}
```

1.24 Template-Metaprogramming

Neben Schleifen sind für eine Konstantenarithmetik (zur Compiletime) auch andere Kontrollstrukturen in Templates simulierbar, vergleiche etwa »Todd Veldhuizen: Template Metaprograms, WWW«.
Damit kann zum Beispiel auch der Sinus einer Konstante zur Übersetzungszeit als Konstante ausgewertet werden.

1.25 Partial Evaluation

Wenn Templates zur Geschwindigkeitssteigerung von (numerischen) Bibliotheken (MTL, Blitz++, ...) benutzt werden, so kann dies als eine spezielle Art von Präprozessing aufgefaßt werden:
Die Template-Funktion

Listing 1.46: statische und dynamische Datenabhängigkeiten

```
template <int k, ...>
long int fkt(int d1 , ...){
    // ...
}
```

hängt von den statischen Daten `k, ...` und den dynamischen Daten `d1, ...` ab.
Ein `Partial Evaluator` kann den Originalquellcode in einen solchen überführen, der nur noch von den dynamischen Daten abhängt.

1.26 Formatierte Ein- und Ausgabe

Zur formatierten Ein- und Ausgabe von Textdateien stehen Ihnen eine Anzahl von Manipulatoren ähnlich wie `endl` zur Verfügung:

endl	Zeilenende anfügen und Puffer leeren
setw(20)	setze die Breite des nächsten Ausgabefeldes hier auf 20 Zeichen
left	Ausgabe linksbündig
right	
internal	Vorzeichen links, Füllzeichen (default: ' '), Zahl rechts
setfill('*')	setze das Füllzeichen für das Ausgabefeld
setprecision(14)	setze bei Gleitkommazahlausgabe die Anzahl signifikanter Ziffern
showpos	zeige das '+'-Vorzeichen
showpoint	zeige einen Dezimalpunkt
dec	Basis dezimal
oct	Basis oktal
hex	Basis sedezimal (hexadezimal)
setbase(8)	setze die Basis hier z.B. auf 8, mögliche Werte: 8, 10, 16
showbase	Basis als führende 0 (oktal) oder führendes 0x (sedezimal) anzeigen
boolalpha	"true" und "false" statt 1 und 0 benutzen
noboolalpha	
fixed	Festkommaformat: 123.4
scientific	wissenschaftliches Gleitkommaformat: 1.234E+002
...	
ws	überliest "white space" bei Eingaben
skipws	
noskipws	

Listing 1.47: iomanip

```
#include <iostream>
#include <iomanip>

using namespace std;

  // ...
  double d1(1.0/3);
  cout << setw(20) << setprecision(10) << d1 << endl;
  cout << setfill ('*') << setw(20) << setprecision(4) << 12.145 << endl;
  cout << internal << showpos << setw(20) << setprecision(4) << 12.145 << endl;
  cout << hex << 255 << endl;
  // ...
```

Für weitere Informationen zu Textstreams siehe etwa »Kapitel 27 - I/O library« des C++-Standards.

Beachten sie insbesondere, daß einige Manipulatoren nur die nächste Ausgabeoperation betreffen (setw()), andere jedoch bis zum expliziten nächsten Wechsel gültig bleiben (setfill()).

Eingaben sollten in der Regel zeichenweise eingelesen und bearbeitet werden. Dabei ist `<strstream>` hilfreich:

Listing 1.48: Aufzählungstyp Wochentag

```
#include      <iostream>
#include      <sstream>
#include      <string>

using namespace std;

// ...

  enum DayType {_Montag, _Dienstag, _Mittwoch, _Donnerstag,
                  _Freitag , _Samstag, _Sonntag};

  static const string DayTable[] =
     {" Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag", "Sonntag"};

  istream& operator>>(istream& is, DayType& d)
  {
        string s;
        is >> s;
        for (int i = 0; i < 7; i++)
                if (s == DayTable[i]) {
                        d = DayType(i);
                        return is;
                }
        is.clear(ios::badbit);
        return is;
  }
  ostream& operator<<(ostream& os, const DayType& d)
  {
        os << DayTable[int(d)];
        return os;
  }

// ...
int main(){
            DayType d2;
            cout << "Bitte einen Wochentag eingeben: ";
          {
                string s;
                getline(cin, s);

                istringstream ss(s);
                ss >> d2;
                if (ss.bad()) {
                        ss.clear();
                        cerr << "Eingabefehler!" << endl;
              };
```

```
                // hier koennten andere Eingabemoeglichkeiten bearbeitet
                // werden
50
            {   string shelp;
                ss >> shelp;
                if (shelp.length()>0)
                    cerr << "Zusaetzlicher Zeileninput ignoriert!" << endl;
55          }
        }
          cout << endl << endl << " ---" << d2 << "---" << endl;
}
```

Die Operationen `bad(), clear(), ...` werden Sie im Folgeabschnitt genauer kennenlernen. Der Datentyp `string` erlaubt in C++ eine sehr viel bequemere Arbeitsweise mit (dynamischen) Zeichenketten als es die `char str[]`-Objekte in C erlauben.

Hinweis: Sollten Ihnen (auf älteren C++-Compilern) einige der oben genannten Manipulatoren fehlen, kann häufig z.B. `setiosflags(ios::hex)` Abhilfe schaffen.

Zusätzliche Hilfe für die Zeichenbearbeitung finden Sie etwa in `<cctype>`: `isdigit(), tolower(), isxdigit() ...`

1.27 Dateiein- und -ausgabe

Textdateien

In `<iostream>` werden Möglichkeiten zur Aus- und Eingabe von Daten in textueller Form bereitgestellt:

Listing 1.49: Datei-IO

```
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

int main(){

  ofstream Ziel("out.txt", ios::noreplace);
  if (!Ziel){
    cerr << endl << "Datei out.txt konnte nicht "
      << " zum Schreiben geoeffnet werden"
      << endl;
    exit(1);
  }

  double d1(123.456789012345678912345678 9);

  Ziel << setprecision(16) << d1 << endl;
  if (!Ziel.good()){
    cerr << endl << "Schreiben auf Datei out.txt fehlerhaft" << endl;
    exit(2);
  }

  Ziel.close();

  // ...

  ifstream Quelle("out.txt");
  if (!Quelle){
    cerr << endl << "Datei out.txt konnte nicht zum Lesen geoeffnet werden"
      << endl;
    exit(3);
  }

  double d2;

  Quelle >> d2;
  if (!Quelle.good()){
    cerr << endl << "Lesen von Datei out.txt fehlerhaft" << endl;
    exit(3);
  }

  Quelle.close();

  // ...

  cout << setprecision(20) << d1 << endl;
  cout << d2 << endl;
  cout << scientific << abs(d1 - d2) << endl;

}
```

Modi für das Öffnen von Dateien können sein:

ios::in	öffne für Input
ios::out	öffne für Output
ios::app	öffne für Anfügen (append)
ios::ate	öffne und ans Ende der Datei positionieren
ios::binary	ändere spezielle Zeichen nicht (EndOfLine, ...)
ios::noreplace	falls Datei schon vorhanden, Fehlerabbruch
ios::nocreate	öffne falls vorhanden, aber nicht neu anlegen
ios::trunc	öffne und verwerfe alten Inhalt

Folgende Attributabfragen stehen bei `iostream`'s zur Verfügung:

eof()	Eingabe am EndOfFile angekommen
good()	
bad()	nichtignorierbarer Lesefehler von Datei oder Schreibfehler in Datei
fail()	Leseoperation fand unerwartete Zeichen

Die Benutzung von Textdateien ist portabel: eine auf Maschine A geschriebene Datei kann problemlos auf Maschine B gelesen werden, selbst wenn diese ein ganz anderes Betriebssystem, einen anderen Compiler, ... hat.

Eine Datei kann auch gleichzeitig zum Lesen und Schreiben geöffnet sein:

Listing 1.50: random access Zugriff auf eine Datei

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main(){
   // Lege Datei "neu.txt" an:

  ofstream Neu("neu.txt");
  if (!Neu){
    cerr << endl << "Datei Neu.txt konnte nicht angelegt werden"
         << endl;
    exit(1);
  }
  Neu.close();

  // Oeffne Datei zum Lesen und Schreiben:
  // Vorbedingung: Datei "neu.txt" existiert bereits!
  fstream Datei("neu.txt", ios::in | ios::out);
  if (!Datei){
    cerr << endl << "Datei IODatei.txt konnte nicht geoeffnet werden"
         << endl;
    exit(2);
  }

  // Schreibe in Datei:

  char outbuf[] = "01234567890123456789012345678901234";
  Datei << outbuf;
  if (!Datei.good()){
    cerr << endl << "Schreibefehler: Streamstatus: "
         << Datei.rdstate() << endl;
    exit(2);
  }

  // Positioniere auf Dateianfang:

  Datei.seekg(0, ios::beg);
  if (!Datei.good()){
    cerr << endl << "Positionierungsfehler: Streamstatus: "
         << Datei.rdstate() << endl;
    exit(2);
  }

  // Lese vom Dateianfang an:

  char inbuf[32];
  inbuf[31] = '\0';

  Datei.get(inbuf, sizeof(inbuf)-1);
  if (!Datei.good()){
    cerr << endl << "Lesefehler: Streamstatus: "
         << Datei.rdstate() << endl;
    exit(3);
  }

  cout << inbuf << endl;

}
```

Literatur: A. Langer u.a.: IOStreams and Locales, Addison-Wesley

Binärdateien
Binärdateien enthalten die interne Bitstruktur der Daten Ihres Computers. Sie sind deshalb fast nie portabel einsetzbar. Benutzt werden sollten diese deshalb nur, wenn es um die Auslagerung von Daten auf Platte und umgehendes Wiedereinlesen geht. Dann ist die Benutzung von Binärdateien zum einen schneller (die Konvertierung in/von Text zur internen Darstellung entfällt) und zum anderen konvertierungsfehlerfrei:

Listing 1.51: Schreibe Binärdatei

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main(){

  ofstream Ziel;
  Ziel.open("real-bin.bin",ios::binary);
  if (!Ziel) throw "Fehler bei Dateioeffnen";

  double v = 1.0/3.0;
  if (!Ziel.write((char *)&v, sizeof(v)))
      throw "Fehler bei write";

  v = 1.0;
  if (!Ziel.write(reinterpret_cast<char *>(&v), sizeof(v)))
      throw "Fehler bei write";
  cout << endl;

}
```

Und schließlich das Leseprogramm:

Listing 1.52: Lese Binärdatei

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main(){

  ifstream Quelle;
  Quelle.open("real-bin.bin",ios::binary);
  if (!Quelle) throw "Fehler bei Dateioeffnen";

  double v;
  if (!Quelle.read((char *)&v, sizeof(v)))
      throw "Fehler bei write";
  cout << v << endl;

  if (!Quelle.read((char *)&v, sizeof(v)))
      throw "Fehler bei write";
  cout << v << endl;

}
```

1.28 Casts

Werte eines Datentyps in korrespondierende Werte eines anderen kann man mittels der Typwandlungsoperationen (casts) wandeln:

Aus C:

Listing 1.53: C casts

```
double r1=1.234567890123456;
float  f;

f = r1  ;          // automatische Typwandlung bei Zuweisung
f = (float) r1   // C cast
```

Neu in C++ sind:

Listing 1.54: zusätzliche C++ casts

```
double r2;
// ...
f = float(r1+r2); // jeder Typname ist zugleich
                  // (ueberladene) Funktion
                  // der Typwandlung, der auch als
f = float r2 ;    // unaerer Operator benutzt werden
                  // kann
f = static_cast<float>(r2);  // neue Templateform
```

Benutzt werden sollten immer die als Templates definierten Casts:

`reinterpret_cast<Zieltyp>(Wert)`	"gefährliche" Uminterpretation
`const_cast<Zieltyp>(Wert)`	zur Anpassung von konstanten aktuellen Parametern an nichtkonstante formale Paramter oder umgekehrt
`static_cast<Zieltyp>(Wert)`	alle sonstigen Casts, außer:
`dynamic_cast<Zieltyp>(Wert)`	vergleiche Kapitel 2 und Vererbung

Den `reinterpret_cast` kennen Sie bereits aus dem vorigen Abschnitt.

Der `const_cast` kann wie in den folgenden Beispielen benutzt werden:

Listing 1.55: const_cast

```
#include <iostream>
using namespace std;
void print(const char* a[], const int n){
  for (int i=0; i<n; i++)
    cout << a[i] << endl;
};
int main(){
  char* a[] = { "abcdefg", "", "abc" };
  print(const_cast<const char**>(a), 3);
}
```

beziehungsweise

Listing 1.56: const_cast Variante 2

```
#include <iostream>
using namespace std;
void print(char* a[], const int n){
  for (int i=0; i<n; i++)
    cout << a[i] << endl;
};

int main(){
  const char* a[] = { "abcdefg", "", "abc" };
  print(const_cast<char**>(a), 3);
}
```

Schöner ist natürlich:

Listing 1.57: Zeichenkettenvektor als Funktionsparameter

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

void print(const char* a[], const int n){
  for (int i=0; i<n; i++)
    cout << a[i] << endl;
};

int main(){
  const char* a[] = { "abcdefg", "", "abc" };
  print(a , 3);
}
```

Wenn Sie jedoch die Funktion `print()` nicht selbst programmiert hätten, sondern von einer erworbenen Bibliothek bezögen, so bliebe Ihnen unter Umständen nichts anderes als eine der beiden obigen Varianten übrig. Bedenken Sie jedoch die semantischen Auswirkungen!

1.29 Invarianten, Vor- und Nachbedingungen

Komentieren Sie den folgenden Quellcode

Listing 1.58: Funktion power()

```
////////////////////////////////
// Datei:   power.cc
// Version: 1.0
// Zweck:   while-Schleife
// Autor:   Hans-Juergen Buhl
// Datum:   17.09.1998
////////////////////////////////

#include      <iostream>
#include      <iomanip>

using namespace std;

double power2(double x, int exp)
{
  double erg(1.0);

  if (exp < 0)
        throw "negativer Exponent bei power2 nicht erlaubt!";
  while ( exp > 0 ) {
        if ((exp % 2 ) != 0) {
                erg *= x;
                exp--;
      }  else   {               // hier  ist  exp gerade
                x = x*x;
                exp = exp/2;
      }
  };
  return erg;
};

int main()
{
 cout << setprecision(10) << power2(13.5, 3) << endl;

 return 0;
}
```

durch Angabe von Schleifeninvariante, -variante, Vor- und Nachbedingung, ... nach dem folgenden Muster:

Listing 1.59: Invarianten, Vor- und Nachbedingungen

```
////////////////////////////////
// Datei:   power.cc
// Version: 0.91
// Zweck:   while-Schleife
// Autor:   Hans-Juergen Buhl
// Datum:   15.09.1998
////////////////////////////////

#include      <iostream>
#include      <iomanip>
```

```
using namespace std;

double power2(double x, int exp)
      //                                  (        Spezifikation )
//   power2:  double x int ---> double
//       exp < 0:   Exception "negativer Exponent bei power2 nicht erlaubt!"
//       exp > 0:   power2(x, exp) == (x ^ exp) * (1 + eps), abs(eps) klein
//
//   Vorbedingung:  exp >= 0
{
  // Sei  x0 = x, exp0 = exp

  double erg(1.0);

  // erg == 1.0
  // x0^exp0 == erg * x^exp

  if (exp < 0)
        throw "negativer Exponent bei power2 nicht erlaubt!";

  // exp >= 0

  for (int i = exp; i > 0; i--){
        //
        // i  in  int ,  i <= exp, i > 0
        //
         //      x0^exp0 == erg * x^i
        //

        erg *= x;

        // Schleifeninvariante :
        //
         //      x0^exp0 == erg * x^(i-1), i-1 >= 0
        //
  };
  //
  // x0^exp0 == erg
  //
  // ( Schleifenvariante  = i)
  //

  return erg;

  // Problemfall: x == 0, exp == 0

};

int main()
{
 cout << setprecision(10) << power2(13.5, 3) << endl;
 cout << setprecision(10) << power2(2.0, 10) << endl;

 return 0;
}
```

Bei der Analyse mit Hilfe der „Zusicherungen“ in Kommentarform wurde der Problemfall `x == 0, exp == 0` als mathematisch nicht korrekt behandelt (und auch nicht richtig spezifiziert) erkannt! Ändern Sie die Spezifikation und das Programm.

1.30 Exceptions

Das Codestück

Listing 1.60: Exceptions

```
...

double power1(double x,int exp)
{
  double erg(1.0);
  if ( exp < 0 ) throw(exp);

  ...

}
```

erzeugt bei Nichterfüllen der Vorbedingung eine (abfangbare) Ausnahmebedingung (Exception). Wenn die erzeugte Exception vom Typ `int` nicht abgefangen wird, so wird je nach Rechnersystem und Entwicklungssoftware ein Programmabbruch mit oder ohne Fehlermeldung erzeugt:

```
Runtime exception error:
Current exception: int
Abort (core dumped) No handler for exception
```

oder

```
Abbrechen (Speicherabbild geschrieben)
```

oder

```
Aborted
```

Dabei wird zum Teil die Abspeicherung eines Speicherabzuges (core), der die Variableninhalte zum Zeitpunkt des Programmabbruchs zur späteren Betrachtung konserviert, durchgeführt.
`workshop -D excep core` oder `kdbg` erlaubt Ihnen, die Stelle des Programmabbruchs im Quellcode zu finden und, falls nötig, die Variableninhalte zum Abbruchzeitpunkt zu inspizieren (klicken Sie im Stacktrace-Fenster auf `main()` beziehungsweise `Display`. Unter Linux können Sie `kdevelop` ähnlich einsetzen.

Schöner ist es jedoch, wenn man die Exception am Ende von `main` (oder sogar in `power1()`) abfängt und selbst eine Fehlermeldung erzeugen kann:

Listing 1.61: Abfangen von Exceptions

```
////////////////////////////////
// Datei:   power1.cc
// Version: 1.1
// Zweck:   while-Schleife, catch
// Autor:   Hans-Juergen Buhl
// Datum:   17.09.1998
////////////////////////////////

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

double power1(double x, int exp)
{
  double erg(1.0);

  if (exp < 0)
        throw "negativer Exponent bei power2 nicht erlaubt!";
  while ( exp > 0 ) {
        if ((exp % 2 ) != 0) {
                erg *= x;
                exp--;
        } else  {               // hier  ist  exp gerade
                x = x*x;
                exp = exp/2;
        }
  };
  return erg;
};

int main()
{
  try{
        cout << setprecision(10) << power2(0.5, -2) << endl;
        return 0;
  }
  catch (const char* err){
        cerr << endl << "Fehler: "<< err << endl << endl;
        return 1;
  }
}
```

Bemerkung: Hier wurde statt einer Exception vom Typ `int` eine solche vom Typ `const char*` benutzt. Im anderen Falle hätte es natürlich `catch(const int& err) ...` heißen müssen.

Bei `catch`-Anweisungen von `try`-Blöcken findet keine automatische Typkonversion statt, weshalb etwa

Listing 1.62: string und char* beim Exceptionabfangen

```
// ...
try{
    // ...
    if (nenner == 0) throw "Fehler: Nenner == 0!";
    // ...
}catch(const string& err){
    cerr << endl << err << endl;
    exit 1;
}
// ...
```

die Exception vom Typ `const char*` nicht abfängt!

Fangen Sie also in diesem Falle eine Exception vom Typ `const char*` ab oder erzeugen Sie eine Exception vom Typ `string`:

```
if (nenner == 0) throw (string)"Fehler: Nenner == 0!";
```

beziehungsweise

```
if (nenner == 0) throw string("Fehler: Nenner == 0!");
```

Schließlich gibt es die Möglichkeit, mehrere `catch`-Anweisungen für einen `try`-Block anzugeben:

Listing 1.63: Abfangsequenz

```
try{
    // ...
} catch (const char* err){
    cerr << endl << "### Fehler: " << err << endl;
    exit (1);
} catch (const string& err){
    // ...
} catch (const int& i_err){
    // ...
} catch (...){
    cerr << endl << "### Fehler: unbekannte Exception" << endl;
    exit (2);
}
```

Dabei müssen die spezielleren vor den allgemeineren Exceptions angegeben werden:

Listing 1.64: Abfangreihenfolge

```
//...
#include <exception>
//...
try{
    // ...
    if (exp < 0) throw range_error("exp invalid, shuld be >= 0");
    // ...
}catch(const std::range_error& e){
    cerr << endl << "### Fehler: " << e.what() << endl;
    exit (1);
}catch(const std:runtime_error& re){
    //...
}catch(const std::bad_alloc& bae){
    //...
}catch(const std::exception& e){
    //...
}catch(...){
    cerr << endl << "### Fehler: unbekannte Exception" << endl;
    exit (2);
}
```

wobei folgende standardmäßig vorhandene `exception`-Hierarchie benutzt wurde:

```
- exception
-- bad_alloc
-- bad_exception
-- bad_cast
-- bad_typeid
-- ios::failure
-- runtime_error
---- range_error
---- overflow
---- underflow
-- logic_error
---- length_error
---- domain_error
---- out_of_range
---- invalid_argument
```

Wollen Sie eine eigene Exception-Hierarchie, etwa

```
- Matherr
-- Overflow
-- ZeroDivide
```

aufbauen, so kann das folgendermaßen geschehen:

Listing 1.65: eigene Exception-Hierarchie

```
// ...
class Matherr {};
class Overflow : public Matherr {};
class ZeroDivide : public Matherr {};
// ...
try{
   // ...
  if (Bed1) throw ZeroDivide();
   // ...
}catch(const ZeroDivide& e){
     // ...
}catch(const Matherr&){
     // ...
}
```

Zusammenfassung: Wird in einer Funktion eine erzeugte Exception nicht abgefangen, so wird diese Funktion abgebrochen und ein Handler für die Exception in der sie aufrufenden Programmeinheit gesucht ...

Ist schließlich auch in `main()` kein Handler (zutreffende `catch`-Anweisung) aufzufinden, wird das gesamte Programm abgebrochen.

1.31 IOStream – Statusabfrage oder Exceptions

Wollen Sie Dateioperationen statt der dauernden Erfolgsüberprüfung nach Kapitel 1.27 lieber mittels Exceptions überwachen, können Sie die Erzeugung von Exceptions folgendermaßen einschalten:

Listing 1.66: Exceptions bei Datei-IO

```
try{
  cout.exceptions(ios::badbit | ios:: failbit );
   // ...
} catch(...){
    if (cout.bad()){     // unrecoverable error
        // ...
   } else if (cout.fail ()){     // retry
       // ...
   }
}
```

1.32 `new` – `null` oder Exceptions

Die Anforderung von neuem Speicherplatz mittels `new()` erzeugt bei nicht mehr genügend vohandenen Zentralspeicher-Ressourcen die Exception `bad_alloc`.

Deshalb sollte in jedem Programm, das `new()` benutzt, eine `catch`-Anweisung für `bad_alloc` vorgesehen werden!
Bei standardkonformen Compilern ist die folgende Programmsequenz

Listing 1.67: Exceptions bei new()

```
//...
double *r = new double(1.15);
if (r == 0){
  // ...
}
```

zwar nicht falsch, aber unnötig, da `r` nie den Wert 0 annehmen kann.

Will man größere ältere Programme mit standardkonformen Compilern übersetzen, so kann man statt einer Neukonzeption von `try-/catch-`Anweisungen auf folgende `new()`-Variante zurückgreifen:

Listing 1.68: news(nothrow)

```
// ...
#include <new>
// ...
double *r = new(nothrow) double(1.15);
if (r == 0){
  // ...
}
```

Zur effektiven Verwaltung von dynamischen Speicherbereichen ist es häufig nötig, eine eigene Speicheranforderung und -rückgabe zu implementieren. Dies kann leicht mittels `set_new_handler()` geschehen und wird beschrieben zum Beispiel in: M. Cline et al.: C++ FAQs, 2nd Edition, Seite 169ff.

1.33 Templates mit Funktionen als statischem Parameter

Schon in C können Sie Funktionen als Parameter von Funktionen nutzen. Das funktioniert aus Abwärtskompatibilitätsgründen natürlich auch in C++:

Listing 1.69: Funktionen als Funktionsparameter

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

typedef double FKT(double);

double func1(double x){
  return 1.0/(1.0 + x);
}

double integrate(FKT f, double a, double b, int numSamplePoints) {
  double delta = (b - a) / (numSamplePoints - 1);
  double sum = 0.0;
  for (int i=0; i < numSamplePoints; i++)
    sum += f(a + i*delta);
  return sum * (b - a) / numSamplePoints;
}

int main(){
  cout << integrate(func1, 1.0, 2.0, 100) << endl;
}
```

`integrate()` kann dabei auch ohne Hilfe von `typedef` deklariert werden:

Listing 1.70: Funktionskopf ohne Benutzung von typedef-Namen

```
double integrate(double(*f)(double),
                 double a, double b,
                 int numSamplePoints)
```

Es gibt jedoch auch die Möglichkeit, `integrate()` als Templatefunktion mit der zu integrierenden Funktion `f()` als (statischen) Templateparameter zu definieren:

Listing 1.71: Funktionen als Template-Parameter

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

double func1(double x){
  return 1.0/(1.0 + x);
}

template<double T_function(double)>
double integrate(double a, double b, int numSamplePoints) {
  double delta = (b - a) / (numSamplePoints - 1);
  double sum = 0.0;
  for (int i=0; i < numSamplePoints; i++)
    sum += T_function(a + i*delta);
  return sum * (b - a) / numSamplePoints;
}

int main(){
  cout << integrate<func1>(1.0, 2.0, 100) << endl;
}
```

Die letzte Variante ist dabei im allgemeinen sehr viel besser optimierbar und deshalb sollte ihr der Vorzug gegeben werden.

1.34 Makefiles

Wenn sie Programme mittels `make` statt mittels direkter Compileraufrufe `CC -g -o p1 p1.cc` beziehungsweise `g++ -g -o p1 p1.cc` übersetzen, hat das den Vorteil, daß die Übersetzung von `p1.cc` in das Binary `p1` nur dann stattfindet, wenn `p1.cc` neueren Datums als `p1` ist, also offensichtlich editiert oder sonstwie verändert wurde nachdem `p1` erzeugt worden ist.

Außerdem kann die Funktionsweise von `make` modifiziert werden, wenn geeignete UNIX-Environmentvariablen gesetzt werden. Wollen Sie etwa die Optimierung des Compilers (Option `-O3`) einschalten, so kann dies auf den Sun-Workstations mittels

```
% setenv CPPFLAGS "-g -O3"
```

beziehungsweise auf den LINUX-Rechnern mit

```
% export CPPFLAGS="-g -O3"
```

geschehen.

Die aktuellen Defaulteinstellungen von make können Sie mittels `make -p` abfragen. Sollte die Steuerung mit Aufrufparametern (vergleiche `man make`) beziehungsweise mit Hilfe von Environmentvariablen zu umständlich sein, kann man für Projekte Makefiles (`makefile` bzw. `Makefile`) einrichten; Fallstudie:

Wenn Sie die Funktionen des letzten Abschnitts auf drei Dateien verteilen

Listing 1.72: func1use.cc

```
// Datei func1use.cc

#include <iostream>
#include "func1.h"
#include "integrate.h"

using namespace std;

int main(){
  cout << integrate<func1>(1.0, 2.0, 100) << endl;
}
```

und

Listing 1.73: integrate.h

```
// Datei integrate.h
// template Funktion integrate()

template<double T_function(double)>
double integrate(double a, double b, int numSamplePoints) {
  double delta = (b - a) / (numSamplePoints - 1);
  double sum = 0.0;
  for (int i=0; i < numSamplePoints; i++)
    sum += T_function(a + i*delta);
  return sum * (b - a) / numSamplePoints;
}
```

sowie

Listing 1.74: func1.h

```
// Datei func1.h

double func1(double x){
  return 1.0/(1.0 + x);
}
```

so erstellt `make func1use` eine aktuelle Version des Binaries `func1use`. Nach einer Änderung von `func1.h` oder `integrate.h` behauptet ein erneuter Aufruf von `make func1use` jedoch immer noch `'func1use' is up to date`.

Bemerkung: Da auf unseren Sun-Workstations der Compiler `CC` Templates mit Funktionen als Templateparametern noch nicht unterstützt, stellen Sie bitte auf diesen Workstations zuvor den von `make` benutzten C++-Compiler mittels `setenv CCC g++` auf den g++-Compiler um. Auf LINUX-Rechnern hieße die entsprechende Environmentvariable zur Umstellung des von `make` zu benutzenden C++-Compilers `CXX`.

Um das `make`-Tool davon zu unterrichten, dass `func1use` von `integrate.h` und `func1.h` abhängt, müssen Sie eine Datei `makefile` mit dem Inhalt

Listing 1.75: makefile

```
func1use: func1use.cc
	$(CCC) $(CPPFLAGS)  -o func1use func1use.cc
func1use.o: func1.h integrate.h
```

einrichten (Die zweite Zeile hat mit einem Tabultorzeichen zu beginnen, unmittelbar gefolgt von "`$(CCC) ...`").

Bei großen Projekten sollten in Headerdateien kein ausführbarer Code sondern lediglich Funktionsdeklarationen und Templatedefinitionen bereitgestellt werden, also:

Listing 1.76: func1.h

```
// Datei func1.h
double func1(double);
```

und

Listing 1.77: func1.cc

```
// Datei func1.cc
#include "func1.h"
double func1(double x){
  return 1.0/(1.0 + x);
}
```

sowie folgendes geändertes `makefile`:

Listing 1.78: makefile Version 2

```
func1use: func1use.cc func1.o
	$(CCC) $(CPPFLAGS) -o func1use func1use.cc func1.o
func1use.o: func1.h integrate.h
func1.o: func1.h
```

Zur Erleichterung (und automatischen Generierung) der durch Include-Anweisungen verursachten Abhängigkeiten existiert auf UNIX-Workstations mit X-Window das Tool `makedepend`. Es kann folgendermaßen benutzt werden (Annahme: Es existiert noch kein Makefile):

```
% touch makefile
makedepend func1.cc func1use.cc
makedepend: warning:  func1use.cc, line 1: cannot find include file "iostream"
        not in iostream
        not in /usr/include/iostream
% cat makefile
# DO NOT DELETE

func1.o: func1.h
func1use.o: func1.h integrate.h
```

Jetzt brauchen Sie nur noch die Anweisung zur Erzeugung des Binaries aus den `*.o`-Dateien am Anfang des Makefiles hinzufügen:

Listing 1.79: makefile Version 3

```
func1use: func1use.cc func1.o
	$(CCC) $(CPPFLAGS) -o func1use func1use.cc func1.o

# DO NOT DELETE

func1.o: func1.h
func1use.o: func1.h integrate.h
```

Üblicherweise fügt man noch ein Make-Ziel `clean` zum Löschen aller Object-Dateien und der Binaries sowie ein Make-Ziel `depend` zum Aktivieren von `makedepend` hinzu

Listing 1.80: makefile Version 4

```
funcluse: funcluse.cc func1.o
	$(CCC) $(CPPFLAGS) -o funcluse funcluse.cc func1.o
clean:
	rm funcluse *.o
depend:
	makedepend \
	-- funcluse.cc func1.cc
# DO NOT DELETE
funcluse.o: func1.h integrate.h
func1.o: func1.h
```

Wurden — wie oben — einige Includedateien nicht gefunden, kann der Pfad zur Suche von Includedateien erweitert werden nach folgendem Muster:

Listing 1.81: makefile-Ergänzung

```
depend:
	makedepend \
	-I /opt/share/SUNWspro/SC5.0/include/CC \
	-- funcluse.cc func1.cc
```

(In LINUX heißt der entsprechende zu ergänzende Pfad statt `/opt/share/SUNWspro/SC5.0/include/CC` `/usr/include/stlport` und `CCC` sollte durch `CXX` ersetzt werden.) Dann können Sie sämtliche Abhängigkeiten erkennen:

Listing 1.82: Abhängigkeiten im makefile

```
...
# DO NOT DELETE

funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/iostream
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/stdcomp.h
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/compnent.h
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/istream
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/ios
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/rwstderr.h
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/stddefs.h
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/stddef.h
funcluse.o: /usr/include/stddef.h /usr/include/sys/isa_defs.h
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/stdarg.h
funcluse.o: /usr/include/stdarg.h /usr/include/sys/va_list.h
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/rwstderr_macros.h
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/rwlocale
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/string
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/string.h
funcluse.o: /usr/include/string.h /usr/include/sys/feature_tests.h
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/ctype.h /usr/include/ctype.h
funcluse.o: /opt/share/SUNWspro/SC5.0/include/CC/wchar.h /usr/include/wchar.h
funcluse.o: /usr/include/stdio_tag.h /usr/include/wchar_impl.h
funcluse.o: /usr/include/time.h /usr/include/sys/types.h
funcluse.o: /usr/include/sys/machtypes.h /usr/include/sys/int_types.h
```

```
25 func1use.o: /usr/include/sys/select.h /usr/include/sys/time.h
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/wctype.h
   func1use.o: /usr/include/wctype.h
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/string_ref
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/memory
30 func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/stdlib.h
   func1use.o: /usr/include/stdlib.h
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/limits.h
   func1use.o: /usr/include/limits.h /opt/share/SUNWspro/SC5.0/include/CC/new
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/exception
35 func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/iterator
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/utility
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/stdmutex.h
   func1use.o: /usr/include/synch.h /usr/include/sys/machlock.h
   func1use.o: /usr/include/sys/time_impl.h /usr/include/sys/synch.h
40 func1use.o: /usr/include/thread.h /usr/include/sys/signal.h
   func1use.o: /usr/include/sys/unistd.h
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/traits
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/stdio.h /usr/include/stdio.h
   func1use.o: /usr/include/stdio_impl.h
45 func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/iotraits
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/iosfwd
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/stdexcept
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/typeinfo
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/locimpl
50 func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/locvector
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/vendor
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/ctype
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/numeral
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/iosbase
55 func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/limits
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/math.h
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/math.h
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/float.h /usr/include/float.h
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/codecvt
60 func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/rw/usefacet
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/ostream
   func1use.o: /opt/share/SUNWspro/SC5.0/include/CC/streambuf func1.h
   func1use.o: integrate.h
   func1.o: func1.h
```

Environmentvariablen für `make` können auch in den Projekt-Makefiles selbst definiert werden:

Listing 1.83: makefile mit Environment-Variablen

```
# CXXFLAGS=-g -DNDEBUG
CXXFLAGS=-g

CXX=CC

CCC=$(CXX)
CPPFLAGS=$(CXXFLAGS)

useGrad: useGrad.cc Grad.o
	$(CCC) $(CXXFLAGS) -o useGrad useGrad.cc Grad.o
clean:
	rm -f *.o *~ useGrad
depend:
	makedepend useGrad.cc Grad.cc
# DO NOT DELETE
useGrad.o: Grad.h
Grad.o: Grad.h
```

Abschließend noch Möglichkeiten zu Fallunterscheidungen in Makefiles (siehe Target `ex_if`) und zur Versionsabfrage der CC- und g++-Compiler (siehe Target `show_version` und die vorhergehenden Zeilen) auf Solaris beziehungsweise LINUX-Rechnern (diese Datei funktioniert sowohl mit Solaris-`make` als auch mit `gmake`):

Listing 1.84: makefile mit Fallunterscheidungen

```
CXXFLAGS=-g -DNDEBUG
# CXXFLAGS=-g

CXX=CC

CCC=$(CXX)
CPPFLAGS=$(CXXFLAGS)

4_2:	4_2.o zeichensatz.o toupper.o rot.o rot13.o
	$(CCC) $(CXXFLAGS) -o 4_2 4_2.o zeichensatz.o toupper.o rot.o rot13.o
clean:
	rm -f *.o *~


#                                       Versionsinformationen
#

G++VERS = $(shell g++ -v  2>&1 | sed -e "1 d" | cut -d" " -f1-3; )

G++VERS:sh  = g++ -v  2>&1 | sed -e "1 d" | cut -d" " -f1-3
CC_VERS:sh  = CC -V 2>&1 | cut -d" " -f2,6-7

show_version:
	echo;echo ----------;echo;\
	if [ "$(OSTYPE)" = "linux" ]; \
	then \
	    echo $(G++VERS); \
	else \
```

```
                echo $(G++VERS); \
                echo $(CC_VERS); \
        fi ; \
        echo;echo ----------;echo;\

#                                       Bourne sh-Konstrukte fuer
#                                       bedingte Ausfuehrung von Anweisungen
#                                       im Makefile
ex_if:
        if [ "$(CXX)" = "CC" ]; \
        then echo Sun C++ Compiler; \
        elif [ "$(CXX)" = "g++" ]; \
        then echo GNU C++ Compiler; \
        fi

depend:
        makedepend 4_2.cc zeichensatz.cc toupper.cc rot.cc rot13.cc

# DO NOT DELETE
```

Durch die Kombination beider Möglichkeiten können Sie betriebssystem-, compiler- und compilerversionsspezifische Sonderfälle in die Makefiles einbauen.
Auch das automatische Abarbeiten der `make`-Kommandos in allen Unter- und Unterunterverzeichnissen ist einfach realisierbar:

Listing 1.85: makefile mit automatischer Rekursion durch alle Unterverzeichnisse

```
# ...

# DIRS=a1 a2 a3
DIRS = $(shell find * -type d -print)
BDIR = $(shell pwd)

DIRS:sh = find * -type d -print | grep -v SunWS_cache
BDIR:sh = pwd

subdirs:
        for SDIR in $(DIRS) ; \
        do \
          echo -------- running make in subdir $${SDIR}; \
          cd $${SDIR} && $(MAKE) -k all; \
          cd ${BDIR}; \
        done

clean:
        rm -f *.o *~ *% 4_2 ; \
        for SDIR in $(DIRS) ; \
        do \
          echo -------- running make clean in subdir $${SDIR}; \
          cd $${SDIR} && $(MAKE) -k clean; \
          cd ${BDIR}; \
        done

# ...
```

Sollen mehrere Makefiles im selben Ordner mit demselben Target gestartet werden, so kann folgendes Makefile weiterhelfen:

Listing 1.86: makefile für den Start mehrerer makes (auch) im selben Ordner

```
which =

all :
	echo ---------- making target $(which) from Makefile_1; \
	make -k -f Makefile_1 $(which); \
	echo ---------- making target $(which) from Makefile_2; \
	make -k -f Makefile_2 $(which);


# Benutzung:
#	make
#	make which=clean
#	make which=depend
#	 ...
```

1.35 Hilfesysteme auf Sun und LINUX-Systemen

Auf unseren Solaris-Sun-Workstations stehen Ihnen zur Zeit die folgenden Hilfssysteme zur Verfügung:

- Sun Workshop Collection über
  http://wmcip3.math.uni-wuppertal.de:8888/ab2/coll.36.5/@Ab2CollView?
- Sun WorkShop Compiler C++ Collection über
  http://wmcip3.math.uni-wuppertal.de:8888/ab2/coll.32.5/@Ab2CollView?
- Sun WorkShopDocumentation über
  file:///usr/local/SUNWspro/DOC5.0/lib/locale/C/html/index.html
- Sun WorkShop Compiler C Collection über
  http://wmcip3.math.uni-wuppertal.de:8888/ab2/coll.33.5/@Ab2CollView?
- Sun WorkShop FORTRAN Collection über
  http://wmcip3.math.uni-wuppertal.de:8888/ab2/coll.34.5/@Ab2CollView?
- Sun WorkShop Overview Collection über
  http://wmcip3.math.uni-wuppertal.de:8888/ab2/coll.377.1/@Ab2CollView?
- Sun WorkShop TeamWare Collection über
  http://wmcip3.math.uni-wuppertal.de:8888/ab2/coll.13.7/@Ab2CollView?
- GNU-Dokumentation in `/opt/local/gnu/manuals`

sowie die Man-Pages zu `CC, make, ....`

Auf unseren LINUX-Rechnern stehen neben den dortigen Man-Pages folgende Hilfemöglichkeiten bereit:

- `xdvi /usr/share/doc/Books/gcc-manual-2.6.dvi` ...
- HOWTO GCC-HOWTO, C++Programming-HOWTO, ... in /usr/share/doc/howto/en
- andere Dokumentation in /usr/share/doc und Unterverzeichnissen
- KDE Help Center `khelpcenter` (Programm Handbücher, dann Entwicklung)
- SuSE Help Center `susehelpcenter` (kdevelop oder gnu, ...)

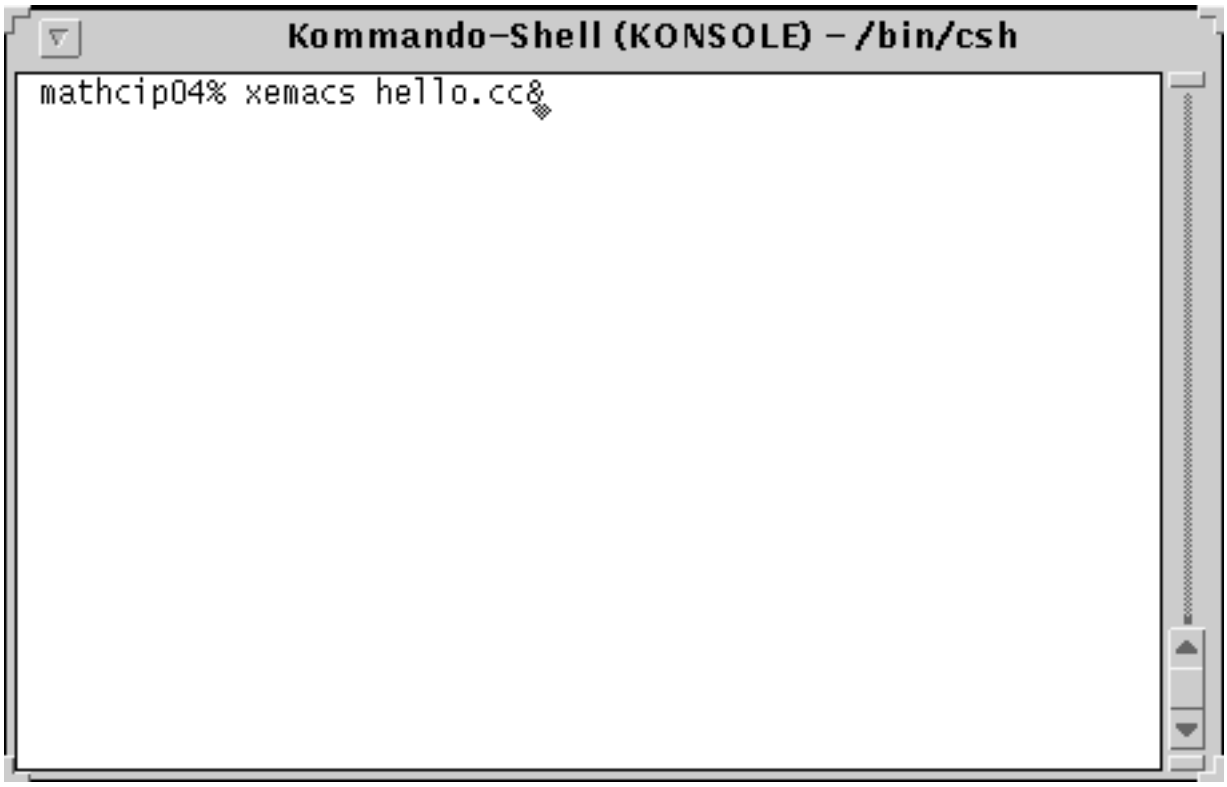
Allgemein können Sie sich über den C++ Standard mittels http://www.math.uni-wuppertal.de/~buhl/c++ informieren.

1.36 Programmentwicklungsumgebungen: Workshop und Kdevelop

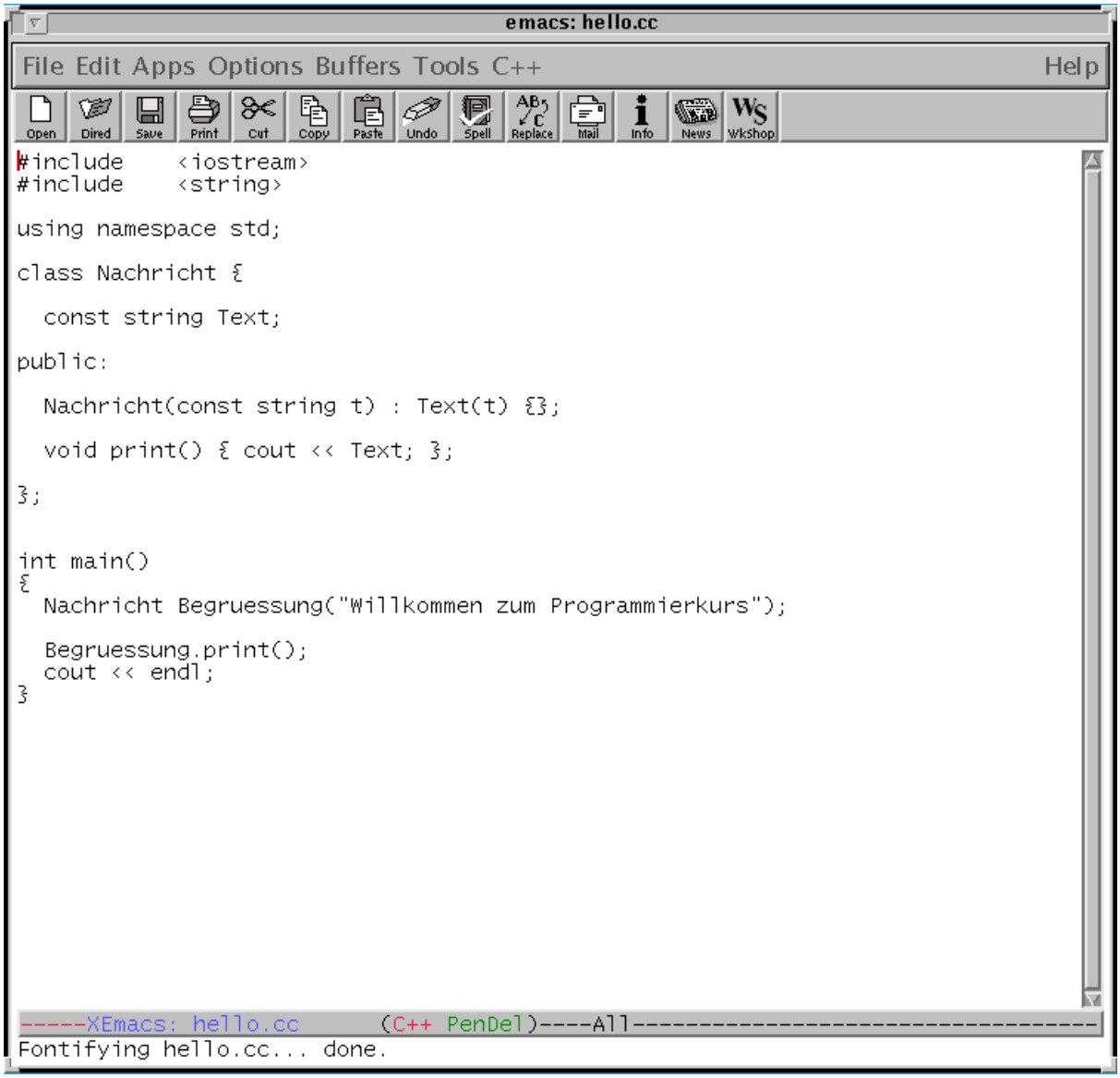
Auf den Sun-Workstations und auch unter LINUX steht Ihnen als Entwicklungstool der `xemacs` zur Verfügung:

Starten der Entwicklungsumgebung:

Im Commandtool



das `xemacs`-Kommando aufrufen und den Text eintippen:



Bemerkung:

a) `endl` steht für `"\n"` (end of line)

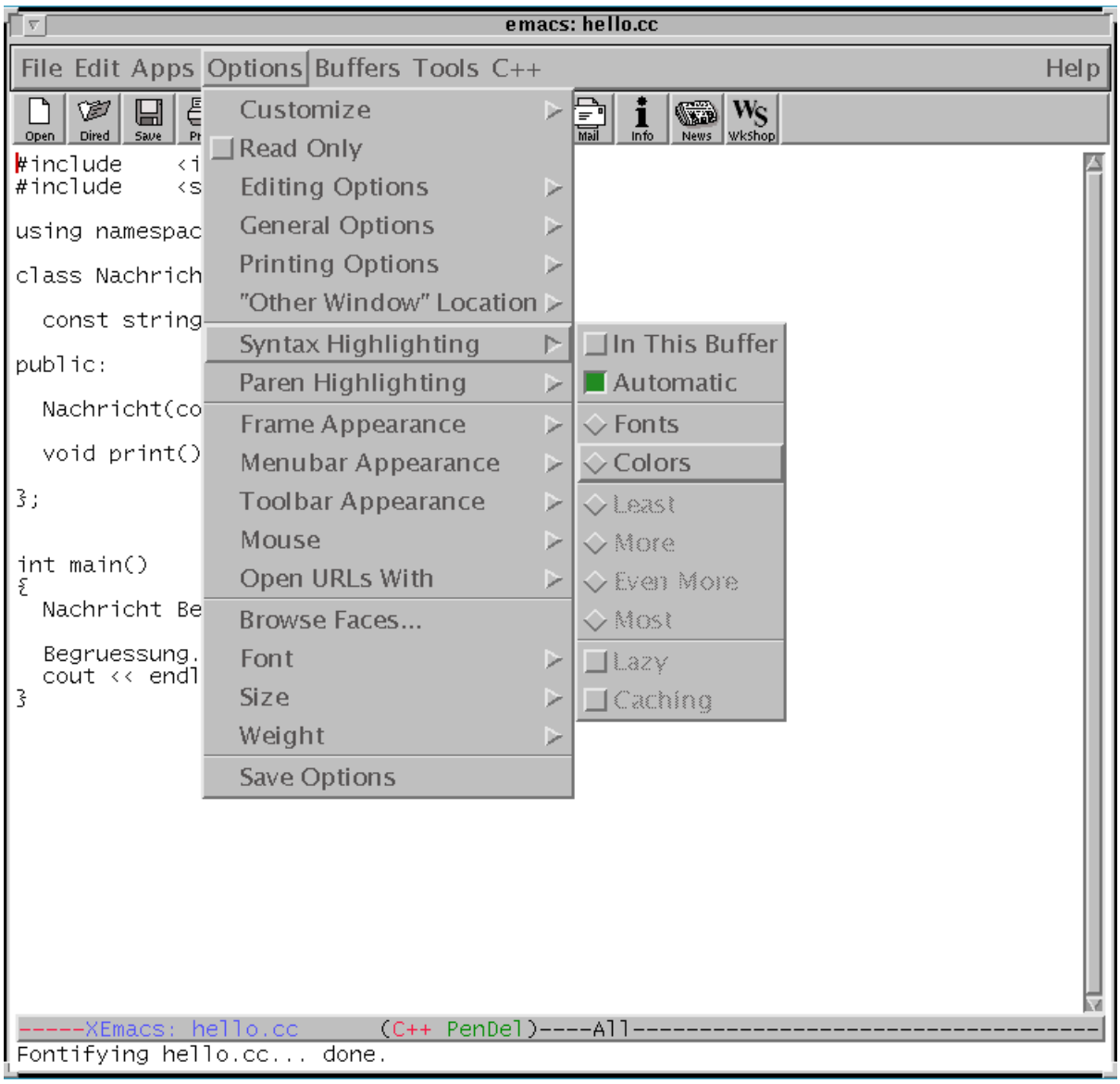
b) Innerhalb einer Klasse können die eigenen Attribute und Methoden ohne

```
Klassenname::Objektname
```

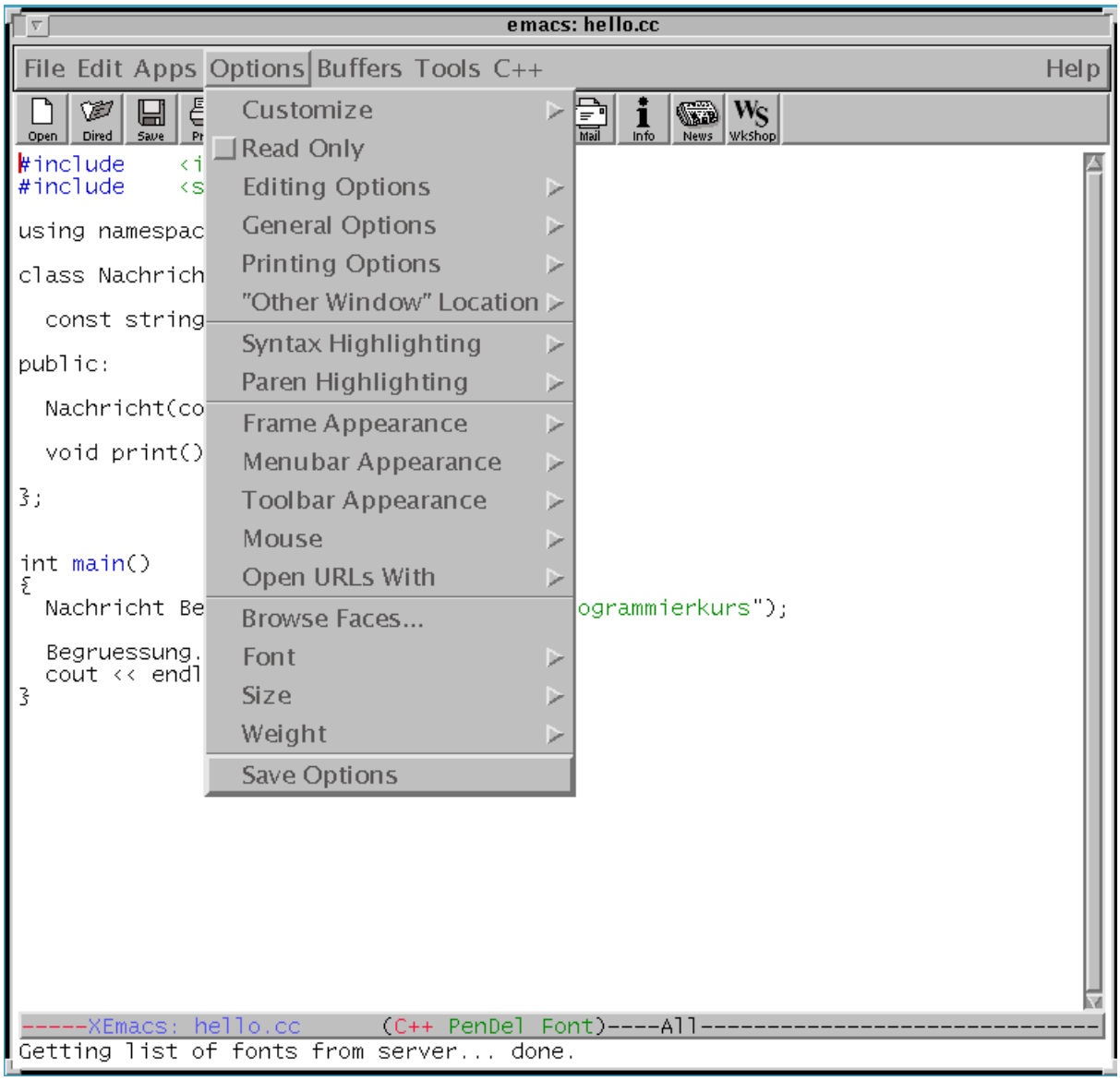
benutzt werden, während bei Trennung von Deklaration und Definition folgendermaßen vorgegangen werden muß:

```
class Nachricht {
   ...
   void print();
};

void Nachricht::print(){
  cout << Text;
};
```

Zuvor `Colors`

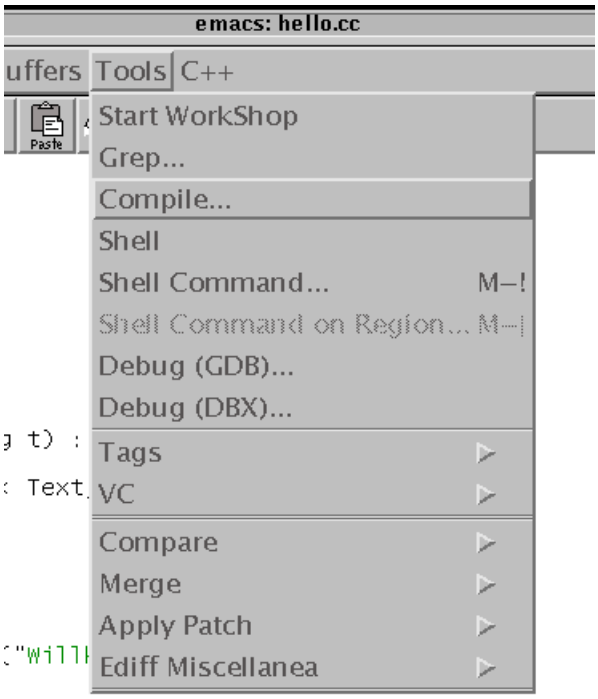


anklicken und permanent speichern:

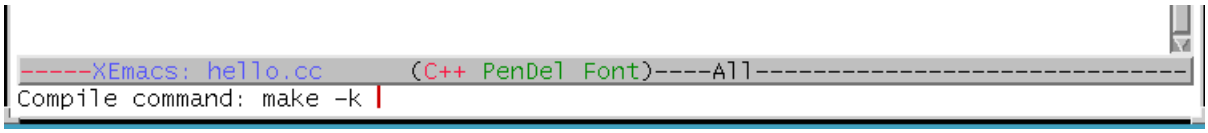


(Save Options)

Nun das Compile-Icon oder den Menüpunkt „Compile“ anwählen:



und die im Editor-Fenster unten erscheinenden Zeile

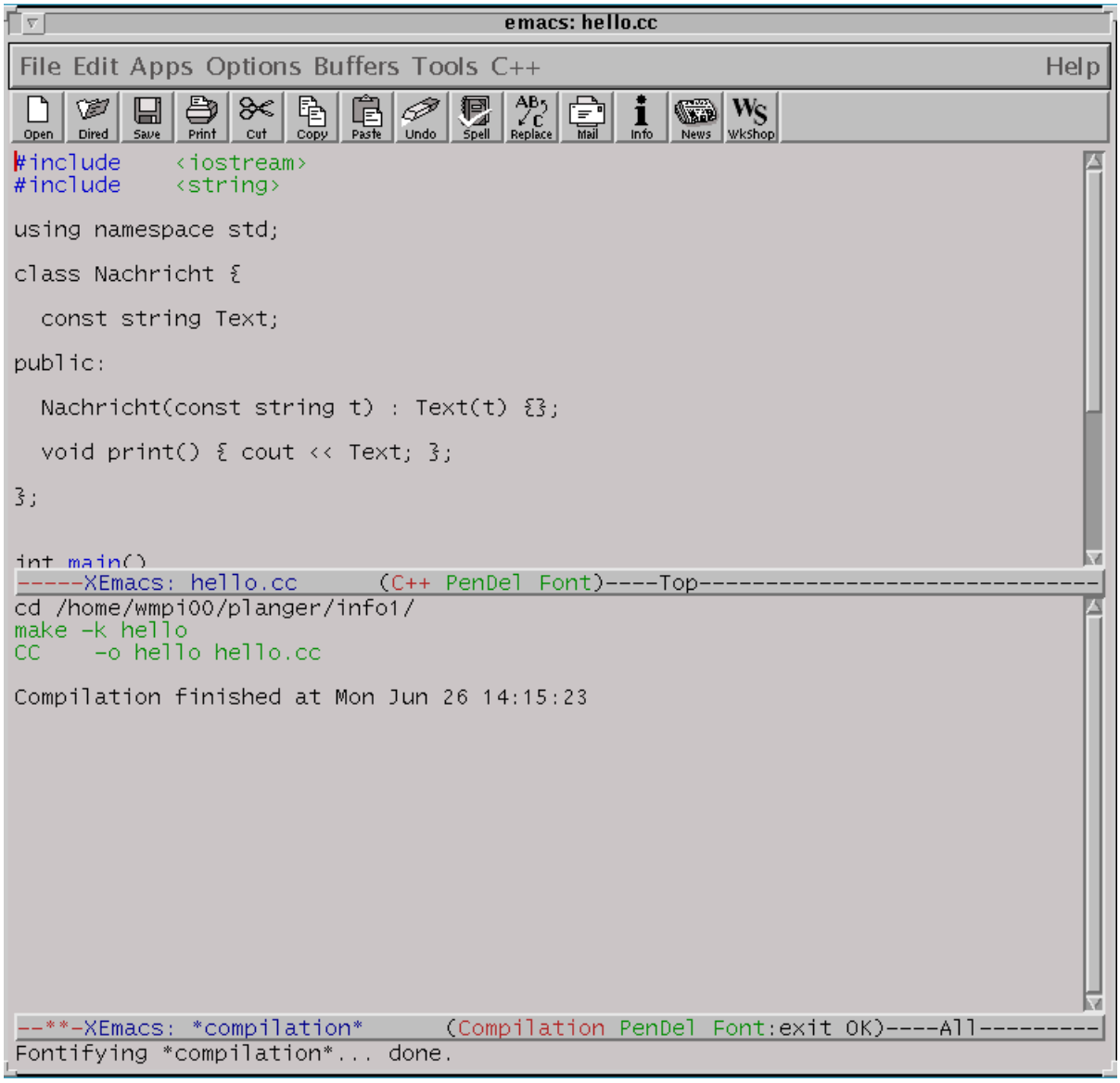


zu

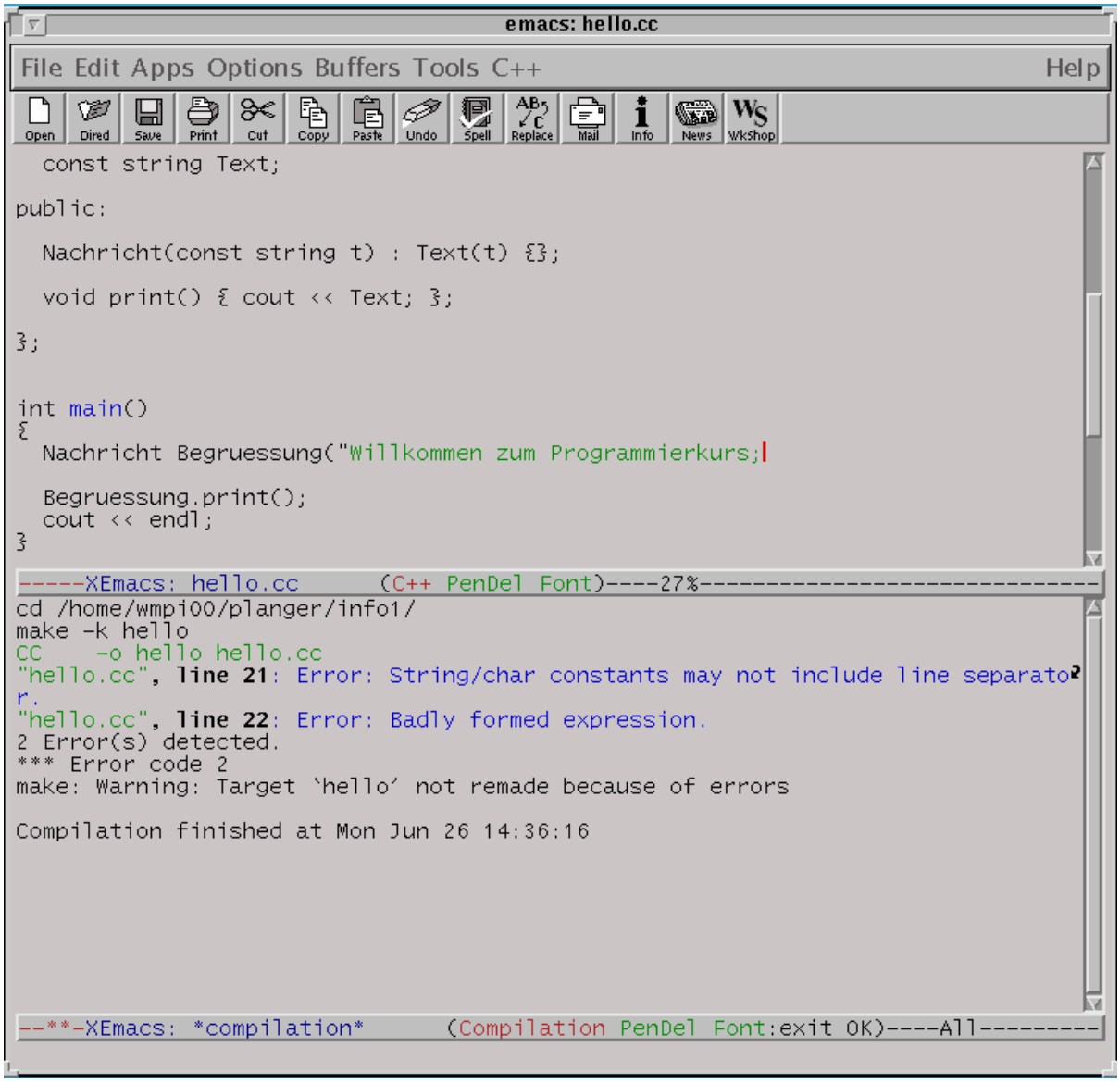
```
Compile command: make -k hello <cr>
```

ergänzen.

Wenn beim Übersetzen keine Fehler auftraten, wird die erfolgreiche Übersetzung in der unteren Hälfte des `xemacs`-Fensters mitgeteilt.



Eventuell auftretende Fehler oder Warnungen werden ebenfalls in der unteren Hälfte des `xemacs`-Fensters angezeigt:



Die Fehlermeldungen werden beim Überstreichen durch die Maus farbig hinterlegt (anklickbar) und Sie können bei Anklicken durch die mittlere Maustaste im Editorfenster direkt an die Fehlerstelle geführt werden.

Bemerkung: Folgefehler

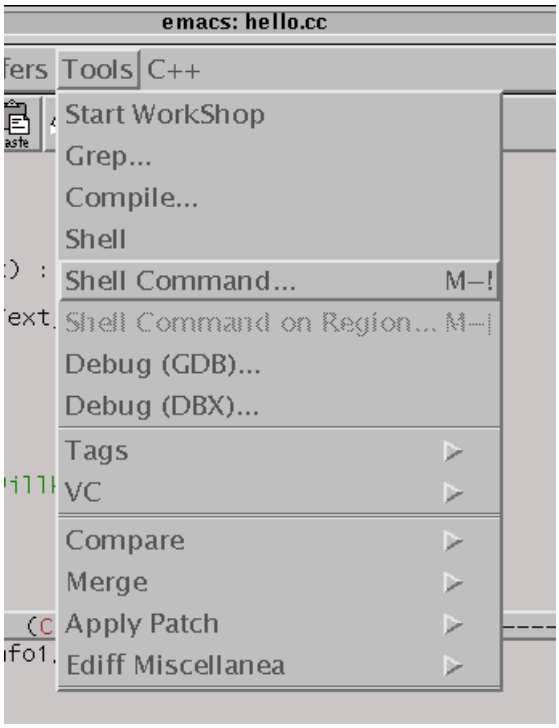
Sind keine Fehler mehr vorhanden, so kann das Programm gestartet werden:

a) In einem Commandtool mit

```
% ./hello <cr>
```

(Vgl. hierzu auch den Aufruf des `xemacs` (s.o.))

b) Im `xemacs`

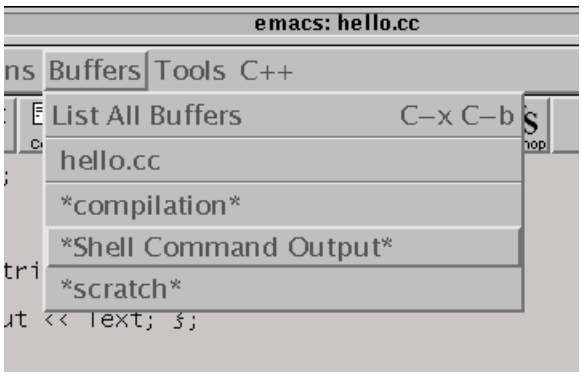


und ähnlich wie bereits oben gesehen die untere Zeile

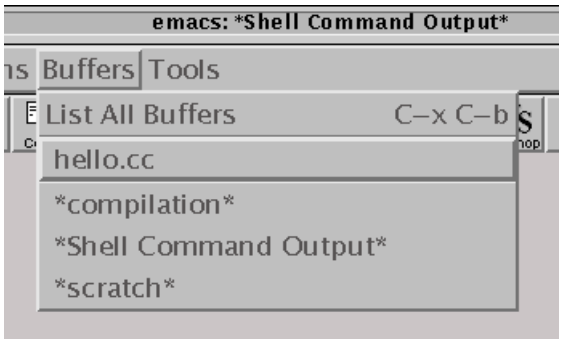
zu

```
Shell command: ./hello <cr>
```

ergänzen.
Die Ausgabe des Programmes kann man sich dann mittels



anschauen. Durch



kehrt man dann wieder zum Quelltext zurück.

Es wird häufig vorkommen, daß das Programm – obwohl es fehlerfrei übersetzt wurde – noch nicht einwandfrei bzw. wunschgemäß funktioniert. Um die noch vorhandenen logischen Fehler („Bugs“) zu beseitigen, wäre es wünschenswert, wenn man Zeile für Zeile nachvollziehen könnte, was das übersetzte Programm tut. Für diesen Zweck gibt es sogenannte *Debugger*.
Auf den Workstations steht uns eine integrierte `C++`-Entwicklungsumgebung, der `workshop`, zur Verfügung, den wir im folgenden exemplarisch benutzen wollen:

a) **Vorbereitung:**
Zu allererst muß man sicherstellen, daß für den Debugger-Betrieb die Environment-Variable `CPPFLAGS` richtig gesetzt ist. Dies erreicht man in einer `C`-Shell dadurch, daß man folgende Schritte ausführt:

- Im `commandtool`

  ```
  % xemacs ~/.cshrc& <cr>
  ```

  eingeben, um die Datei mit den Systemeinstellungen zu editieren.
- Gegebenenfalls noch die Zeile

  ```
  setenv CPPFLAGS -g
  ```

  ergänzen und die Datei abspeichern.

b) **Workshop starten:**
Es gibt 2 Möglichkeiten, `workshop` zu starten:

(a) Im gestarteten `xemacs` auf das `workshop`-Icon klicken.

(b) Im `commandtool`

```
% workshop&  <cr>
```

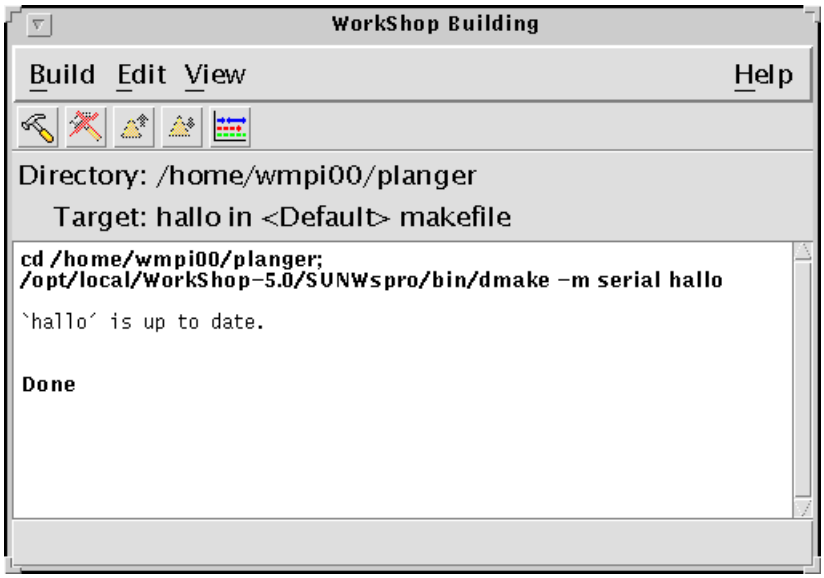
eingeben.

Es erscheint folgendes Fenster:

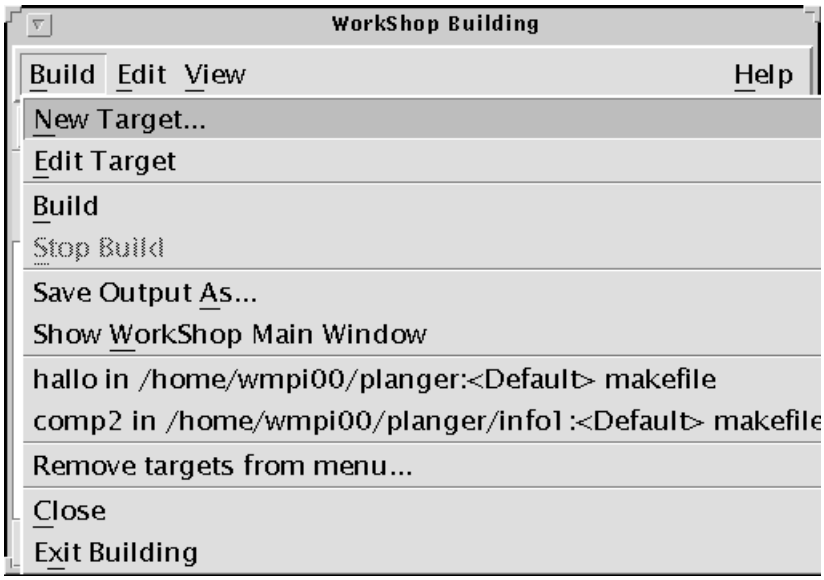
Hat man `workshop` nach der zweiten Methode gestartet (also im

`commandtool`), so kann man durch einen Klick auf das linke Icon in der `workshop`-Menüleiste eine Datei auswählen, die dann in einen neuen, von `workshop` gestarteten `xemacs` geladen wird.

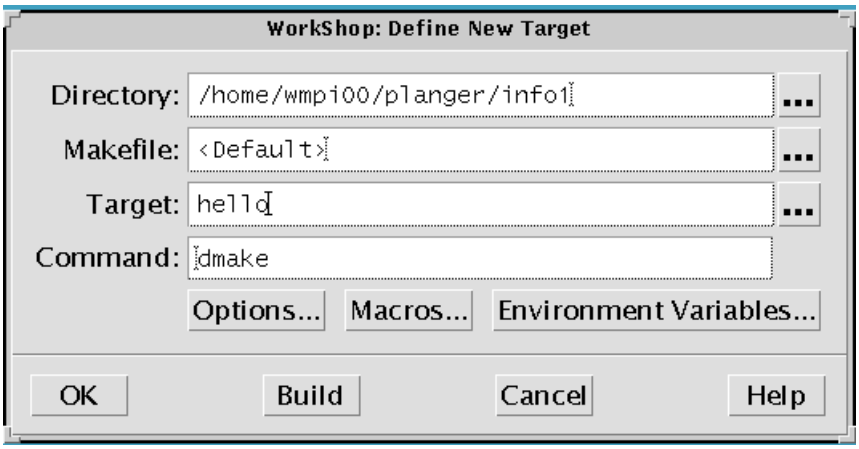
c) **Projekt übersetzen bzw. compilieren:**
Hierzu klickt man das zweite Icon von links an, woraufhin folgendes Fenster geöffnet wird:



Also wählt man im Menü `Build` den Punkt `New Target` an.



In dem nun erscheinenden Fenster trägt man in dem String-Feld `Target` den Namen der zu compilierenden Datei (ohne Endung `.cc`) ein, hier also z.B. `hello`:



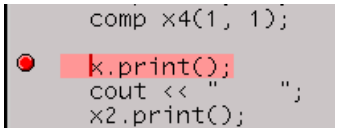
Nun kann man das Projekt compilieren. Dies geschieht durch einen Klick auf das `compile`-Icon entweder im `xemacs`-Fenster, im `workshop`-Fenster oder im zuletzt aufgerufenen `WorkShop Building`-Fenster. Natürlich kann man das Projekt auch auf die bereits bekannte Weise compilieren (s.o.).

d) **Projekt debuggen:**
Trat beim Compilieren kein Fehler auf, kann man nun den Debugger starten. Dies erfolgt durch einen Klick auf das `Debug`-Icon (drittes Icon von links im `workshop`-Fenster). Es erscheint nun das Debugger-Fenster auf dem Bildschirm. Der Debugger bietet folgende Möglichkeiten, den Ablauf eines Programmes nachzuvollziehen:

- Setzen von sogenannten *Breakpoints*
  Hierzu geht man wieder in das `xemacs`-Fenster und wählt im `main`-Block eine Zeile aus, indem man mit dem Cursor eine Zeile anklickt. Nun kann man den Breakpoint setzen, indem man das zweite Icon von links in der Werkzeugleiste des `xemacs` anklickt.

  Die gewählte Zeile wird nun im Editorfenster graphisch hervorgehoben:



Einen gesetzten Breakpoint kann man löschen, indem man wieder mit dem Cursor die Zeile des Breakpoints anwählt und in der Werkzeugleiste das Icon für das Löschen eines Breakpoints anwählt (viertes Icon von links).

- Ablauf des Programmes im Debugger
  Wenn man nun im Debugger-Fenster auf das `Start`-Icon klickt, wird das Programm bis zum Erreichen eines evtl. gesetzen Breakpoints „normal“ ausgeführt. An der Stelle des Breakpoints hält der Debugger die Abarbeitung des Programmes an. Auf diese Weise kann man „in aller Ruhe“ analysieren, was das Programm bis zu diesem Zeitpunkt bewirkt hat. Den bisherigen Output des Programmes kann man im Fenster mit dem Titel `WorkShop Program Input/Output` betrachten. Die aktuelle Zeile, an der sich das Programm gerade befindet, wird im `xemacs`-Fenster übrigens durch einen grünen Pfeil nach rechts gekennzeichnet. Will man die Abarbeitung des Programmes fortsetzen, so klickt man in der Werkzeugleiste des Debuggers auf das Icon mit dem grünen Pfeil nach unten. Das Programm läuft nun weiter bis zum nächsten gesetzten Breakpoint, u.s.w.
  Auf diese Weise (d.h. durch das Setzen von mehreren Breakpoints) kann man das Programm in mehrere übersichtliche Abschnitte unterteilen, deren Ablauf man leichter untersuchen kann.
  An besonders kritschen Stellen des Programmes kann man auch in den Einzelschrittbetrieb übergehen. Hierzu klickt man nach Erreichen eines Breakpoints (oder am Beginn des Programmes) in der Werkzeugleiste des Debuggers einfach auf das Icon mit dem grünen Pfeil nach rechts. Es wird nur eine einzige Zeile ausgeführt und dann sofort wieder angehalten. Durch Klick auf das Icon mit dem grünen Pfeil nach unten wird das Programm wieder „normal“ bis zum Erreichen des nächsten Breakpoints abgearbeitet.
  Hier noch ein Bild der Werkzeugleiste des Debuggers:

- Beobachten von Variablen und Ausdrücken:
  Häufig gewinnt man durch bloßes Setzen von Breakpoints noch keinen Aufschluß über eventuelle logische Fehler. Es wäre z.B. wünschenswert, wenn man das Programm nicht nur an Hand des Outputs beobachten könnte, sondern wenn man auch die Änderung einzelner Variablen nachvollziehen könnte. Auch hierfür stellt unser Debugger ein geeignetes Instrument zur Verfügung. Bearbeitet man beispielsweise ein Programm, in dessen `main`-Block die `integer`-Variable `x` vorkommt, so trägt man dies einfach in das `Expression`-Feld des Debuggers ein:

  Durch Klick auf den `Display`-Button öffnet sich ein Fenster mit dem Titel `WorkShop Data Display`. Wenn das Programm gerade nicht läuft, steht im Fenster `x = <not active>`.
  Startet man nun das Programm im Debugger (s.o.), kann man im `Display`-Fenster die Änderung von `x` im Programmverlauf nachvollziehen.

Unter Linux können Sie mit `kdevelop` (vgl. `khelpcenter`: Programm Handbücher, dann Entwicklung) ähnlich komfortabel Programme entwickeln.

1.37 Funktionssignaturen und Exceptions

Zu einer Funktionssignatur gehört eigentlich auch die Aufzählung aller Exceptions, die die Funktion werfen kann:

Listing 1.87: Exceptioninformationen in Funktionssignaturen

```
// ...
double f1(double x){
   // ...
};

double f2(double x) throw (std::bad_alloc){
   // ...
};

double f3(double x) throw (){
   // ...
};

double f4(double x) throw (std::bad_alloc, int){
   // ...
};

// ...
```

`f1()` kann hier alle möglichen Exceptions werfen, `f2()` nur die Exception `std::bad_alloc`, `f3()` kann keine Exception werfen, ...
Sollte zum Beispiel in `f3()` oder einer von `f3()` aufgerufenen Funktion eine nicht in `f3()` abgefangene Exception erzeugt werden, so wird diese nicht die Aufrufhierarchie hinauf zurückgereicht, sondern die Funktion `std::unexpected()` aufgerufen.
Näheres zur sinnvollen Benutzung dieser Möglichkeiten lesen Sie etwa im C++-Standard (Kapitel 18.6) oder z.B. in „Scott Meyers: More Effective C++, Addison-Wesley, 1996, Item 72“ nach.

1.38 Template-Verbunde und Traits

Sie können Template-Verbunde benutzen, um etwa in

Listing 1.88: Template-Verbunde

```
// ...
template<class T>
structure Point{
  T x;
  T y;
};
// ...
```

den Datentyp `Point` für beliebige Skalarbereiche (`int, long int, float, double, ...`) bereitzustellen.

Der eingebaute Datentyp `complex<T>` ist ein solcher generischer Datentyp, in dem Sie den Datentyp für Real- und Imaginärteil frei wählen können.

Da in Strukturen auch `typedef`'s erlaubt sind kann man unter Zuhilfenahme von Template-Verbunden eine Abbildung von Typen auf andere Typen realisieren (sogenannte Traits):
Nehmen wir an, Sie wollen eine Template-Funktion `average()` zur Berechnung des arithmetischen Mittels der Komponenten eines Feldes schreiben. Je nach Komponententyp ist es dann sinnvoll, den Mittelwert entweder mit `float`- oder mit `double`-Arithmetik zu berechnen, nicht jedoch mit `int`-Arithmetik:

Komponententyp	*Arithmetiktyp*
char	float
int	float
long int	double
float	float
double	double
complex<float>	complex<double>

Realisiert werden kann das folgendermaßen:

Listing 1.89: Traits

```
#include <iostream>
#include <iomanip>
#include <complex>
using namespace std;
    //                          default : float arithmetic
template<class T>
struct arithmetic_trait {
```

```
  typedef float T_arithmetic;
};
    //                                  Sonderfaelle durch Spezialisierung
template<>
struct arithmetic_trait<long int> {
  typedef double T_arithmetic;
};
template<>
struct arithmetic_trait<double>{
  typedef double T_arithmetic;
};
template<>
struct arithmetic_trait< complex<double> >{
                                  // Vorsicht: nicht ... complex<double>>
  typedef complex<double> T_arithmetic;
};
    //                                  generische Funktion average()
template<class T>
typename arithmetic_trait<T>::T_arithmetic
                                average(const T data[], int numElements){
  typename arithmetic_trait<T>::T_arithmetic sum=0.0;
  for (int i=0; i < numElements; i++)
    sum += data[i];
  return sum / (double)numElements;
};

int main(){
 char v [] = {127, 127, 127, 127, 127};
 float f = average(v, 5);
 cout << setiosflags(ios::showpoint) << f << endl;

 complex<double> w[] = {complex<double>(1,1), complex<double>(3,2)};
 complex<double> g = average(w, 2);
 cout << g << endl;
}
```

1.39 Template-Verbunde als bessere Implementierung von Compiletime Assertions

Der spezialisierte Template-Verbund

Listing 1.90: Compile time assertion vs. 2

```
//...
template<bool> struct CompileTimeError;
template <> struct CompileTimeError<true> {};

#define CT_ASSERT(expr) (CompileTimeError < (expr) != 0 > ())
// ...
  CT_ASSERT(LDBL_DIG > DBL_DIG);
```

kann ebenfalls für Zusicherungen zur Übersetzungszeit eines Programmes benutzt werden (vergleiche auch Abschnitt 1.23).
Statt der Fehlermeldung

```
"test1.cc", line 35: Error: The type "CompileTimeError<0>"
                     is incomplete.
```

im Falle der Verletzung der Zusicherung kann auch etwas Benutzerfreundlicheres als Fehlermeldung erzeugt werden

```
"test2.cc", line 51: Error: Cannot cast from ERROR_No_Higher_Accuracy
                     to CompileTimeChecker<0>.
```

falls Sie

Listing 1.91: Compile time assertion vs. 3

```
// ...
template<bool> struct CompileTimeChecker
{
        CompileTimeChecker(...);
};
template <> struct CompileTimeChecker<false> {};

#define CT_ASSERT(expr, msg)\
  {\
    class ERROR_##msg {};\
    ( void)sizeof(CompileTimeChecker< (expr) != 0>((ERROR_##msg())));\
  }
// ...
  CT_ASSERT(LDBL_DIG > DBL_DIG, No_Higher_Accuracy);
// ...
```

benutzen.

Die hier dargestellte Version von Compiletime-Assertions funktioniert im Gegensatz zu denen aus Abschnitt (1.23) **auch** innerhalb von rekursiv sich selbst aufrufenden Templatefunktionen:

Listing 1.92: Compiletime-Fehlermeldung in rekursiven Templates vs. 2

```
// ...
template<int k>
long int faku(){
    CT_ASSERT(k >= 0, Domain_Faku);
    return faku<k-1>() * k;
};

template<> long int faku<0>(){
    return 1;
};
// ...
```

Kapitel 2

Abstrakte Datentypen in C++ (Klassenkonzept)

In C++ können Sie für eigene Datentypen (die Sie als Verbund `struct` beziehungsweise `class` implementieren) auch all die Operatoren `+, -, *, /, <<, >>, ...` definieren, die Ihnen bei den eingebauten Datentypen `int, ...` zur Verfügung stehen:

Listing 2.1: useint.cc

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

int main(){
  int g1;
  cout << g1 << endl;          // undefined
  int g2(g1);
  int g3;
  g3 = g1;
  int* pg;
  pg = &g1;

  int g4(-7.1);                // liefert -7
  cout << g4 << endl;

  int g6;
  cout << endl;
  cout << "Bitte einen int-wert eingeben: ";
  cin >> g6;
  cout << g6 << endl;          // 66.8 liefert 66 ohne Warnung

  g6 = -5.25;
  cout << g6 << endl;          // liefert -5

  int g7 (-5.25);              // analog
  cout << g7 << endl;

  g7 = g7+g7;
  cout << g7 << endl;                  // +: int x int -> int

  if (g7 == g7)
```

```
    cout << "gleich" << endl;
  else
   cout << "ungleich" << endl;

  g7 = g6 * 3;
  cout << g7 << endl;

  g7 = (-4) * g6;
  cout << g7 << endl;

  int g8(-5.12);
  cout << endl;
  cout << g8 << endl;
  double gn1(g8);
  cout << gn1 << endl;
}
/*************************************************************************
g++:
useDouble.cc: In function `int main()':
useDouble.cc:15: warning: initialization to `int' from `double'
useDouble.cc:24: warning: assignment to `int' from `double'
useDouble.cc:27: warning: initialization to `int' from `double'
useDouble.cc:44: warning: initialization to `int' from `double'
---
CC:
"useDouble.cc", line 8: Warning: The variable g1 has not yet been assigned a value.
1 Warning(s) detected.
*************************************************************************/
```

Beachten Sie jedoch die zuweilen unangenehmen Auswirkungen der automatischen Typkonversion.

Hier ein Beispiel für einen solchen eigenen Datentyp namens `AltGrad`. Zunächst ein Projekt-Makefile:

Listing 2.2: AltGrad Makefile

```
# ########## Makefile ##################################
# CXXFLAGS=-g -DNDEBUG
CXXFLAGS=-g

#CXX=g++
CXX=CC

CCC=$(CXX)
CPPFLAGS=$(CXXFLAGS)

useGrad: useGrad.cc Grad.o
	$(CCC) $(CXXFLAGS) -o useGrad useGrad.cc Grad.o
clean:
	rm -f *.o *~ useGrad
depend:
	makedepend useGrad.cc Grad.cc

# DO NOT DELETE

useGrad.o: Grad.h
Grad.o: Grad.h
```

Dieses `Makefile` ist sowohl auf den Sun-Workstations mit `CC` und `g++` als auch auf den LINUX-Rechnern mit `g++` benutzbar und soll als Beispiel für plattformunabhängige Programmentwicklung dienen.

Die Definitionen der Datenfelder und der Operationen für den Datentyp `AltGrad` sowie `Rad`:

Listing 2.3: Altgrad.h

```
//
// Datei: Grad.h
//

#ifndef GRAD_MY
class Altgrad{
  long int voll;
  long int minuten;   // 0   <= minuten  < 60
  double   sekunden;  // 0.0 <= sekunden < 60.0

                                                  // Problem:
                                                  // auf diese Art sind alle
                                                  // Winkel w mit
                                                  //    -1 Grad < w < 0 Grad
                                                  // n i c h t    darstellbar!
  void Normalize();
 public:
  Altgrad(long int v = 0,
          long int m = 0,
          double s = 0.0); // Vollgrad/Minuten/Sekunden:
                                                   //       m >= 0, s >= 0

  operator double() const  ;                      // Typkonversion zu double
  double get_sekunden() const  ;                  // Observatoren
   // ...
  void set_sekunden(double s);  /* s >= 0.0 */  // Modifikatoren

  friend std::ostream& operator<<(std::ostream& o, const Altgrad g);
                                                 // Ausgabeoperator (Observator))
  friend std::istream& operator>>(std::istream& i, Altgrad& g);
                                                 // Eingabeoperator (Modifikator)
  // Altgrad& operator=(double d  );                //   Wertzuweisungsoperator
  Altgrad(double d    );                          // Konstruktor fuer
                                                  // double-Argumente
  bool operator==(const Altgrad& g) const;      // Wertevergleich
  friend Altgrad operator*(const Altgrad& g, long int i);
  friend class Neugrad;
};
class Rad{
  double winkel;
 public:
  Rad(double r);
  Rad(const Altgrad& ag);
  friend std::ostream& operator<<(std::ostream& o,
                                  const Rad g);
                                          //  Ausgabeoperator (Observator));
   // ...
};
#define GRAD_MY
#endif
```

Nun die Implementierung:

Listing 2.4: Altgrad.cc

```
//
// Datei: Grad.cc
//
#include <iostream>
#include <iomanip>
#include <exception>
#include <cassert>
#include <cmath>
#include "Grad.h"
using namespace std;

double sign(double d){
  if (d >= 0.0)
    return 1.0;
  else
    return -1.0;
}

Altgrad::Altgrad(long int v, long int m, double s){
  if ((m < 0)||(s < 0.0)) throw
                "  Altgrad: Minuten oder Sekunden haben unerlaubte Werte!";
   long int uebertrag = static_cast<long int>(s) / 60;
  sekunden = s - uebertrag * 60;
  long int m0 = m;
  m += uebertrag;
  assert((0 <= sekunden)&&(sekunden < 60));

  minuten = m % 60;
  assert((0 <= minuten)&&(minuten < 60));

  if (v >= 0)
    voll = v + (m / 60);
  else
    voll = v - (m / 60);

  assert ( ((v*60+sign(v)*m0)*60+sign(v)*s)==
          ((  voll*60+sign(voll)*minuten)*60+sign(voll)*sekunden) );
#ifndef NDEBUG
  cerr << "*****" << voll << " Grad " << minuten << " Min "
       << sekunden << " Sek " << endl;
#endif
};

Altgrad::operator double() const {
  return voll+sign(voll)*(sekunden/60+minuten)/60;
};

double Altgrad::get_sekunden() const {
  return sekunden;
};

void Altgrad::set_sekunden(double s){
  if (s < 0.0) throw "Altgrad: sekunden mit unerlaubtem Wert";
  sekunden = s;
  this->Normalize();
  // geeignetes assert!
};
```

```
void Altgrad::Normalize    (){                          // private Hilfsmethode
   long int uebertrag = static_cast<long int>(sekunden) / 60;
  sekunden -= uebertrag * 60;
  minuten  += uebertrag;
  assert((0 <= sekunden)&&(sekunden < 60));

  uebertrag = minuten / 60;
  minuten = minuten % 60;
  assert((0 <= minuten)&&(minuten < 60));

  if ( voll >= 0)
    voll += uebertrag;
  else
    voll -= uebertrag;
  // geeignetes assert!
};

ostream& operator<<(ostream& o, const Altgrad g){
  o << g.voll << " Altgrad " << g.minuten << " Minuten "
    << g.sekunden << " Sekunden ";
  return o;
};

istream& operator>>(istream& i, Altgrad& g){
  i >> g.voll >> g.minuten >> g.sekunden;
  if ((g.minuten < 0)||(g.sekunden < 0)) throw
              "  Altgrad >>- Operator: Minuten oder Sekunden negativ";
  g.Normalize();
                                  // besser: zeilen- oder zeichenweise Eingabe
                                  // mit geeigneter Exceptionerzeugung und
                                                 // -behandlung
  return i;
};

/*
Altgrad& Altgrad::operator=(double d){
   voll = static_cast<long int>(d);
  double Rest = abs(d - voll);
  Rest *= 60;
  minuten = static_cast<long int>(Rest);
  sekunden = (Rest - minuten)*60;
  return *this;
};
*/

Altgrad::Altgrad(double d){
   voll = static_cast<long int>(d);
  double Rest = abs(d - voll);
  Rest *= 60;
  minuten = static_cast<long int>(Rest);
  sekunden = (Rest - minuten)*60;
};

 //             eigenen Operator +: Altgrad x Altgrad -> Altgrad als Aufgabe

bool Altgrad::operator== (const Altgrad& g) const {
  return ((voll == g.voll)&&(minuten == g.minuten)&&(sekunden == g.sekunden));
};

/*
long int abs(long int i){                // Workaround fuer g++ Fehler
   if (i >= 0)                           // Bitte nur bei Benutzung von g++
```

```
    return i;                                  // entkommentieren!
  else
    return -i;
};
*/

Altgrad operator*(const Altgrad& g, long int i){
  Altgrad Hilf(g);
  Hilf.voll *= i;
  Hilf.minuten *= abs(i);
  Hilf.sekunden *= abs(i);
  Hilf.Normalize();
  return Hilf;
};

/* ------------------------------------------------------------------------

Rad::Rad(double r) : winkel(r) {};

Rad::Rad(const Altgrad& ag){
  winkel = static_cast<double>(ag) * M_PI / 180.0;
};

ostream& operator<<(ostream& o, const Rad g){
  o <<  setiosflags(ios::scientific) << setprecision(16)
    << g.winkel << " Radiant " ;
  return o;
};
```

Und schließlich das Testrahmenprogramm (die Applikation):

Listing 2.5: useGrad.cc

```
#include <iostream>
#include "Grad.h"

using namespace std;

int main(){
  Altgrad g1;
  Altgrad g2;
  Altgrad g3;
  g3 = g1;
  Altgrad* pg;
  pg = &g1;

  Altgrad g4(-7.1, 90, 119.999999999999999);
  Altgrad g5(-3, 0, 3601);
  Altgrad h1(-4, 15);

  cout << g4 << endl;
  cout << g5 << endl;
  cout << h1 << endl;

  cout << g5.get_sekunden() << endl;

  Altgrad g6(4,59,0);
  cout << g6 << endl;
  g6.set_sekunden(121);
  cout << g6 << endl;

  // cout << g6 << endl;                     // jetzt  mit neuem <<
```

```
30
   cout << endl;
   cout << "Bitte einen Altgradwert eingeben: ";
   // cin  >> g6;
   cout << g6 << endl;
35
   g6 = -5.25;
   cout << g6 << endl;   // liefert  15,0,0 !!!

   Altgrad g7(-5.25);    // Zuweisung ok, aber Konstruktor noch falsch
40 cout << g7 << endl;

                         // nach Konstruktor ist Zuweisungsop unnoetig!
   g7 = g7+g7;
   cout << g7 << endl;   // +: double x double -> double
45
   if (g7 == g7)
     cout << "gleich" << endl;
   else
    cout << "ungleich" << endl;
50
   g7 = g6 * 3;
   cout << g7 << endl;
   g7 = g6 * (-3);
   cout << g7 << endl;
55
   g7 = (-4) * g6;
   cout << g7 << endl;

   Altgrad g8(90.0);
60 cout << endl;

   cout << g8 << endl;
   Rad gn1(g8);
   cout << gn1 << endl;
65
   Altgrad t1(0,10,25);   // falsches Ergebnis!!!
   g7 = t1 * (-1);
   cout << g7 << endl;

70 }
```

Wie Sie sehen, unterscheidet sich die Operatorbenutzung nicht von derjenigen des Datentyps `double`. Auch unschöne (eigentlich nicht wünschenswerte) Effekte der automatischen Typkonversion sollten Sie dazu veranlassen, die Kombination erwünschter Operationen bei eigenen Datentypen sorgfältig zu planen!

Zusammenfassung: Ein abstrakter Datentyp fasst eine Menge von Objekten und die mit diesen Objekten möglichen Operationen zusammen. Das Klassenkonzept (Wortsymbole `struct` bzw. `class`) in `C++` erlaubt die Realisierung solcher abstrakter Datentypen.

Beispiel 2.1
Abstrakter Datentyp „complex“:
Paare von reellen Zahlen
und
arithmetische Operationen für diese Paare.
Weitere typische Operationen (für Realisierung auf einer Rechneranlage):
- Zugriff auf Real- und Imaginärteil
- Anlegen einer Kopie
- Zuweisung
- Ein-/Ausgabe
- Funktionen wie sin(), exp(), ... für solche Paare
- Typkonversion: reell nach komplex

Beispiel 2.2
Mitarbeiter eines Betriebes, abstrakter Datentyp „mitarbeiter“:
Karteikarten mit Einträgen
- Name, Vorname
- Anschrift
- Geschlecht
- Familienstand
- Anzahl der Kinder
- Alter
- Mitarbeiter seit
- Stellung im Betrieb (Lehrling, Arbeiter, Manager, ...)
- Gehalt
- Resturlaub
- zu vergütende Überstunden

und Operationen
- Neue Karteikarte anlegen
- Karteikarte entfernen
- Karteikarten sortieren (nach Name, nach Alter, nach Wohnort, ...)
- Karteikarteneinträge aktualisieren
- ...

Weitere typische Operationen (für eine Realisierung auf einer Rechneranlage):
- Anlegen einer Kopie
- Zuweisung
- Ein-/Ausgabe
- Zugriff auf Einträge wie Alter, Gehalt, ...
- Erstellen einer Verteilers mit den Anschriften aller Mitarbeiter.

Zum ersten Datentyp wollen wir eine Implementierung betrachten:

Listing 2.6: Klasse complex I

```
//Realisierung "complex":

#include <iostream>
using namespace std;

class complex {// Klassendefinition
 private :      //  Zugriffsspezifizierung
                // Datenmember, auch Attribute genannt
   double re;  // Realteil
   double im;  // Imaginaerteil
 public :       // Auch ausserhalb  des Klassenrumpfes ansprechbare Memberfunktionen
   void setRe(const double x) { re= x; return; }
   void setIm(const double x) { im= x; return; }

   double Re() const { return re; } // implizit vorhandenes Argument ist const
   double Im() const { return im; }

   complex add(complex z){
     re+= z.re; // ausfuehrlich : (* this ). re+= z.re;
     im+=z.re; //                 (* this ). im+=z.im;
     return *this;
   }
}; // Strichpunkt am Ende der Klassendefinition ist notwendig!

int main(){
   complex w, r;
   w.setRe(1.3); // Aufruf einer  Memberfunktion: Objektname.f(...)
   // w.re= 1.3; // Falsch , da Attribut re  mit private-Zugriff
                 // Zugriff  ausserhalb des Klassenrumpfes nicht moeglich
                 // Keine versehentliche  Aenderung (Programmsicherheit)
   cout << w.Re() << endl;
   return 0;
}
```

Der Zugriff (also lesend sowie schreibend) auf die Klassenattribute ist nur mittels Memberfunktionsaufrufen möglich. Die tatsächliche Realisierung der Attribute (z.B. als zwei `double` Komponenten oder z.B. als Feld mit zwei Komponenten) wird vor dem Benutzer versteckt (`information hiding`) und ist für die öffentlichen Daten- und Funktionsmember bestimmt. Sie bilden die Benutzerschnittstelle.

Memberfunktionen wie z.B. `setRe()` haben implizit als erstes Argument immer einen Zeiger auf dasjenige Objekt, für das die Funktion aufgerufen wird. Dieser Zeiger heißt „`this`“. Es ist beim Aufruf einer Memberfunktion also immer klar, für welches Objekt diese aufgerufen wird. Die Definition von `setRe()` lautet demnach also ausführlicher

```
void setRe (double x) {  (*this).re= x;  }
```

Beim Aufruf „`w.setRe(1.0);`“ ist der implizit an `setRe` übergebene `this`-Zeiger ein Zeiger auf das Objekt `w`.

Eine Memberfunktion zur Addition zweier komplexer Zahlen könnte dann folgendermaßen aussehen:

```
complex add(complex z){
  re+= z.re; // ausfuehrlich :  (* this ). re+= z.re; bzw. (*this ). re = (*this ). re+z.re;
  im+= z.im;
}
```

Sind nun `w`, `r` Objekte der Klasse `complex`, so kann diese Funktion mit

```
w.add(r );  // r  wird zu w addiert und das Ergebnis in w gespeichert
```

aufgerufen werden.

Für die Realisierung von Operatoren stehen unterschiedliche Möglichkeiten zur Verfügung

a) **Memberfunktion:**
Ermöglicht den Zugriff auf `private`-Elemente, ein `this`-Pointer ist implizit vorhanden.

b) **`friend`-Funktion:**
Ermöglicht den Zugriff auf `private`-Elemente, es ist kein `this`-Pointer implizit vorhanden.

c) **globale Funktion:**
Ermöglicht Zugriffe nur über die `public`-Schnittstelle (d.h. die `public`-Attribute und `public`-Funktionen) der Klasse.

In `C++` werden

Operatoren auf Funktionen zurückgeführt.

Also entspricht einem Operator `*` eine Funktion mit dem Namen „`operator*`“. Es sind dann zwei äquivalente Aufrufarten möglich (als Funktion oder über Operatorschreibweise). Dies gibt aber auch die Möglichkeit, den Operator zu überladen.

Wir wollen eine erweiterte Realisierung des abstrakten Datentyps „`complex`“ betrachten:

Listing 2.7: Klasse complex II

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

class complex {// Klassendefinition
 private :       //  Zugriffsspezifizierung
                 // Datenmember, auch Attribute genannt
   double re;  // Realteil
   double im;  // Imaginaerteil
 public :        //  oeffentliche   Schnittstelle
   void setRe(const double x) { re= x; return; }
   void setIm(const double x) { im= x; return; }

   double Re() const { return re; }
   double Im() const { return im; }

   complex operator+(complex z) // Memberfunktion
   {
     complex c;
     c.re= re+z.re;  // ausfuehrlich : c.re= (*this).re+ z.re;
     c.im= im+z.im;  //               c.im= (*this).im+ z.im;
     return c;
   }
   friend complex operator-(complex a, complex b); // friend-Funktion
}; // Strichpunkt am Ende der Klassendefinition ist notwendig!

complex operator-(complex a, complex b) // Definition der friend-Funktion
{
  complex c;
  c.re= a.re-b.re;  // friend-Funktion darf auf die private-Daten re und im zugreifen
  c.im= a.im-b.im;
  return c;
}

complex operator*(complex a, complex b) // globale Funktion
{
  complex c; // Globale Funktionen haben keinen Zugriff auf die private-Elemente
  c.setRe(a.Re()*b.Re()-a.Im()*b.Im());
  c.setIm(a.Re()*b.Im()+a.Im()*b.Re());
  return c;
}


int main(){
   complex w, r;
   w+r; // wird automatisch vom Compiler umgesetzt in w.operator+(r);
   w-r; // aequivalent zu operator-(w,r);
   w*r; // aequivalent zu operator*(w,r);
/* Nicht funktionieren wuerden
  3+w; // wuerde 3.operator+(w) entsprechen
   w+3; // wuerde w.operator+(3) entsprechen
  3*w; // waere Fehler: no match for'int * complex &'
   w*3; // s.o., aequivalent zu operator* (w,3);
*/
   return 0;
}
```

Operatoren können also überladen werden (im Wesentlichen wie Funktionen). Es können nur Operatorsymbole überladen werden, die in C++ vorhanden sind. Selbstdefinierte Operatorensymbole sind also nicht möglich. Die Operatorpriorität ist nicht änderbar (d.h. * vor + gilt immer)!

Listing 2.8: Klasse complex III

```
/*

Konstruktoren, Referenzuebergabe, Verwendung von const,
Operatoren fuer Zuweisung, Prae- und Postinkrement, Ein-/Ausgabe

*/

#include <iostream>
using namespace std;

class complex { // Klassendefinition
 public:
   // Konstruktur zur Erzeugung und Initialisierung von Klassenobjekten:
   complex(const double r=0, const double i=0) { re= r; im= i; return; }
   // Kopierkonstruktor
   complex(const complex& z) { re= z.re; im= z.im; return; }

   void setRe(const double x) { re= x; return; }
   void setIm(const double x) { im= x; return; }

   double Re() const { return re; } // implizit vorhandenes Argument ist const
   double Im() const { return im; }

   // Arithmetische Operatoren alle global!

   // Zuweisung:
   complex& operator=(const complex& w) // Elementfunktion mit einem Argument
   {
       if (this==&w) return *this; // es ist dann nichts zu tun
       re= w.re;
       im= w.im;
       return *this;
   }
   // Weiterer Zuweisungsoperator:
   complex& operator+=(const complex& w) // Elementfunktion mit einem Argument
   {
       re+= w.re;
       im+= w.im;
       return *this;
   }

   // Praefixform: erhoehen und holen
   complex& operator++() { (*this).re+= 1; return *this; }

   // Postfixform: holen und erhoehen
   const complex operator++(int) // int-Arg. nur aus syntaktischen Gruenden
   {
       complex oldValue= *this;
     ++(*this); // verwende Praefixform
       return oldValue;
   }

   // Gleichheit:
```

```
    bool operator==(const complex& w) { return (re==w.re) && (im== w.im); }

    // Ein-/Ausgabeoperatoren:
    friend istream& operator>>(istream& is, complex& w)
    {
       is >> w.re >> w.im;
       return is;
    }

    friend ostream& operator<<(ostream& os, const complex& w)
    {
       os << w.re << " + " << w.im << "*i" << endl;
       return os;
    }
 private: // Datenmember
    double re; // Realteil
    double im; // Imaginaerteil
}; // class complex

complex operator+(const complex& a, const complex& b){
  complex c;
  c.setRe( a.Re() + b.Re() );
  c.setIm( a.Im() + b.Im() );
  return c;
}

complex operator-(const complex& a, const complex& b){
  complex c;
  c.setRe( a.Re() - b.Re() );
  c.setIm( a.Im() - b.Im() );
  return c;
}

complex operator*(const complex& a, const complex &b)
{
  complex c;
  c.setRe( a.Re()*b.Re() - a.Im()*b.Im() );
  c.setIm( a.Re()*b.Im() + a.Im()*b.Re() );
  return c;
}

complex operator/(const complex& a, const complex &b)
{
  complex c;
  double h= b.Re()*b.Re() + b.Im()*b.Im(); // Nenner
  c.setRe( ( a.Re()*b.Re() + a.Im()*b.Im())/h );
  c.setIm( (-a.Re()*b.Im() + a.Im()*b.Re())/h );
  return c;
}

int main(){
  complex w(1,2), z(3,4);

  z++; // aequivalent zu z.operator++(0);
 ++z; // aequivalent zu z.operator++();

 ++++z; // aequivalent zu (z.operator++()).operator++();
  // z++++; geht nicht, da z.. kein lvalue  ist

  cin >> w >> z;  // aequivalent zu operator>>(operator>>(cin,w), z);
  cout << w << z; // aequivalent zu operator<<(operator<<(cout,w), z);
  cout << "w: " << w;
```

```
  w= 3+z*(w+z)/w-5;
  w=complex(3,7)
  return 0;
}

/*
Eingabe:
1 2
3 4
Ausgabe
1 + 2*i
3 * 4*i
w: 1 + 2*i
*/
```

Synonyme:

Klassenobjekt, Instanz, Klassenvariable
Elementfunktion, Memberfunktion, Methode
Datenelement, Datenmember, Attribut

Wir betrachten nun ein Protokoll mit Konstuktor- und Destruktoraufrufen, `static`-Datenelementen, und `static`-Funktionen, zum Zählen von Instanzen:

Listing 2.9: Klasse complex IV

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

// Steuert die Einrueckung
void out(int blanks)
{ for(int i=0; i<2*blanks; i++) cout << " "; return; }

class complex { // Klassendefinition
 public:
   static int anzComplex; // Anzahl der angelegten Klassenvariablen
   static int blanks ;      // Einrueckung

   static void info ()      // static-Memberfunktion
   { out(blanks; cout << "Version ... der Klasse complex" << endl; }

   // Konstruktur zur Erzeugung und Initialisierung von Klassenobjekten:
   complex(const double r=0, const double i=0)
   {
     re= r;
     im= i;
     out(blanks++);
     cout << "Anlegen von Objekt    " << ++anzComplex << endl;
     return;
   }
   // Kopierkonstruktor
   complex(const complex& z)
   {
     re= z.re; // ohne selbstdefinierten Copy-Konstruktor macht der
     im= z.im; // Compiler genau dies
     out(blanks++);
     cout << "Anlegen von Objekt    " << ++anzComplex << endl;
```

```
      return;
    }

    void setRe(const double x) { re= x; return; }
    void setIm(const double x) { im= x; return; }

    double Re() const { return re; }
    double Im() const { return im; }

    // Zuweisung:
    complex& operator=(const complex& w) // Elementfunktion mit einem Argument
    {
        if (this==&w) return *this; // es ist dann nichts zu tun
        re= w.re;
        im= w.im;
        return *this;
    }

    friend ostream& operator<<(ostream& os, const complex& w)
    {
      os << w.re << " + " << w.im << "*i" << endl;
      return os;
    }

    ~complex() // Destruktor
    {
      out(--blanks);
      cout << "Zerstoerung von Objekt " << anzComplex-- << endl;
      return;
    }

  private: // Datenmember
    double re; // Realteil
    double im; // Imaginaerteil
}; // class complex

complex operator+(const complex& a, const complex& b){
  complex c;
  c.setRe( a.Re() + b.Re() );
  c.setIm( a.Im() + b.Im() );
  out(complex::blanks);
  cout << "Ruecksprung aus operator+()" << endl;
  return c;
}

int complex::anzComplex= 0; // Initialisierung der static-Attribute
int complex::blanks=0;


complex globVar(1,3); // Globale Variable ausserhalb von main() definieren


int main(){
  out(complex::blanks);
  cout << "Routine main() wird ausgefuehrt!" << endl;
  complex::info (); // Aufruf der static-Memberfunktion

  complex z;
/*
  cout << "z: " << z << endl;
*/
  z= z + globVar;
```

```
95    // Zugriff auf static-Attribut blanks der Klasse complex:
      out(complex::blanks);
      cout << "Vor dem Ruecksprung aus Routine main()" << endl;

      return 0;
100 }
```

Dies hat als Ausgabe

```
Anlegen von Objekt     1
  Routine main() wird ausgefuehrt!
  Version ... der Klasse complex
  Anlegen von Objekt      2
    Anlegen von Objekt      3
      Ruecksprung aus operator+()
      Anlegen von Objekt      4
      Zerstoerung von Objekt 4
    Zerstoerung von Objekt 3
    Vor dem Ruecksprung aus Routine main()
  Zerstoerung von Objekt 2
Zerstoerung von Objekt 1
```

`static`-Attribute einer Klasse sind nicht einzelnen Instanzen, sondern der Klasse selbst zugeordnet.
Aufruf: Klassenname::Attributname[1]

`static`-Memberfunktionen werden qualifiziert mit dem Klassennamen ohne Bezug auf ein konkretes Klassenobjekt aufgerufen.

Konstruktoren: Erzeugen und Initialisieren von Klassenobjekten (Resourcenbeschaffung)

Destruktoren: Aufräumarbeiten für nicht mehr benötigte Klassenobjekte (Resourcenfreigabe)

[1] der Operator :: heißt **Scope**-Operator

2.1 Umgang mit dynamischem Speicher

Kommen Zeiger auf dynamischen Speicher als Datenmember vor, so sind ein allgemeiner Konstruktor, ein Kopierkonstruktor, ein Zuweisungskonstruktor und ein Destruktor selbst zu definieren!
Werden diese nicht definiert, generiert der Compiler Default-Versionen mit falscher Semantik!

Listing 2.10: dynamische Speicher

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

class dyn{ // Klasse mit Zeiger auf dynamischen Speicher
 public:
   dyn (int k=0) { p=new int(k);} // allgemeiner Konstruktor
   // Besorgt Speicher fuer ein int und initialisiert diesen mit Wert von k

   ~dyn() { delete p; } // Destruktor
   // Gibt den Speicher, auf den p zeigt, wieder frei
 public:
   int* p; // Datenmember: Zeiger auf ein int
};

int main()
{
  dyn a, b;
  *a.p= 5;
  cout << "*a.p: " << *a.p << "  *b.p: " << *b.p << endl;
  b= a; // Datenmember werden elementweise kopiert (flache Kopie)!
  cout << "*a.p: " << *a.p << "  *b.p: " << *b.p << endl;
  *b.p= 9;
  cout << "*a.p: " << *a.p << "  *b.p: " << *b.p << endl;

  return 0;
}
/*
*a.p: 5  *b.p: 0
*a.p: 5  *b.p: 5
*a.p: 9  *b.p: 9
*/
```

Die obige Zuweisung `b = a;` (**flache Kopie**) veranlasst den Compiler, den Wert des Datenelements `a.p` (dies ist ein Adresswert) in das Datenelement `b.p` des Objekts `b` zu übertragen. Die beiden Datenelemente zeigen danach auf die gleiche Stelle im Heap. Eine Veränderung des dort gespeicherten `int`-Wertes (oben mittels `*b.p = 9;`) ändert also auch `*a.p` (genauer gesagt ist `*a.p` immer gleich `*b.p`).
Eine **tiefe Kopie** legt dagegen eine echte Kopie des Wertes `*a.p` im Heap an, auf die der Pointer `b.p` zeigt. Dazu ist ein selbstgeschriebener Zuweisungsoperator notwendig!

Listing 2.11: dynamische Speicher

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

class dyn{ // Klasse mit Zeiger auf dynamischen Speicher
 public:
   dyn (int k=0) { p=new int(k);} // allgemeiner Konstruktor
   // Besorgt Speicher fuer ein int und initialisiert diesen mit Wert von k

   ~dyn() { delete p; } // Destruktor
   // Gibt den Speicher, auf den p zeigt, wieder frei

   dyn(const &dyn x) { // Kopierkonstruktor (treffender: copy initializer)
      p= new int(*x.p); // Aufruf z.B. bei der Variablendefinition dyn b(a);
      return;
   }

   dyn& operator=(const dy& x) // Zuweisungsoperator
   {
      if (this!=&x) {
         delete p ;          // gebe Speicher frei, auf den (*this).p zeigt
         p= new int (*x.p); // allokiere neu und initialisiere
      }
      return *this;
   }

 public:
   int* p; // Datenmember: Zeiger auf ein int
};

int main()
{
  dyn a, b;
  *a.p= 5;
  cout << "*a.p: " << *a.p << "  *b.p: " << *b.p << endl;
  b= a; // Datenmember werden elementweise kopiert (flache Kopie)!
  cout << "*a.p: " << *a.p << "  *b.p: " << *b.p << endl;
  *b.p= 9;
  cout << "*a.p: " << *a.p << "  *b.p: " << *b.p << endl;

  return 0;
}
/*
*a.p: 5  *b.p: 0
*a.p: 5  *b.p: 5
*a.p: 9  *b.p: 9
*/
```

Entsprechend wird zur korrekten Initialisierung ein Kopierkonstruktor benötigt (dieser wird nur bei Initialisierungen, nicht aber bei Zuweisungen und bei Aufrufen eines allgemeinen Konstruktors aufgerufen; die Übergabe von Objekten per Wert an eine Funktion und die Rückgabe eines Ergebnisobjektes werden als Initialisierungen betrachtet!). Ergänzt man obiges Programm durch den Zuweisungsoperator, so ergibt sich:

```
*a.p: 5  *b.p: 0
*a.p: 5  *b.p: 5
*a.p: 5  *b.p: 9
```

2.2 Implizite Benutzung des Kopierkonstruktors

Ein Protokoll für Kopierkonstruktoraufrufe

Listing 2.12: Kopierkonstruktor

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

class dyn{ // Klasse mit Zeiger auf dynamischen Speicher
 public:
   dyn (int k=0) { p=new int(k);} // allgemeiner Konstruktor
   // Besorgt Speicher fuer ein int und initialisiert diesen mit Wert von k

   ~dyn() { delete p; } // Destruktor
   // Gibt den Speicher, auf den p zeigt, wieder frei

   dyn(const dyn& x) {  // Kopierkonstruktor (treffender: copy initializer )
      cout << "Kopierkonstruktor arbeitet ... " << endl;
      p= new int(*x.p);  // Aufruf z.B. bei der Variablendefinition dyn b(a);
      return;
   }

   dyn& operator=(const dyn& x) // Zuweisungsoperator
   {
      if (this!=&x) {
         delete p ;          // gebe Speicher frei , auf den (*this).p zeigt
         p= new int (*x.p); // allokiere neu und initialisiere
      }
      return *this;
   }

 public:
   int* p; // Datenmember: Zeiger auf ein int
};

dyn& f0(dyn& x) { return x; } // Beachte Referenz- bzw. Wertuebergaben
dyn  f1(dyn& x) { return x; }
dyn  f2(dyn  x) { return x; }

int main()
{
  dyn a, b; // allgemeiner Konstruktor, also keine Aufrufe des Kopierkonstruktor
  a= f0(b); // kein Aufruf
  a= f1(b); // ein Aufruf
  a= f2(b); // zwei Aufrufe des Kopierkonstruktors

  return 0;
}
/*
Kopierkonstruktor arbeitet ...
Kopierkonstruktor arbeitet ...
Kopierkonstruktor arbeitet ...
*/
```

(typisch:

```
class T {...};

T f(const T& x)
{
 T y;
  ...
 return y;
}
```

) zeigt Ihnen, dass der Kopierkonstruktor benutzt wird bei

Wertübergabe von x: zum Anlegen einer lokalen Kopie wird der Kopierkonstruktor aufgerufen;

Rückgabe eines Objektes per Wert: Im rufenden Programm wird mit einer Kopie von y weitergearbeitet (also Kopierkonstruktoraufruf).

Ein Aufruf erfolgt außerdem bei

```
T x(y);
```

(wenn y vom Typ T ist).

2.3 friend-Klassen

Eine Klasse kann als `friend` einer anderen Klasse spezifiziert werden. Die als `friend` gekennzeichnete Klasse hat dann Zugriff auf alle Daten- und Funktionsmember der sie als `friend` spezifizierenden Klasse.

Listing 2.13: friend-Klassen

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

class A{
 private:
   int wert;
   int quadrat() { return wert*wert; }
   friend class B; // Alle Attribute und Methoden der Klasse A koennen
                   // in B angesprochen werden (als waeren sie public)
};

class B{
 public:
   void set(A& a, int k) { a.wert=k; }
   void out(A& a) { cout << a.quadrat() << endl; }
};

int main()
{
  A a;
  B b;
  b.set(a,7);
//cout << a.wert << endl; nicht moeglich, da private
  b.out(a);
  return 0;
}
```

Verwendung von `friend`-Funktionen bzw. `friend`-Klassen bedeutet eine kontrollierte Aufweichung des Prinzips der Datenkapselung. Also nur in begründeten Ausnahmefällen verwenden!

2.4 Template-Klassen

`template`-Klassen sind nach Typen parametrisierte Klassen. Als Beispiel betrachten wir die Klasse `dyn` als `template`-Klasse

Listing 2.14: Template-Klassen

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

template<class T> class dyn{ // Klasse mit Zeiger auf dynamischen Speicher
 public:
   dyn (T k=0) { p=new T(k);} // allgemeiner Konstruktor
   // Besorgt Speicher fuer ein int und initialisiert diesen mit Wert von k

   ~dyn() { delete p; } // Destruktor
   // Gibt den Speicher, auf den p zeigt, wieder frei

   dyn(const dyn& x) { // Kopierkonstruktor (treffender: copy initializer)
      cout << "Kopierkonstruktor arbeitet ... " << endl;
      p= new T(*x.p);  // Aufruf z.B. bei der Variablendefinition dyn b(a);
      return;
   }

   dyn& operator=(const dyn& x) // Zuweisungsoperator
   {
      if (this!=&x) {
         delete p ;          // gebe Speicher frei, auf den (*this).p zeigt
         p= new T (*x.p); // allokiere neu und initialisiere
       }
      return *this;
   }

 public:
   T* p; // Datenmember: Zeiger auf ein int
};

typedef dyn<int> Dyn; // Dyn steht abkuerzend fuer den Typ dyn<int>

Dyn  f1(Dyn& x) { return x; }

int main()
{
  Dyn a, b; // allgemeiner Konstruktor, also keine Aufrufe des Kopierkonstruktor
  a= f1(b); // ein Aufruf des Kopierkonstruktors
  dyn<double> d;
  *d.p = 3.7;
  dyn<double> e(d); // Aufruf des Kopierkonstruktors
  cout << "*e.p: " << *e.p << endl;
  dyn<char> c('x'); // allgemeiner Konstruktoraufruf
  cout << "*c.p: " << *c.p << endl;
  return 0;
}
/*
Kopierkonstruktor arbeitet ...
Kopierkonstruktor arbeitet ...
*e.p: 3.7
*c.p: x
*/
```

2.5 Templates mit Template-Parametern

Templates dürfen auch weiter (Klassen-)Templates als Parameter benutzen:

```
template<class T,
         template <class> class Speicherstrategie = InDerLaengeBeschraenkt>
class SequentiellerContainer
                              :      Speicherstrategie<T>
{
    // ...
};

template<class T>
struct InDerLaengeBeschraenkt { // ...; };

template<class T>
struct InDerLaengeUnbeschraenkt { // ...; };

// ...
```

So können dann Benutzer der Template-Bibliothek wahlweise verschiedene Ausprägungen für verschiedenen Anwendungen generieren:

```
    // ...
    SequentiellerContainer<double> vek1;
    SequentiellerContainer<Figure, InDerLaengeUnbeschraenkt> graphik;
    // ...
```

Genaueres kann man in „Andrei Alexandrescu: Modern C++ Design, Addison-Wesley, 2001“ nachlesen. Die geschilderte Art des Bibliotheksdesigns nennt man **policy-based design**.

2.6 Matrix-Vektor-Operationen (dynamisch)

Zu berechnen ist

$$A \cdot b$$

wobei A eine (m, n)-Matrix und b ein Vektor mit n Elementen ist. Definiert man nun

```
class Matrix {  };
class Vektor {  };
```

so könnte man $A \cdot b$ mittels `A.operator*(b)` berechnen, wobei dann `operator*` eine Memberfunktion der Klasse `Matrix` wäre.

Aus Effizienzgründen sollen dann Matrix- und Vektorelemente direkt, also nicht erst durch Zugriffsfunktionen ansprechbar sein. Als Memberfunktion von `Matrix` trifft dies auf Matrixelemente zu. Direkten Zugriff auf Vektorelemente erreicht man durch die Deklaration von `Matrix::operator*()` als `friend`-Funktion der Klasse `Vektor`.

Listing 2.15: Minimal-Hauptprogramm

```
#include "matvekop2.h"
 ...

int main()
{
  int n = 3; // auch einlesbar!
  Matrix A(3,3);
  Vektor b(3);
  for(int i=1; i<=n; i++)
    for(int j=1; j<=n; j++)
      A(i,j)=i+j;
  cout << "Matrix A:\n" << A;
  b=A[1];  // erste Zeile von A
  cout << " A*b: \n" << A*b;

  return 0;
}
```

Man beachte

a) Indizierung von 1 ab

b) Ausgabeoperator ist überladen

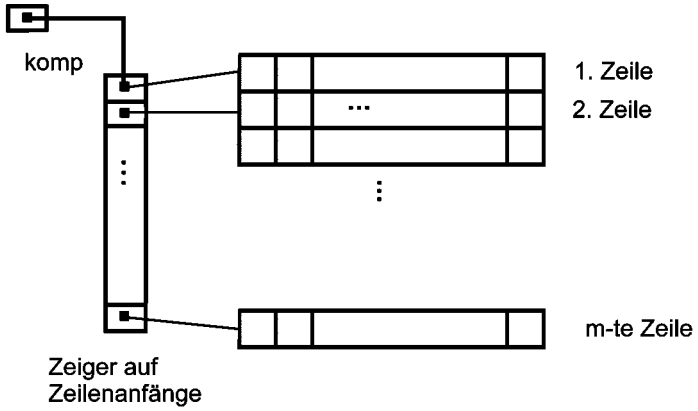
c) Elementzugriff mittels ()

d) Zeilenzugriff mittles []

Es gibt hierzu eine Splittung in drei Dateien

`main.cpp` `matvekop2.cpp` (Definition der Klassenmember) `matvekop2.h` (Klassen mit Deklarationen)

Die vorgestellte Implementierung einer (m, n)-Matrix hat nun folgende Gestalt



Eine alternative Implementierung könnte auf einer Indexarithmetik basieren, die den Doppelindex auf ein eindimensionales Feld abbildet (vgl. http://www.math.uni-wuppertal.de/~buhl/teach/exercises/Inf1-WS0001/script.pdf#section.2.6).

Listing 2.16: Datei matvekop2.h

```
/*
Matrix-/Vektor-Operationen Headerfile
Indizierung von 1 bis n (also nicht von 0 bis n-1)
Keine Indexueberpruefungen!
*/

#ifndef MatVekOp2Included //Schutz vor Mehrfacheinbindung dieser Headerdatei
#define MatVekOp2Included

#include <iostream>

class Vector; //weiter unten folgt die Definition dieser Klasse

class Matrix {
 // oeffentliche Schnittstelle , nur Deklarationen
 // Definitionen ausserhalb des Klassenrumpfes
 public:
   Matrix(int, int ); //allgemeiner Konstruktor
   Matrix(const Matrix& A); //Kopierkonstruktor
   ~Matrix (); //Destruktor
   Vector operator*(const Vector&); //Matrix ueber this
   Vector operator[](int z) const; //Zeile einer Matrix
```

```
    double  operator() (int z, int s) const; //Elementzugriff lesend bei const
    double& operator() (int z, int s);  //Elementzugriff
    Matrix& operator=(const Matrix& A); //Zuweisungsoperator

    friend ostream& operator<<(ostream& os, const Matrix& A);
  private:
    int  zeilen , spalten;
    double** komp; //zur Adressierung der Matrixkomponenten
};


class Vector {
    friend ostream& operator<<(ostream& os, const Vector& x);
    //Matrix-Memberfunktion operator*() soll befreundet sein:
    friend Vector Matrix::operator*(const Vector&);
    friend Vector Matrix::operator[](int z) const;
  public:
    Vector(int); //Konstruktor
   ~Vector();   //Destruktor
    Vector(const Vector& a); //Kopierkonstruktor
    double  Vector::operator[](int z) const;
    double& Vector::operator()(int z);

  private:
    int anzKomp;  //Laenge des Vektors
    double* komp; //Zeiger auf Anfang des Vektors
};

#endif
```

Zunächst eine kurzer Einschub:

Listing 2.17: Referenzen als Funktionsergebnis

```
//Funktion, die eine Referenz als Erg. liefert
//Beachte: keine Ref. auf lokale Groessen zurueckgeben!

#include <iostream>
using namespace std;

int& max(int& a, int& b) { return a>b?a:b; } //Referenz als Erg.
int  min(int& a, int& b) { return a<b?a:b; } //Rueckgabe per Wert
//int& falsch() { int r; return r; }
// waere falsch!  warning: reference to local variable `r'

int main(){
    int x=3, y=5;
    max(x,y)= 9; // max(x,y) liefert hier eine Referenz auf y,
                 // y wird also der Wert 9 zugewiesen
    //x= max(3, 9); waere Fehler:
    // initialization  of non-const reference type `int &'
    //from rvalue of type `int'

    cout << "x: " << x << "   y: " << y << endl;
    //min(x,y)= 9; //non-lvalue in assignment
}
/*
x : 3    y: 9
*/
```

Listing 2.18: Datei matvekop2.cpp

```
/*
Matrix-/Vektor-Operationen (Implementation)
Indizierung von 1 bis n  (also nicht von 0 bis n-1)
Keine Indexueberpruefungen!
*/

//#include <iostream>
//using namespace std;
#include "matvekop2.hpp"

//Definitionen der Memberfunktionen ausserhalb des Klassenrumpfes,
//deshalb Qualifizierung mit Matrix::

Matrix::Matrix(int z, int s) : zeilen(z), spalten(s) //Konstruktor
{
   komp= new double*[zeilen]; //Vektor von Zeigern auf Zeilenanfaenge
   for (int i=0; i<zeilen; ++i)
   {
      komp[i]= new double[spalten]; //Zeile i
      for (int j=0; j<spalten; ++j)
         komp[i][j] = 0.0;
   }
}

Matrix::~Matrix() //Destruktor
{
   for (int i=0; i<zeilen; ++i)
      delete[] komp[i]; //zunaechst Zeilen loeschen
   delete[] komp;       //Feld der Zeiger auf Zeilenanfaenge freigeben
}

Matrix::Matrix(const Matrix& A) //Kopierkonstruktor
{
   zeilen= A.zeilen;
   spalten= A.spalten;
   komp= new double*[zeilen]; //Vektor von Zeigern auf Zeilenanfaenge
   for (int i=0; i<zeilen; ++i)
   {
      komp[i]= new double[spalten]; //Zeile i
      for (int j=0; j<spalten; ++j)
         komp[i][j] = A.komp[i][j];
   }
   return;
}

Matrix& Matrix::operator=(const Matrix& A) //tiefe Kopie bei Zuweisung
{
   if (this != &A) //Bei A=A; soll nichts getan werden
   {
      for (int i=0; i<zeilen; ++i)
         delete[] komp[i]; //zunaechst Zeilen loeschen
      delete[] komp;       //Feld der Zeiger auf Zeilenanfaenge freigeben
      zeilen= A.zeilen;
      spalten= A.spalten;
      komp= new double*[zeilen]; //Vektor von Zeigern auf Zeilenanfaenge
      for (int i=0; i<zeilen; ++i)
      {
         komp[i]= new double[spalten]; // Zeile i
         for (int j=0; j<spalten; ++j)
            komp[i][j] = A.komp[i][j];
```

```
      }
    }
    return (*this);
}

//Ueberladung des Funktionsaufrufoperators ()
double Matrix::operator()(int z, int s) const  //Elementzugriff nur lesend
{
    return (*this).komp[z-1][s-1]; //Indizierung von 1 bis n
}

double& Matrix::operator()(int z, int s) //Elementzugriff
{
    return (*this).komp[z-1][s-1];
}


ostream& operator<<(ostream& os, const Matrix& A) //Ausgabe der Matrix
{
    for (int i=1; i<=A.zeilen; i++)
    {
       os << "Zeile " << i << ":" << endl;
       for (int j=1; j<=A.spalten; j++)
      {
          os << A(i, j) << "  ";
      }
       os << endl;
    }
    os << endl;
    return os;
}


Vector::Vector(int k ) : anzKomp(k) //Konstruktor
{
    komp= new double[anzKomp];
    for (int i=0; i<anzKomp; ++i)
        komp[i]= 0.0;
}

Vector::~Vector() { delete[] komp; } //Destruktor


Vector::Vector(const Vector& a) : anzKomp(a.anzKomp)
{ //Kopierkonstruktor
    komp= new double[anzKomp];
    for (int i=0; i<anzKomp; i++) komp[i]= a.komp[i];
}

//Ueberladung des Indexoperators []:
double Vector::operator[](int z) const  //nur lesend
{
    return komp[z-1];
}

double& Vector::operator()(int z)
{
    return komp[z-1];  //
}

ostream& operator<<(ostream& os, const Vector& x)
{
```

```
    for (int i=1; i<=x.anzKomp; i++)
        cout << x[i] << endl;
    return os;
}

Vector Matrix::operator[](int z) const //Zeile einer Matrix
{
    Vector tmp(spalten);
    for (int j=0; j<spalten; j++)
        tmp.komp[j]= komp[z-1][j];
    return tmp;
}

// Matrix-Vektor-Multiplikation A*b:
Vector Matrix::operator*(const Vector& b) //Matrix ueber this
{
    Vector res(zeilen);
    for (int i=0; i<zeilen; i++)
    {
        res.komp[i]= 0.0;
        for (int j=0; j<spalten; j++)
            res.komp[i]+= komp[i][j]*b.komp[j];
    }
    return res; //Kopierkonstr. wird aufgerufen, also tiefe Kopie
}
```

Listing 2.19: Datei main.cpp

```
/*
Test der Matrix-/Vektor-Operationen
Indizierung laeuft von 1 bis n
*/

#include <iostream>
#include "matvekop2.hpp" //Headerdatei mit Mat-Vek-Op
using namespace std;

int main() {
    Matrix A(2,2);
    Vector b(2), res(2);
    A(1, 1) = 1;
    A(1, 2) = 2;
    A(2, 1) = 3;
    A(2, 2) = 4;

    const Matrix AA(A); //Initialisierung mittels Copy-Konstr.
    //AA(1,1)= 17; //waere Fehler, da AA als const spezifiziert
    cout << "Matrix AA:\n" << A << endl;

    cout << AA(2,2) << endl;

    cout << "Zeile 1 von A als Vektor: \n" << A[1] << endl;

    b(1) = 2;
    b(2) = 1;
    res= A*(A*b); //aeq. zu res= A.operator*(b);
    cout << "A*(A*b): \n" << res << endl;

    res= A*A[1]; //A*erste Zeile von A
    cout << "A*A[1]: \n" << res << endl;
```

```
    int n= 3; //kann auch eingelesen werden

    Matrix B(n,n);
    for (int i=1; i<=n; i++)
        for(int j=1; j<=n; j++)
            B(i,j)= i+j;

    cout <<"Matrix B:\n" << B;

    Matrix BB(1,2); //Matrix mit einer Zeile und zwei Spalten
    BB= B; //Zuweisungsoperator, BB hat danach 3 Zeilen und 3 Spalten
    BB(1,1)= -1; //setze Element links oben auf -1
    cout << "Matrix BB:\n" << BB << endl;
    Vector v(n), r(n);
    for (int i=1; i<=n; i++) v(i)= 1;
    r=B*v;
    cout << "r=B*v:\n" << r;

    return 0;
}
/*
Matrix AA:
Zeile 1:
1 2
Zeile 2:
3 4

4
Zeile 1 von A als Vektor:
1
2

A*(A*b):
24
52

A*A[1]:
5
11

Matrix B:
Zeile 1:
2 3 4
Zeile 2:
3 4 5
Zeile 3:
4 5 6

Matrix BB:
Zeile 1:
-1 3 4
Zeile 2:
3 4 5
Zeile 3:
4 5 6

r=B*v:
9
12
15
*/
```

2.7 Automatische Typkonversion

Listing 2.20: Automatische Typkonversion: Konstruktor

```
/*
      Automatische Typkonversion double -> Vector
*/
#include <iostream>
#include "matvekop2.hpp"
using namespace std;
int main() {
   Vector b(2);
   b(1) = 1;
   b(2) = 2;
   cout << b << endl;

   b = 3;
   cout << b << endl;
}
```

liefert den Ausdruck

```
1
2

0
0
0
```

weil hier eine automatische Typkonversion von `int` nach `Vector` mit Hilfe des Konstruktors `Vector(int)` durchgeführt wird.
Um solche unerwünschten (impliziten) Anwendungen des Konstruktors zu unterdrücken, kann und sollte man ihn als `explicit` deklarieren, sofern der Parameter des Konstruktors nichts mit dem Wert sondern eher etwas mit der Struktur des zu konstruierenden Objekts zu tun hat:

Listing 2.21: expliziter Konstruktor

```
class Vector {
   friend ostream& operator<<(ostream& os, const Vector& x);
   //Matrix-Memberfunktion operator*() soll befreundet sein:
   friend Vector Matrix::operator*(const Vector&);
   friend Vector Matrix::operator[](int z) const;
 public:
    explicit  Vector(int); //Konstruktor
   ~Vector();        //Destruktor
   Vector(const Vector& a); //Kopierkonstruktor
   double  Vector::operator[](int z) const;
   double& Vector::operator()(int z);
 private:
   int anzKomp;  //Laenge des Vektors
   double* komp; //Zeiger auf Anfang des Vektors
};
```

Weitere Probleme durch die automatische Typkonversion:

Listing 2.22: Typkonversion

```
// Zur automatischen Typkonversion (TK)

#include <iostream>
using namespace std;

struct A { //alles standardmaessig public
 public:
  A(int k=1) {d= k;}
  friend A operator+(const A& x, const A& y){ A r; r.d=x.d+y.d; return r; }
  int d;
};

struct B {
  B()   {d=0;  }
  B(A a) {d= a.d;}
  friend B operator+(const B& x, const B& y){ B r; r.d=x.d+y.d; return r; }
  int d;
};

struct C {
  C(int k=1) {d= k;}
  C operator+(const C& x){ d+=x.d; return *this; }
  int d;
};

int main() {
  A x;
  x= x+3.1; // OK, genau ein impl. Aufruf der selbstdef. TK
  x= 3+x;
  B z;
  z= (A)3+z; // auf (A)3 vom Typ A wird impl. eine selbstdef. TK angewandt
  //z= z+3;  // geht nicht, da nur maximal eine implizite  selbstdef. TK
  C w;
  w= w+3;  // w.operator+(3);
}
```

2.8 Inline-Funktionen

Funktionsaufrufe von inline-Funktionen sollten vom Compiler überall durch den entsprechenden Code des Funktionsrumpfes ersetzt werden. Es ergibt sich ein Laufzeitgewinn (keine Parameterübergabe, ...).

Nur einfach(st)e Funktionen sollten als inline deklariert werden. Der Compiler kann „inline“ ignorieren.

Listing 2.23: Inlinefunktionen

```
/*
Inline-Funktionen
=================

Funktionsaufrufe von inline-Funktionen sollten vom Compiler ueberall
durch den entsprechenden Code des Funktionsrumpfes ersetzt werden.
--> Laufzeitgewinn (keine Parameteruebergabe, ...)

Nur einfach(st)e Funktionen sollten als inline deklariert werden.
Der Compiler kann "inline" ignorieren
*/

#include <iostream>
#include <ctime> // Zeitmessung

using namespace std;

void start_clock(clock_t& t1); // Startet den Timer
void print_time_used(clock_t t1);

//inline
double sum(double a, double b)
{
   return a+b;
}

int main()
{
   const long int anz=100000000;
   cout << "anz: " << anz << endl;
   double a=0, x=0;

   clock_t t; // Datentyp in <ctime> definiert
   cout << "Zeitmessung wird gestartet ...\n";
   start_clock(t); //Startzeitpunkt wird in t gespeichert
   cin >> a;
   for(long int i=0; i< anz; i++)
   {
      x= sum(2.0,3.0);
      //Bei inline: Ersetzung durch x = 2.0+3.0; oder sogar durch c= 5.0;
      x= sum(a,1.0);
      //Bei inline: Ersetzung durch x= a+1.0;
   }
   print_time_used(t); //Vergangene Zeit seit Zeitpunkt t
   cout << " beendet \n";
```

```
    return 0;
}

void start_clock(clock_t& t1)
{
    //clock_t
    t1= clock();
    if (t1 == clock_t(-1))  // Abbruch, falls timer nicht  richtig  arbeitet
    {
        cerr << "sorry, no clock\n";
        exit (1);
    }
    return;
}

void print_time_used(clock_t  t1)
{
    clock_t  t2= clock();
    if (t2 == clock_t(-1))
    {
       cerr<< " sorry, clock overflow\n";
       exit (2);
    }
    cout << "Time  used: " << 1000*double(t2-t1)/CLOCKS_PER_SEC
         << " msec" << endl;
    return;
}
/*
Uebersetzung mit  g++ -O2 ...  auf Sun

O h n e   inline :
anz: 100000000
Zeitmessung wird gestartet  ...
4  <-- Eingabe
Time  used: 17820 msec

M i t     inline :
anz: 100000000
Zeitmessung wird gestartet  ...
4  <-- Eingabe
Time  used: 280 msec

Compileraufruf mit Optimierungsoption verwenden.
Ergebnisse haengen stark vom verwendeten Compiler ab!

*/
```

2.9 Zeitmessung für Inline-Funktionen

Listing 2.24: Zeitmessung bei Inlinefunktionen

```
//Zeitmessung fuer inline-Memberfunktionen

#include <iostream>
#include <ctime> // Fuer Zeitmessung
using namespace std;

void start_clock(clock_t& t1); // Startet den Timer

void print_time_used(clock_t t1);

class real
{
   double d;
 public:
   real(double x=0);
   real(const real& a);
   friend real operator*(const real&, const real&);
};

inline
real :: real(double x=0) { d= x; }      //allg. Konstr.
inline
real :: real(const real& a) { d= a.d; } //Copy-Konstr.

inline
real operator*(const real& a, const real& b)
{
   return real(a.d*b.d);
}

int main()
{
   real a(3), b(1);

   const long int anz=10000000;
   cout << "Schleifenlaenge: " << anz << endl;

   clock_t t; // Datentyp in <ctime> definiert
   cout << "Zeitmessung wird gestartet .... \n";
   start_clock(t); // Startzeitpunkt wird in t gespeichert

   for(long int i=0; i< anz; i++)
   {
      b= a*a*a;
   }
   print_time_used(t); // Vergangene Zeit seit Zeitpunkt t

   cout << "... beendet \n";

   return 0;
}

void start_clock(clock_t& t1)
{
   //clock_t
   t1= clock();
   if (t1 == clock_t(-1)) // Abbruch, falls timer nicht richtig arbeitet
```

```
    {
        cerr << "sorry, no clock\n";
        exit (1);
    }
    return;
}

void print_time_used(clock_t t1)
{
    clock_t t2= clock();
    if (t2 == clock_t(-1))
    {
        cerr<< " sorry, clock overflow\n";
        exit (2);
    }
    cout << "Time  used: " << 1000*double(t2-t1)/CLOCKS_PER_SEC
        << " msec" << endl;
    return;
}

/*
g++  o h n e  Optimierungsoption auf Sun:

O h n e  inline :
Schleifenlaenge: 10000000
Time  used: 4150 msec
 ...  beendet


M i t  inline :
Schleifenlaenge: 10000000
Time  used: 3060 msec
 ...  beendet


g++ -O2 ... also  m i t  Optimierungsoption auf Sun:

O h n e  inline :
Schleifenlaenge: 10000000
Time  used: 2090 msec
 ...  beendet


M i t  inline :
Schleifenlaenge: 10000000
Time  used: 420 msec
 ...  beendet
*/
```

2.10 Inline in Verbindung mit generischer Programmierung

Meistens ist es günstiger Inlinefunktionen anstatt von `friend`-Funktionen zu benutzen.

Listing 2.25: Inlinefunktionen mit templates

```
#include <iostream>
#include <ctime> // Fuer Zeitmessung
using namespace std;

#define INLINE inline
//Macro ersetzt textuell INLINE durch inline oder aber durch nichts

void start_clock(clock_t& t1); // Startet den Timer

void print_time_used(clock_t t1);

template <class T> class real
{
 private:
   T d;
 public:
   real(T x =0);            //allgem. Konstruktor
   real(const real<T>&);  //Copy-Konstruktor
   T get() const;
   //friend real<T> operator*<>(const real<T>&, const real<T>&);
};

//Beachte: inline muss n a c h template <class T> stehen:
template <class T> INLINE real<T>::real(T x=0) { d= x; }     //allg. Konstr.

template <class T> INLINE real<T>::real(const real<T>& a) { d= a.d; } //C-K

template <class T> INLINE T real<T>::get() const { return d; }

template <class T> INLINE real<T> operator*(const real<T>& a, const real<T>& b)
{
   return real<T>(a.get()*b.get());
}

int main()
{
   real<double> a(2), b(3);

   const long int anz=10000000;
   cout << "Schleifenlaenge: " << anz << endl;

   clock_t t; // Datentyp in <ctime> definiert
   cout << "Zeitmessung wird gestartet .....\n";
   start_clock(t); // Startzeitpunkt wird in t gespeichert

   for(long int i=0; i< anz; i++)
   {
      b= a*a*a;
   }
   print_time_used(t); // Vergangene Zeit seit Zeitpunkt t
```

```
    cout << "... beendet \n";

    return 0;
}

void start_clock(clock_t& t1)
{
    //clock_t
    t1= clock();
    if (t1 == clock_t(-1))  // Abbruch, falls timer nicht richtig arbeitet
    {
        cerr << "sorry, no clock\n";
        exit(1);
    }
    return;
}

void print_time_used(clock_t t1)
{
    clock_t t2= clock();
    if (t2 == clock_t(-1))
    {
       cerr<< " sorry, clock overflow\n";
       exit(2);
    }
    cout << "Time  used: " << 1000*double(t2-t1)/CLOCKS_PER_SEC
        << " msec" << endl;
    return;
}

/*

g++ -O2 ... also mit Optimierungsoption auf Sun:

Ohne inline:
Schleifenlaenge: 10000000
Time used: 2710 msec

Mit inline:
Schleifenlaenge: 10000000
Time used: 420 msec

Haeufig kann und sollte damit ohne Effizienzverlust auf friend-Funktionen
verzichtet werden! Man sollte inline-Zugriffsfunktionen fuer Zugriffe
auf Klassenattribute verwenden!
*/
```

2.11 Matrix- und Vektoroperationen als Templates

Zunächst wird der Typ complex als `template` realisiert, dabei wird noch einmal die Deklaration von `friend`-Funktionen in `template`-Klassen gezeigt.

Listing 2.26: Matrix-/Vektoroperationen und complex als templates

```
/*
Der Skalartyp ist selbst wieder ein Template-Typ

Indizierung von 1 bis n (also nicht von 0 bis n-1)
Keine Ueberpruefungen!
*/

#include <iostream>
using namespace std;


template<class S> class complex;
template<class S> istream& operator>>(istream& is, complex<S>& w);


template<class S> class complex { // Klassendefinition
 public:
   // Konstruktur zur Erzeugung und Initialisierung von Klassenobjekten:
   complex(const S r=0, const S i=0) { re= r; im= i; return; }
   // Kopierkonstruktor
   complex(const complex<S>& z) { re= z.re; im= z.im; return; }

   void setRe(S x) { re= x; return; }
   void setIm(S x) { im= x; return; }

   S Re() const { return re; } // implizit vorhandenes Argument ist const
   S Im() const { return im; }

   // Arithmetische Operatoren alle global!

   // Zuweisung:
   complex& operator=(const complex& w) // Elementfunktion mit einem Argument
   {
       if (this==&w) return *this; // es ist dann nichts zu tun
       re= w.re;
       im= w.im;
       return *this;
   }
   // Weiterer Zuweisungsoperator:
   complex& operator+=(const complex& w) // Elementfunktion mit einem Argument
   {
       re+= w.re;
       im+= w.im;
       return *this;
   }

   // Ein-/Ausgabeoperatoren:
   friend istream& operator>> <> (istream& is, complex<S>& w);
/*
   {
      is >> w.re >> w.im;
```

```
      return is;
    }
*/
    friend ostream& operator<< (ostream& os, const complex& w)
    {
      os << w.re << " + " << w.im << "*i" << endl;
      return os;
    }
  private: // Datenmember
    double re; // Realteil
    double im; // Imaginaerteil
};


template<class T> istream& operator>>(istream& is, complex<T>& w)
    {
      is >> w.re >> w.im;
      return is;
    }


template<class S> complex<S> operator+(const complex<S>& a, const complex<S>& b){
  complex<S> c;
  c.setRe( a.Re() + b.Re() );
  c.setIm( a.Im() + b.Im() );
  return c;
}

template<class S> complex<S> operator-(const complex<S>& a, const complex<S>& b){
  complex<S> c;
  c.setRe( a.Re() - b.Re() );
  c.setIm( a.Im() - b.Im() );
  return c;
}

template<class S> complex<S> operator*(const complex<S>& a, const complex<S> &b)
{
  complex <S> c;
  c.setRe( a.Re()*b.Re() - a.Im()*b.Im() );
  c.setIm( a.Re()*b.Im() + a.Im()*b.Re() );
  return c;
}

template<class T> class Vector;

template<class T> class Matrix {
  public:
    Matrix(int, int ); // Konstruktor
    ~Matrix ();         // Destruktor
    // Matrix-Vektor-Multiplikation:
    void matXvec(const Vector<T>&, Vector<T>&); // Matrix ueber this-Zeiger
    T& elem(int, int); // Zugriff auf ein Matrixelement
    void matOut(); // Ausgabe der Matrix zeilenweise

  private:
    int zeilen , spalten;
    T** komp; //
};

template<class T> Matrix<T>::Matrix(int z, int s) : zeilen(z), spalten(s) {
    komp= new T*[zeilen];
    for (int i=0; i<zeilen; ++i)
```

```
    {
115     komp[i]= new T[spalten];
        for (int j=0; j<spalten; ++j)
            komp[i][j] = 0.0;
    }
}
120
template <class T> Matrix<T>::~Matrix() {
    for (int i=0; i<zeilen; ++i)
        delete[] komp[i];
    delete[] komp;
125 }

template <class T> T& Matrix<T>::elem(int z, int s) {
    if (z<=0 || z>zeilen || s<=0 || s>spalten) {
        cout << "Ungueltiger Index" << endl;
130     exit (1);
    }
    return komp[z-1][s-1];
}

135 template<class T> void Matrix<T>::matOut(){ // Ausgabe der Matrix
    for (int i=0; i<zeilen; i++)
    {
      cout << endl;
      for (int j=0; j<spalten; j++)
140   {
        cout << elem(i+1, j+1);
      }
    }
    cout << endl;
145 };


template <class T> class Vector {
    friend void Matrix<T>::matXvec (const Vector<T>&, Vector<T>&);
150  public:
    Vector(int); // Konstruktor
   ~Vector(); // Destruktor
    T& elem(int);  // Zugriff auf Elemente
    void vecOut(); // Ausgabe eines Vektors
155  private:
    int anzKomp; // Laenge des Vektors
    T* komp; // Zeiger auf Anfang des Vektors
};

160 template<class T> Vector<T>::~Vector() { delete[] komp; }

template<class T> Vector<T>::Vector(int k) : anzKomp(k) {
  komp= new T[anzKomp];
  for (int i=0; i<anzKomp; ++i)
165   komp[i]= 0.0;
}

template<class T> T& Vector<T>::elem(int k) {
    return komp[k-1];
170 }

template<class T> void Vector<T>::vecOut() {
    for (int j=1; j<=anzKomp; j++)
        cout << elem(j);
175 cout << endl;
```

```
    return;
}


template<class T> void Matrix<T>::matXvec(const Vector<T>& v, Vector<T>& res) {
    for (int i=0; i<zeilen; i++) {
        res.komp[i]= 0.0;
        for (int j=0; j<spalten; j++)
            res.komp[i]+= komp[i][j]*v.komp[j];
    }
    return;
}

typedef Matrix<complex<double> >  matComplDouble;

int main() {
    Matrix<complex<double> > A(2,2);
    matComplDouble W(5,5);
    Vector< complex<double> > b(2), res(2);
    A.elem(1, 1) = 1;
    A.elem(1, 2) = 2;
    A.elem(2, 1) = 3;
    A.elem(2, 2) = 4;
    b.elem(1) = 5;
    b.elem(2) = 6;
    A.matXvec(b, res);
    cout << res.elem(1) << res.elem(2) << endl;

    Matrix<double> AA(2,2);
    Vector<double> bb(2), rr(2);
    AA.elem(1, 1) = 1;
    AA.elem(1, 2) = 2;
    AA.elem(2, 1) = 3;
    AA.elem(2, 2) = 4;
    bb.elem(1) = 5;
    bb.elem(2) = 6;
    AA.matXvec(bb, rr);
    cout << rr.elem(1) <<"  " <<  rr.elem(2) << endl;


    {
        int n;
        n= 3; // Kann auch eingelesen werden
        Matrix<complex<double> > B(n,n);
        for (int i=1; i<=n; i++)
            for(int j=1; j<=n; j++)
                B.elem(i,j)= i+j;

        B.matOut();

        Vector<complex<double> > V(n), R(n);

        for (int i=1; i<=n; i++) V.elem(i)= complex<double>(1,2);
        V.vecOut();

        B.matXvec(V,R);
        for (int i=1; i<=n; i++) cout << R.elem(i);

    }  // Fuer Vektoren V, R und Matrix B werden Destruktoren aufgerufen

    return 0;
}
```

```
/*
17 + 0*i
39 + 0*i

17  39

2 + 0*i
3 + 0*i
4 + 0*i

3 + 0*i
4 + 0*i
5 + 0*i

4 + 0*i
5 + 0*i
6 + 0*i

1 + 2*i
1 + 2*i
1 + 2*i

9 + 18*i
12 + 24*i
15 + 30*i
*/
```

2.12 Templateklassen mit Typangleich der Operanden bei binären Operatoren

Benutzen Sie die `Vector`-Klasse des vorangehenden Abschnitts auf die folgende Weise

```
  // ...

template <class T>
Vector<T> operator+(
Vector<T>& v1, Vector<T>& v2){
  if (v1.length() != v2.length()) throw
    " Vector: operator+, Operanden haben ungleiche äLnge!";

  Vector<T> result(v1.length());
  for (int i=1; i <= v1.length(); i++)
      result(i) = v1(i)+v2(i);
  return result;
};

  // ...

  Vector<double> D(4);
  Vector<int>    I(4);

  D(1) = 1; D(2) = 2; D(3) = 3; D(4) = 4;
  I(1) = 10; I(2) = 20; I(3) = 30; I(4) = 40;

  cout << D << endl;
  cout << I << endl;

  cout << D+I << endl;
```

so werden Sie leider nur eine Fehlermeldung der Art

```
"vector_test.cc", line 62: Error: The operation
                "   Vector<double> + Vector<int>" is illegal.
```

erhalten. Hier muß ein Template mit zwei Parametern und für den Ergebnistyp eine Typangleichung (type promotion) desselben an den „höherwerigen“ der beiden Operandentypen nach folgendem Muster mit Hilfe von Traits selbst realisiert werden:

```
template<class T1, class T2>
struct promote_trait {};

#define DECLARE_PROMOTE(A,B,C)                \
  template <> struct promote_trait<A,B> {     \
    typedef C T_promote;                      \
  };

DECLARE_PROMOTE(int, double, double);
DECLARE_PROMOTE(double, int, double);
DECLARE_PROMOTE(float, double, double);
DECLARE_PROMOTE(double, float, double);
// DECLARE_PROMOTE(double, complex<float>, complex<float>);
 // ...
```

```
15
template <class Op1, class Op2>
Vector< typename promote_trait<Op1,Op2>::T_promote >
operator+(Vector<Op1>& v1, Vector<Op2>& v2){
   if (v1.length() != v2.length()) throw
      " Vector:␣operator+,␣Operanden␣haben␣ungleiche␣äLnge!";

   typedef Vector< promote_trait<Op1,Op2>::T_promote> result_type;

   result_type  result(v1.length());
   for (int i=1; i <= v1.length(); i++)
        result(i) = v1(i)+v2(i);
   return result;

}
 // ...
```

Kapitel 3

Vererbung und Polymorphie

Zunächst einige wichtige Stichworte:

Vererbung: Basisklasse, Oberklasse, abgeleitete Klasse, Unterklasse, Klassenhierarchie

Diese Begriffe wollen wir nun im Folgenden erklären und auch ihren Zusammenhang deutlich machen. Zunächst ein Beispiel:

Listing 3.1: Vererbung

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

class mensch { // Oberklasse
 public:
   string name;
   mensch(string x="Name ist unbekannt") : name(x) {}
   void out() { cout << name << endl;}
};

//abgeleitete Klasse
class frau : public mensch // Die Klasse frau erbt von der Klasse mensch
{
 public:
   frau(){} //implizit wird der Konstr. der Oberklasse aufgerufen;
            //das Attribut mutterschutz wird nicht initialisiert !
   frau(string a, bool b=0) : mensch(a), mutterschutz(b){}
   bool mutterschutz;
   void out() {
      mensch::out(); cout << "  Schutz: " << mutterschutz << endl;
   }
};

int main()
{
   mensch albert;
   cout << albert.name << endl;

   albert.name="Albert";
   cout << albert.name << endl;
   albert.out();
```

```
    frau eva;
    eva.out();
    frau maria("maria");
    maria.out();
    frau gisela("Gisela",true);
    gisela.out();
    mensch menschY;
    menschY= gisela;
    menschY.out(); //entspricht menschY.mensch::out();
    //eva= menschY; //nicht erlaubt! Attr. mutterschutz?
    return 0;
}
/*
Name ist unbekannt
Albert
Albert
Name ist unbekannt
  Schutz: -4263676
maria
  Schutz: 0
Gisela
  Schutz: 1
Gisela
*/
```

Jedes Objekt einer abgeleiteten Klasse enthält ein (anonymes) Objekt vom Typ der Oberklasse. Dieses sogenannte **Subobjekt** wird durch einen impliziten Aufruf des Oberklassenkonstruktors angelegt. Jede Elementfunktion der Oberklasse kann auf ein Objekt vom Typ der Oberklasse der abgeleiteten Klasse angewendet werden. Insbesondere der Aufruf einer Operation für ein Objekt lässt nicht erkennen, ob die Operation der Klasse des Objektes oder aber der Oberklasse zugeordnet ist. Eine abgeleitete Klasse enthält in der Regel zusätzliche Attribute und Methoden. Methoden einer Oberklasse dürfen in einer abgeleiteten Klasse überschrieben werden. In der abgeleiteten Klasse können Elementfunktionen mit dem gleichen Namen und der gleichen Signatur wie eine Elementfunktion in der Oberklasse definiert werden (Umdefinition). Die Umdefinition von nicht-virtuellen Methoden ist nicht empfehlenswert: Vergleiche Item 37 in „Effective C++, Scott Meyers, Addison-Wesley, 1992“. Ein Objekt vom Typ der abgeleiteten Klasse ist zuweisungskompatibel zu einem Objekt der Oberkasse. Dabei wird das anonyme Subobjekt dem Objekt der Oberklasse zugewiesen; eine Zuweisung Unterklassenobjekt=Oberklassenobjekt hingegen ist nicht erlaubt. Das Unterklassenobjekt hätte nämlich dann möglicherweise nicht-initialisierte Attribute. Methoden einer Unterklasse sollten sämtliche Invarianten der Oberklasse intakt lassen. Umdefinitionen sollten unter gleichen oder allgemeineren Vorbedingungen gleiche oder speziellere Nachbedingungen garantieren.
Wird von einer Oberklasse abgeleitet, so brauchen in der Unterklasse nur die Abweichungen zur Oberklasse beschrieben werden.

Vererbung funktioniert auch mit Template-Klassen.

Listing 3.2: Vererbung mit Templates

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

class Mitarbeiter {
 public:
   Mitarbeiter(const string vn, const string nn) : vname(vn), nname(nn) {}
   void print() const { cout << vname + "  " + nname << endl; }
 private:
   string vname, nname;
};

template <class Base> class TelCont: public Base //
{
 public:
   TelCont(string vn, string nn, string pn)
      : Base(vn,nn), phone(pn) {}
   void print() const
   { Base::print(); cout << "  Tel: " + phone << endl; }
 private:
   string phone;
};

template <class Base> class EMailCont: public Base
{
 public:
   EMailCont(string vn, string nn, string em)
      : Base(vn,nn), email(em) {}
   void print() const
   { Base::print(); cout << "  EMail: " + email << endl; }
 private:
   string email;
};

int main(){
   Mitarbeiter m1("Albert","Einstein");
   m1.print();

   TelCont<Mitarbeiter> m2("Fred","Feuerstein","0202-439-3060");
   m2.print();
   EMailCont<Mitarbeiter> m3("Teddy", "Baer", "teddy@uni-wuppertal.de");
   m3.print();

   //Das Folgende geht nicht:

   typedef TelCont<Mitarbeiter> TCM;
   //EMailCont<TCM> m4("AA","BB","CC","DD");
   // no matching function for call to
   // 'EMailCont<TelCont<Mitarbeiter> >::EMailCont
   // (const char [3], const char [3], const char [4], const char [4])'
   Mitarbeiter* p= &m1;
   p->print();
   p= &m2;      // p zeigt auf TelCont<Mitarbeiter>
   p->print(); // trotzdem wird Mitarbeiter::print() aufgerufen
   TelCont<Mitarbeiter>* q= &m2;
   q->print(); // TelCont<Mitarbeiter>::print() wird aufgerufen
   // q= &m1; // waere nicht erlaubt
   return 0;
```

```
}

/*
Albert  Einstein
Fred  Feuerstein
  Tel: 0202-439-3060
Teddy  Baer
  EMail: teddy@uni-wuppertal.de
Albert  Einstein
Fred  Feuerstein
Fred  Feuerstein
  Tel: 0202-439-3060
*/
```

Zeiger auf Basisklassenobjekte dürfen auch auf Objekte von abgeleiteten Klassen zeigen.

Beim Zugriff auf Objekte über Zeiger ist beim Aufruf von Memberfunktionen der sogenannte **statische Datentyp** des Zeigers (d.h. der Typ, der bei dessen Vereinbarung im Quelltext angegeben ist) ausschlaggebend. Dies trifft nicht mehr zu, wenn Memberfunktionen als virtuell (`virtual`) vereinbart werden.

3.1 Polymorphie: virtuelle Funktionen

Listing 3.3: Vererbung und virtuelle Funktionen

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

class Base {
 public:
   static string kennung;
   virtual void print(){ cout << kennung << endl; }
};
string Base::kennung="Basisklasse";

class A : public Base
{
 public:
    void print(){ cout << "Klasse A"<< endl; }
};

class B: public Base
{
 public:
   void print(){ cout << "Klasse B"<< endl; }
};

class CC: public A
{
 public:
   void print()
   {
       cout << "Klasse CC"<< endl;
       cout << "  direkt abgeleitet von " ;
       A::print ();
   }
   Base y;
};


int main() {
   Base x;
   x.print ();
   A a;
   a.print ();
   B b;
   CC cc;
   Base* p;
   p= &x;
   p->print();
   p= &a;
   p->print();
   p= &b;
   p->print();
   p= &cc;
   p->print();
   cc.y.print ();
   // p->y.print()  weare nicht erlaubt; p zeigt nur auf Subobjekt!
   // Das referenzierte Subobjekt vom Typ Base hat aber kein Attribut y!
   return 0;
}
```

```
/*
Basisklasse
Klasse A
Basisklasse
Klasse A
Klasse B
Klasse CC
  direkt abgeleitet von Klasse A
Basisklasse
*/
```

Der statische Datentyp des Pointers `p` ist *Zeiger auf Objekt der Klasse Base*. Der dynamische Typ des Zeigers wechselt, wenn dieser Zeiger auf ein Objekt einer von `Base` abgeleiteten Klasse zeigt. Ist in der Basisklasse eine Memberfunktion als `virtual` deklariert, so wird zur Auswahl der geeigneten Funktion beim Zugriff auf Objekte über einen Zeiger der dynamische Typ dieses Zeigers verwendet.

3.2 Vererbung über mehrere Stufen, Ableitungsgraph

Betrachtet man

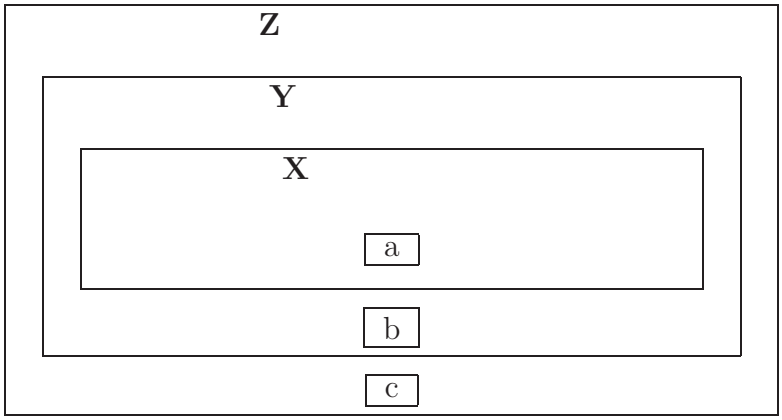
Listing 3.4: Mehrfache Vererbung

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

struct X { int a; };

struct Y : X { int b; };

struct Z : Y { int c; } obj; // obj wird als Objekt der Klasse Z angelegt

int main() {
   obj.c= 1; //ist aeq. zu:
   obj.Z::c= 1;
   obj.b= 2; //ist aeq. zu:
   obj.Y::b= 2;
   obj.Z::Y::b= 2;
   obj.a= 3; //ist aeq. zu:
   obj.X::a= 3;
   obj.Y::a= 3;
   obj.Z::a= 3;
   obj.Y::X::a= 3;
   obj.Z::Y::X::a= 3;
   obj.Z::X::a= 3;
}
```

so kann man dies auch als Graph darstellen.

$$Z \longrightarrow Y \longrightarrow X$$

Y⟶X bedeutet: „X ist direkte Basisklasse von Y“ beziehungsweise „Y ist ein X“ im Falle einer „public“ Oberklasse X oder „Y wird implementiert mit Hilfe von X“ im Falle einer „private“ Oberklasse X.



3.3 Namenskonflikte bei Vererbung

Listing 3.5: Namenskonflikte bei Vererbung

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

struct X { int a; };

struct Y : X { int a; };

struct Z : Y { int a; } z; // z wird als Objekt der Klasse Z angelegt

int main() {
   z.a= 1;
   z.Y::a= 2; //aeq. zu z.X::Y::a
   z.X::a= 3;
   cout << "z.a, z.Z::a, z.Y::a, z.X::a " << endl;
   cout << z.a <<" "<< z.Z::a <<" "<<  z.Y::a <<" "<< z.X::a << endl;
   cout << "z.Y::X::a: "<<  z.Y::X::a << endl;
   // Es wird diejenige Komp. a ausgegeben, die Y von X erbt und an Z
   // weitervererbt
}
/*
Jede Instanz der Klasse Z hat 3 int-Komponenten
*/
```

Zugriff mittels Zeiger auf die Basisklasse:

Listing 3.6: Zeiger auf Basisklasse

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

class X {
public:
  X(int k=1) : a(k){}  // Nach dem Konstruktornamen kann man getrennt durch
                       // Kommata eine Initialisiererliste anfuegen
                       // anstatt dies im Rumpf zu schreiben
  int a;
};

class Y : public X {
  Y(int j=2): X(5), b(j){} // Konstruktor von X aufrufen, dann den des
                           // Datentyps von b
  int b;
};

int main()
{
 X x;
 Y y;
 X* p;
 p= &x;
 p= &y; // *p.b gibt es dann hier nicht! p zeigt nur auf ein Subobjekt vom Typ
        // Typ X. Es ist also nur (*p).a ansprechbar!

 return 0;
}
```

3.4 Zugriffskontrolle bei der Vererbung mittels private/protected/public

Bei der Vererbung hat man die Möglichkeit zu spezifizieren, welche Zugriffsrechte die abgeleitete Klasse auf die Attribute der Oberklasse besitzt.

Listing 3.7: Zugriffskontrolle

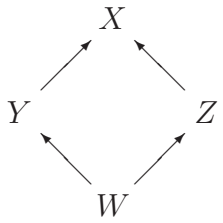
```
#include <iostream>
using namespace std;

class X {
 private:
   int a;    //kann auch in Unterklassen nicht angesprochen werden
 protected: //Elemente koennen in Unterklassen angesprochen werden
   int b;
 public:
   int c;  //kann ueberall angesprochen werden
} x;

class Y : private X { // Member von X sind ab Unterklasse Y  privat.
 public:
   void print() { cout << b << endl; }
};

//Beim weiteren Ableiten von Y gelten die Attribute b und c als privat.
//Deshalb geht folgendes nicht:
/*
class Z : public Y {
 public:
   void print() { cout << c << endl; }
};
*/

int main() {
  x.c= 1; //OK
//x.b= 2; //nein, da protected (also nur in Unterklassen ansprechbar)
//x.a= 3; //nein, da private und damit nur im Klassenrumpf ansprechbar
}
```

3.5 Mehrfachvererbung

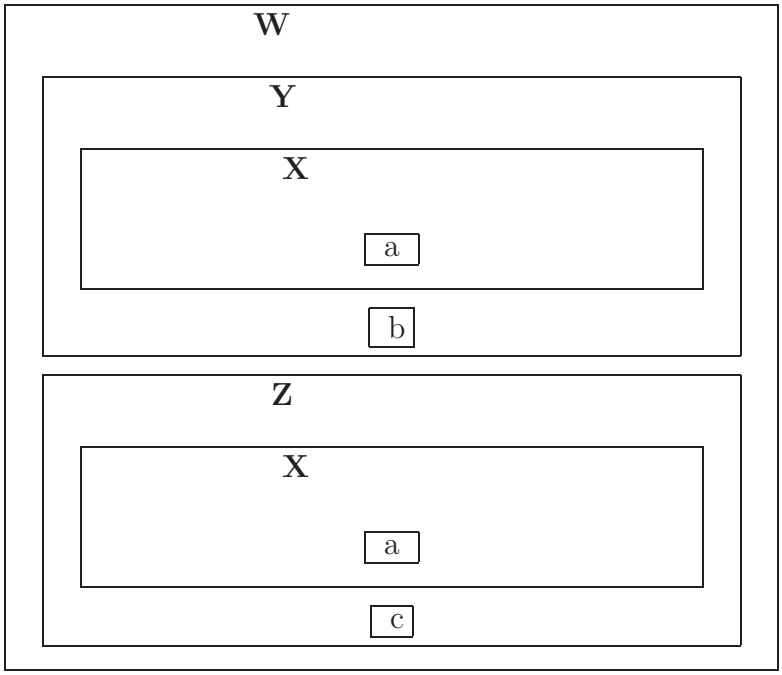
Listing 3.8: Mehrfachvererbung mit mehrfachen (anonymen) Objekten der Basisklasse

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

struct X {
    int a;
};

struct Y : X {
    int b;
};

struct Z : X {
    int c;
};

struct W : Y, Z {};

//struct V : X, Y {}; // ergibt:
//warning: direct base 'X' inaccessible in 'V' due to ambiguity

int main(){
    W w;
    w.Y::a= 1; // a-Attribut von X geerbt ueber Y
    w.Z::a= 2; // weiteres a-Attribut von X geerbt ueber Z
    cout << w.Y::a << "␣" << w.Z::a << endl;
    //w.X::a= 3;   // geht nicht, da mehrdeutig
    //w.Y::X::a= 4; // geht auch nicht
    return 0;
}
/*
1 2
*/
```

Der Ableitungsgraph hat dann folgende Gestalt



bzw.



Es werden mehrere Basisklassen(sub)objekte erzeugt. Will man dies nicht, so muss man virtuelle Basisklassen verwenden.

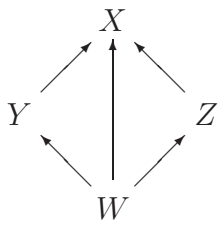
3.6 Virtuelle Basisklassen

Listing 3.9: virtuelle Basisklassen

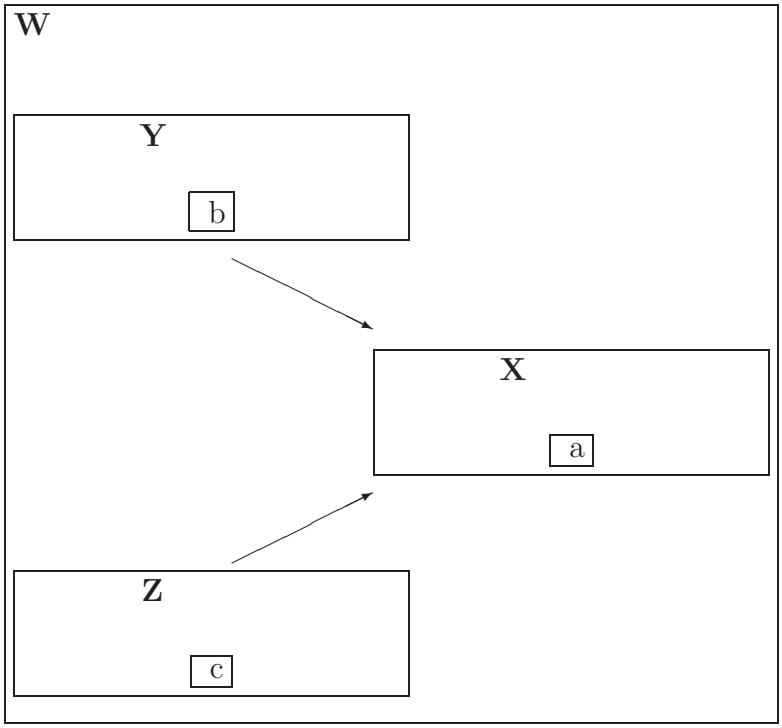
```
#include <iostream>
using namespace std;

struct X {
    int a;
};

struct Y : virtual X {
    int b;
};

struct Z : virtual X {
    int c;
};

struct W : virtual X, Y, Z {} w;


int main(){
    w.Y::a= 1;
    w.Z::a= 2;
    w.a= 3;
    cout << w.Y::a << " " << w.Z::a << " " << w.a <<  endl;
    // Es gibt nur ein einziges Attribut mit Namen a
}
/*
3 3 3
*/
```

Von virtuellen Basisklassen wird nur ein Subobjekt, auf das über verschiedene Vererbungswege zugegriffen werden kann, erzeugt.

Der Ableitungsgraph hat dann folgende Gestalt



bzw.


W
Y
b
X
a
Z
c

3.7 Initialisierungsreihenfolge von Unterklassen

Die Initialisierungsreihenfolge bei Verwendung virtueller Basisklassen zeigt folgendes Beispiel:

Listing 3.10: Initialisierungsreihenfolge

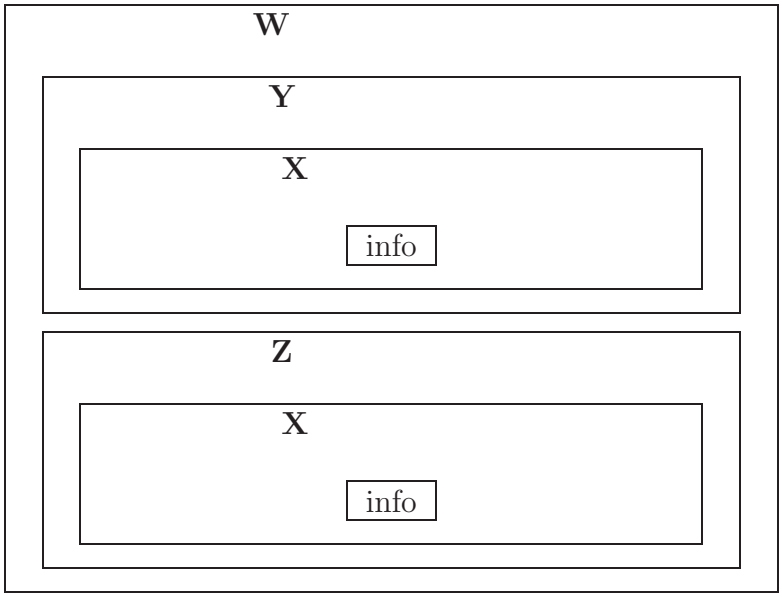
```
#include <iostream>
#include <string>
using namespace std;

struct X {
  X(string s="Standardkonstruktor in X ") : info(s) { cout << s << endl; }
  string info;
};

struct Y : virtual X {
  Y(string s="X() in Y ") : X(s) { cout << "Konstr. in Y "<< endl;}
};

struct Z : virtual X {
  Z(string s="X() in Z ") : X(s) { cout << "Konstr. in Z "<< endl;}
} z;

struct W : Y, Z {
  W() : X("X() in W"), Y("Y in W") , Z("Z in W"){ }
  // Beachte:  Basisklasseninitialisierer  der Konstruktoren
  // Y() und Z() werden ignoriert!
} w;


int main() {
  cout << "w: " << w.info << endl;
  cout << "z: " << z.info << endl;
}

/*
X() in Z
Konstr. in Z

X() in W
Konstr. in Y
Konstr. in Z

w: X() in W
z: X() in Z
*/
```

Nur der Basisklasseninitialisierer des Konstruktors eines vollständigen Objektes (also eines Objektes, das selbst nicht Subobjekt eines anderen Objektes ist) wird verwandt, um Subobjekte von virtuellen Basisklassen zu initialisieren. Ist kein Konstruktor vorhanden, so wird der Standardkonstruktor der virtuellen Basisklasse aufgerufen.

Die Initialisierung bei Verwendung nicht-virtueller Basisklassen:

Listing 3.11: Initialisierungsreihenfolge bei nicht-virtuellen Basisklassen

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

struct X {
  X(string s="Standardkonstruktor in X ") : info(s) { cout << s << endl; }
  string info;
};


struct Z : X {
  Z(string s="X() in Z ") : X(s) { cout << "Konstr. in Z "<< endl;}
};

struct Y : X {
  Y(string s="X() in Y ") : X(s) { cout << "Konstr. in Y "<< endl;}
};

struct W : X, Y, Z // <-- bestimmt Reihenfolge der Konstr-Aufr.
{
  W() : Y("Y() in W"),  Z("Z() in W") { }
  // warning: direct base 'X' inaccessible in 'W' due to ambiguity
  // Bem: Es wird automatisch X() aufgerufen!
} w;


int main() {
  cout << "w.Y::info: " << w.Y::info << endl;
  cout << "w.Z::info: " << w.Z::info << endl;
}

/*
Standardkonstruktor in X
Y() in W
Konstr. in Y
Z() in W
Konstr. in Z

w.Y::info: Y() in W
w.Z::info: Z() in W
*/
```



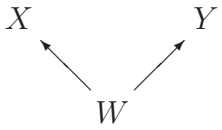
Beachte: Die Reihenfolge der Konstruktoraufrufe der Basisklasse hängt **nicht** von der Reihenfolge in der Initialisierungsliste ab! Dies hat z.B. Auswirkungen, wenn Attribute der Klasse mit Attributwerten von Subobjekten initialisiert werden. Virtuelle Oberklassen werden abweichend von der Reihenfolge in der struct-/class-Kopfzeile immer zuerst initialisiert!

Eine fehlerhafte Initialisierungsreihenfolge zeigt

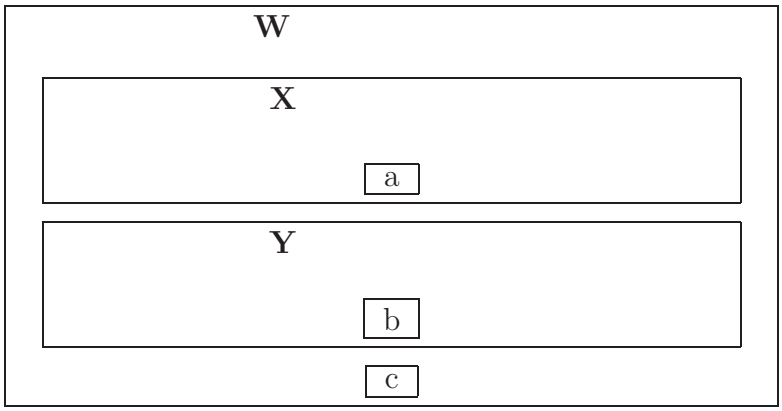
Listing 3.12: fehlerhafte Initialisierungsreihenfolge

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

struct X {
   X(int k=1) : a(k) {}
   int a;
};

struct Y {
   Y(int k=2) : b(k) {}
   int b;
};


struct W : X, Y { // <-- Reihenfolge ist ausschlaggebend fuer Initialisierung
   int c;
   W() : Y(7), X(b), c(a) {}; //Fehlerhaft, da zuerst X(b) und dann erst Y(7)!
                              //Attribut Y::b ist nicht rechtzeitig init.
} w;

int main(){
   cout << w.c << endl;
```

```
}
/*
 0     // Es wird also nicht, wie wohl erwartet, 7 ausgegeben
*/
```



bzw.



3.8 Polymorphie und virtuelle Destruktoren

Listing 3.13: Virtuelle Destruktoren

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

class X {
 public:
   X(int k=1) : a(k) { cout << "Konstr. der Basisklasse" << endl; }
   virtual ~X() { cout << "Destr. in X " << endl; } // virtueller Destr.
   virtual void info() { cout << " X-Objekt " << endl; }
   int a;
};

class Y : public X {
 public:
   Y() : X(5) { cout << "Konstr. der abgel. Klasse" << endl;}
   virtual ~Y() { cout << "Destr. in Y " << endl; }
   void info() { cout << " Y-Objekt " << endl; }
};


int main(){
   const int n=2;
   X* feld[n]; //Feld von Zeigern auf Basisklassenobjekte
   feld[0]= new X(1);
   feld[1]= new Y;
   for (int i=0; i<n; i++)
   {
      (* feld[i]). info ();
       // dyn. Typ des Zeigers feld[i] ist ausschlaggebend!
       delete feld[i];
   }
}
/*
Konstr. der Basisklasse
Konstr. der Basisklasse
Konstr. der abgel. Klasse
 X-Objekt
Destr. in X
 Y-Objekt
Destr. in Y
Destr. in X
*/
```

Werden Basisklassenzeiger oder -referenzen auf dynamisch erzeugte Objekte benutzt, so sollten virtuelle Destruktoren verwendet werden!

3.9 Rein virtuelle Funktionen, abstrakte Basisklassen

Wird die Funktionsdeklaration einer virtuellen Funktion durch „`= 0;`“ statt „`{ ...; }`“ abgeschlossen, so handelt es sich um eine **rein virtuelle** Funktion. Dies bedeutet, dass auf eine Realisierung dieser Funktion in der Klasse verzichtet wird. Eine solche Funktion ist nur dafür vorgesehen, in abgeleiteten Klassen definiert und verwendet zu werden. Lediglich das Interface (=Signatur) dieser Funktion wird vorgeschrieben und vererbt, nicht jedoch die (nicht vorhandene) Implementierung.

Eine Klasse mit einer rein virtuellen Funktion heißt **abstrakte** Klasse. Von einer abstrakten Klasse können keine Objekte erzeugt werden. Zeiger auf Objekte einer solchen Klasse sind dagegen erlaubt (Verwendungszweck: Polymorphie)!

Listing 3.14: Rein virtuelle Funktionen

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

class Base { //abstrakte (Basis-)Klasse
 public:
  virtual void print() = 0;  //rein  virtuelle  Funktion
}; //Es koennen keine Objekte dieser Klasse erzeugt werden

class X: public Base {
 public:
  virtual void print() { cout << "print() von Klasse X" << endl; }
} x;

int main() {
    Base* p= &x; //erlaubt
    x.print();
    p->print();  //dynamischer Typ des Pointers ist ausschlaggebend
}
/*
print() von Klasse X
print() von Klasse X
*/
```

Ein Feld von (Basisklassen-)Zeigern kann verwendet werden, um eine heterogene Struktur aufzubauen (Datenbank, o.ä.). Dies zeigt das folgende Beispiel:

Listing 3.15: heterogenes Feld

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

struct Fahrzeug { //abstrakte (Basis-)Klasse
  virtual void print() = 0;  //rein virtuelle Funktion
}; //Es sind Zeiger auf abstr. Klasse moeglich

struct Auto : Fahrzeug {
  virtual void print() { cout << "Auto" << endl; }
  int a;
   //...
};

struct Boot : Fahrzeug {
  virtual void print() { cout << "Boot" << endl; }
  int b;
   //...
};


int main() {
   const int n=4;
   Fahrzeug* p[n]; //Feld von Zeigern auf Basisklasse
   p[0]= new Boot;
   p[1]= new Auto;
   p[2]= new Auto;
   p[3]= new Boot;
   for(int i=0; i<n; i++) { cout << i << " "; p[i]->print(); }
}
/*
0 Boot
1 Auto
2 Auto
3 Boot
*/
```

(Nicht rein) Virtuelle Funktionen dienen als polymorph benutzbare Methoden, bei denen das Interface und eine Defaultimplementierung vorgegeben ist und vererbt wird.

Nichtvirtuelle Funktionen schreiben Interface und Implementierung fest vor und vererben sie.

Kapitel 4

Funktionsobjekte

Funktionsobjekte sind gewöhnliche Objekte einer Klasse. Für sie ist der Funktionsaufrufoperator `()` überladen, so dass „Aufrufe“ der Form `objektname(...)` möglich sind (äußere Form eines Funktionsaufrufs).

Listing 4.1: Funktionsobjekte

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

class FunkObj { //Klasse, die den Funktionsobjekten zugrunde liegt
   double a, b, c; //Koeff. des quadr. Polynoms a*x*x + b*x + c
 public:
   FunkObj(double a, double b=0, double c=0) {//Allgem. Konstruktor
      FunkObj::a=a; FunkObj::b=b; FunkObj::c=c;
   }
   void operator()(double a, double b, double c) { //Koeff. neu waehlen
      FunkObj::a=a; FunkObj::b=b; FunkObj::c=c;
   }
   double operator()(double x) { return a*x*x + b*x + c; }  //Auswertung
   friend FunkObj operator+(FunkObj& f, FunkObj& g) { //!
      return FunkObj(f.a+g.a, f.b+g.b, f.c+g.c );       //!
         }                                              //!
};

int main() {
   FunkObj p(1,2,3), q (0,1,1);  //Polynome p(x):= x*x+2*x+3 und q(x):=x+1
   cout << p(3) << endl;
   p (1,1,1);  //Polynom p(x)= x*x+x+1
   cout << p(3) << endl;
   cout << (p+q)(3) << endl; //!
}
/*
18
13
 17   //!
*/
```

Funktionsobjekte können wie Funktionen aufgerufen werden.

4.1 Ein Zufallszahlengenerator als Funktionsobjekt

Hier das Beispiel eines Zufallszahlengenerators (pseudo random number generator) zum Erzeugen von ganzen Zahlen im Bereich 0 bis 2147483647-1 (linearer Kongruenzgenerator für Pseudozufallszahlen):

Listing 4.2: Funktionsobjekte zum Erzeugen von Zufallszahlen

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

class RandGen { //Klasse, fuer die Funktionsobjekte erzeugt werden
   long z; //aktuelle  Zufallszahl
   static long a, m, q, r; //statische  Attribute
 public:
     RandGen(long seed=314159); //allgem. Konstruktor
     long operator()(); //Ueberladung des Funktionsaufrufoperators
};

long RandGen::a= 16807;
long RandGen::m= 2147483647; //Modul
long RandGen::q= m/a; //ganzzahlige Division
long RandGen::r= m%a; //Rest bei ganzzahliger Division

RandGen::RandGen(long seed=314159){ z= seed; } //Def. des Konstruktors

long RandGen::operator()() {
   long gamma;
   gamma= a*(z%q) - r*(z/q);
   if(gamma>0) z= gamma; else z= gamma + m;
   return z;
}

int main() {
   RandGen rand;
   for (int i= 0; i<10; i++) cout << rand()%100 << endl;
}
/*
19
10
47
39
39
59
83
65
28
57
*/
```

4.2 Funktionsobjekte mit Parametern

Funktionsobjekte können Parameter enthalten:

Listing 4.3: Funktionsobjekte mit Paramater

```
#include <iostream>
#include<cassert> //Prueft Bedingungen: Abbruch, falls nicht erfuellt
#include<cmath>   //fabs()

using namespace std;

class FunkObj {//Klasse, die den Funktionsobjekten zugrunde liegt
   double a, b, c; //Koeff. des quadr. Polynoms a*x*x + b*x + c
 public:
   FunkObj(double a, double b=0, double c=0) {//Allgem. Konstruktor
      FunkObj::a=a; FunkObj::b=b; FunkObj::c=c;
   }
   void operator()(double a, double b, double c) { //Koeff. neu waehlen
      FunkObj::a=a; FunkObj::b=b; FunkObj::c=c;
   }
   double operator()(double x) { return a*x*x + b*x + c; }  //Auswertung
   friend FunkObj operator+(FunkObj& f, FunkObj& g) { //!
      return FunkObj(f.a+g.a, f.b+g.b, f.c+g.c );       //!
         }                                               //!
};

double root(double links, double rechts, FunkObj f) {//Halbierungsverfahren
   const double eps= 0.00001; //Abbruchbedingung
   double mitte;
   assert( links <= rechts );       //Pruefe  Intervalleigenschaft
   assert(f(links)*f(rechts)<=0);//Pruefe VZ-Bedingung
   do {
      mitte= links + 0.5*(rechts-links);
      if (f(links)*f(mitte)>0) links= mitte; else rechts= mitte;
   } while (fabs(f(mitte)) > eps);
   return mitte;
}

int main() {
   FunkObj p(1,0,-1); //Polynom x*x-1
   cout << root(0,2.3,p) << endl;
   p(1,0,-0.49); //Polynom x*x-0.49
   cout << root(0,2.3,p) << endl;
   FunkObj q  (1,0,-0.36);                  //!
   double r    ;                            //!
   cout << (r = root(0,2.3,p+q)) << endl; //!
   cout << (p+q)(r) << endl ;               //!
}
/*
1
0.7
 0.65192       //!
-7.91135e-07  //!
*/
```

4.3 Funktionsobjekte als Templateparameter

Listing 4.4: Funktionsobjekte als Templateparameter

```
#include <iostream>
#include<cassert> //Prueft Bedingungen: Abbruch, falls nicht erfuellt
#include<cmath>   //fabs()

using namespace std;

class FunkObj {//Klasse, die den Funktionsobjekten zugrunde liegt
   double a, b, c; //Koeff. des quadr. Polynoms a*x*x + b*x + c
 public:
   FunkObj(double a, double b=0, double c=0) {//Allgem. Konstruktor
      FunkObj::a=a; FunkObj::b=b; FunkObj::c=c;
   }
   void operator()(double a, double b, double c) { //Koeff. neu waehlen
      FunkObj::a=a; FunkObj::b=b; FunkObj::c=c;
   }
   double operator()(double x) { return a*x*x + b*x + c; }  //Auswertung
   friend FunkObj operator+(FunkObj& f, FunkObj& g) { //!
      return FunkObj(f.a+g.a, f.b+g.b, f.c+g.c );       //!
   }
};

template<class T>
double root(double links, double rechts, T f) {//Halbierungsverfahren
   const double eps= 0.001; //Abbruchbedingung
   double mitte;
   assert(links <= rechts );       //Pruefe  Intervalleigenschaft
   assert(f(links)*f(rechts)<=0);//Pruefe VZ-Bedingung
   do {
      mitte= links + 0.5*(rechts-links);
      if (f(links)*f(mitte)>0) links= mitte; else rechts= mitte;
   } while (fabs(f(mitte)) > eps);
   return mitte;
}

int main() {
   FunkObj p(1,0,-0.49); //Polynom x*x-0.49
   FunkObj q(1,0,-0.36); //         x*x-0.36
   cout << root(0,2.3,p+q) << endl;
   cout << root(2.5,3.8,sin) << endl;
}
/*
0.651929
3.14238
*/
```