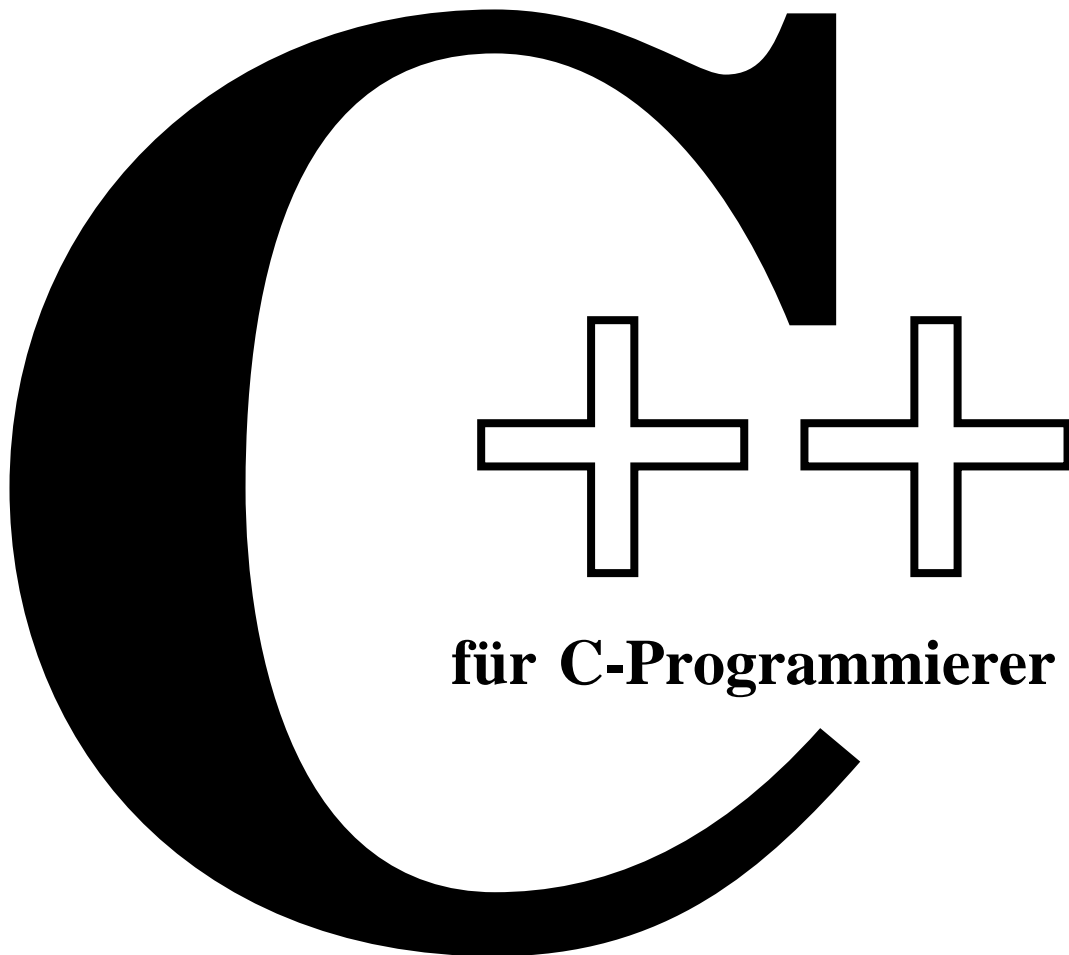


C und C++ für UNIX, DOS und MS-Windows, Teil 3:



für C-Programmierer

Dies ist weder ein Manual noch ein "normales" Vorlesungs-Skript ("normale" Vorlesungen über eine Programmiersprache sind wohl ohnehin langweilig). Es soll in erster Linie eine Hilfe zum Selbststudium sein (und wird deshalb als "Tutorial" bezeichnet).

Die im Skript abgedruckten Programme (Quelltext) können über die Internet-Adresse

http://www.fh-hamburg.de/rzbt/dankert/c_tutor.html

kopiert werden.

Inhalt (Teil 3)

12	Einführung in C++ für C-Programmierer	201
12.1	Einige eher formale Unterschiede zur Sprache C	201
12.2	Klassen und Kapselung	207
12.2.1	Daten und Methoden in der Klasse	208
12.2.2	Konstruktoren und Destruktoren	212
12.2.3	Objektorientiertes Programmieren fordert objektorientiertes Denken	215
12.3	Vererbung	218
12.4	Virtuelle Funktionen, abstrakte Klassen (Polymorphismus)	225
12.5	Überladen	235
12.6	Eingabe und Ausgabe, Arbeiten mit Dateien	239
12.6.1	Das Klassen-Objekt cout	239
12.6.2	Dateien	241
12.6.3	Die Methoden ostream::write und istream::read, Binär-Dateien	245
12.7	Was man sonst noch wissen sollte	249
12.7.1	Arbeiten mit friend-Funktionen und friend-Klassen	249
12.7.2	Der this-Pointer	250
12.7.3	Mehrfach-Vererbung	251
12.7.4	Virtuelle Basisklassen	255
12.7.5	Das Schlüsselwort const	256
12.7.6	inline-Funktionen	257
12.7.7	static-Variablen in Klassen-Deklarationen	258

"Im Vergleich mit so schönen Namen wie 'Fortran', 'Algol', 'Pascal', 'Modula', 'Ada', 'Smalltalk' oder 'Java' klingt 'C' doch recht bieder."
"C++ klingt sogar etwas nach Waschmittel-Werbung."

12 Einführung in C++ für C-Programmierer

Die gute Nachricht zuerst: Die Programmiersprache C ist in der Sprache C++ komplett enthalten, so daß nichts von dem, was beim Durcharbeiten dieses Tutorials bisher gelernt wurde, ungültig wird.

Die folgende Aussage sollte man nicht als die "schlechte Nachricht" auffassen: C++ ist nicht nur eine Erweiterung der Programmiersprache C, sondern fordert einen ganz neuen Denkansatz, wenn man die Vorteile dieser Sprache voll nutzen will. Der entscheidende Vorteil ist die Unterstützung der sogenannten "objektorientierten Programmierung" (dazu mehr ab Abschnitt 12.2).

12.1 Einige eher formale Unterschiede zur Sprache C

Weil die Programmiersprache C komplett in C++ enthalten ist, liefern einige Compiler-Hersteller nur noch die C++-Versionen ihrer Produkte aus. Wer also seine C-Übungen schon auf einem C++-Compiler absolviert hat, muß sich im Umgang mit dem Compiler (und gegebenenfalls der gesamten Entwicklungsumgebung) kaum umstellen. Wer einen GNU-Compiler unter UNIX benutzt hat, muß eventuell den Compiler-Aufruf von **gcc** nach **g++** ändern. Zu beachten ist, daß einige Compiler den Quellcode in Files erwarten, deren Namen mit ganz bestimmten Extensions versehen sind, üblich sind **.cpp** oder **.cxx**.

Die nachfolgenden Programme **hllworld.cpp** und **reihe05.cpp** entsprechen in ihrer Funktionalität etwa den C-Programmen **hllworld.c** aus dem Abschnitt 3.4 bzw. **reihe01.c** aus dem Abschnitt 3.7. Sie zeigen einige eher formale Unterschiede der C++-Syntax zur C-Syntax, demonstrieren aber auch, daß die C-Syntax in C++ komplett enthalten ist und mit dieser beliebig gemischt werden darf:

- ◆ **Kommentar** darf in C++ mit // eingeleitet werden. Nach diesen beiden Zeichen gilt der gesamte Rest einer Zeile als Kommentar (Abschluß des so eingeleiteten Kommentars ist jeweils das Zeilenende).

Empfehlung: Man verwende ausschließlich die C++-Version, auch wenn bei Kommentaren über mehrere Zeilen die C-Version bequemer erscheinen mag. Die C-Version bleibt dann verfügbar, um längere Code-Passagen bei Bedarf "herauskommentieren" zu können, ohne daß unerlaubte Kommentar-"Schachtelung" zu befürchten ist.

- ◆ **Variablen dürfen an beinahe beliebiger Stelle in einem Programm definiert werden.** Während in C die Definitionen am Anfang eines Blocks stehen müssen (ein Block wird durch die öffnende Klammer { eingeleitet), gilt in C++ nur die Bedingung, daß sie **vor ihrer ersten Verwendung** zu definieren sind. In jedem Fall ist die Gültigkeit einer Variablen auf den Block beschränkt, in dem sie definiert ist. Zu empfehlen ist, von der zusätzlich in C++ gegebenen Freiheit dann Gebrauch zu machen, wenn die Variable nur in unmittelbarer Umgebung ihrer Definition gebraucht wird, z. B. kann eine nur im Kopf einer **for**-Schleife benutzte Zählgröße entsprechend

```
for ( int i = 0 ; i < n ; i++ )
```

genau an dieser Stelle auch definiert werden (vgl. auch die Definition der Pointer-Variablen im Zusammenhang mit dem Allokieren von dynamischem Speicherplatz im Programm **division.cpp** am Ende dieses Abschnitts).

- ◆ Das **"Casten" einer Variablen** auf einen bestimmten Typ, das in C z. B. als

```
(double) ixyz
```

codiert wird, darf in C++ auch in der Form

```
double (ixyz)
```

geschrieben werden.

- ◆ In den nachfolgenden Programmen scheint der Unterschied zwischen den Bibliotheksfunktionen von C für die **Aus- und Eingabe (printf bzw. scanf)** und den "Objekten" **cout** und **cin**, die dafür in C++ vorgesehen sind, auch eher formaler Natur zu sein. Der Schein trügt. Hinter diesen Objekten steckt schon eine für die objektorientierte Sprache ganz typische Strategie, die hier noch nicht erläutert werden soll (außerdem gilt natürlich nach wie vor die Bemerkung aus dem Abschnitt 3.12, daß die "Schönheit der Bildschirmausgabe" der Windows-Programmierung vorbehalten sein sollte). Der "Hello World"-Klassiker kann in C++ z. B. wie folgt geschrieben werden:

```
// "Hello World"-Programm in C++ (Programm hllworld.cpp)
// =====
#include <iostream.h>                // ... fuer cout
main ()
{
    cout << "HELLO, 'C++'-WORLD\n" ;
    return 0 ;
}
```

- ◆ Registrieren Sie zunächst nur, daß man z. B. mit

```
cout << "Text"
```

einen Text zur Standardausgabe (das ist in der Regel der Bildschirm) schicken kann (natürlich sind auch Formatierungen möglich) und mit

```
cin >> xyz
```

von der Standardeingabe (das ist in der Regel die Tastatur) einen Wert auf eine Variable überträgt (demonstriert im nachfolgenden Programm). Bei der Verwendung von **cin** und **cout** ist die Header-Datei **iostream.h** einzubinden.

```

// Untersuchung der harmonischen Reihe (Programm reihe05.cpp)
// =====
/* Programm demonstriert
   * die Moeglichkeit, C++-Syntax mit C-Syntax zu mischen (dieser Kommentar
     wird z. B. im C-Stil geschrieben, die Ueberschrift hat die Syntax
     von C++-Kommentar),
   * die Freiheit bei der Positionierung von Variablen-Definitionen,
   * die verschiedenen Syntax-Varianten fuer "Casts",
   * jeweils eine ganz einfache Anwendung von 'cin' und 'cout'.          */
#include <stdio.h>                // ... fuer printf
#include <iostream.h>            // ... fuer cin und cout

main ()
{
    long    zaehler = 2 , nenner    = 1 , grenze ;

    cout << "Bitte Obergrenze fuer Reihensumme eingeben: " ; // Ausgabe ...
    cin  >> grenze ; // ... und Eingabe mit C++-Objekten

    double  summe    = 0. , zielsumme = 1. , dnenner ; // ... zeigt, dass
                // Definitionen von Variablen an beinahe
                // beliebigen Stellen im Programm stehen duerfen

    while (zielsumme <= grenze)
    {
        dnenner = double (nenner) ; // "Cast" mit C++-Syntax und ...
        summe += (double) zaehler / (dnenner * dnenner) ; // ... C-Syntax
        if (summe >= zielsumme)
        {
            printf ("Summe %8.2lf erreicht mit %10ld Summanden\n" ,
                    summe , nenner) ; // Ausgabe mit C-Funktion printf
            zielsumme += 1. ;
        }
        zaehler++ ;
        nenner++ ;
    }

    return 0 ;
}

```

In der C-Programmierung gilt die Regel, daß die Werte von Argumenten, die an eine Funktion übergeben werden, nicht geändert werden können, die aufgerufene Funktion erhält nur Kopien der Werte ("Call by value"). C++ kennt dagegen auch die Übergabe von Argumenten "**by reference**". Dies wird bei Definition und Deklaration (Prototyp) einer Funktion durch ein dem Parameter voranzustellendes **&** veranlaßt.

Damit ist eine einfachere Möglichkeit gegeben, Werte von Argumenten ändern zu lassen. Der Zwang, in einem solchen Fall mit Pointern arbeiten zu müssen, entfällt.

Empfehlung: Während der Compiler bei Arrays in C (und damit auch in C++), wenn sie als Argumente eines Funktions-Aufrufs erscheinen, dafür sorgt, daß nur die Pointer übergeben werden, erfolgt bei Strukturen (vgl. Kapitel 7) die Übergabe "by value". Deshalb sollte der C-Programmierer bei umfangreichen Strukturen selbst für eine Pointer-Übergabe sorgen. Dies kann in C++ durch die Übergabe "by reference" ersetzt werden. Es ist in jedem Fall sinnvoll, Strukturen (und ganz besonders Klassen-Objekte, die im Abschnitt 12.2 eingeführt werden) "by reference" zu übergeben.

Die in der Programmiersprache C geltende Regel, daß durch den Aufruf einer Funktion

```
return_wert = aufgerufene_funktion (argument) ;
```

der Wert der Variablen **argument** nicht geändert werden kann (auch nicht bei Arrays und Strings, denn dafür werden ja nur die nicht zu ändernden Pointer übergeben), gilt in C++ nicht. Mit genau diesem Funktionsaufruf kann der Wert von **argument** in der aufgerufenen Funktion verändert werden, wenn diese z. B. in der Form

```
int aufgerufene_funktion (int &argument)
{ // ...
}
```

definiert wird. Das Zeichen **&** vor dem Namen des Arguments veranlaßt den Compiler, dieses "by reference" zu übergeben. Es kann in der aufgerufenen Funktion geändert werden, diese Änderung gilt auch für den Wert in der aufrufenden Funktion.

Bei großen Strukturen und besonders bei Klassen-Objekten (Abschnitt 12.2) ist die Übergabe "by reference" in jedem Fall empfehlenswert.

Das nachfolgende Programm demonstriert die unterschiedlichen Möglichkeiten der Definition von Funktionen an einem einfachen Beispiel:

```
// Tauschen zweier Werte in C und C++ (Programm tauschen.cpp)
// =====

// Programm demonstriert

// * die Moeglichkeit, in C++ Argumente an eine Funktion "by reference"
//   zu vermitteln,

// * die Verknuepfung mehrerer Bestandteile einer Ausgabe, die an cout
//   geschickt wird, durch den Operator <<.

// Es werden drei Funktionen benutzt, mit denen zwei Werte vertauscht werden
// sollen:

// - An change_err werden die Argumente "by value" vermittelt, die Funktion
//   bekommt nur Kopien der Werte, das Tauschen misslingt.

// - An change_c werden die Pointer auf die Werte vermittelt, die Funktion
//   kennt also die "Adressen" und kann die Werte aendern.

// - Die Funktion change_cpp stellt die nur in C++ vorgesehene zusaetzliche
//   Moeglichkeit dar, Argumente "by reference" zu uebergeben. Der Aufruf
//   dieser Funktion sieht exakt wie der Aufruf der Funktion change_err aus,
//   der Prototyp von change_cpp (und natuerlich die Kopfzeile der Funktions-
//   Definition) veranlassen den Compiler allerdings, die Argumente
//   "by reference" zu uebergeben, so dass ihre Werte geaendert werden
//   koennen.

#include <iostream.h>

void change_err (int , int ) ;
void change_c  (int * , int *) ;
void change_cpp (int & , int &) ;

main ()
{
  int wert1 = 10 , wert2 = 20 ;
```

```

cout << "Vorbelegung:          wert1 = " << wert1
      << ",          wert2 = " << wert2 ;

change_err (wert1 , wert2) ;
cout << "\nNach change_err:    wert1 = " << wert1
      << ",          wert2 = " << wert2 ;

change_c (&wert1 , &wert2) ;
cout << "\nNach change_c:      wert1 = " << wert1
      << ",          wert2 = " << wert2 ;

change_cpp (wert1 , wert2) ;
cout << "\nNach change_cpp:    wert1 = " << wert1
      << ",          wert2 = " << wert2 ;

cout << "\n\nDer Tauschversuch mit change_err misslingt (natuerlich!)."
      << "\n\nAn change_c werden die Pointer auf die zu tauschenden"
      << " \nWerte vermittelt, die Funktion kann die Werte tauschen."
      << "\n\nDer Aufruf von change_cpp sieht exakt so aus wie der"
      << " \nAufruf von change_err, trotzdem kann change_cpp die Werte"
      << " \nntauschen, weil die Funktion sie \"by reference\" erhaelt"
      << " \n(veranlasst durch die Zeichen & im Protoyp und in der"
      << " \nKopfzeile der Funktion)." ;

return 0 ;
}

void change_err (int w1 , int w2)
{
    int ws ;

    ws = w1 ;          // ... und das ist alles vergeblich, weil change_err
    w1 = w2 ;          //          nur Kopien der Werte bekommt, in der aufrufenden
    w2 = ws ;          //          Funktion aendert sich kein Wert.
}

void change_c (int *w1_p , int *w2_p)
{
    int ws ;

    ws    = *w1_p ; // ... hier kommen die Pointer w1_p und w2_p an, die
    *w1_p = *w2_p ; //          dereferenziert werden, die Werte der Variablen
    *w2_p = ws      ; //          in der aufrufenden Funktion aendern sich.
}

void change_cpp (int &w1 , int &w2)
{
    int ws ;

    ws = w1 ;          // ... und das geht nur in C++: Es sieht so aus wie in
    w1 = w2 ;          //          change_err, der Compiler sorgt aber dafuer, dass
    w2 = ws ;          //          es wie in change_c ablaeuft.
}

```

- ◆ Man beachte, daß **am Funktionsaufruf nicht zu erkennen ist**, ob die Argumente "by value" oder "by reference" übergeben werden.

Für das **dynamische Allokieren von Speicherplatz** steht in C++ der Operator **new** zur Verfügung. Er ist vergleichbar mit der C-Funktion **calloc**, allerdings einfacher zu handhaben: Dem Operator **new** müssen der Datentyp und gegebenenfalls zusätzlich die Anzahl der Daten (in eckigen Klammern) folgen, für die Speicherplatz bereitgestellt werden soll, er liefert einen Pointer auf eine Variable dieses Typs ab (im Unterschied zur Funktion **calloc**, die einen "Pointer auf void" liefert).

```

double *a_p , *xy_p ;
// ...
a_p = new double ;
xy_p = new double [100] ;

```

... stellt Speicher für einen double-Wert bereit, auf den der Pointer **a_p** zeigt, und reserviert einen Speicherbereich für 100 double-Werte, auf den **xy_p** pointert.

Bei einem Mißerfolg der **new**-Operation wird der **NULL**-Pointer abgeliefert, dies sollte in jedem Fall abgefragt werden. Mit **new** allokierte Speicherbereiche können mit dem Operator **delete** (entspricht weitgehend der C-Funktion **free**) wieder freigegeben werden, für das betrachtete Beispiel wäre

```

delete a_p ;
delete xy_p ;

```

zu codieren.

Das nachfolgende Programm **division.cpp** demonstriert das Arbeiten mit den Operatoren **new** und **delete**. Zur Funktionalität des Programms: Bei der Division zweier ganzer Zahlen entsteht entweder ein endlicher oder ein periodischer Dezimalbruch. Bei der Berechnung der Nachkommastellen beginnt eine Wiederholung einer bereits vorher vorhandenen Ziffernfolge immer dann, wenn sich ein schon einmal aufgetretener "Divisionsrest" erneut ergibt.

```

// Ganzzahl-Division mit exaktem Ergebnis (Programm division.cpp)
// =====

// Der Quotient zweier positiver ganzer Zahlen wird exakt berechnet. Die
// Rechnung bricht erst ab, wenn die exakte Dezimalbruch-Darstellung
// erreicht ist oder eine Periode erkannt wird.

// Um die Periode zu erkennen, muessen alle "Divisionsreste" gespeichert
// werden. Da bei einer Division durch q maximal q-1 unterschiedliche
// Divisionsreste moeglich sind, muss ein entsprechend grosses Array
// bereitgestellt werden.

// Das Programm demonstriert das dynamische Allokieren von Speicherplatz
// mit dem Operator 'new' und die Freigabe mit 'delete'.

#include <iostream.h>

main ()
{
    long i , j , p , q , divid , ziffer ;

    cout << "Exakte Berechnung des Quotienten p/q\n"
         << "(p und q sind positive ganze Zahlen)\n"
         << "=====\n\n" ;

    do { cout << "Dividend:      p = " ; cin >> p ;
         cout << "Divisor:       q = " ; cin >> q ;
         } while (p < 0 || q <= 0) ;

    long *rest_p = new long [q] ; // ... allokiert Speicherplatz fuer q
                                // long-Werte, liefert Pointer auf
                                // Pointer-Variable 'rest_p' ab, die
                                // gleichzeitig definiert wird (vgl.
                                // Kommentar am Programmende).

    if (rest_p == NULL)
    {
        cout << "Sorry, nicht genuegend Speicherplatz!\n" ;
        return 1 ;
    }
}

```

```

cout << "\n" << p << "/" << q << " = " << p/q << ", " ;
*rest_p = p % q ; // ... ist der erste "Divisionsrest".

for (i = 1 ; i < q ; i++)
{
    divid = *(rest_p + i - 1) * 10 ;
    ziffer = divid / q ;
    cout << ziffer ;
    *(rest_p + i) = divid - ziffer * q ;

    if (*(rest_p + i) == 0) break ;

    for (j = 0 ; j < i ; j++)
    {
        if (*(rest_p + j) == *(rest_p + i))
        {
            cout << "... \n*** Periodischer Dezimalbruch, "
                << "Periode hat " << i - j << " Ziffer(n) ***" ;
            i = q ; // ... sorgt fuer Abbruch der aeusseren Schleife.
            break ;
        }
    }
}

cout << "\n" ;

delete rest_p ; // ... weil es guter Programmierstil ist, Speicherplatz
               // wird am Ende des Programms ohnehin freigegeben.
return 0 ;
}

// Natuerlich kann man die Definition der Pointer-Variablen auch an den
// Programmanfang stellen:

//         long *rest_p ;
//         ...
//         rest_p = new long [q] ;

// waere also gleichwertig. Auch die Erfolgsabfrage kann man gleich in
// die Speicherplatzanforderung integrieren:

//         if ((rest_p = new long [q]) == NULL)
//         {
//             cout << "Sorry, nicht genuegend Speicherplatz!\n" ;
//             return 1 ;
//         }

```

12.2 Klassen und Kapselung

Die **Klasse (class)** ist der zentrale Begriff der **objektorientierten Programmierung** in der Sprache C++. Die Klasse ist eine Erweiterung der aus der C-Programmierung bekannten **Struktur (struct)**. Hier werden zunächst nur zwei markante neue Eigenschaften betrachtet (daß in C++ die Strukturen auch mit diesen Eigenschaften ausgestattet sind, wird nicht weiter erwähnt, es ist empfehlenswert, Objekte grundsätzlich als Klassen zu vereinbaren):

- ◆ Die Daten-Elemente einer Klasse sind per Voreinstellung **private**, auf sie kann nicht (wie auf die **public** voreingestellten Daten einer Struktur) direkt zugegriffen werden.
- ◆ Neben den Daten enthält eine Klasse auch Funktionen, mit denen die Daten der Klasse manipuliert werden können. Diese zur Klasse gehörenden Funktionen werden

im folgenden als **Methoden** bezeichnet (es sind auch die Begriffe "Elementfunktionen", "Schnittstellenfunktionen" bzw. "Member Functions" gebräuchlich). Die Methoden sind per Voreinstellung **public**, können also aus allen Programmteilen aufgerufen werden, für die die Klassendeklaration "sichtbar" ist.

Hier wird zunächst eine besonders wichtige Eigenschaft der Klassen betrachtet: Man kann die in einer Klasse definierten Daten **kapseln**, sie gewissermaßen vor dem direkten Zugriff des Programmierers schützen, der sie nur über die zur Klasse gehörenden Methoden manipulieren kann.

Natürlich ist diese Art der Programmierung aufwendiger, aber auch deutlich sicherer und vor allem wartungsfreundlicher. Sie wurde lange vor der Erfindung der inzwischen zum Modewort verkommenen "objektorientierten" Vorgehensweise von Projektgruppen praktiziert, die große Softwarepakete herstellen. Der wesentliche Vorteil ist die Möglichkeit, die in einer Klasse definierte Datenstruktur zu ändern (z. B. mehrere Variablen des gleichen Typs zu einem Array zusammenzufassen), ohne daß in den übrigen Programmteilen geändert werden muß, wenn man die Methoden der geänderten Datenstruktur anpaßt (die Prototypen der Methoden dürfen sich natürlich nicht ändern).

12.2.1 Daten und Methoden in der Klasse

Obwohl es für die Elemente einer Klasse Voreinstellungen hinsichtlich ihrer Eigenschaften gibt, werden im folgenden immer trotzdem die Schlüsselwort **private:** und **public:** bei der Klassen-Deklaration verwendet, was ohnehin empfehlenswert ist. Diese Schlüsselworte können beliebig oft in der Deklaration einer Klasse verwendet werden, sie gelten jeweils bis zum Auftreten des nächsten Schlüsselwortes.

Bemerkung zur Wortwahl: Am Beginn des Kapitels 5 wurde auf den Unterschied zwischen Deklaration (Beschreibung von Eigenschaften, z. B.: Prototyp einer Funktion) und Definition (Bereitstellung von Speicherplatz bzw. des kompletten Funktions-Codes) hingewiesen. Bei der Deklaration einer Klasse werden auch nur die Eigenschaften festgelegt, es wird noch kein Objekt erzeugt. Da aber zu einer Klassen-Deklaration auch komplett ausprogrammierte Funktionen (Methoden) gehören können, wird häufig der Begriff "Klassen-Definition" verwendet. Hier wird dafür konsequent der Begriff "Deklaration" verwendet, zumal in der Regel innerhalb der Klassen-Deklaration die Methoden nur durch Prototypen vertreten sind.

Die folgenden Beispiel-Programme enthalten keine nennenswerte Funktionalität, sie dienen nur der Demonstration der Syntax bei der Deklaration von Klassen. Daß diese außerhalb der Funktion **main** deklariert werden, ist nicht zwingend, Klassen können (wie Strukturen) innerhalb einer Funktion deklariert werden, sind dann allerdings auch nur innerhalb dieser Funktion "sichtbar". Für die Anwendung typisch ist der Zugriff aus verschiedenen Programmteilen, so daß sich die Deklaration der Klassen in (über **#include**-Statements einzubindenden) Header-Dateien anbietet. Das erste Beispiel zeigt nur die Syntax der Deklaration einer Klasse und der Erzeugung einer **Instanz** dieser Klasse. Dabei wird deutlich, daß zur Klasse sowohl Daten als auch die Funktionen für den Zugriff auf die Daten (Methoden) gehören:

```

// Klasse und Methoden (Programm class01.cpp)
// =====

// Das Programm demonstriert das Deklarieren einer Klasse, die Definition
// einer Instanz der Klasse und den Zugriff auf Klassen-Daten ueber Methoden.

#include <iostream.h>

// Es wird eine Klasse 'point' deklariert, die zwei double-Variablen
// x und y enthaelt. Diese werden als 'privat' deklariert und
// koennen deshalb nicht direkt aus einem Programm angesprochen werden.

// Dagegen werden die vier zur Klasse gehoerenden Methoden als
// 'public' deklariert, koennen im Programm benutzt werden und
// dienen als Zugriffs-Funktionen zu den 'private'-Daten.

class point
{
    private:
        double x ;
        double y ;
    public:
        void set_x (double xc) { x = xc ; }
        void set_y (double yc) { y = yc ; }
        double get_x () { return x ; }
        double get_y () { return y ; }
} ;

main ()
{
    point center ; // ... definiert eine "Instanz" der Klasse
                  // 'point' mit dem Namen 'center'.

// Die nachfolgende (herauskommentierte) Anweisung wuerde einen
// Fehler beim Compilieren verursachen (probieren Sie es aus
// durch Entfernen der Kommentarstriche!). Auf 'private' definierte
// Daten einer Klasse kann nicht direkt zugegriffen werden:
// center.x = 4. ; // Falsch!!!

    center.set_x (4.) ; // Zugriff ueber die definierten
    center.set_y (7.) ; // Methoden

    cout << "Mittelpunkt: x = " << center.get_x ()
         << "\n" << "y = " << center.get_y () << "\n" ;

    return 0 ;
}

```

- ◆ Das Programm **class01.cpp** demonstriert mit der Klasse **point** den typischen Vertreter einer "gekapselten" Datenstruktur mit den beiden Koordinaten und den vier Methoden, zwei Methoden zum Ändern der Werte der Daten und zwei Methoden zur Abfrage. Da die vier Methoden nur ganz einfache Operationen ausführen, wurden sie komplett innerhalb der Klassen-Deklaration definiert. Dies ist eher die Ausnahme, im allgemeinen werden in Klassen-Deklarationen nur Prototypen aufgenommen.
- ◆ In **main** wird die **Instanz center** der Klasse **point** definiert. Von einer Klasse können beliebig viele Instanzen erzeugt werden (die Klasse **point** ist ein Datentyp, die Definition einer Instanz sieht auch genau so aus wie die Definition einer Variablen eines vordefinierten Typs, z. B. einer double-Variablen). Bei der Definition einer Instanz einer Klasse wird jeweils Speicherplatz für alle Daten-Elemente der Klasse reserviert, die zur Klasse gehörenden Methoden existieren nur einmal.

- ◆ Die Syntax für die Verwendung der Methoden entspricht der Syntax, mit der auf Komponenten einer Struktur zugegriffen wird: "Name der Instanz, Punkt, Name der Methode" entspricht "Name der Struktur, Punkt, Name der Komponente".

Das nachfolgende Programm **class02.cpp** hat die gleiche Funktionalität wie **class01.cpp**, es wird die typische Definition der Methoden außerhalb der Klassen-Deklaration gezeigt:

```
// Klasse und Methoden (Programm class02.cpp)
// =====

// Das Programm hat die gleiche Funktionalitaet wie class01.cpp.
// Es demonstriert die Definition von Methoden ausserhalb der
// Klassen-Deklaration.

#include <iostream.h>

// In der Klassen-Deklaration sind die vier Methoden nur durch
// Prototypen vertreten:

class point
{
    private:
        double x ;
        double y ;
    public:
        void    set_x (double) ;           // Prototypen der
        void    set_y (double) ;           // zur Klasse
        double  get_x () ;                 // gehoerenden
        double  get_y () ;                 // Methoden
} ;

// Die Definition der zur Klasse 'point' gehoerenden Methoden wird
// jeweils mit 'point::' eingeleitet (:: ist der "Gueltingkeits-
// operator", der den Bezug zur Klasse herstellt):

void  point::set_x (double xc) { x = xc ; }
void  point::set_y (double yc) { y = yc ; }
double point::get_x ()        { return x ; }
double point::get_y ()        { return y ; }

main ()
{
    point center ;

    center.set_x (4.) ;
    center.set_y (7.) ;

    cout << "Mittelpunkt:  x = " << center.get_x ()
         << "\n          y = " << center.get_y () << "\n" ;

    return 0 ;
}
```

Neben der Erhöhung der Sicherheit erfüllt die Daten-Kapselung einen entscheidenden weiteren Zweck:

Am Beginn der Bearbeitung eines Software-Projektes ist für den Entwurf der Datenstruktur stets (der niemals verfügbare) geniale, alles vorhersehende Programmierer gefragt. Die Kapselung der Datenstruktur gestattet nachträgliche Änderungen und verzeiht damit die (ohnehin unvermeidlichen) Entwurfsfehler.

Mit einer dritten Variante des Demonstrations-Programms soll gezeigt werden, daß sich eine Änderung der Datenstruktur innerhalb einer Klasse nicht auf den Programm-Code außerhalb der Klasse auswirken muß, weil die Daten "gekapselt" sind. Es müssen lediglich die zur Klasse gehörenden Methoden angepaßt werden:

```
// Klasse und Methoden (Programm class03.cpp)
// =====

// Das Programm hat die gleiche Funktionalitaet wie class02.cpp.
// Es demonstriert die Aenderung der Datenspeicherung in einer
// Klasse, die nur zu Aenderungen der zur Klasse gehoerenden Methoden
// zwingt, waehrend der restliche Programmcode davon nicht betroffen ist.

#include <iostream.h>

class point
{
    private:
        double xyz [3] ;
    public:
        void set_x (double) ; // Die Klasse 'point' wird
        void set_y (double) ; // erweitert und kann nun die
        void set_z (double) ; // Koordinaten eines 3D-Punktes
        double get_x () ; // in einem Array speichern.
        double get_y () ; // Zwei Methoden werden
        double get_z () ; // ergaenzt, die Prototypen
                          // der anderen Methoden
                          // sind unveraendert.
};

// Die Methoden, die bereits im Programm class02.cpp definiert wurden,
// muessen der geaenderten Datenstruktur angepasst werden:

void point::set_x (double xc) { xyz[0] = xc ; }
void point::set_y (double yc) { xyz[1] = yc ; }
void point::set_z (double zc) { xyz[2] = zc ; }
double point::get_x () { return xyz[0] ; }
double point::get_y () { return xyz[1] ; }
double point::get_z () { return xyz[2] ; }

// Von der Aenderung der Datenstruktur innerhalb der Klasse ist die
// Funktion 'main' nicht betroffen (Code identisch mit class02.cpp):

main ()
{
    point center ;

    center.set_x (4.) ;
    center.set_y (7.) ;

    cout << "Mittelpunkt: x = " << center.get_x ()
         << "\n" << "y = " << center.get_y () << "\n" ;

    return 0 ;
}
```

- ◆ Mit der Deklaration einer Klasse ist ein Datentyp entstanden, der mit ähnlicher Syntax das Erzeugen von Instanzen dieses Typs wie mit den vordefinierten Datentypen gestattet, z. B.:

```
double xyz ;
point center ;
```

Es fehlt noch die Möglichkeit, die Instanzen (wie die vordefinierten Datentypen) bei der Definition initialisieren zu können. Dies ist das Thema des folgenden Abschnitts.

12.2.2 Konstruktoren und Destruktoren

Beim Erzeugen einer Instanz einer Klasse wird immer eine spezielle Funktion aufgerufen, der sogenannte **Konstruktor**. Dieser wird entweder vom Compiler automatisch bereitgestellt ("Standard-Konstruktor"), kann (und sollte!) jedoch vom Programmierer geschrieben werden (der Compiler integriert in den vom Programmierer geschriebenen Konstruktor die Funktionalität, die ansonsten im Standard-Konstruktor steckt). Als Pendant dazu existiert immer ein **Destruktor**, der beim Erlöschen der Gültigkeit einer Instanz (z. B. am Ende des Blockes, in dem sie erzeugt wurde) aufgerufen wird.

Regeln für das Schreiben von Konstruktor und Destruktor:

- ◆ Der Name des Konstruktors muß immer mit dem Namen der Klasse identisch sein (der Compiler unterscheidet auf diese Weise den Konstruktor von den übrigen zur Klasse gehörenden Methoden).
- ◆ Der Name des Destruktors muß aus dem Namen der Klasse mit einer vorangestellten Tilde ~ gebildet werden.
- ◆ Der Konstruktor kann Argumente übernehmen (dies ist der Regelfall, da er vorwiegend zur Initialisierung von Daten der Klasse benutzt wird). Ein Destruktor kann grundsätzlich keine Argumente übernehmen.
- ◆ Weder Konstruktor noch Destruktor liefern einen Return-Wert ab, haben also auch keinen Typ (nicht einmal **void**).

Das nachfolgende Programm demonstriert den Gebrauch von Konstruktor und Destruktor mit den für beide Funktionen typischen Aufgaben:

- ◆ Mit dem Konstruktor werden die Daten einer neuen Instanz einer Klasse initialisiert. Dies kann durch im Konstruktor festgelegte Werte geschehen oder durch Steuerung über die Argumente, mit denen der Konstruktor (beim Erzeugen einer Instanz) aufgerufen wird. Im Programm **class04.cpp** wird die letztgenannte Variante verwendet, wobei zusätzlich die Möglichkeit genutzt wird, Default-Werte für die Initialisierung festzulegen.
- ◆ Die in **class04.cpp** gezeigte Verwendung von Default-Werten für Argumente ist nicht auf Konstruktoren beschränkt, sondern gilt in C++ für beliebige Funktionen: Bei der Deklaration können für die letzten (bzw. alle) Parameter Vorgabewerte festgelegt werden, die dann gelten, wenn die (letzten) Parameter beim Funktionsaufruf weggelassen werden.
- ◆ Im Konstruktor kann (mit **new**) Speicherplatz für die Daten der Klasse angefordert werden, der im Destruktor (mit **delete**) wieder freigegeben wird. Im Programm **class04.cpp** wird die Menge des anzufordernden Speicherplatzes sogar noch von den Argumenten beim Aufruf des Konstruktors abhängig gemacht (dies ist auch eine typische Strategie, wenn eine Klasse Strings enthält).

```

// Konstruktor und Destruktor (Programm class04.cpp)
// =====

// Das Programm ist eine Erweiterung des Programms class03.cpp, die Klasse
// 'point' wird fuer wahlweise Verwendung fuer 2D- bzw. 3D-Punkte ausgelegt.

// Das Programm demonstriert

// * die Definition von Konstruktor und Destruktor fuer eine Klasse,
// * das Erzeugen von Instanzen mit und ohne Nutzung der Vorbelegung von
//   Initialisierungswerten und teilweiser Nutzung der Vorbelegungswerte,
// * das dynamische Allokieren von Speicherplatz innerhalb des
//   Konstruktors, die Freigabe des Speicherplatzes im Destruktor.

#include <iostream.h>

void testproc () ;

class point
{
private:
    double *xyz ;           // Fuer die Aufnahme der Daten
public:                    // wird nur ein Pointer (kein
    void set_x (double) ;  // Speicherplatz fuer die
    void set_y (double) ;  // double-Werte) vereinbart.
    void set_z (double) ;
    double get_x () ;
    double get_y () ;
    double get_z () ;

    // Prototypen von Konstruktor und Destruktor (fuer die
    // Parameter des Konstruktors werden Default-Werte festgelegt):
    point (int dim = 3 , double xc = 0. ,
           double yc = 0. , double zc = 0.) ;
    ~point () ;
} ;

// Die Methoden, die bereits im Programm class03.cpp definiert wurden,
// muessen der geaenderten Datenstruktur angepasst werden:
void point::set_x (double xc) { *(xyz) = xc ; }
void point::set_y (double yc) { *(xyz+1) = yc ; }
void point::set_z (double zc) { *(xyz+2) = zc ; }
double point::get_x () { return *(xyz) ; }
double point::get_y () { return *(xyz+1) ; }
double point::get_z () { return *(xyz+2) ; }

// Definition des Konstruktors:
point::point (int dim , double xc , double yc , double zc)
{
    cout << "Konstruktor arbeitet\n" ; // ... nur zur Information
    if (dim != 2) dim = 3 ;           // ... damit garantiert
                                     // nur die Werte 2 oder 3 moeglich sind
    xyz = new double [dim] ;         // ... Speicherplatz fuer
                                     // die Koordinaten (2 oder 3 Werte)
    *(xyz) = xc ;                    // ... Initialisierung der Koordinaten
    *(xyz + 1) = yc ;                // ...
    if (dim > 2) *(xyz + 2) = zc ;
}

// Definition des Destruktors:
point::~~point ()
{
    cout << "Destruktor arbeitet\n" ; // ... nur zur Information
    delete xyz ;                     // ... Freigabe des Speicherplatzes
}

```

```

main ()
{
    point center (3 , 1. , 2. , 3.) , origin ; // ... erzeugt eine
        // Instanz center (3D-Punkt, der mit den in der Klammer
        // angegebenen Werten initialisiert wird) und eine Instanz
        // origin, fuer die die Default-Initialisierung (3D-Null-
        // punkt) verwendet wird.

    cout << "Mittelpunkt:  x = " << center.get_x ()
        << "\n          y = " << center.get_y ()
        << "\n          z = " << center.get_z () ;

    cout << "\nNullpunkt:  x = " << origin.get_x ()
        << "\n          y = " << origin.get_y ()
        << "\n          z = " << origin.get_z () << "\n" ;

    // ... zeigt, dass der Zugriff auf die Daten der Klasse auch nach der
    // erneuten Aenderung der Datenstruktur innerhalb der Klasse
    // ungeaendert gegenueber den Vorgaenger-Programmen erfolgt.

    testproc () ; // ... wird doppelt aufgerufen, um das jeweilige
    testproc () ; // Erzeugen und Zerstoeren der Klassen-
                  // Instanzen zu zeigen.

    return 0 ; // ... und hier wird der Destruktor fuer die
              // Instanzen center und origin aktiviert.
}

void testproc ()
{
    point p1 (2 , 1. , 1.) , p2 (2) ; // ... erzeugt eine Instanz
        // p1 (2D-Punkt, der mit den in der Klammer angegebenen
        // Werten initialisiert wird) und eine Instanz p2, die
        // ein 2D-Nullpunkt ist

    cout << "Punkt p1:    x = " << p1.get_x ()
        << "\n          y = " << p1.get_y () ;
    cout << "\nPunkt p2:    x = " << p2.get_x ()
        << "\n          y = " << p2.get_y () << "\n" ;

    // ... und hier endet jeweils das Leben der Instanzen p1 und p2
    // (Aufruf des Destruktors fuer beide Instanzen).
}

/* Man beachte, dass in den Methoden der Klasse point keine Sicherung
vorgesehen ist, die einen Zugriff auf die z-Koordinate einer als
2D-Punkt erzeugten Instanz abfaengt. Um dies zu realisieren,
muesste eine weitere Komponente in den Klassen-Daten vorgesehen werden,
die die Information speichert, ob ein 2D- oder 3D-Punkt vorliegt. */

```

- ◆ Die Funktion **testproc** wurde in das Programm **class04.cpp** aufgenommen, um zu zeigen, daß das Leben der beim Aufruf der Funktion erzeugten Instanzen der Klasse **point** mit dem Ende der Funktion endet, wobei für jede Instanz der Destruktor aufgerufen wird. Aus dem gleichen Grund wird die Funktion zweimal aufgerufen, um das "Ableben und die Wiedergeburt" (mit neuerlicher Initialisierung) zu zeigen.
- ◆ Damit das Arbeiten von Konstruktor und Destruktor beim Programmlauf bemerkt wird, sind (nur zur Information) entsprechende Ausgabeanweisungen in beide Funktionen eingebaut worden.

Nachfolgend wird die Ausgabe des Programms **class04.cpp** aufgelistet. Der rechts stehende Kommentar der Ausgabezeilen dient der Erläuterung (und wird nicht vom Programm erzeugt):

```

Konstruktor arbeitet          ... Instanz center
Konstruktor arbeitet          ... Instanz origin
Mittelpunkt:  x = 1
                y = 2
                z = 3
Nullpunkt:    x = 0
                y = 0
                z = 0          ... sind die bei der Initialisie-
                                rung erzeugten Werte.
Konstruktor arbeitet          ... Instanz p1
Konstruktor arbeitet          ... Instanz p2
Punkt p1:     x = 1
                y = 1
Punkt p2:     x = 0
                y = 0          ... Ausgabe in testproc
Destruktor arbeitet           ... p1 "stirbt"
Destruktor arbeitet           ... p2 "stirbt"
Konstruktor arbeitet          ... p1 entsteht neu
Konstruktor arbeitet          ... p2 entsteht neu
Punkt p1:     x = 1
                y = 1
Punkt p2:     x = 0
                y = 0
Destruktor arbeitet           ... p1
Destruktor arbeitet           ... p2
Destruktor arbeitet           ... center
Destruktor arbeitet           ... origin

```

Ausgabe des Programms class04.cpp

12.2.3 Objektorientiertes Programmieren fordert objektorientiertes Denken

Obwohl die wesentlichen Hilfsmittel für das objektorientierte Programmieren erst in den nachfolgenden Abschnitten behandelt werden, soll schon hier auf die Konsequenzen (und die Möglichkeiten) aufmerksam gemacht werden, die sich mit der Verwendung von Klassen im Sinne von Objekten ergeben.

Der C⁺⁺-Programmierer darf sich "Objekte" als "Variablen in einem wesentlich erweiterten Sinne" vorstellen. Das Objekt hat einen **Typ**, der als Bestandteil der Programmiersprache vordeklariert ist (z. B.: **int** oder **double**) oder durch eine Deklaration (z. B. mit dem Schlüsselwort **class**) erzeugt wurde.

Beim Erzeugen eines Objektes (Definition) wird Speicherplatz bereitgestellt, **und es werden gegebenenfalls (möglicherweise sehr aufwendige) Algorithmen abgearbeitet.** Letzteres kann sich auf das Initialisieren von Variablen beschränken (mit der Anweisung **int i = 0 ;** wird Speicherplatz für eine Integer-Variable reserviert, und die Variable wird mit einer 0 initialisiert), im Konstruktor einer Klasse kann unter Umständen jedoch der wesentliche Algorithmus eines Programms erledigt werden.

Während man in der klassischen Programmierung beim Erzeugen eines Objektes nur an dessen Existenz im Programm denken mußte (z. B. eine **int**-Variable, die einen bestimmten Wert hat), darf man die Vorstellung nun wesentlich erweitern: Wenn in einem Windows-Programm das Objekt "Hauptfenster der Applikation" erzeugt wird (durch Definition einer Instanz der entsprechenden Klasse), wird Speicherplatz für zahlreiche Variablen reserviert (Position, Abmessungen, Hintergrundfarbe, ...), diese werden initialisiert, **und das Fenster wird (von entsprechenden Algorithmen des Konstruktors) auch tatsächlich auf dem Bildschirm erzeugt.**

Das nachfolgend angegebene Beispielprogramm **global1.cpp** soll einen ersten Eindruck davon vermitteln, daß objektorientiertes Programmieren den Programmierer zu einem anderen Denken zwingt. Das Hauptprogramm (Funktion **main**) enthält (außer der Return-Anweisung) keine Anweisung, nicht einmal die Definition einer Variablen. Ein C-Programm (vgl. **minimain.c** im Abschnitt 3.3) würde auch keine für den Benutzer sichtbare Aktion ausführen.

Das C++-Programm **global1.cpp** definiert allerdings ein "globales Objekt" vom Typ **vector3d** (die Klasse **vector3d** wird in der Datei **vector.h** deklariert, die von **global1.cpp** inkludiert wird). Und diese Definition eines Objektes, die natürlich auch lokal in der Funktion **main** angesiedelt sein könnte, "bringt Leben in das Programm", denn das Objekt hat einen recht aktiven Konstruktor (die lokale oder globale Definition beliebiger Variablen im C-Programm **minimain.c** hätte natürlich keine Aktion zur Folge).

```
// Definition einer globalen Instanz einer Klasse (Programm global1.cpp)
// =====

#include "vector.h" // ... enthaelt Deklaration der Klasse vector3d
vector3d vec ; // ... ist eine globale Instanz der Klasse

main () { return 0 ; }

// Include-Datei vector.h mit Definition der Klasse vector3d
// =====

#include <iostream.h>
#include <math.h>

class vector3d
{
private:
    double xyz [3] ;
public:
    vector3d () ; // Konstruktor
    ~vector3d () ; // Destruktor
} ;

vector3d::vector3d ()
{
    cout << "Bitte x-Koordinate eingeben: x = " ;
    cin >> xyz [0] ;
    cout << "Bitte y-Koordinate eingeben: y = " ;
    cin >> xyz [1] ;
    cout << "Bitte z-Koordinate eingeben: z = " ;
    cin >> xyz [2] ;
    cout << "Betrag (Laenge) des Vektors: l = "
        << sqrt (xyz[0] * xyz[0] + xyz[1] * xyz[1] + xyz[2] * xyz [2]) ;
}

vector3d::~~vector3d () {}
```

Eine global definierte Instanz eines Objektes wird **vor der Abarbeitung von main** erzeugt. Der Programmierer muß sich also von der Vorstellung verabschieden, daß "vor dem Beginn der Abarbeitung von **main** nichts Erkennbares passiert", im Gegenteil: Die komplette (wenn auch sehr bescheidene) Funktionalität von **global1.c** wird vor dem Eintritt in **main** abgearbeitet.

Und man sollte diese Art der Programmierung auch nicht für eine exotische Spielerei halten, denn bei der MS-Windows-Programmierung mit MFC, wofür im Kapitel 11 ein erstes kleines Beispiel gezeigt wurde, geschehen ganz entscheidende Dinge im Konstruktor eines global erzeugten Objektes (es wird "eigentlich nur ein Objekt 'Applikation' definiert").

Aber auch auf ein "leeres" Hauptprogramm kann nicht verzichtet werden. Abgesehen davon, daß bei nicht existierender Funktion **main** der Linker sich mit einem Fehler meldet, die "Abarbeitung der leeren Funktion **main**" bestimmt den Zeitpunkt, **vor** dem die globalen Objekte erzeugt und **nach** dem sie (Aufruf der Destruktoren!) zerstört werden. Probieren Sie es aus, indem Sie den Destruktor der Klasse **vector3d** (in **vector.h**) folgendermaßen ergänzen:

```
vector3d::~vector3d ()
{
    cout << "\nEnde der Existenz des Objektes vom Typ vector3d\n" ;
}
```

Um zu zeigen, wann Konstruktor und Destruktor für das global erzeugte Objekt aufgerufen werden, wird eine Ausgabeanweisung in der Funktion **main** ergänzt, das nachfolgend gelistete Programm **global2.cpp** unterscheidet sich von **global1.cpp** nur durch diese eine zusätzliche Anweisung:

```
// Definition einer globalen Instanz einer Klasse (Programm global2.cpp)
// =====
#include "vector.h" // ... enthaelt Deklaration der Klasse vector3d
vector3d vec ; // ... ist eine globale Instanz der Klasse

void main ()
{
    cout << "\nStart der Funktion main" ;

    return 0 ;
}
```

Mit dieser geänderten Programmversion kann sich z. B. folgender Dialog ergeben:

```
Bitte x-Koordinate eingeben: x = 3
Bitte y-Koordinate eingeben: y = 4
Bitte z-Koordinate eingeben: z = 5
Betrag (Laenge) des Vektors: l = 7.07107
Start der Funktion main
Ende der Existenz des Objektes vom Typ vector3d
```

12.3 Vererbung

Das Definieren von Klassen einschließlich aller zugehörigen Methoden kann recht aufwendig sein. Deshalb wird der Programmierer typischerweise immer wieder verwendbare Klassen in Include-Dateien zusammenfassen, die er in seine Programme einbindet, was darüber hinaus den Vorteil hat, daß in verschiedenen Files garantiert mit den gleichen Definitionen gearbeitet wird. Ein typisches Problem dabei ist, daß eine bereits existierende Klassendeklaration zwar weitgehend, aber eben nicht exakt zum gerade anstehenden Problem paßt.

Mit der Möglichkeit, bei der Deklaration einer neuen Klasse auf bereits existierende Deklarationen zurückzugreifen, wobei in die neue Klasse ("abgeleitete Klasse") sämtliche Daten und Methoden übernommen werden, bietet C++ die wohl wichtigste Erweiterung gegenüber der Programmiersprache C. Man spricht von **Vererbung** ("inheritance"), wenn die existierende Klasse alle ihre Eigenschaften an die **abgeleitete Klasse** weitergibt. Die neue Klasse ist eine Spezialisierung der alten Klasse im Sinne einer Erweiterung.

Die nachfolgend aufgelistete Datei **basecl1.h** (Name steht für "Basisklassen") enthält eine Basisklasse **point2d** und eine von dieser abgeleitete Klasse **point2d_init**:

```
// Include-Datei basecl1.h mit Definition der Basisklassen
// =====

// Die Klasse point2d enthaelt die beiden Koordinaten x und y und
// (einschliesslich Konstruktor und Destruktor) sieben Methoden:

class point2d
{
    private:
        double x ;
        double y ;
    public:
        point2d ( ) ;           // Konstruktor
        ~point2d ( ) ;         // Destruktor
        void    set_x (double) ; // ... setzt Koordinate x
        void    set_y (double) ; // ... setzt Koordinate y

        double get_x ( ) ;     // ... liefert Koordinate x
        double get_y ( ) ;     // ... liefert Koordinate y
        void    input ( ) ;    // ... setzt x und y durch Einlesen
                                // ueber cin
};

point2d::point2d ( )
{
    x = 0. ;                   // Konstruktor wird ohne Argumente
    y = 0. ;                   // aufgerufen und initialisiert
                                // beide Koordinaten
}

point2d::~~point2d ( ) {}     // Destruktor ohne besondere Aufgaben

void    point2d::set_x (double xc) { x = xc ; }
void    point2d::set_y (double yc) { y = yc ; }
double point2d::get_x ( )         { return x ; }
double point2d::get_y ( )         { return y ; }

void    point2d::input ( )
{
    cout << "Bitte x-Koordinate eingeben:  x = " ;
    cin  >> x ;
    cout << "Bitte y-Koordinate eingeben:  y = " ;
    cin  >> y ;
}
```

```
// Die Klasse point2d_init wird aus der Klasse point2d abgeleitet (Syntax
// dafuer ist 'class point2d_init : public point2d') und "erbt" damit
// saemtliche Daten und Methoden der Basisklasse. Als einzige eigene
// Besonderheit enthaelt point2d_init einen Konstruktor, der einen String
// uebernimmt, diesen als Ueberschrift ausgibt, um danach die point2d-Methode
// input() aufzurufen, die die beiden Koordinaten von der Standard-Eingabe
// abfordert (man beachte die Syntax des Aufrufs der point2d-Methode):

class point2d_init : public point2d
{
    public:
        point2d_init (char *) ;
        ~point2d_init () ;
} ;

point2d_init::point2d_init (char *output)
{
    cout << output << "\n" ;
    point2d::input () ;
}

point2d_init::~~point2d_init () {}
```

- ◆ Auf die **private** deklarierten Daten der Basisklasse können die Methoden der abgeleiteten Klasse nicht direkt zugreifen (so großzügig sind die "Erbgesetze" nicht), sie können nur über die Methoden der Basisklasse erreicht werden. Die "vererbende Klasse" kann jedoch Daten auch **protected** deklarieren, auf die aus den Methoden der abgeleiteten Klassen direkt zugegriffen werden kann, während sie vor direktem Zugriff aus anderen Funktionen wie **private**-Daten geschützt sind. Im allgemeinen ist die (etwas aufwendiger zu programmierende) Variante "**private**-Daten und Zugriffs-Methoden" aus Sicherheitsgründen vorzuziehen.
- ◆ Mit dem Doppelpunkt nach dem Namen der neuen Klasse wird bei der Klassen-Definition der Bezug auf eine Basisklasse eingeleitet. Der Zusatz **: public point2d**, mit dem in dem betrachteten Beispiel die Ableitung der Klasse **point2d_init** aus der Klasse **point2d** veranlaßt wird, kann gegebenenfalls auch mit den Schlüsselworten **protected** bzw. **private** gebildet werden, was eine Veränderung der Zugriffsrechte für die aus der Basisklasse ererbten Objekte zur Folge hätte (würde sich z. B. bei einer Ableitung einer weiteren Klasse aus der abgeleiteten Klasse auswirken). Diese Varianten haben keine nennenswerte praktische Bedeutung.

Beim Erzeugen einer Instanz einer abgeleiteten Klasse wird (automatisch) **erst der Konstruktor der Basisklasse aufgerufen, anschließend der Konstruktor der abgeleiteten Klasse**. Für das betrachtete Beispiel im File **basecl1.h** bedeutet das, daß für eine Instanz der abgeleiteten Klasse **point2d_init** zuerst (vom Konstruktor **point2d**) die beiden Koordinaten "0-initialisiert" werden und danach (vom Konstruktor **point2d_init**) über die Standard-Eingabe ihre aktuellen Werte erhalten.

Was zu beachten ist, wenn dem Konstruktor einer Basisklasse Argumente übergeben werden müssen, wird im nachfolgenden Programm **schwerp1.cpp** kommentiert.

Im folgenden Programm wird aus der Klasse **point2d_init**, die selbst aus **point2d** abgeleitet ist, eine neue Klasse **area** abgeleitet (die "Erbfolge" kann beliebig fortgesetzt werden).

Das Problem, das mit **schwerp1.cpp** behandelt wird, ist einfach zu beschreiben: Für eine ebene Fläche, die man sich aus Teilflächen zusammengesetzt denken kann (Ausschnitte können als "negative Flächen" erfaßt werden), kann man die Gesamtfläche A und die Koordinaten des Schwerpunkts x_S und y_S aus den Teilflächen A_i und den Schwerpunktkoordinaten der Teilflächen x_i und y_i (Koordinaten beziehen sich auf ein beliebiges kartesisches Koordinatensystem) nach folgenden Formeln berechnen:

$$A = \sum_{i=1}^n A_i \quad ; \quad x_S = \frac{1}{A} \sum_{i=1}^n A_i x_i \quad ; \quad y_S = \frac{1}{A} \sum_{i=1}^n A_i y_i \quad .$$

(vgl. z. B. "Dankert/Dankert: Technische Mechanik, computerunterstützt", Seite 28).

Das Programm **schwerp1.cpp** fordert die Anzahl der Teilflächen n an und legt (dynamisch mit **new**) für jede Teilfläche eine Instanz der Klasse **area** (aus der Klasse **point2d_init** abgeleitete Klasse) an. Dabei werden jeweils die Konstruktoren von Basisklasse und abgeleiteter Klasse aktiv, über die die Parameter der Teilfläche eingelesen werden. Die Pointer auf die Instanzen der Klasse **area** werden in einem Array gespeichert, so daß die Daten aller Instanzen (über die Methoden!) abgefordert werden können. Die Methoden sind mit der aus dem Abschnitt 7.2 bekannten Syntax für den Zugriff auf die Komponenten einer durch einen Pointer identifizierten Struktur anzusprechen (mit dem Operator **->**):

```
// Schwerpunkt einer zusammengesetzten Flaechе (Programm schwerp1.cpp)
// =====

#include <iostream.h>
#include <math.h>
#include "basecl1.h"          // ... enthaelt Basisklassen-Definitionen

class area : public point2d_init
{
private:
    double a ;                // Flaechе

public:
    area (char *) ;
    ~area () ;
    double get_a () ;        // Zugriff auf Flaechе a
} ;

// Weil der Konstruktor der Klasse point2d_init, aus der die Klasse area
// abgeleitet wird, mit einem Argument (String) aufgerufen wird, muss
// der Aufruf des Konstruktors der Basisklasse angegeben werden
// (das Argument output, mit dem der Konstruktor area gerufen
// wird, wird an den Konstruktor point2d_init "durchgereicht"):

area::area (char *output) : point2d_init (output)
{
    cout << "Bitte Flaechе eingeben:          A = " ;
    cin  >> a ;
}

area::~~area () { }

double area::get_a () { return a ; }

main ()
{
    int      i , n = 0 ;
    double   ai , a = 0. , sx = 0. , sy = 0. ;
```

```

area *area_array [10] ; // Feld fuer maximal 10 Pointer auf Instanzen
                        // der Klasse area

cout << "Schwerpunkt einer zusammengesetzten Flaechen\n" ;
cout << "=====\n\n" ;

do {
    cout << "Anzahl der Teilflaechen (max. 10):      n = " ;
    cin  >> n ;
    } while (n <= 0 || n > 10) ;

// Bei jedem Erzeugen einer Instanz der Klasse area werden der
// Konstruktor der Basisklasse (point2d_init) und der Konstruktor
// der Klasse area gerufen, wobei die Schwerpunktkoordinaten und die
// Flaechen jeweils einer Teilflaechen angefordert werden:

for (i = 0 ; i < n ; i++)
    {
    if ((area_array [i] = new area ("\nTeilflaechen")) == NULL)
        {
        cout << "Fehler beim Allokieren von Speicherplatz\n" ;
        return 1 ;
        }
    }

// Sowohl die in der Klasse area definierte Methode als auch die in
// der "Erbfolge" von point2d ueber point2d_init an area gelangten
// Methoden get_x und get_y werden mit dem vorangestellten Namen
// der Instanz angesprochen. Weil im Feld area_array die Pointer auf
// die Instanzen gespeichert sind, wird (analog zur Syntax bei der
// Verwendung von Struktur-Pointern) die Methode z. B. mit
// area_array[i]->get_x identifiziert:

for (i = 0 ; i < n ; i++)
    {
    ai = area_array[i]->get_a() ;
    a += ai ;
    sx += ai * area_array[i]->get_x() ;
    sy += ai * area_array[i]->get_y() ;
    }

cout << "\nFlaechen          A = " << a ;
if (fabs (a) > 1.e-20)
    {
    cout << "\nSchwerpunkt-Koordinaten:   xS = " << sx / a ;
    cout << "\n                          yS = " << sy / a << "\n" ;
    }

return 0 ;
}

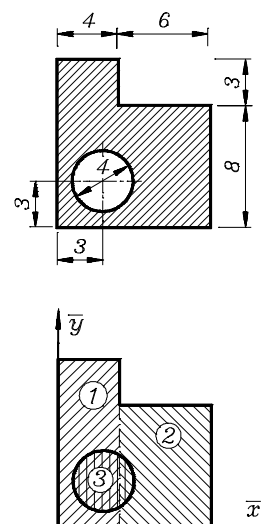
```

Wenn das Programm mit der Berechnung für die nebenstehend skizzierte Fläche (entnommen aus "Dankert/Dankert: Technische Mechanik, computerunterstützt", Seite 214) getestet wird, erhält man bei Einteilung in drei Teilflächen unter Verwendung eines Koordinatensystems entsprechend der unteren Skizze bei Eingabe der Werte

n = 3	x1 = 2	y1 = 5.5	A1 = 44
	x2 = 7	y2 = 4	A2 = 48
	x3 = 3	y3 = 3	A3 = -12.566

die Ergebnisse:

Flaechen:	A = 79.434
Schwerpunkt-Koordinaten:	xS = 4.863
	ys = 4.989



- ◆ Natürlich hätte man die Funktionalität des Programms **schwerp1.cpp** auch wesentlich einfacher realisieren können (ohne Arrays, Pointer und Klassen, indem in der Eingabeschleife gleich gerechnet wird). Es soll der Demonstration der "Vererbung" dienen und wurde deshalb in der angegebenen Form geschrieben. Wenn man das Programm komfortabler gestalten will (z. B. durch die Möglichkeit der nachträglichen Korrektur eingegebener Werte), müssen die Eingabewerte jedoch gespeichert werden, und dafür ist die realisierte Variante sicher eine besonders geeignete Form.

Namenskonflikte, Funktionen redefinieren:

In einer abgeleiteten Klasse können Namenskonflikte mit (**private** deklarierten) Variablen aus den Basisklassen nicht auftreten, weil diese über die Methoden der Basisklassen angesprochen werden. Anders ist dies bei (**public** deklarieren) Funktionen: Eine Funktion in der abgeleiteten Klasse (mit gleichem Namen, gleicher Anzahl von Argumenten und gleichen Argument-Typen) "verdeckt" die entsprechende Funktion einer Basisklasse.

So können Methoden aus den Basisklassen mit den exakt zur abgeleiteten Klasse passenden Eigenschaften **redefiniert** werden, ohne daß ein Konflikt entsteht. Sollte trotzdem Bedarf nach dem Zugriff auf die "verdeckte" Funktion bestehen, so kann sie unter zusätzlicher Angabe der Klasse, in der sie deklariert wurde, mit der Syntax

Name_der_Instanz.Name_der_Basisklasse::Name_der_Funktion

angesprochen werden.

Prinzipiell gilt: Der Programmierer muß von den Basisklassen nur die Eigenschaften kennen, die er selbst ausnutzen möchte, und kann (ohne Furcht vor Konflikten) eigene Daten und Methoden ergänzen. Die Möglichkeit, auf spezielle Methoden der Basisklassen zuzugreifen, wird ihm damit nicht genommen.

Das folgende Programm **schwerp2.cpp** demonstriert das Anlegen einer **verketteten Liste** aus Instanzen einer Klasse (vgl. Abschnitt 7.3, wo verkettete Listen mit Strukturen erzeugt wurden). Die Funktionalität dieses Programms ist einfach:

Für eine Fläche, die von einem geschlossenen Polygon begrenzt wird (die Skizze zeigt ein Polygon, das durch 6 Punkte definiert wird), können die Schwerpunkt-Koordinaten nach

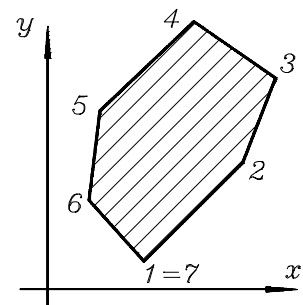
mit

$$x_S = \frac{S_y}{A} \quad ; \quad y_S = \frac{S_x}{A}$$

$$A = \frac{1}{2} \sum_{i=1}^n (y_{i+1} x_i - y_i x_{i+1}) \quad ,$$

$$S_x = \frac{1}{6} \sum_{i=1}^n (y_i + y_{i+1}) (y_{i+1} x_i - y_i x_{i+1}) \quad ,$$

$$S_y = \frac{1}{6} \sum_{i=1}^n (x_i + x_{i+1}) (y_{i+1} x_i - y_i x_{i+1})$$



berechnet werden (vgl. "Dankert/Dankert: Technische Mechanik, computerunterstützt", Seite 219). Dabei müssen die Punkte so numeriert werden, daß die Fläche beim Durchlaufen des

Polygons immer links liegt. Für den Punkt $n+1$, dessen Koordinaten bei einem Polygon mit n Punkten in den Formeln benötigt wird, müssen die Koordinaten des Startpunktes (Punkt 1) noch einmal verwendet werden. Die Formeln gelten auch für mehrfach zusammenhängende Bereiche (vgl. Beispiel am Ende dieses Abschnittes), wenn nur gewährleistet ist, daß die Fläche immer links vom Polygon liegt, Startpunkt und Endpunkt identisch sind und Linien, die nicht zum Rand gehören, doppelt (in entgegengesetzten Richtungen) durchlaufen werden.

Im Programm **schwerp2.cpp** werden Punktkoordinaten angefordert, bis ein Punkt eingegeben wird, dessen Koordinaten mit denen des Startpunktes identisch sind (Prüfung auf Identität in der Funktion **identical_points**). In $n+1$ Instanzen der Klasse **polygon** werden die Koordinaten für $n+1$ Punkte abgelegt (der letzte Punkt hat die gleichen Koordinaten wie der Startpunkt).

Die Klasse **polygon** wird aus der Klasse **point2d** (definiert in **basecl1.h**) abgeleitet, erbt damit die beiden Punktkoordinaten und fügt als eigenen Beitrag einen Pointer auf den eigenen Typ hinzu (zur Realisierung der Verkettung). Neben Konstruktor und Destruktor werden zwei Methoden für das Arbeiten mit diesem Pointer definiert:

```
// Schwerpunkt einer Polygonflaeche (Programm schwerp2.cpp)
// =====

#include <iostream.h>
#include <math.h>
#include "basecl1.h"          // ... enthaelt Definition der Basisklasse

class polygon : public point2d
{
private:
    polygon *next ;          // ... fuer die Verkettung
public:
    polygon () ;
    ~polygon () ;
    void anchor_pp (polygon *) ;
    polygon *get_next () ;
} ;

polygon::polygon ()
{
    point2d::input () ;      // Eingabe der Koordinaten ueber cin
    next = NULL ;
}

polygon::~~polygon () { }

void polygon::anchor_pp (polygon *anchor)
{
    next = anchor ;          // ... haengt neues Listenelement an
}

polygon *polygon::get_next ()
{
    return next ;            // ... liefert Pointer auf Nachfolgeelement
}

int identical_points (polygon *p1 , polygon *p2)
{
    const double eps = 1.e-10 ; // ... willkuerlicher Wert

    if (fabs (p1->get_x () - p2->get_x ()) <= eps &&
        fabs (p1->get_y () - p2->get_y ()) <= eps)
        return 1 ;           // Punkte sind identisch
    else
        return 0 ;           // Punkte sind nicht identisch
}
```

```

main ()
{
    polygon *first_point = NULL , *last_point , *new_point ;
    int     ende = 0 , i = 0 ;
    double  ai , a = 0. , sx = 0. , sy = 0. ;

    cout << "Schwerpunkt einer Polygonflaeche\n" ;
    cout << "=====\n" ;

    while (!ende)
    {
        i++ ;
        cout << "\nPunkt " << i << "\n" ;
        if ((new_point = new polygon) == NULL) // ... neue Instanz
        {
            cout << "Fehler beim Allokieren von Speicherplatz\n" ;
            return 1 ;
        }

        if (first_point == NULL)
        {
            first_point = new_point ; // Startelement der Liste
            last_point  = new_point ;
        }
        else
        {
            last_point->anchor_pp (new_point) ;
            last_point = new_point ;
            ende = identical_points (last_point , first_point) ;
        }
    }

    last_point = first_point ;

    while (last_point->get_next () != NULL)
    {
        new_point = last_point->get_next () ;
        ai = last_point->get_x () * new_point->get_y () -
            last_point->get_y () * new_point->get_x () ;
        a += ai ;
        sx += ai * (last_point->get_y () + new_point->get_y ()) ;
        sy += ai * (last_point->get_x () + new_point->get_x ()) ;
        last_point = new_point ;
    }

    a = a / 2 ;
    sx = sx / 6 ;
    sy = sy / 6 ;

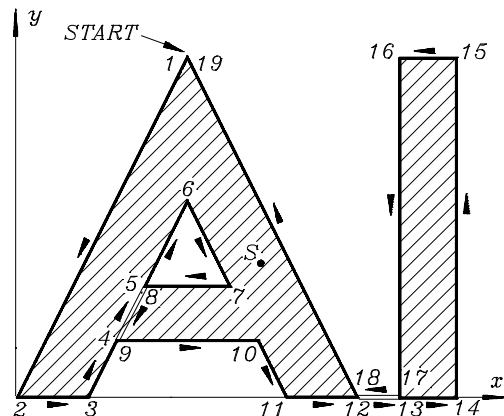
    cout << "\nFlaeche:                A = " << a ;
    if (fabs (a) > 1.e-20)
    {
        cout << "\nSchwerpunkt-Koordinaten:   xS = " << sy / a ;
        cout << "\n                                   yS = " << sx / a << "\n" ;
    }

    return 0 ;
}

```

- ◆ Man beachte, daß die verkettete Liste mit den Klassen-Objekten nach dem gleichen Prinzip erzeugt wird, wie es im Kapitel 7 unter Verwendung von Strukturen demonstriert wurde. Auch der Speicherbedarf ist nicht größer als bei Strukturen, weil die Methoden nur einmal existieren. Natürlich sind auch andere Datenstrukturen (wie binäre Bäume, vgl. Kapitel 8) mit Klassen-Objekten realisierbar.

Die nebenstehende Skizze zeigt eine Fläche, die aus zwei nicht miteinander verbundenen Teilflächen besteht, von denen die linke Fläche einen dreieckigen Ausschnitt hat (entnommen aus "Dankert/Dankert: Technische Mechanik, computerunterstützt", Seite 36). Der Gesamt-Schwerpunkt kann mit dem Programm **schwerp2.cpp** berechnet werden, wenn die Punkte in der angegebenen Reihenfolge eingegeben werden (Koordinaten siehe Tabelle unten).



Die Bedingung, daß die Fläche immer links vom Rand liegt, ist erfüllt (im Gegensatz zur Außenkontur wird die Ausschnittkontur rechtsherum durchlaufen). Die beiden Linien, die die Teilflächen bzw. die Außenkontur und die Innenkontur verbinden, werden jeweils doppelt (im entgegengesetzten Sinn) durchlaufen, ihre Anteile heben sich in der programmierten Formel auf. Das Programm liefert

$$A = 318 \quad ; \quad x_s = 17,1321 \quad ; \quad y_s = 10,0168 \quad .$$

Koordinaten des Polygons in der Reihenfolge, in der sie eingegeben werden:

Pkt.	1	2	3	4	5	6	7	8	9
x;y	12 ; 24	0 ; 0	5 ; 0	7 ; 4	9 ; 8	12 ; 14	15 ; 8	9 ; 8	7 ; 4
	10	11	12	13	14	15	16	17	18
	17 ; 4	19 ; 0	24 ; 0	27 ; 0	31 ; 0	31 ; 24	27 ; 24	27 ; 0	24 ; 0
									19
									12 ; 24

12.4 Virtuelle Funktionen, abstrakte Klassen (Polymorphismus)

Wie Strukturen können auch Instanzen einer Klasse einander komplett zugewiesen werden. Es ist sogar möglich, eine Instanz einer abgeleiteten Klasse einer Instanz der zugehörigen Basisklasse zuzuweisen, obwohl im Regelfall dabei nicht alle Daten übernommen werden können, weil eine abgeleitete Klasse zusätzliche Daten enthalten kann.

Der umgekehrte Fall ist aus naheliegenden Gründen nicht erlaubt: Einer Instanz einer abgeleiteten Klasse kann nicht eine Instanz der Basisklasse zugewiesen werden, einzelne Daten würden keine Werte erhalten können (diese Aussage gilt nicht für Pointer auf die Klassen, weil dabei zunächst keine Daten "bewegt werden", aber vor einer solchen Zuweisung wird ausdrücklich gewarnt).

Natürlich werden nach einer Zuweisung einer Instanz einer abgeleiteten Klasse an eine Instanz der Basisklasse für diese nur noch die Methoden der Basisklasse verwendet, denn die Methoden werden über den Namen der Instanz identifiziert.

Beispiel:

Mit der nachfolgend angegebenen Basisklasse und einer daraus abgeleiteten Klasse werden erlaubte und verbotene Operationen demonstriert:

```

class point2d
{
    private:
        double x ;
        double y ;

    public:
        // ...
        double get_x () ;
        // ...
} ; // ... ist die Basisklasse

class polygon : public point2d
{
    private:
        polygon *next ;

    public:
        // ...
} ; // ... ist die abgeleitete Klasse

main ()
{
    point2d pt1 , pt2 ;
    polygon pol1 , pol2 ;
    double xc ;

    // ...

    pt1 = pol1 ; // ... ist erlaubt.

    xc = pt1.get_x () ; // ... benutzt in jedem Fall
                        // die Methode get_x der Basisklasse point2d,
                        // auch wenn in der abgeleiteten Klasse eine
                        // Methode gleichen Namens existieren wuerde,
                        // weil bereits der Compiler mit dem Typ der
                        // Instanz pt1 diese Zuordnung trifft.

    pol2 = pt2 ; // ... ist NICHT ERLAUBT
                // (wird bereits vom Compiler beanstandet),
                // weil der Pointer next in der Klasse
                // Polygon keinen Wert aus einer Instanz der
                // Klasse point2d erhalten kann.

    // ...
}

```

Häufig ist es bei der Arbeit mit abgeleiteten Klassen ein erheblicher Vorteil, wenn man die Basisklasse gewissermaßen weiter als "gemeinsamen Nenner" der Instanzen der abgeleiteten Klassen benutzen kann, indem man z. B. in einem Pointer-Array oder einer verketteten Liste alle Instanzen verwaltet (Beispiel: Als Instanzen der Basisklasse 'Flaeche' werden die Instanzen der abgeleiteten Klassen 'Rechteck', 'Kreis' usw. mit verwaltet). Trotzdem sollen die jeweils zuständigen Methoden der abgeleiteten Klassen benutzt werden (bei der Eingabe der Rechteck-Definition sollen Breite und Höhe, bei der Eingabe der Definition eines Kreises nur der Durchmesser abgefragt werden). Dieses Ziel erreicht man durch

Virtuelle Funktionen:

Durch das Schlüsselwort **virtual**, das man bei der Deklaration einer Methode **in der Basisklasse** voranstellt, wird der Compiler veranlaßt, folgendes zu organisieren:

- ◆ Für jede Zuweisung einer Instanz einer abgeleiteten Klasse an eine Instanz der Basisklasse wird registriert, von welcher abgeleiteten Klasse ein Objekt "angeliefert" wurde (da dies erst zur Laufzeit des Programms bekannt ist, kann der Compiler hierfür nur die "organisatorische Vorarbeit" leisten).
- ◆ Wenn die als **virtual** deklarierte Methode in der Basisklasse tatsächlich auch definiert wird (das ist durchaus nicht erforderlich, diese Variante wird auch noch in diesem Abschnitt behandelt), ist sie nur dann die zuständige Methode,
 - wenn das Objekt tatsächlich der Basisklasse (und nicht einer von ihr abgeleiteten Klasse) entstammt, oder
 - für die abgeleitete Klasse keine eigene Methode (gleichen Namens) definiert wurde (das bedeutet, daß zu virtuellen Funktionen in einer Basisklasse in jeder abgeleiteten Klasse Methoden gleichen Namens definiert werden können, die bei Instanzen, die ursprünglich zur abgeleiteten Klasse gehörten, Vorrang haben).

Da erst zur Laufzeit des Programms bekannt ist, welche Instanzen abgeleiteter Klassen den Instanzen der Basisklasse zugewiesen werden, spricht man von "später oder dynamischer Bindung", weil beim Compilieren noch nicht entschieden wird, welche Methode auf ein Objekt der Basisklasse anzuwenden ist.

- ◆ Da die Instanz der abgeleiteten Klasse mit dem Konstruktor dieser Klasse erzeugt wurde, sollte sie unbedingt auch mit dem Destruktor der abgeleiteten Klasse zerstört werden (unter Umständen ist Speicherplatz angefordert worden, der vom Destruktor wieder freigegeben wird). Dies kann trotz Zuweisung zu einer Instanz der Basisklasse dadurch gesichert werden, daß man auch den Destruktor einer Basisklasse, die virtuelle Methoden enthält, als **virtual** deklariert.

Das nachfolgende Programm **schwerp3.cpp** demonstriert das Arbeiten mit virtuellen Methoden: Die Basisklasse **base_area** beschreibt eine Fläche durch ihre Schwerpunkt-Koordinaten und den Flächeninhalt **a** und deklariert zwei Funktionen für den Zugriff auf ihre Daten als **virtual**. In der aus **base_area** abgeleiteten Klasse **circle**, die einen Kreis mit seinen Mittelpunkt-Koordinaten und seinem Durchmesser verwaltet, werden eigene Methoden gleichen Namens definiert.

```
// Schwerpunkt einer zusammengesetzten Flaechе (Programm schwerp3.cpp)
// =====
#include <iostream.h>
#include <math.h>
#include "basec11.h"          // ... enthaelt Definition der Klasse point2d

// Die Klasse base_area wird aus der Klasse point2d abgeleitet, die beiden
// Werte x und y aus point2d sind die Schwerpunkt-Koordinaten der Flaechе,
// die von einer base_area-Instanz beschrieben wird:
```

```

class base_area : public point2d
{
    private:
        double a ;
    public:
        base_area () ;
        virtual ~base_area () ;
        virtual void input_a () ; // ... werden 'virtual' definiert,
        virtual double get_a () ; // weil aus base_area abgeleitete
} ; // Klassen die Flaechе mit ganz
// anderen Parametern beschreiben
base_area::base_area () // koennen, auf die dann mit
{ // eigenen Methoden zugegriffen
    a = 0. ; // werden muss.
}

base_area::~~base_area () { }

void base_area::input_a ()
{
    cout << "*** Schwerpunkt der Flaechе ***\n" ;
    point2d::input () ;
    cout << "Bitte Flaechе eingeben: A = " ;
    cin >> a ;
}

double base_area::get_a () { return a ; }

// Die Klasse circle beschreibt eine Kreisflaechе mit dem Mittelpunkt
// (ist ueber "Erbfolge" point2d --> base_area --> circle von point2d bis
// circle "durchgereicht" worden) und dem Durchmesser d. Die virtuellen
// Funktionen der Klasse base_area werden durch eigene Methoden input_a
// bzw. get_a ueberschrieben:

class circle : public base_area
{
    private:
        double d ;

    public:
        circle () ;
        ~circle () ;
        void input_a () ;
        double get_a () ;
} ;

circle::circle ()
{
    d = 0. ;
}

circle::~~circle () {}

// Die aus base_area abgeleitete Klasse circle bekommt zwei Methoden mit
// den gleichen Namen wie die virtuellen Methoden der Basisklasse:

void circle::input_a ()
{
    cout << "*** Mittelpunkt ***\n" ;
    point2d::input () ;
    cout << "*** Durchmesser ***\nd = " ;
    cin >> d ;
}

const double pi_4 = atan (1.) ;

double circle::get_a () { return (pi_4 * d * d) ; }

```

```

main ()
{
    int      i , n = 0 , id ;
    double   ai , a = 0. , sx = 0. , sy = 0. ;

    base_area *area_array [20] ; // Feld fuer maximal 20 Pointer auf
                                // Instanzen der Klasse base_area

    cout << "Schwerpunkt einer zusammengesetzten Flaechen\n" ;
    cout << "=====\n\n" ;

    do {
        cout << "Anzahl der Teilflaechen (max. 20):      n = " ;
        cin  >> n ;
    } while (n <= 0 || n > 20) ;

    for (i = 0 ; i < n ; i++)
    {
        do {
            cout << "\n1 - Allgemeine Flaechen, 2 - Kreis\n" ;
            cout << "Typ der Teilflaechen " << i + 1 << " : " ;
            cin  >> id ;
        } while (id <= 0 || id > 2) ;

        switch (id)
        {
            // Hier entscheidet sich erst zur Laufzeit, welcher Typ
            // erzeugt und im "Basisklassen-Pointer-Array" registriert
            // wird:
            case 1: area_array [i] = new base_area ; break ;
            case 2: area_array [i] = new circle   ; break ;
        }

        if (area_array [i] == NULL)
        {
            cout << "Fehler beim Allokieren von Speicherplatz\n" ;
            return 1 ;
        }

        // Weil die Methode input_a in der Basisklasse base_area virtuell
        // deklariert wurde, wird abhaengig davon, ob in area_array[i] ein
        // Pointer auf die Basisklasse oder auf die abgeleitete Klasse
        // gespeichert ist, jeweils die zur entsprechenden Klasse gehoernde
        // Methode input_a verwendet (Entscheidung zur Laufzeit des
        // Programms ----> "Spaete Bindung"):
        area_array[i]->input_a () ;
    }

    for (i = 0 ; i < n ; i++)
    {
        // Fuer die virtuelle Funktion get_a gilt sinngemaess die gleiche
        // Aussage wie fuer input_a:
        ai = area_array[i]->get_a() ;
        a += ai ;
        sx += ai * area_array[i]->get_y() ;
        sy += ai * area_array[i]->get_x() ;
    }

    cout << "\nFlaechen          A = " << a ;
    if (fabs (a) > 1.e-20)
    {
        cout << "\nSchwerpunkt-Koordinaten:   xS = " << sy / a ;
        cout << "\n                               yS = " << sx / a << "\n" ;
    }

    return 0 ;
}

```

- ◆ Der Vorteil der verwendeten Strategie mit virtuellen Funktionen zeigt sich erst in **main** beim Aufruf der Methoden **input_a** und **get_a**: Ohne **switch**-Anweisung werden die Methoden der "richtigen Klasse" verwendet. Auch bei der Programm-Erweiterung (zusätzliche abgeleitete Klassen) braucht an diesen Stellen nichts geändert zu werden.

Die Berechnung des Schwerpunkts der skizzierten Fläche (Kreis mit rechteckigem Ausschnitt) führt mit folgendem Eingabe-Dialog zu dem angegebenen Ergebnis:

```
Schwerpunkt einer zusammengesetzten Flaechе:
=====
```

```
Anzahl der Teilflaechen (max. 20):      n = 2
```

```
1 - Allgemeine Flaechе, 2 - Kreis
```

```
Typ der Teilflaechе 1: 2
```

```
*** Mittelpunkt ***
```

```
Bitte x-Koordinate eingeben:    x = 0
```

```
Bitte y-Koordinate eingeben:    y = 0
```

```
*** Durchmesser ***
```

```
d = 80
```

```
1 - Allgemeine Flaechе, 2 - Kreis
```

```
Typ der Teilflaechе 2: 1
```

```
*** Schwerpunkt der Flaechе ***
```

```
Bitte x-Koordinate eingeben:    x = 10
```

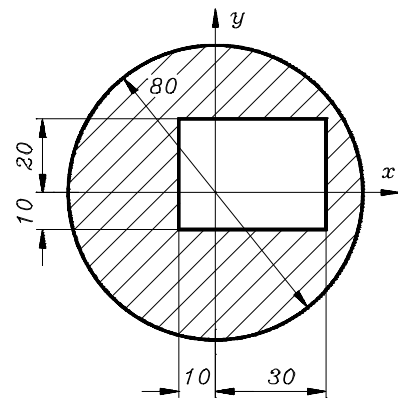
```
Bitte y-Koordinate eingeben:    y = 5
```

```
Bitte Flaechе eingeben:        A = -1200
```

```
Flaechе:                        A = 3826.55
```

```
Schwerpunkt-Koordinaten:      xS = -3.13599
```

```
                                yS = -1.56799
```



Schwerpunkt des Ausschnitts liegt bei
x = 10 und y = 5

Auch bei einer Erweiterung des Programms mit zusätzlichen abgeleiteten Klassen für weitere spezielle Flächen wird die Variable **a** in der Basisklasse für die abgeleiteten Klassen nicht benötigt. Es ist deshalb konsequent, die im Programm **schwerp3.cpp** als Basisklasse behandelte "Allgemeine Fläche" auch als abgeleitete Klasse zu definieren. Dann kann die Variable **a** in der Basisklasse weggelassen werden, **und damit hätten auch die beiden (virtuellen) Funktionen input_a und get_a für die Basisklasse keine sinnvolle Funktionalität mehr, wenn sie nicht als "formale Platzhalter" der Methoden der abgeleiteten Klassen erforderlich wären.**

Genau mit der letztgenannten Eigenschaft werden sie aber weiterhin benötigt, was folgendermaßen realisiert wird:

Mit dem Zusatz **= 0** beim Prototyp einer virtuellen Funktion in der Basisklasse wird signalisiert, daß diese Funktion **für die Basisklasse nicht definiert wird.**

- ◆ Dadurch wird die Basisklasse zu einer **abstrakten Klasse**, für die **keine Instanzen definiert werden können.**
- ◆ Eine aus einer abstrakten Klasse abgeleitete Klasse muß für die virtuellen Funktionen der Basisklasse eigene Funktionen definieren, um nicht selbst zur abstrakten Klasse zu werden.

Das Programm **schwerp4.cpp** demonstriert die Definition einer abstrakten Klasse **abstract_area**. Die aus ihr abgeleiteten Klassen **plain_area** (für die "Allgemeine Fläche", die im Programm **schwerp3.cpp** durch die Basisklasse repräsentiert wurde), **rectangle** (Rechteck) und **circle** (Kreis) werden bei der Eingabe durch die Indikatoren 1, 2 und 3 als Teilflächen bzw. -1, -2 und -3 als Ausschnitte gekennzeichnet:

```
// Schwerpunkt einer zusammengesetzten Flaechе (Programm schwerp4.cpp)
// =====

// Das Programm definiert eine abstrakte Klasse abstract_area, aus der die
// Klassen  plain_area (eingelесen werden Schwerpunkt und Flaecheninhalte),
//          rectangle (eingelесen werden zwei Eckpunkte einer Diagonalen),
//          circle    (eingelесen werden Mittelpunkt und Durchmesser)
// abgeleitet werden. Alle abgeleiteten Klassen muessen eigene Funktionen
// input_a (Einlesen der Parameter der Flaechе) und get_a (Abliefern des
// Flaecheninhalts) enthalten, da diese nur durch Prototypen (mit dem Zusatz
// = 0) in der Basisklasse vertreten sind.

// Die Komponente area_or_hole in der Basisklasse wird 'protected' angelegt,
// so dass alle abgeleiteten Klassen direkt auf sie zugreifen koennen. Das
// ist bequem, aber nicht so sicher wie eine 'private' angelegte Komponente,
// fuer die dann Zugriffs-Methoden zu definieren waeren.

#include <iostream.h>
#include <math.h>
#include "basecl1.h"           // ... enthaelt Definition von point2d

class abstract_area : public point2d
{
    protected:
        int area_or_hole ;           // 1 --> Flaechе, -1 --> Ausschnitt
    public:
        abstract_area () ;
        virtual ~abstract_area () ;
        virtual void input_a (int id) = 0 ;
        virtual double get_a  ()      = 0 ;
} ;

abstract_area::abstract_area ()
{
    area_or_hole = 1 ;
}

abstract_area::~~abstract_area () { }

class plain_area : public abstract_area
{
    private:
        double a ;

    public:
        plain_area () ;
        ~plain_area () ;
        void input_a (int id) ;
        double get_a  () ;
} ;

plain_area::plain_area ()
{
    a = 0. ;
}

plain_area::~~plain_area () {}
```

```

void plain_area::input_a (int id)
{
    cout << "*** Schwerpunkt der Flaechе ***\n" ;
    point2d::input () ;
    cout << "Bitte Flaechе eingeben:          A = " ;
    cin  >> a ;
    if (a < 0.)
        {
            area_or_hole = - 1 ;
            a = - a ;
        }
    else
        area_or_hole = id ;
}

double plain_area::get_a () { return (a * area_or_hole) ; }

class rectangle : public abstract_area
{
    private:
        double b , h ;

    public:
        rectangle () ;
        ~rectangle () ;
        void  input_a (int id) ;
        double get_a  () ;
} ;

rectangle::rectangle ()
{
    b = h = 0. ;
}

rectangle::~~rectangle () {}

void rectangle::input_a (int id)
{
    double x1 , y1 ;
    cout << "*** Beliebiger Eckpunkt ***\n" ;
    point2d::input () ;
    x1 = point2d::get_x () ;
    y1 = point2d::get_y () ;
    cout << "*** Eckpunkt diagonal zum ersten Punkt ***\n" ;
    point2d::input () ;
    b = fabs (x1 - point2d::get_x ()) ;
    h = fabs (y1 - point2d::get_y ()) ;
    point2d::set_x ((x1 + point2d::get_x ()) / 2) ;
    point2d::set_y ((y1 + point2d::get_y ()) / 2) ;
    area_or_hole = id ;
}

double rectangle::get_a () { return (b * h * area_or_hole) ; }

class circle : public abstract_area
{
    private:
        double d ;

    public:
        circle () ;
        ~circle () ;
        void  input_a (int id) ;
        double get_a  () ;
} ;

```

```

circle::circle ()
{
    d = 0. ;
}

circle::~~circle () {}

void circle::input_a (int id)
{
    cout << "*** Mittelpunkt ***\n" ;
    point2d::input () ;
    cout << "*** Durchmesser ***\nd = " ;
    cin >> d ;
    area_or_hole = id ;
}

const double pi_4 = atan (1.) ;

double circle::get_a () { return (pi_4 * d * d * area_or_hole) ; }

main ()
{
    int      i , n = 0 , id ;
    double   ai , a = 0. , sx = 0. , sy = 0. ;

    abstract_area *area_array [20] ; // Feld fuer maximal 20 Pointer auf
                                     // Instanzen der Klasse abstract_area

    cout << "Schwerpunkt einer zusammengesetzten Flaechen\n" ;
    cout << "=====\n\n" ;

    do {
        cout << "Anzahl der Teilflaechen (max. 20):      n = " ;
        cin >> n ;
    } while (n <= 0 || n > 20) ;

    for (i = 0 ; i < n ; i++)
    {
        do {
            cout << "\n1 - Allgemeine Flaechen, 2 - Rechteck, 3 - Kreis" ;
            cout << "\n(negativer Typ-Indikator --> Ausschnitt)\n" ;
            cout << "Typ der Teilflaechen " << i + 1 << ": " ;
            cin >> id ;
        } while (id < -3 || id == 0 || id > 3) ;

        switch (id < 0 ? - id : id)
        {
            case 1: area_array [i] = new plain_area ; break ;
            case 2: area_array [i] = new rectangle  ; break ;
            case 3: area_array [i] = new circle      ; break ;
        }

        if (area_array [i] == NULL)
        {
            cout << "Fehler beim Allokieren von Speicherplatz\n" ;
            return 1 ;
        }

        area_array[i]->input_a (id < 0 ? - 1 : 1) ;
    }

    for (i = 0 ; i < n ; i++)
    {
        ai = area_array[i]->get_a() ;
        a += ai ;
        sx += ai * area_array[i]->get_y() ;
        sy += ai * area_array[i]->get_x() ;
    }
}

```

```

cout << "\nFlaeche"                A = " << a ;
if (fabs (a) > 1.e-20)
{
    cout << "\nSchwerpunkt-Koordinaten:  xS = " << sy / a ;
    cout << "\n"                          yS = " << sx / a << "\n" ;
}
return 0 ;
}

```

Das Programm wird mit der nachfolgend skizzierten Fläche getestet. Sie besteht aus einem doppelt-symmetrischen Sechseck, aus dem ein Rechteck und ein Kreis ausgeschnitten wurden. Es wird das skizzierte Koordinatensystem verwendet. Das Sechseck hat die Fläche $90 \cdot 45 = 4050$, sein Schwerpunkt liegt im Ursprung des gewählten Koordinatensystems, es wird als "Allgemeine Fläche" eingegeben. Die beiden Ausschnitte werden mit negativen Identifikatoren eingegeben.

Es ergibt sich der nachfolgend gelistete Eingabe-Dialog, man erhält die angegebenen Ergebnisse:

Schwerpunkt einer zusammengesetzten Flaeche
=====

Anzahl der Teilflaechen (max. 20) n = 3

1 - Allgemeine Flaeche, 2 - Rechteck, 3 - Kreis
(negativer Typ-Indikator --> Ausschnitt)

Typ der Teilflaeche 1: 1

*** Schwerpunkt der Flaeche ***

Bitte x-Koordinate eingeben: x = 0

Bitte y-Koordinate eingeben: y = 0

Bitte Flaeche eingeben: A = 4050

1 - Allgemeine Flaeche, 2 - Rechteck, 3 - Kreis
(negativer Typ-Indikator --> Ausschnitt)

Typ der Teilflaeche 2: -2

*** Beliebiger Eckpunkt ***

Bitte x-Koordinate eingeben: x = -15

Bitte y-Koordinate eingeben: y = -35

*** Eckpunkt diagonal zum ersten Punkt ***

Bitte x-Koordinate eingeben: x = 5

Bitte y-Koordinate eingeben: y = -5

1 - Allgemeine Flaeche, 2 - Rechteck, 3 - Kreis
(negativer Typ-Indikator --> Ausschnitt)

Typ der Teilflaeche 3: -3

*** Mittelpunkt ***

Bitte x-Koordinate eingeben: x = 5

Bitte y-Koordinate eingeben: y = 20

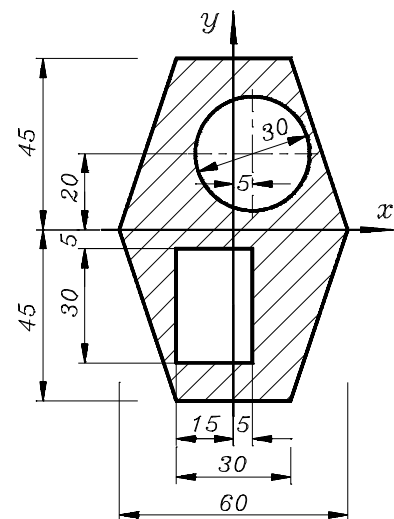
*** Durchmesser ***

d = 30

Flaeche A = 2743.14

Schwerpunkt-Koordinaten: xS = -0.194774

 yS = -0.779095



12.5 Überladen

Im nachfolgend gelisteten Programm **distance.cpp** werden 4 verschiedene Funktionen mit dem gleichen Namen **dist** definiert. Sie berechnen alle den Abstand eines durch 2 bzw. 3 Koordinaten definierten Punktes vom Nullpunkt, in zwei Funktionen sind die Koordinaten als double-Argumente anzugeben, in den beiden anderen Funktionen als int-Argumente.

Diese in C++ gegebene Möglichkeit, Funktionen mit gleichen Namen zu definieren, die vom Compiler nur durch die Argumentanzahl bzw. die Typen der Argumente voneinander zu unterscheiden sind, wird als "**Überladen von Funktionen**" bezeichnet.

```
// Abstand eines Punktes vom Nullpunkt (Programm distance.cpp)
// =====

// Es wird die Technik des "Ueberladens von Funktionen" demonstriert: Vier
// verschiedene Varianten einer Funktion, die den Abstand eines zwei- bzw.
// dreidimensionalen Punktes vom Nullpunkt berechnet, werden definiert. Sie
// unterscheiden sich jeweils durch die Anzahl bzw. die Typen der Argumente.

#include <iostream.h>
#include <math.h>

double dist (double , double , double) ;      // ... 3D mit double-Werten
double dist (double , double)                ; // ... 2D mit double-Werten
double dist (int   , int   , int)           ; // ... 3D mit int-Werten
double dist (int   , int)                   ; // ... 2D mit int-Werten

main ()
{
    // Der Compiler kann anhand der Argumentanzahl bzw. der Argumenttypen
    // entscheiden, welche Funktion aufzurufen ist:
    cout << "   dist = " << dist (2. , 3. , 4.) << "\n" ;
    cout << "   dist = " << dist (2. , 3.)   << "\n" ;
    cout << "   dist = " << dist (2 , 3 , 4) << "\n" ;
    cout << "   dist = " << dist (2 , 3)   << "\n" ;
    return 0 ;
}

// Die Ausgabeanweisungen wurden in die Funktionen eingebaut, damit erkennbar
// ist, dass tatsaechlich die "richtigen" Funktionen aufgerufen werden:

double dist (double x , double y , double z)
{
    cout << "Funktion dist (3D) mit double-Argumenten:" ;
    return sqrt (x * x + y * y + z * z) ;
}

double dist (double x , double y)
{
    cout << "Funktion dist (2D) mit double-Argumenten:" ;
    return sqrt (x * x + y * y) ;
}

double dist (int x , int y , int z)
{
    cout << "Funktion dist (3D) mit int-Argumenten:   " ;
    return sqrt (double (x * x + y * y + z * z)) ;
}

double dist (int x , int y)
{
    cout << "Funktion dist (2D) mit int-Argumenten:   " ;
    return sqrt (double (x * x + y * y)) ;
}
```

Das Programm liefert folgende Ausgabe:

```
Funktion dist (3D) mit double-Argumenten: dist = 5.38516
Funktion dist (2D) mit double-Argumenten: dist = 3.60555
Funktion dist (3D) mit int-Argumenten:    dist = 5.38516
Funktion dist (2D) mit int-Argumenten:    dist = 3.60555
```

Beim **Überladen von Funktionen** ist folgendes zu beachten:

- ◆ Der Compiler muß eindeutig entscheiden können, welche Funktion aufgerufen werden soll. Er entscheidet anhand der Anzahl und der Typen der Argumente, der **Typ des Return-Wertes der Funktion wird nicht zur Entscheidung herangezogen**.
- ◆ Bei Funktionen, für die **Standardwerte für Argumente** vorgesehen sind (vgl. Programm **class04.cpp** im Abschnitt 12.2.2), ergeben sich in jedem Fall verschiedene Aufrufmöglichkeiten mit unterschiedlicher Argument-Anzahl. In diesem Fall kann es für den Compiler zu einem nicht auflösbaren Konflikt kommen, wenn zusätzlich überladene Funktionen vorhanden sind, die untereinander nur über die Argument-Anzahl unterschieden werden.

Eine Besonderheit stellt die Möglichkeit des **Überladens von Operatoren** dar. Man kann den vordefinierten Operatoren (wie z. B. + - * / < <= == usw., neue Operatoren können nicht definiert werden) eine zusätzliche Funktionalität zukommen lassen, wenn mindestens ein Operand einen nicht-vordefinierten Typ hat, also ein Klassen-Objekt ist (damit ist klar, daß die Wirkungsweise der Operanden auf die vordefinierten Typen nicht verändert werden kann, das kleine Einmaleins kann also nicht manipuliert werden).

Dazu wird das Schlüsselwort **operator**, gefolgt von dem zu überladenden Operationssymbol benutzt. Man darf sich z. B. die Multiplikation als Funktion mit dem Namen **operator*** vorstellen. Es gibt zwei Möglichkeiten, diesen Operator zu überladen (die Syntax gilt natürlich sinngemäß auch für die anderen Operatoren):

- ◆ Es wird eine Funktion

```
Return_Typ operator* (Typ1 Operand1 , Typ2 Operand2) { ... }
```

definiert, die vom Compiler dann verwendet wird, wenn in einer Operation **x*y** der erste Faktor **x** vom **Typ1** ist und der zweite Faktor **y** vom **Typ2**. Dabei kann es Probleme mit den Zugriffsrechten auf Daten in Klassen-Objekten geben (das Stichwort für die Lösung dieses Problems lautet "**friend-Funktionen**", wird später besprochen).

- ◆ Vorzuziehen (wenn möglich) ist in jedem Fall die Bereitstellung einer Methode für die Klasse, die den Typ des ersten Operanden bestimmt, der selbst dann nicht als Argument übergeben wird, so daß die Definition genau einen Parameter weniger vorsehen muß:

```
Return_Typ Klassen_Name::operator* (Typ2 Operand2) { ... }
```

(der **Klassen_Name** in dieser Definition entspricht dem **Typ1** in der ersten Variante der Funktions-Definition).

Das nachfolgende Programm **vecalg1.cpp** zeigt, wie für eine Klasse die Wirkungsweise einiger Operatoren überladen wird. Die entsprechenden Funktionen, die die Operatoren überladen, werden dann vom Compiler berücksichtigt, wenn für die Kombination "Operator und Typ der Operanden" eine geeignete Funktion gefunden wird:

```
// Vektor-Algebra (Programm vecalg1.cpp)
// =====

// Fuer eine Klasse vector werden exemplarisch einige Operationen,
// die den Regeln der Vektor-Algebra entsprechen, durch Ueberladen
// der Standard-Operatoren definiert.

#include <iostream.h>

class vector
{
private:
    double xc , yc , zc ;
public:
    vector (double x = 0. , double y = 0. , double z = 0.) ;
    ~vector () ;
    void print (char *) ;
    vector operator+ (vector&) ; // Vektor-Addition
    double operator* (vector&) ; // Skalarprodukt
    vector operator* (double) ; // Produkt 'Vektor*Skalar'
    vector operator- () ; // Vorzeichen (unaerer Operator)
} ;

vector::vector (double x , double y , double z)
{
    xc = x ; yc = y ; zc = z ;
}

vector::~vector () {}

void vector::print (char *text)
{
    cout << text << "[" << xc << " , " << yc << " , " << zc << "]" \n" ;
}

// Bei binaeren Operatoren kann auf die Daten der Instanz links vom
// Operator direkt zugegriffen werden, die Instanz rechts vom Operator
// wird der Methode als Argument uebergeben (hier: op2):

vector vector::operator+ (vector &op2)
{
    return vector (xc + op2.xc , yc + op2.yc , zc + op2.zc) ;
    // Dies ist eine besondere Art, ein Klassen-Objekt (gewissermassen
    // fuer "fluechtige" Verwendung, in diesem Fall fuer den Return-Wert)
    // zu erzeugen. Die Syntax ist analog zum Aufruf des Konstruktors der
    // Klasse vector und entspricht damit dem Definieren einer Variablen
    // mit Initialisierung.
}

double vector::operator* (vector &op2)
{
    return xc * op2.xc + yc * op2.yc + zc * op2.zc ;
}

// Die Methode 'operator* (vector)' wird mit einer Methode gleichen
// Namens 'operator* (double)' ueberladen:

vector vector::operator* (double scal)
{
    return vector (xc * scal , yc * scal , zc * scal) ;
}
```

```

// Man beachte: Es wurde eine Methode definiert, mit der 'Vektor*Skalar'
//              berechnet werden kann, fuer 'Skalar*Vektor' kann keine
//              Methode definiert werden, weil der erste Operand immer
//              vom Typ der Klasse selbst sein muss.

// Bei einem unaeren Operator (hier: Minuszeichen, das ist nicht das
// Subtraktionssymbol) wird kein Argument uebergeben:

vector vector::operator- ()
{
    return vector (- xc , - yc , - zc) ;
}

// Man beachte: Mit den definierten Methoden koennen Ausdruecke wie
//              -a+b oder a+(-b) berechnet werden, wenn a und b vom
//              Typ vector sind, aber nicht a-b, dafuer muesste die
//              Methode 'operator- ()' mit einer zusaetzlichen Methode
//              'operator- (vector)' ueberladen werden.

// Nachfolgend wird der Multiplikations-Operator ein weiteres Mal
// ueberladen, um auch die Multiplikation 'Skalar*Vektor' ausfuehren
// zu koennen. Dies kann nicht durch eine Methode der Klasse vector
// realisiert werden, deshalb muessen BEIDE Operanden an die
// Funktion uebergeben werden:

vector operator* (double scal , vector op2)
{
    return op2 * scal ;    // ... nutzt die oben definierte Methode
                          // fuer die Operation 'Vektor*Skalar'
}

main ()
{
    vector a (2. , -3. , 5.) , b (3. , 4. , -7.) , c ;

    a.print ("Vektor a:  ") ;
    b.print ("Vektor b:  ") ;
    c = a + b ;
    c.print ("a + b      = ") ;
    c = - a + b ;
    c.print ("- a + b     = ") ;
    cout << "Skalarprodukt a*b = " << a * b << "\n" ;
    c = (a + b) * 1.5 + 2 * b ;
    c.print ("(a + b) * 1.5 + 2 * b = ") ;

    // Der Compiler fuehrt eine relativ scharfe Typ-Ueberpruefung aus.
    // Eine Anweisung wie
    //              c = a * b + b ;
    // wuerde beanstandet werden, weil nach dem Berechnen des Skalar-
    // Produkts eine skalare Groesse zu einem Vektor addiert werden
    // muesste. Dagegen ist die folgende Anweisung natuerlich korrekt:

    c = (a * b) * b + b ;
    c.print ("(a * b) * b + b      = ") ;

    return 0 ;
}

```

Das Programm produziert folgende Ausgabe:

```

Vektor a:  [2 , -3 , 5]
Vektor b:  [3 , 4 , -7]
a + b      = [5 , 1 , -2]
- a + b     = [1 , 7 , -12]
Skalarprodukt a*b = -41
(a + b) * 1.5 + 2 * b = [13.5 , 9.5 , -17]
(a * b) * b + b      = [-120 , -160 , 280]

```

Man beachte:

Auch bei überladenen Operatoren bleiben die Vorrangregeln bei der Behandlung zusammengesetzter Ausdrücke erhalten.

12.6 Eingabe und Ausgabe, Arbeiten mit Dateien

Wenn hier trotz der kompletten Verfügbarkeit aller C-Routinen für die Ein- und Ausgabe und der mehrfach geschriebenen Bemerkung, daß komfortable Ein- und Ausgabe von Daten der Windows-Programmierung vorbehalten sein sollte, noch ein spezieller Abschnitt zu diesem Thema eingefügt wird, hat das folgende Gründe:

- ◆ Die Realisierung der Ein- und Ausgabe in C++ ist ein geradezu klassisches Beispiel für das Arbeiten mit Klassen und das Überladen von Operatoren. Sie sollten die nachfolgenden Erläuterungen als Test dafür ansehen, ob Sie diese in den vorigen Abschnitten behandelten Themen auch wirklich verstanden haben.
- ◆ Zum Arbeiten mit Dateien sind ohnehin noch einige bisher nicht erwähnte C++-spezifische Informationen zu behandeln.

An dieser Stelle ist eine grundsätzliche Bemerkung zur Portabilität von C++-Programmen angebracht. Die Programmiersprache C++ ist noch relativ jung, die meisten verfügbaren Compiler wurden bereits zu einer Zeit konzipiert, als Standardisierungs-Bemühungen überhaupt noch nicht angelaufen waren. Glücklicherweise haben sich die wichtigsten heute verfügbaren Compiler an einem "Quasi-Standard" orientiert, der von B. Stroustrup, dem "Erfinder" von C++, und M. A. Ellis mit dem 1990 erschienenen Buch "The Annotated C++ Reference Manual" gesetzt wurde. Deshalb sind z. B. alle bisher behandelten Beispiel-Programme ohne jede Anpassung u. a. mit Microsoft- und Borland-C++-Compilern und auch mit dem GNU-C++-Compiler zu übersetzen.

In speziellen Nischen ergeben sich bei einem fehlenden Standard aber zwangsläufig Unterschiede, im Abschnitt 12.6.3 wird auf einen solchen Unterschied aufmerksam gemacht.

12.6.1 Das Klassen-Objekt `cout`

Nachfolgend wird die Realisierung der Ausgabe mit `cout` beschrieben, alle Erläuterungen gelten sinngemäß auch für die Eingabe über `cin` und die Ausgabe über `cerr` bzw. `clog` (`cin` ist im Regelfall mit der Tastatur verbunden, Ausgaben über `cout`, `cerr` und `clog` landen üblicherweise auf dem Bildschirm, wobei die für Fehlermeldungen vorgesehenen Objekte `cerr` und `clog` sich nur in der Art der Pufferung der Ausgabe unterscheiden).

- ◆ Die für das Arbeiten mit `cout`, `cin`, `cerr` und `clog` erforderlichen Definitionen und Deklarationen werden über `iostream.h` eingebunden (nicht alle findet man direkt in dieser Datei, sie inkludiert selbst weitere Dateien).
- ◆ `cout` ist eine Instanz einer Klasse `ostream_withassign` und wird dementsprechend mit einer "normalen Definitionsanweisung" als globale Variable erzeugt (darum braucht sich der Programmierer nicht zu kümmern, die Objekte `cout`, `cin`, `cerr` und `clog` sind vordefiniert). Die Klasse `ostream_withassign` ist abgeleitet aus der Klasse `ostream` und diese wiederum aus der Klasse `ios`, so daß zum Objekt `cout` zahlreiche Daten und Methoden gehören, die entlang dieser Erbfolge "eingesammelt" wurden.

- ◆ Der Operator `<<` dient in der Sprache C (und damit auch in C++) für die bitweise Verschiebung nach links innerhalb einer Variablen ($j = i \ll 2$ verschiebt z. B. alle Bits der Variablen `i` um 2 Positionen nach links und weist das Ergebnis der Variablen `j` zu). Dieser Operator wird in der Klasse **ostream** mehrfach überladen, so daß er für alle möglichen Typen des rechten Operanden benutzt werden kann, wenn der linke Operand vom Typ **ostream** oder einer aus ihr abgeleiteten Klasse ist (seine ursprüngliche Bedeutung geht dabei natürlich nicht verloren, weil die Methoden, die in **ostream** den Operator überladen, natürlich vom Compiler nur dann verwendet werden, wenn der linke Operand den passenden Typ des Klassen-Objekts hat).
- ◆ Da Operatoren `<<` bei mehrfachem Auftreten in einer Anweisung von links nach rechts abgearbeitet werden, sind auch die verketteten Ausgaben, wie sie im Beispiel-Programm **tauschen.cpp** im Abschnitt 12.1 erstmalig kommentiert wurden, möglich. Der Programmierer darf die Vorstellung haben, einen "Ausgabestrom zum Objekt **cout** zu leiten".
- ◆ In den Ausgabestrom dürfen sogenannte **Manipulatoren** eingefügt werden. Dies sind Funktionen, die z. B. Format-Informationen einfügen. Zu den **argumentlosen Manipulatoren** gehören z. B. **endl** (Einfügen des "End of Line"-Zeichens, Leeren des Ausgabepuffers) und **dec**, **hex** und **oct**, die die Ausgabe aller nachfolgenden ganzen Zahlen als "dezimal", "hexadezimal" bzw. "oktal" veranlassen (gilt jeweils bis zum nächsten Manipulator aus dieser Gruppe, Voreinstellung ist "dezimal"). Im Gegensatz dazu muß z. B. der Manipulator **setw** (**int sz**) mit einem **int**-Argument aufgerufen werden, das die Breite des (rechtsbündig zu füllenden) Feldes für die nachfolgende Ausgabe festlegt. Die Programmzeile


```
cout << "3429 = " << hex << setw (6) << 3429 << " (hex.)" << endl ;
```

 produziert unter Verwendung von 6 Positionen für die Hexadezimalzahl die Ausgabe:


```
3429 =      d65 (hex.)
```

 (bei der Verwendung von Manipulatoren mit Argumenten muß die Header-Datei **iomanip.h** zusätzlich eingebunden werden).
- ◆ Alternativ zu den Manipulatoren, die als "normale" Funktionen definiert sind, kann man auch die von der Klasse **ios** geerbten Methoden benutzen, die mit dem Namen des Objekts (**cout**) aufgerufen werden müssen. Z. B. legt man mit dem Aufruf

```
cout.width (30) ;
```

die Breite des nächsten (rechtsbündig zu füllenden) Ausgabefeldes auf 30 Positionen fest, gleichwertig wäre das Einfügen des Manipulators **setw(30)** in den Ausgabestrom.

Die beschriebene Funktionalität von **cout** soll als Beispiel für die Anwendung eines Klassen-Objektes dienen und die konsequente Realisierung der Ausgabe im C++-Stil demonstrieren. Es gibt weit mehr Funktionalität als hier beschrieben wurde, der Programmierer, der solche Programme mit zeilenweiser Ausgabe schreibt, sollte sich in den Handbüchern informieren.

Auf eine Darstellung der mit **cin** für die Eingabe gegebenen Möglichkeiten wird auch deshalb verzichtet, weil hierfür ohnehin nur die "klassische Variante" mit "Schreiben eines Prompts und Eingabe nur eines Wertes" verwendet werden sollte.

12.6.2 Dateien

Die Deklarationen und Definitionen, die für das Arbeiten mit Dateien erforderlich sind, werden über die Datei **fstream.h** eingebunden. Da die wichtigsten Klassen, die in **fstream.h** deklariert werden, aus Basisklassen abgeleitet werden, die in **iostream.h** deklariert werden, wird diese Datei von **fstream.h** inkludiert. Wenn also (für das Arbeiten mit Dateien) **fstream.h** in ein Programm eingebunden werden muß, sollte auf das zusätzliche Einbinden von **iostream.h** verzichtet werden.

Die Strategie der Arbeit mit Dateien ist dem im Kapitel 6 beschriebenen Vorgehen sehr ähnlich: Eine zu bearbeitende Datei muß "geöffnet" werden, es folgen die Lese- bzw. Schreib-Aktionen, und schließlich wird die Datei wieder geschlossen. In C++ kann dies alles jedoch konsequent objektorientiert ablaufen:

- ◆ Für die Arbeit mit Dateien stehen die Klassen **ifstream** (für das Lesen), **ofstream** (für das Schreiben) und **fstream** (für Lesen und Schreiben) zur Verfügung. Sie sind aus den Klassen **istream**, **ostream** bzw. **iostream** abgeleitet, so daß alle für das Arbeiten mit den Standard-I/O-Objekten vorgesehenen Methoden (und natürlich noch wesentlich mehr) verfügbar sind.
- ◆ Im Gegensatz zu den Standard-I/O-Objekten **cout**, **cin**, **cerr** und **clog**, die vordefiniert sind, müssen die für das Arbeiten mit Dateien zu verwendenden Objekte vom Programmierer definiert werden (Erzeugen einer Instanz der entsprechenden Klasse mit einem frei festzulegenden Namen). Mit dem Erzeugen einer Instanz einer der Klassen **ifstream**, **ofstream** oder **fstream** wird diese nicht automatisch mit einer bestimmten Datei verbunden. Dies kann entweder durch den Aufruf der Methode **open** oder beim Definieren des Objekts mit einem speziellen Konstruktor-Aufruf realisiert werden. Das nachfolgend gelistete Programm **file2.cpp** realisiert die zweite Variante und erläutert im Kommentar am Programm-Ende die alternative Möglichkeit.

Mit der Methode **close** kann eine Datei geschlossen werden, und das definierte Objekt kann gegebenenfalls mit erneutem Aufruf von **open** mit einer anderen Datei verknüpft werden. Im Regelfall verzichtet der Programmierer auf einen Aufruf der Methode **close**, weil die von **close** zu leistende Arbeit auch vom Destruktor der Klasse erledigt wird. Dieser wird bekanntlich automatisch aktiv, wenn die Instanz ihre Existenz aufgibt, spätestens also beim Programm-Ende.
- ◆ Eine konsequente Unterscheidung zwischen "Sequentiellen Files" und "Direct-Access-Files" wie in anderen höheren Programmiersprachen kennt C++ nicht. Alle Files können sequentiell geschrieben und gelesen werden, trotzdem kann man sich "relativ frei bewegen" und exakt einzelne Positionen ansteuern (dabei ist man nicht wie in anderen Sprachen auf feste oder variable "Record-Längen" festgelegt). Hilfreich ist die Vorstellung eines Magnetbandes, über das sich ein "Lese-Schreib-Kopf" bewegt. Die Anfangsposition in der Datei hat die "Adresse 0", jedes Byte in der Datei ist gegebenenfalls über eine positive ganze Zahl als "Adresse" erreichbar.
- ◆ In der Klasse **ios** ("Urahn" aller hier besprochenen Klassen) finden sich zahlreiche (**public** deklarierte) "Bit-Flags", die für das Arbeiten mit den Methoden nützlich sind. Sie können, wenn dies sinnvoll ist, mit dem "Logischen Oder" | kombiniert werden.

Im nachfolgenden Beispiel-Programm wird dies beim Aufruf des Konstruktors mit dem zweiten Argument demonstriert, das entsprechend

```
ios::in | ios::nocreate
```

das Flag **ios::in** (Datei öffnen für Eingabe) mit dem Flag **ios::nocreate** (nur bereits existierende Datei öffnen) verbindet.

- ◆ Für das Registrieren eines Mißerfolgs einer Datei-Operation dienen mehrere "Error-Flags", die einzeln oder pauschal abgefragt werden können (sollte natürlich immer geschehen). Das nachfolgende Programm demonstriert die Verwendung der Methode **good()** für eine solche Abfrage.

```
// Oeffnen eines Files, Lesen vom File, File schliessen (file2.cpp)
// =====
// Programm muss aufgerufen werden mit
//
//           file2 filename
//
// (filename ist der Name eines beliebigen Files) und ermittelt
// die Anzahl der Zeichen im File.
//
// Demonstriert werden
//
// * das Oeffnen eines Files mit Hilfe des Konstruktors, Ueberpruefung,
//   ob Aktion erfolgreich war,
//
// * die istream-Methoden seekg und tellg.
#include <fstream.h>
main (int argc , char *argv [])
{
    if (argc > 1)           // ... steht ein File-Name in der Kommandozeile
    {
        // Definieren einer Instanz (hier gewaehlter Name: file) der
        // Klasse ifstream, durch den Konstruktor wird ein File mit dem
        // Namen argv[1] geoeffnet (wenn existent):
        ifstream file (argv [1] , ios::in | ios::nocreate) ;
                        // ... oeffnet File zum Lesen, erzeugt Fehler,
                        //      wenn File nicht existiert

        if (!file.good ()) // ... fragt ab, ob ein Fehler-Bit gesetzt
        {                  //      wurde
            cout << "Fehler beim Oeffnen des Files " << argv [1] << "\n" ;
            return 1 ;
        }

        file.seekg (0L , ios::end) ; // ... bewegt imaginaeren Lesekopf
                                    //      an das File-Ende

        cout << "Anzahl der Zeichen: " << file.tellg () << endl ;
                                    // ... gibt aktuelle Position des
                                    //      Lesekopfes aus
    }

    return 0 ;
}

// Beim Definieren einer Instanz der Klasse ifstream wird nicht zwangs-
// laeufig auch ein File geoeffnet. Gleichwertig mit der oben gewaehlten
// Anweisung waere das Oeffnen mit der Methode ifstream::open:
```

```

// ifstream file ; // ... erzeugt Instanz mit dem Standard-Konstruktor
// file.open (argv [1] , ios::in | ios::nocreate) ; // ... oeffnet File

// Die fuer das zweite Argument (in ios definierten) Werte koennen mit
// dem "Bitweisen Oder" | kombiniert werden (hier: "File oeffnen fuer
// Eingabe" und "File wird bei Nicht-Existenz NICHT erzeugt").

// Durch "Erbfolge" gehoeren zu ifstream alle Daten und Methoden der
// Klassen istream und ios.

// Die Methode ios::good prueft alle Fehler-Bits und gibt nur dann einen
// Wert ungleich Null zurueck, wenn kein Fehler-Bit gesetzt wurde.

// Die Methode istream::seekg bewegt den imaginaeren Lesekopf auf den
// angegebenen Wert. Sie kann mit der absoluten Position (ein Argument)
// aufgerufen werden, z. B.:

//                                     file.seekg (17L) ;

// steuert Byte-Position 17 (long-Wert) an. Beim Aufruf mit zwei Argumenten
// ist das erste Argument (long-Wert) ein Offset, das zweite Argument
// kann einer der in ios public definierten Werte sein:

//          ios::beg      -->  File-Anfang
//          ios::end      -->  File-Ende
//          ios::cur      -->  Aktuelle Position des imaginaeren Lesekopfes

// Beispiel:
//          file.seekg (-10L , ios::end) ;

// ... postiert den imaginaeren Lesekopf 10 Positionen vor dem File-Ende.

```

Das folgende Beispiel-Programm **femfile5.cpp** entspricht in der Funktionalität dem Programm **femfile1.c** aus dem Abschnitt 6.3. Es demonstriert das Lesen eines ASCII-Files im typisch "sequentiellen Stil". Man beachte folgende Besonderheiten:

- ◆ Von einem Objekt, das mit einem File verknüpft ist, kann man im gleichen Stil ("Eingabestrom") wie vom Objekt **cin** mit Hilfe des überladenen Operators **>>** Daten auf Programm-Variablen übertragen. Als **Delimiter** ("Begrenzer") eines Wertes werden Leerzeichen, TAB-Zeichen und "Newline"-Zeichen akzeptiert.
- ◆ In der Klasse **ios** wird der Operator **!** ("Negations-Operator") überladen. Dies ermöglicht die Abfrage des Erfolgs einer Operation mit Objekten von Klassen, die aus **ios** abgeleitet sind. Auch die Operationen mit den überladenen Operatoren **<<** bzw. **>>** können auf diese Weise überprüft werden.
- ◆ Von den zahlreichen in der Erbfolge erworbenen Methoden wird die sehr nützliche **istream**-Methode **getline** vorgestellt, die (per Voreinstellung, auch das ist durch Angabe eines zusätzlichen Arguments änderbar) bis einschließlich des "Newline"-Zeichens liest und einen String (ohne das "Newline"-Zeichen, dafür mit der begrenzenden "ASCII-Null") abliefert.
- ◆ Das hier nicht demonstrierte Schreiben eines ASCII-Files im "sequentiellen Stil" erfolgt analog zur Ausgabe auf **cout** (vgl. Abschnitt 12.6.1), wobei natürlich z. B. ein Objekt der Klasse **ofstream** erzeugt werden muß. Programme, die ASCII-Files in diesem Stil schreiben und lesen, müssen in den Aus- bzw. Eingabeanweisungen natürlich exakt aufeinander abgestimmt sein. Allerdings können solche Files gegebenenfalls sogar zwischen unterschiedlichen Betriebssystemen ausgetauscht werden.

```

// Identifikation eines in einem File gespeicherten
// Berechnungsmodells (Programm femfile5.c)
// =====

// Das Programm femfile5.cpp ist die C++-Version des Programms femfile1.c
// aus dem Abschnitt 6.3 (Analyse eines FEM-Modell-Files). Es wird mit

//          femfile5 filename

// aufgerufen (wenn der File-Name in der Kommandozeile fehlt, wird
// "femmod.dat" angenommen), liest die ersten 4 Zeilen des Files und
// versucht zu entschluesseln, was fuer ein Berechnungsmodell beschrieben
// wird. Das Programm kann getestet werden mit den Files femmod1.dat
// bis femmod4.dat.

#include <fstream.h>
#include <string.h>
#include <stdio.h>

main (int argc , char *argv [])
{
    char    flname [FILENAME_MAX+1] = "femmod.dat" ;
    char    line   [80] ;
    int     kx , kf , ke , kp , ne , nk ;
    int     bekannt = 1 ;

    if (argc > 1) strcpy (flname , argv [1]) ;

    ifstream femfile (flname , ios::in | ios::nocreate) ;

    if (!femfile) // ... gleichwertig mit:      if (!femfile.good ())
    {
        cout << "Fehler beim Oeffnen des Files " << flname << endl ;
        return 1 ;
    }

    // Die Methode istream::getline liest eine Zeile, maximale Anzahl der zu
    // lesenden Zeichen wird als zweites Argument angegeben. Da der Operator !
    // in ios ueberladen wird, koennen der Erfolg der Methoden und das Ergebnis
    // des (ebenfalls ueberladenen) Operators >> abgefragt werden:

    if (!femfile.getline (line , 80)) goto Fehler ; // ... liest eine Zeile
    if (!(femfile >> kx >> kf >> ke >> kp)) goto Fehler ; // ... liest 4 Werte

    if (!femfile.getline (line , 80)) goto Fehler ; // ... liest Rest der Zeile
        // (erforderlich, weil Zeilenende-Kennzeichen noch nicht gelesen wurde)
    if (!femfile.getline (line , 80)) goto Fehler ; // ... liest eine Zeile
    if (!(femfile >> ne >> nk)) goto Fehler ;      // ... liest 2 Werte

    cout << "File " << flname << " beschreibt ein " ;

    if (ke == 2)
    {
        switch (kx)
        {
            case 2: cout << "zweidimensionales " ; break ;
            case 3: cout << "dreidimensionales " ; break ;
            default: bekannt = 0 ; break ;
        }
        if (bekannt)
        {
            switch (kf)
            {
                case 2: cout << "Fachwerk" ; break ;
                case 3: if (kx == 3) cout << "Fachwerk" ;
                    else cout << "Rahmentragwerk" ;
                    break ;
                case 6: cout << "Rahmentragwerk" ; break ;
            }
        }
    }
}

```

```

        default: bekannt = 0 ; break ;
    }
}
else
    bekannt = 0 ;

if (!bekannt) cout << "unbekanntes Gebilde" ;
cout << "\nmit " << ne << " Elementen und " << nk << " Knoten\n" ;

return 0 ;

Fehler:
    cout << "Fehler beim Lesen vom File " << fname << endl ;
    return 1 ;
}

```

- ◆ Die beiden Beispiel-Programme dieses Abschnitts haben (bis auf den Aufruf der Methode **seekg** im Programm **file2.cpp**) von der Möglichkeit, bestimmte Positionen im File anzusteuern, keinen Gebrauch gemacht. Weil dies im folgenden Abschnitt bei der Arbeit mit Binär-Dateien demonstriert wird, soll hier noch einmal ausdrücklich darauf hingewiesen werden, daß diese Möglichkeiten auch für die Text-Dateien verfügbar sind.

12.6.3 Die Methoden **ostream::write** und **istream::read**, Binär-Dateien

Die Methode **ostream::write** fügt eine vorzugebende Anzahl Bytes in den Ausgabestrom ein. Es findet weder eine Formatierung noch eine Konvertierung statt, es wird exakt das "Bitmuster" eingefügt, das im Speicher des Programms existierte. Wenn man auf diese Weise z. B. int- oder double-Werte in eine Datei schreibt, ist es nicht sehr hilfreich, sich diese anschließend mit einem Editor anzusehen. Wenn man allerdings dieses "Bitmuster" mit der Methode **istream::read** auf Variablen gleichen Typs wieder einliest, wird es natürlich wieder richtig interpretiert.

Ausgesprochen bequem (und natürlich schnell) ist die Methode **write** für das Speichern größerer Datenmengen, die in einem Array oder in einer Struktur zusammengefaßt sind. Gerade bei Strukturen könnte eine "intelligente" Speicherung bei unterschiedlichen Typen der einzelnen Komponenten sehr aufwendig sein.

Die Syntax für den Aufruf von **write** ist angenehm einfach. Es müssen nur zwei Argumente übergeben werden, ein char-Pointer, der den Anfang des auszugebenden Bereichs festlegt (andere Datentypen müssen auf char* "gecastet" werden), und die Anzahl von Bytes, die geschrieben werden sollen. Die Syntax von **read** ist analog dazu (ebenfalls zwei Argumente mit diesen Typen).

Man sollte prinzipiell Dateien, die mit diesen Methoden bearbeitet werden, als Binär-Dateien behandeln, damit auch wirklich nur die "Bitmuster" abgelegt und in keinem Fall interpretiert werden (bei Text-Dateien hat das "Newline"-Zeichen eine besondere Bedeutung, aber genau dessen "Bitmuster" kann natürlich in den Bytes, die z. B. int- oder double-Werte darstellen, auch vorkommen, ohne daß es eine besondere Bedeutung hat).

Laut Voreinstellung wird beim Öffnen stets der Modus "Text-Datei" angenommen, die Absicht, die Datei als Binär-Datei zu behandeln, muß explizit beim Öffnen erklärt werden (bei Microsoft- und Borland-Compilern mit dem Flag `ios::binary`, der GNU-C++-Compiler sieht dafür das Flag `ios::bin` vor).

Das Programm `femfile6.cpp` legt mit den Zahlenwerten, die von einer anderen Datei gelesen, eine Binär-Datei an, die vom Programm `femfile7.cpp` gelesen wird:

```
// Schreiben eines binären Files (Programm femfile6.cpp)
// =====

// Es werden wie im Programm femfile6.cpp die ersten Zeilen eines FEM-
// Modell-Files gelesen (dabei werden im Gegensatz zu femfile5.cpp die
// Zahlenwerte in zwei Strukturen gespeichert). Es wird ein Binär-File
// "femmod.bin" zur Ausgabe geöffnet, um einmal vier und danach noch
// einmal zwei int-Werte binär abzulegen. Dabei wird das Prinzip einer
// "Datei mit Index" demonstriert (am Anfang der Datei werden die
// Positionen verzeichnet, bei denen die Datengruppen beginnen).
// Auf eine Absicherung der Ausgabe (Abfrage des Erfolgs der Aktion)
// wird hier verzichtet. Dies kann analog zur Absicherung der Eingabe-
// aktionen programmiert werden (z. B. mit dem ueberladenen Operator !).

#include <fstream.h>
#include <string.h>
#include <stdio.h>

main (int argc , char *argv [])
{
    char    flname  [FILENAME_MAX+1] = "femmod.dat" ;
    char    line    [80] ;
    int     startpos [] = {0 , 0} ;
    struct  { int     kx , kf , ke , kp ;
             } femmod_char ;
    struct  { int     ne , nk ;
             } femmod_size ;

    if (argc > 1) strcpy (flname , argv [1]) ;

    ifstream femfile (flname , ios::in | ios::nocreate) ;
    if (!femfile)
    {
        cout << "Fehler beim Oeffnen des Files" << flname << endl ;
        return 1 ;
    }

    // Oeffnen des Files "femmod.bin" zur Ausgabe im "Binary-Mode"
    // (Achtung, Unterschiede beachten: Fuer Microsoft- und Borland-
    // Compiler heisst das Flag ios::binary, der GNU-C++-Compiler
    // erwartet ios::bin):

    ofstream fembin ("femmod.bin" , ios::out | ios::binary) ;
    if (!fembin)
    {
        cout << "Fehler beim Oeffnen des Files femmod.bin" << endl ;
        return 1 ;
    }

    if (!femfile.getline (line , 80)) goto Fehler ;
    if (!(femfile >> femmod_char.kx >> femmod_char.kf >>
          femmod_char.ke >> femmod_char.kp)) goto Fehler ;

    // Schreiben von 2 int-Werten (Array startpos) als "Platzhalter" fuer
    // die spaeter einzusetzenden Positions-Informationen (ausfuehrlicher
    // Kommentar zur Methode ostream::write am Programm-Ende):
    fembin.write ((char *) startpos , 2 * sizeof (int)) ;
}
```

```

// Die Position des imaginaeren "Schreibkopfes" wird registriert:
startpos [0] = fembin.tellp () ;

// Struktur femmod_char wird geschrieben:
fembin.write ((char *) &femmod_char , sizeof (femmod_char)) ;

if (!femfile.getline (line , 80)) goto Fehler ; // ... liest Rest der Zeile
if (!femfile.getline (line , 80)) goto Fehler ; // ... liest eine Zeile
if (!(femfile >> femmod_size.ne >> femmod_size.nk)) goto Fehler ;

// Neue Position des imaginaeren "Schreibkopfes" wird registriert:
startpos [1] = fembin.tellp () ;

// Struktur femmod_size wird geschrieben:
fembin.write ((char *) &femmod_size , sizeof (femmod_size)) ;

// Ruecksetzen des "Schreibkopfes" an den File-Anfang:
fembin.seekp (0L , ios::beg) ;

// Schreiben der Positions-Informationen:
fembin.write ((char *) startpos , 2 * sizeof (int)) ;

cout << "File 'femmod.bin' wurde erzeugt" << endl ;

return 0 ;

Fehler:
    printf ("Fehler beim Lesen vom File \"%s\"\n" , filename) ;
    return 1 ;
}

// Fuer das Arbeiten mit Binaer-Files sind die ostream-Methode write und
// die istream-Methode read besonders geeignet. Beide erwarten zwei
// Argumente, einen Pointer auf ein char-Array und einen int-Wert, der die
// Anzahl der zu schreibenden bzw. zu lesenden char-Werte (Byte)
// festlegt. Wenn ein File im Binary-Mode geoeffnet wurde, wird exakt die
// so festgelegte Byte-Anzahl (ohne Konvertierung oder Formatierung,
// einfach als "Bitmuster") uebertragen. Damit sind natuerlich auch
// beliebige andere Datentypen uebertragbar, nur muss der Pointer auf den
// Anfang des Datenbereichs auf den Typ char* "gecastet" werden.

//          fembin.write ((char *) startpos , 2 * sizeof (int)) ;

// ... uebertraegt Elemente des Arrays startpos auf das Objekt fembin. Der
// Pointer auf das erste Element des Arrays (repraesentiert durch den
// Array-Namen startpos) wird auf char* "gecastet". Da es ein int-Array
// ist, wird die zu uebertragene Byte-Anzahl durch sizeof(int),
// multipliziert mit der Anzahl der Array-Elemente (hier: 2), ermittelt.

// Aehnlich ist die Anweisung

//          fembin.write ((char *) &femmod_size , sizeof (femmod_size)) ;

// zu interpretieren: Es wird die komplette Struktur femmod_size
// uebertragen. Der Pointer auf die Struktur &femmod_size wird nach
// char* "gecastet", sizeof(femmod_size) liefert die zu uebertragene
// Byte-Anzahl.

```

Das Anlegen der Datei "femmod.bin" mit einem Index ("Inhaltsverzeichnis") hat nichts mit dem Thema "Binär-Datei" zu tun. Es soll nur demonstrieren, welche Möglichkeiten gegeben sind, wenn man sich in der Datei "frei bewegen" kann. In einem Index können natürlich auch weitere Informationen (Datentypen, Anzahl der Werte, ...) verschlüsselt werden, so daß schließlich "intelligente Files" mit Programmen korrespondieren, die diese Informationen auswerten. Das Programm **femfile7.cpp** liest die Binär-Datei, die vom Programm **femfile6.cpp** erzeugt wurde und gibt zur Kontrolle die eingelesenen Werte über **cout** aus:

```

// Lesen eines binären Files (Programm femfile7.c)
// =====

// Es wird die Datei "femmod.bin", die mit dem Programm femfile6.c
// geschrieben wird, gelesen. Zum Verständnis der Aktionen in
// femfile7.cpp sind die Kommentare im Programm femfile6.cpp
// hilfreich.

// Im Gegensatz zum Programm femfile6.cpp wird jeweils der Erfolg
// der Aktionen mit dem File überprüft. Es wird gezeigt, dass
// dafür die bereits bekannte Strategie mit dem überladenen
// Operator ! verwendet werden kann.

#include <fstream.h>
#include <string.h>
#include <stdio.h>

main ()
{
    int    startpos [2] ;

    struct { int    kx , kf , ke , kp ;
            } femmod_char ;
    struct { int    ne , nk ;
            } femmod_size ;

    // Öffnen des Files "femmod.bin" zur Eingabe im "Binary-Mode"
    // (Achtung, Unterschiede beachten: Für Microsoft- und Borland-
    // Compiler heisst das Flag ios::binary, der GNU-C++-Compiler
    // erwartet ios::bin):

    ifstream fembin ("femmod.bin" , ios::in | ios::binary) ;
    if (!fembin)
    {
        cout << "Fehler beim Öffnen des Files femmod.bin" << endl ;
        return 1 ;
    }

    // Lesen der Positions-Informationen ab Datei-Anfang:
    if (!(fembin.read ((char *) startpos , 2 * sizeof (int)))) goto Fehler ;

    // In startpos[0] steht die File-Position für die Informationen,
    // die in die Struktur femmod_char übertragen werden sollen,
    // positionieren des imaginären "Lesekopfes" und Lesen der
    // entsprechenden Byte-Anzahl:
    if (!(fembin.seekg (startpos [0]))) goto Fehler ;
    if (!(fembin.read ((char *) &femmod_char , sizeof (femmod_char))))
        goto Fehler ;

    // Dieselbe Aktion für die Struktur femmod_size:
    if (!(fembin.seekg (startpos [1]))) goto Fehler ;
    if (!(fembin.read ((char *) &femmod_size , sizeof (femmod_size))))
        goto Fehler ;

    cout << "File 'femmod.bin':" << endl
         << "kx = " << femmod_char.kx << endl
         << "kf = " << femmod_char.kf << endl
         << "ke = " << femmod_char.ke << endl
         << "kp = " << femmod_char.kp << endl
         << "ne = " << femmod_size.ne << endl
         << "nk = " << femmod_size.nk << endl ;

    return 0 ;

Fehler:
    cout << "Fehler beim Lesen vom File femmod.bin" << endl ;
    return 1 ;
}

```

12.7 Was man sonst noch wissen sollte

Dieses Tutorial kann und will nicht vollständig sein in der Beschreibung der behandelten Programmiersprachen, schon gar nicht in diesem Kapitel "C++ für C-Programmierer". Nach dem Durcharbeiten der Abschnitte 12.1 bis 12.6 sind aber die wesentlichen Hilfsmittel der objektorientierten Programmierung bekannt, auch wenn einige Aspekte bewußt weggelassen wurden, um die ohnehin schwierige Materie nicht noch komplizierter erscheinen zu lassen. Es ist auch wesentlich wichtiger, die Prinzipien der objektorientierten Programmierung zu verstehen als jede Feinheit der syntaktischen Varianten zu kennen.

Deshalb folgender Rat: Wenn Ihnen die Themen "Vererbung", "Arbeit von Konstruktoren und Destruktoren", "Überladen" und "Polymorphismus" immer noch nur schemenhaft etwas sagen, dann sollten Sie eine Wiederholung der Abschnitte 12.1 bis 12.6 dem Weiterlesen vorziehen. Wenn Sie allerdings meinen, das Zurückliegende einigermaßen verstanden zu haben, dürfen Sie diesen Abschnitt 12.7 "diagonal lesen", dabei registrieren, was es auch noch gibt, und darauf gelegentlich bei Bedarf zurückkommen. Denn eigentlich ist jetzt der Punkt erreicht, an dem man ernsthaft an eigene Programmieraufgaben herangehen kann.

In diesem Abschnitt wird in wahlloser Reihenfolge einiges nachgeliefert, was thematisch auch in die vorigen Abschnitte sehr gut gepaßt hätte. Um den Blick für das Wesentliche nicht zu verstellen, blieben die nachfolgend behandelten Aspekte bisher unerwähnt.

12.7.1 Arbeiten mit friend-Funktionen und friend-Klassen

Die Sicherheit, die durch Kapselung von Daten erreicht wird, kann durchaus zu erheblichem Mehraufwand bei der Programmierung führen. Die konsequente Befolgung der Regel, Daten einer Klasse nur von den zur Klasse gehörenden Methoden manipulieren zu lassen (durch **private**-Deklaration der Daten), kann sehr schnell zu einer Überfrachtung der Klasse mit einer sehr großen Anzahl von Methoden oder zur Anhäufung von sehr vielen Daten in einer Klasse führen, was für eine saubere Strukturierung des Programms nicht förderlich ist.

Nicht selten tritt der Fall ein, daß man bestimmte Daten in einer Klasse A eigentlich nur innerhalb der Klasse A und innerhalb einer einzigen anderen Klasse B (mit deren Methoden) manipulieren muß. In diesem Fall müßte bei konsequenter Kapselung die Klasse A für alle Daten Zugriffsmethoden bereitstellen, die dann natürlich nicht nur von den Methoden der Klasse B, sondern auch von jeder anderen Funktion benutzt werden könnten (was aus Sicherheitsgründen unter Umständen unerwünscht ist).

Es gibt noch eine Reihe anderer Gründe, weshalb mit der Möglichkeit, "Freunde zu deklarieren", die "Privatsphäre" einer Klasse etwas zugänglicher gemacht werden kann.

Eine Klasse kann Funktionen, die nicht zur Klasse gehören, oder andere Klassen (und damit alle Methoden dieser Klassen) zu "Freunden" erklären (mit dem Schlüsselwort **friend**) und ihnen damit direkten Zugriff auf ihre privaten Daten gestatten.

- ◆ Grundsätzlich "sucht sich eine Klasse ihre Freunde selbst aus". Es ist nicht möglich, daß sich eine Funktion oder eine andere Klasse Zugang zu **private**-Daten einer Klasse verschafft, indem sie diese "zum Freund erklärt".
- ◆ Bei verantwortungsbewußtem Umgang mit dem Schlüsselwort **friend** ist keine nennenswerte Verringerung der Sicherheit zu befürchten, im Gegenteil: Wenn man "nur den (selbst sorgfältig auszusuchenden!) Freunden" Zugriff auf sensible private Daten gestattet (und nicht durch die Definition von Methoden auch dem "Rest der Welt"), kann sogar eine Erhöhung der Sicherheit erreicht werden.

Beispiel:

Im Programm **vecalg1.cpp** (Abschnitt 12.5) konnte das Überladen des Multiplikations-Operators für die Operation "Skalar * Vektor" aus dort kommentierten Gründen nicht als Methode der Klasse **vector** realisiert werden. Die Funktion **operator* (double , vector)** muß auf die Daten der Klasse **vector** zugreifen, um die Operation auszuführen (in **vecalg1.cpp** wurde ein Trick benutzt, den direkten Zugriff zu umgehen, indem ausgenutzt wurde, daß die Operation "Vektor * Skalar", für die eine Methode definiert wurde, mit "Skalar * Vektor" mathematisch gleichwertig ist, eine solche Möglichkeit ist natürlich im Regelfall nicht verfügbar).

Es gibt zwei Möglichkeiten, das Zugriffsproblem zu lösen: In der Klasse **vector** können Methoden für den Zugriff auf die privaten Daten definiert werden, oder die Klasse **vector** erklärt die Funktion **operator* (double , vector)** zur "befreundeten Funktion". Dies wird durch die Deklaration in einem beliebigen Bereich der Klasse **vector** in der Form

```
friend vector operator* (double , vector&) ;
```

realisiert. Dann kann (außerhalb der Klasse **vector**) die Definition dieser Funktion z. B. folgendermaßen aussehen:

```
vector operator* (double scal , vector &op2)
{
    return vector (op2.xc * scal , op2.yc * scal , op2.zc * scal) ;
}
```

Man beachte: Diese Funktion greift auf die **private**-Daten der Klasse **vector** wie eine Methode dieser Klasse zu. Der Programmierer der Funktion **operator* (double , vector&)** hätte allerdings keine Chance, an diese Daten heranzukommen, wäre er nicht auch der Programmierer der Klasse **vector** (oder mindestens ein "Freund dieses Programmierers"), so daß er die entscheidende **friend**-Anweisung in die Klasse **vector** hineinbringen kann.

In dem zum Tutorial gehörenden Programm **vecalg2.cpp** (Erweiterung des Programms **vecalg1.cpp**) ist dieses kleine Beispiel realisiert.

12.7.2 Der this-Pointer

Die zu einer Klasse gehörenden Methoden werden von Funktionen außerhalb der Klasse stets mit Angabe des Objekts aufgerufen, auf das sie angewendet werden sollen (die Methode **get_x()** der Klasse **point** wird für ein Objekt, das mit **point p1 ;** vereinbart wurde, in der Form **p1.get_x** aufgerufen). Es gibt Situationen, in denen eine Methode den Pointer des Objektes benötigt, mit dem sie aufgerufen wurde (dessen Daten sie gerade bearbeitet).

Das Schlüsselwort **this** repräsentiert diesen Pointer, mit der Konstruktion

```
this->xc
```

könnte also eine Methode auf die zur Klasse gehörende Komponente **xc** zugreifen, was sinnvollerweise so nicht codiert wird, weil die Methode auf **xc** natürlich direkt zugreifen kann.

Sinnvoll ist die Verwendung des **this**-Pointers vor allen Dingen dann, wenn das aktuelle Objekt als Ganzes angesprochen werden muß, wenn also z. B. der Return-Wert der Methode ein Pointer auf das aktuelle Objekt oder das Objekt selbst ist. Dies ist häufig der Fall, wenn die Daten eines Objekts von einer Methode verändert werden und das geänderte Objekt sofort weiterverarbeitet werden soll.

Beispiel:

Im Programm **vecalg2.cpp** ist auch ein kleines Beispiel für die Verwendung des **this**-Zeigers zu finden. In Erweiterung des Programms **vecalg1.cpp**, in dem der "Additions-Operator +" überladen wurde, wird in **vecalg2.cpp** auch noch der "unäre Operator +" überladen. Mit dem Programm **vecalg1.cpp** sind Vektor-Additionen **a+b** möglich (**a** und **b** sind Objekte der Klasse **vector**), eine Anweisung wie **+a+b** würde dagegen einen Compiler-Fehler erzeugen, weil der unäre Operator + (das "Vorzeichen") nicht für Objekte der Klasse **vector** definiert ist (eigentlich braucht man den unären Operator + im Gegensatz zum unären Operator - nicht, aber für die Standard-Typen ist er auch erlaubt und dürfte bei einer "ordentlich eingerichteten" Klasse **vector** natürlich nicht fehlen).

Der unäre Operator + muß gar keine Operationen ausführen, sondern nur das Objekt, auf das er angewendet wird, ungeändert als Return-Wert abliefern, am einfachsten also zu realisieren durch:

```
vector vector::operator+ () { return *this ; }
```

(weil **this** der Pointer auf das aktuelle Objekt ist, wird mit ***this** das Objekt selbst angesprochen).

12.7.3 Mehrfach-Vererbung

Die wichtigste Aussage über die im Abschnitt 12.3 behandelte Ableitung einer Klasse aus einer Basisklasse war, daß die **Basisklasse alle Daten und Methoden an die abgeleitete Klasse vererbt**. Es wurde gezeigt, daß dies mehrstufig möglich ist: Eine abgeleitete Klasse kann selbst Basisklasse einer weiteren abgeleiteten Klasse sein. Jede abgeleitete Klasse besitzt mindestens die Daten und Methoden ihrer Basisklasse, so daß entlang einer Erbfolge die Klassen immer mächtiger werden. Namenskollisionen (sowohl für Daten als auch für Methoden) stellen kein Problem dar, gegebenenfalls muß mit dem Klassennamen und dem Operator **::** Eindeutigkeit hergestellt werden.

Das Konzept der Mehrfach-Vererbung gestattet die Ableitung einer Klasse aus mehreren Basisklassen. Es war in den ersten Definitionen der Sprache C⁺⁺ nicht vorgesehen und wird von älteren Compiler-Versionen möglicherweise nicht unterstützt. Die Mehrfach-Vererbung ist jedoch ein sehr wichtiges Konzept bei der Erfüllung des Anspruchs der objektorientierten Programmierung, die reale Welt im Programm adäquat nachbilden zu können.

Die Syntax der Mehrfach-Vererbung ist nicht komplizierter als die Syntax der einfachen Vererbung. Die Basisklassen werden bei der Deklaration der abgeleiteten Klasse, durch Komma voneinander getrennt, aufgelistet, Beispiel:

```
class Basis_Klasse_1
{ // ...
} ;
class Basis_Klasse_2
{ // ...
} ;
class Abgeleitete_Klasse : public Basis_Klasse_1 , public Basis_Klasse_2
{ // ...
} ;
```

... leitet eine Klasse aus zwei Basisklassen ab, wobei sämtliche Daten und Methoden beider Basisklassen in die abgeleitete Klasse einfließen. Eventuelle Namenskonflikte sind problemlos zu handhaben, indem mit dem Zuordnungsoperator `::` Eindeutigkeit hergestellt wird (der Compiler achtet darauf, daß keine Zweifel bestehen, welche Daten oder Methoden verwendet werden sollen).

Beim Aufruf der Konstruktoren bleibt die für die einfache Vererbung geltende Regel erhalten, daß zuerst die Konstruktoren der Basisklassen aufgerufen werden, danach der Konstruktor der abgeleiteten Klasse. Die Reihenfolge der Konstruktor-Aufrufe für die Basisklassen wird vom Programmierer der abgeleiteten Klasse bestimmt, entweder durch die Reihenfolge der Basisklassen bei der Deklaration der abgeleiteten Klasse (siehe oben) oder aber durch die Reihenfolge der Konstruktor-Aufrufe bei der Definition des Konstruktors der abgeleiteten Klasse. Diese könnte für das gewählte Beispiel folgendermaßen aussehen:

```
// Definition des Konstruktors fuer Abgeleitete_Klasse:
Abgeleitete_Klasse::Abgeleitete_Klasse (int a , int b , double c) :
    Basis_Klasse_2 (a) , // Aufruf des Konstruktors von Basis_Klasse_1
    Basis_Klasse_1 (c) // Aufruf des Konstruktors von Basis_Klasse_2
{ // ...
}
```

... würde festlegen, daß beim Erzeugen einer Instanz von **Abgeleitete_Klasse** zuerst der Konstruktor von **Basis_Klasse_2** (mit dem Argument **a**) aufgerufen wird, danach der Konstruktor von **Basis_Klasse_1** (mit dem Argument **c**) und schließlich der Konstruktor von **Abgeleitete_Klasse**.

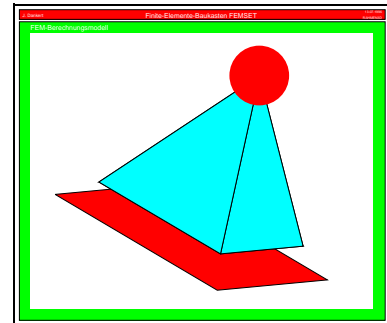
Es gibt bei aller Einfachheit der Syntax (und der Chance, in der Kombination von "Erbfolgen" und Mehrfach-Vererbung ausgesprochen komplexe Gebilde realitätsnah im Programm als Objekte abzubilden) auch einige Besonderheiten, die die erhöhte Aufmerksamkeit des Programmierers erfordern:

- ◆ **Virtuelle Methoden** (vgl. Abschnitt 12.4) von Basisklassen (nicht zu verwechseln mit den nachfolgend noch zu besprechenden "virtuellen Basisklassen") werden für Objekte der abgeleiteten Klasse nur dann verwendet, wenn in dieser nicht eine Methode gleichen Namens definiert wurde. Wenn bei Mehrfach-Vererbung virtuelle Methoden gleichen Namens schon in den Basisklassen existieren, kann dieser Konflikt nur gelöst werden, wenn in der abgeleiteten Klasse auch eine Methode mit diesem Namen definiert wird (in diesem Fall besteht also ein Zwang zur Definition einer eigenen Methode wie bei einer Ableitung aus einer abstrakten Klasse).

- ◆ Eine nicht einmal seltene Konflikt-Situation entsteht, wenn die Basisklassen einen gemeinsamen "Urahn" besitzen. In diesem Fall sind die Daten und Methoden auf verschiedenen "Erbswegen" tatsächlich mehrfach in den Objekten der abgeleiteten Klasse präsent, was durchaus erwünscht oder aber auch unerwünscht sein kann. Da dieses Problem eng mit der Frage nach dem Entwurf einer geeigneten Klassen-Hierarchie für eine Programmier-Aufgabe zusammenhängt, wird nachfolgend dazu ein Beispiel etwas ausführlicher behandelt.

Beispiel:

Die nebenstehende Abbildung zeigt ein Symbol für ein sogenanntes räumliches "Loslager". Es ist in dem CAMMPUS-FEM-Programm (vgl. <http://www.fh-hamburg.de/rzbt/dnksoft/cammpus>) "Räumliche Rahmentragwerke" als Klasse **Loslager_3D** realisiert, die sich aus den drei Basisklassen **Rechteck_3D**, **Pyramide** und **Kugel** ableitet, wobei (Mehrfach-Vererbung) sämtliche Daten und Methoden der drei Basisklassen in die Klasse **Loslager_3D** einfließen.



Die Basisklassen leiten sich aber auch aus einer ganzen Reihe anderer Klassen ab, unter anderem haben alle drei Basisklassen die Klasse **obj_cols** in ihrer "Ahnenreihe":

Objekt der Klasse **Loslager_3D**, die sich aus 3 Basisklassen ableitet

```
class obj_cols
{
private:
    int obj_col ;    // Farbe des geometrischen Objekts
    int line_col ;  // Farbe fuer Raender und Kanten
    int hl_col  ;   // "Highlight"-Farbe
    // ...

public:
    obj_cols (int oc = 7 , int lc = 0 , int hc = 4) ;
    int get_line_col () { return line_col ; }
    // ...
} ;

obj_cols::obj_cols (int oc , int lc , int hc)    // Konstruktor
{
    obj_col = oc ;
    line_col = lc ;
    hl_col = hc ;
}
```

... ist ein Auszug aus der Deklaration der Klasse **obj_cols**, in der alle Daten und Methoden zusammengefaßt sind, die für die Farbgebung bei der Darstellung eines geometrischen Objekts erforderlich sind. Es werden jeweils nur die für das Verständnis der behandelten Problematik wesentlichen Anweisungen wiedergegeben, in der Deklaration der Klassen **Rechteck_3D** und **Pyramide** findet man also als Basisklasse jeweils **obj_cols**:

```
class Rechteck_3D : public obj_cols    // ...
{
public:
    Rechteck_3D (int oc = 0 , int lc = 0 , int hc = 0) ; // Konstruktor
    // ...
} ;

Rechteck_3D::Rechteck_3D (int oc , int lc , int hc) : obj_cols (oc , lc , hc)
{ // ...
}
```

```

class Pyramide : public obj_cols // ...
{
    public:
        Pyramide (int oc = 0 , int lc = 0 , int hc = 0) ; // Konstruktor
        // ...
} ;
Pyramide::Pyramide (int oc , int lc , int hc) : obj_cols (oc , lc , hc)
{ // ...
}

```

Auf entsprechende Weise steckt die Klasse **obj_cols** auch in der Klasse **Kugel** (wie Sie bemerken, wird innerhalb des CAMMPUS-Programms konsequent "Denglisch" gesprochen, bei der Realisierung größerer Software-Projekte ist man froh, für das permanente Problem der Namensfindung mindestens zwei Sprachen verfügbar zu haben).

Der Programmierer, der die Klasse **Loslager_3D** deklariert, steht nun vor der Entscheidung, ob alle Elemente des Loslagers mit dem gleichen Satz von Farben gezeichnet werden sollen, oder ob Rechteck, Pyramide und Kugel unterschiedliche Farben haben können. Bei der Deklaration der Klasse

```

class Loslager_3D : public Rechteck_3D , public Pyramide , public Kugel
{ // ...
} ;

```

... bringt jede der drei Basisklassen ohnehin einen eigenen kompletten Satz von Daten der Klasse **obj_cols** ein, so daß es naheliegend ist, die Möglichkeit der unterschiedlichen Farbgebung zu nutzen. Dann könnte man bei der Definition des Konstruktors von **Loslager_3D** veranlassen, daß die Konstruktoren der Basisklassen gleich die komplette Farb-Definition ausführen:

```

Loslager_3D::Loslager_3D (int coltab []) : //.. erwartet "Farben-Tabelle"
    Rechteck_3D (coltab[0] , coltab[1] , coltab[2]) ,
    Pyramide (coltab[3] , coltab[4] , coltab[5]) ,
    Kugel (coltab[6] , coltab[7] , coltab[8])
{ // ...
} ;

```

Beim Erzeugen einer Instanz der Klasse **Loslager_3D** kann man also die gesamte Farbgebung für alle Teile des Lagersymbols individuell festlegen. Zu beachten ist, daß die Methoden der Klasse **obj_cols** auf eindeutige Weise angesprochen werden. Ein Aufruf für ein Objekt der Klasse **Loslager_3D** mit dem Namen **lager1** in der Form

```

lager1.get_line_col ()

```

würde vom Compiler beanstandet werden, weil nicht zu ersehen ist, ob die Farbe für das Rechteck, die Pyramide oder die Kugel angefordert ist. Hier hilft der Zuordnungsoperator, die nachfolgend gelistete Funktion **main** zeigt das korrekte Erzeugen einer Instanz der Klasse **Loslager_3D** und eindeutige Aufrufe der Methode **get_line_col**:

```

main ()
{
    int coltab [] = { 3 , 1 , 4 , 2 , 6 , 4 , 0 , 5 , 4 } ;
    Loslager_3D lager1 (coltab) ;
    cout << "Parameter line_col fuer" << endl
        << "Rechteck: " << lager1.Rechteck_3D::get_line_col ()
        << ", Pyramide: " << lager1.Pyramide::get_line_col ()
        << ", Kugel: " << lager1.Kugel::get_line_col () << endl ;
    return 0 ;
}

```

Alle hier aufgeführten Beispiel-Anweisungen sind zum Programm **mulinh1.cpp** zusammengefaßt worden. Dieses liefert das erwartete Ergebnis:

```
Parameter line_col fuer
Rechteck: 1,   Pyramide: 6,   Kugel: 5
```

Wenn der Programmierer entscheiden würde, daß generell alle Teile eines Lagersymbols mit dem gleichen Farbensatz dargestellt werden sollen, stehen ihm z. B. folgende Möglichkeiten zur Verfügung:

- ◆ Es wird nur die Farbtabelle stets so eingerichtet, daß für alle Teile des Lagersymbols die gleichen Farben eingetragen sind. Dem Vorteil, daß man es sich jederzeit bei minimalem Änderungsaufwand wieder anders überlegen kann, stehen folgende Nachteile gegenüber: "Speicherverschwendung" und der "Zwang zur Festlegung einer Zuordnung zu einer Basisklasse" beim Aufruf einer Methode aus **obj_cols**, obwohl dies jeweils zum gleichen Ergebnis führt.
- ◆ Naheliegender ist die Variante, die Basisklassen **Rechteck_3D**, **Pyramide** und **Kugel** nicht aus der Klasse **obj_cols** abzuleiten, dafür aber **Loslager_3D** direkt zusätzlich aus **obj_cols** abzuleiten. Diese in sich sehr logische Lösung hat den erheblichen Nachteil, daß Objekte, die direkt aus einer der drei Basisklassen erzeugt werden (z. B. eine Pyramide, die nicht zum Loslager gehört), dann "farblos" sind.
- ◆ Die in den meisten Fällen beste Lösung erreicht man mit **virtuellen Basisklassen**. Diese werden im folgenden Abschnitt vorgestellt.

12.7.4 Virtuelle Basisklassen

Wenn bei der Ableitung einer Klasse aus einer Basisklasse dieser das Schlüsselwort **virtual** vorangestellt wird, sorgt der Compiler dafür, daß die Daten der nun **virtuellen Basisklasse** nur genau einmal in abgeleiteten Klassen vorhanden sind. Dies soll an dem im vorigen Abschnitt behandelten Beispiel demonstriert werden:

```
class Pyramide : virtual public obj_cols // ...
{
    public:
        Pyramide (int oc = 0 , int lc = 0 , int hc = 0) ; // Konstruktor
        // ...
} ;
```

... mit der als **virtual** deklarierten Basisklasse **obj_cols** hat für das Definieren von Objekten vom Typ **Pyramide** keine Auswirkungen (alle Daten der Basisklasse gehören auch zur abgeleiteten Klasse und können über deren Methoden manipuliert werden). Denkbar wäre allerdings, daß bei einer zusätzlichen Ableitung von **Pyramide** aus weiteren Basisklassen (z. B. könnte **Pyramide** sinnvoll zusätzlich aus **Rechteck_3D** abgeleitet werden, weil die Grundfläche der Pyramide ein Rechteck ist) auf diesem Wege **obj_cols** noch einmal in der Ahnenreihe steht. Dann wird mit dem Schlüsselwort **virtual** dem Compiler das Signal gegeben, die Daten von **obj_cols** nur einmal in die abgeleitete Klasse einzufügen.

Bei dem im vorigen Abschnitt behandelten Beispiel müßten die Basisklassen **Rechteck_3D** und **Kugel** bei ihrer Ableitung aus **obj_cols** natürlich auch das Schlüsselwort **virtual**

einfügen. Dann ist gesichert, daß bei der Ableitung von **Loslager_3D** aus den drei Basis-Klassen **Rechteck_3D**, **Pyramide** und **Kugel** die abgeleitete Klasse nur einmal die Daten von **obj_cols** enthält. Im Programm **mulinh2.cpp**, das genau diese Modifikationen gegenüber dem Programm **mulinh1.cpp** enthält, wird die Klasse **Loslager_3D** folgendermaßen deklariert:

```
class Loslager_3D : public Rechteck_3D , public Pyramide , public Kugel
{ // ...
  public:
    Loslager_3D (int oc , int lc , int hc) ;
} ;
Loslager_3D::Loslager_3D (int oc , int lc , int hc) :
    obj_cols (oc , lc , hc)
{ // ...
} ;
```

Man beachte den veränderten Konstruktor: Weil nur noch ein Farbensatz verwendet wird, wurde auf die Farben-Tabelle verzichtet, und es wird direkt der Konstruktor von **obj_cols** aktiviert, dem die nun für das gesamte Loslager geltenden Farben übergeben werden. Außerdem ist folgendes ist zu beachten:

- ◆ Auch für die Basisklassen erfolgt ein automatischer (parameterloser) Konstruktor-Aufruf, wenn (wie im Beispiel) die Konstruktoren der Basisklassen beim Konstruktor der abgeleiteten Klasse nicht aufgeführt sind. Dies wird vom Compiler in diesem Fall nur akzeptiert, weil in den Konstruktoren der Basisklassen für alle Argumente Default-Werte vorgesehen sind.
- ◆ Auch für die virtuellen Basisklassen erfolgen Konstruktor-Aufrufe. Diese werden vor den Konstruktor-Aufrufen der anderen Klassen abgearbeitet.
- ◆ Da die Daten der Klasse **obj_cols** nur einmal in **Loslager_3D** existieren, vereinfachen sich die Aufrufe der Methoden dieser Klasse wieder auf "Objektname, Punkt, Methodennamen". Die nachfolgend gelistete Funktion **main** des Programms **mulinh2.cpp** zeigt dies:

```
main ()
{
  Loslager_3D lager1 (2 , 3 , 4) ;
  cout << "Parameter line_col: " << lager1.get_line_col () << endl ;
  return 0 ;
}
```

12.7.5 Das Schlüsselwort **const**

Konstanten (unveränderliche Werte, die mit einem Namen angesprochen werden) können in der Programmiersprache C (und damit auch in C++) mit der Präprozessor-Anweisung **#define** (vgl. Abschnitt 3.5) oder mit dem Schlüsselwort **const** definiert werden. Die Definition mit **const** ist zu bevorzugen, weil dem Compiler die Chance zur Typ-Überprüfung gegeben wird (mit **#define** erzeugte Konstanten bekommt der Compiler gar nicht zu sehen). In C++ wurde deshalb mit dem Ziel, **#define** überflüssig zu machen, das Verhalten von **const** etwas abweichend von der C-Definition dieses Schlüsselwortes festgelegt (und für die Möglichkeit, Makros mit **#define** zu definieren - beschrieben am Ende des Abschnitts 8.4.1 - wurden in C++ als Ersatz die **inline**-Funktionen kreiert, die im folgenden Abschnitt beschrieben werden).

Beim Einsatz des Schlüsselwortes **const** in C++ ist folgendes zu beachten:

- ◆ Eine Konstante wird mit der Syntax "const Typ Name = Wert ;" definiert, z. B.:

```
const double xyz = 3.45 ;
```

Die Typ-Bezeichnung darf fehlen, dann wird die Konstante als **int**-Wert interpretiert. Auch Instanzen von Klassen dürfen mit **const** definiert werden.

- ◆ Weil in C++ **const**-Anweisungen in Header-Dateien die **#define**-Anweisungen ersetzen sollen, ist die Gültigkeit der mit **const** erzeugten Werte auf die Datei beschränkt, in der sie (außerhalb von Funktionen) erscheint (dies ist unterschiedlich zu C).
- ◆ Mit **const** definierte Konstanten dürfen in C++ (nicht in C) bei der Definition von Arrays zur Festlegung der Anzahl der Elemente verwendet werden, z. B.:

```
const n_elem = 200 ;
double x [n_elem] ;
int k [n_elem] ;
```

- ◆ Auch Methoden können mit dem Schlüsselwort **const** versehen werden (bei Deklaration und dann unbedingt auch bei der Definition). Das Schlüsselwort muß zwischen der schließenden Klammer der Funktionsparameter und dem Semikolon (bei Deklarationen) bzw. der öffnenden geschweiften Klammer (bei der Definition) stehen, z. B.:

```
class obj_cols
{ // ...
  public:
    int get_line_col () const ;
  // ...
} ;
int obj_cols::get_line_col () const
{ // ...
}
```

Eine mit dem Schlüsselwort **const** definierte Methode kann keine Daten der Klasse ändern (und auch keine anderen Methoden aufrufen, die dazu in der Lage wären, weil sie nicht mit **const** definiert wurden).

Für Objekte einer Klasse, die mit **const** erzeugt wurden (konstante Instanzen der Klasse) dürfen nur Methoden aufgerufen werden, die als **const** gekennzeichnet sind.

12.7.6 inline-Funktionen

Bei Funktionen, die nur sehr wenige Operationen ausführen, ist häufig der "Overhead" für die Argument-Übergabe und Übernahme des Return-Wertes aufwendiger als die eigentliche Arbeit der Funktion. Für solche Funktionen bietet sich die **inline**-Definition an, bei der vom Compiler an jeder Stelle, wo die Funktion aufgerufen wird, der komplette Funktionscode eingesetzt wird. Realisiert wird dies durch einfaches Voranstellen des Schlüsselwortes **inline** bei der Funktions-Definition, z. B.:

```
inline double max (double x , double y) { return x > y ? x : y ; }
```

Im Vergleich mit der Makro-Definition mit einer **#define**-Anweisung entsprechend

```
#define max (x , y) ((x) > (y) ? (x) : (y))
```

wird vor allen Dingen deutlich, daß der Compiler bei **inline**-Funktionen eine Typ-Überprüfung vornehmen kann, andererseits ist das Makro für unterschiedliche Datentypen verwendbar (man kann natürlich auch die **inline**-Funktionen überladen und damit für verschiedene Datentypen bereitstellen).

12.7.7 static-Variablen in Klassen-Deklarationen

Wenn eine Variable in einer Klassen-Deklaration mit dem Zusatz **static** definiert wird, hat dies eine andere Bedeutung im Vergleich mit **static**-Definitionen von Variablen in Funktionen. Während letztere die Eigenschaft haben, ihre Gültigkeit bei Verlassen der Funktion nicht zu verlieren (und damit ihren Wert bei einem Wiederaufruf der Funktion immer noch zu besitzen), gilt für

static-Variablen in Klassen-Deklarationen:

Wenn eine Variable **x** innerhalb einer Klassen-Deklaration mit dem Zusatz **static** deklariert wird, existiert für diese bei beliebiger Anzahl von Instanzen, die von dieser Klasse erzeugt werden, nur eine einzige Variable **x**, die von allen Instanzen manipuliert werden kann.

Die **static**-Variablen einer Klasse gehören also nicht zu einem Objekt. Daraus ergibt sich für ihre Initialisierung eine Besonderheit, die das folgende Beispiel-Programm zeigt. **Außerhalb der Klassendeklaration** wird die einzige Instanz dieser Variablen (vor dem Erzeugen der ersten Instanz der Klasse) definiert und initialisiert. Das Programm **static.cpp** nutzt eine **static**-Variable der Klasse **stat_val**, um die aktuelle Anzahl der Klassen-Instanzen zu zählen:

```
// Statische Variable in einer Klasse (Programm static.cpp)
// =====
#include <iostream.h>

class stat_val
{
private:
    static int obj_num ;

public :
    stat_val () { obj_num++ ; }
    ~stat_val () { obj_num-- ; }
    void print ()
    {
        cout << "Anzahl der stat_val-Instanzen: " << obj_num << endl ;
    }
} ;

int stat_val::obj_num = 0 ; // ... initialisiert static-Klassen-Variable
```

```
main ()
{
    stat_val a , b , c ;      // ... erzeugt 3 Objekte
    a.print () ;
    {
        stat_val d , e ;      // ... erzeugt 2 weitere Objekte
        b.print () ;
    }                          // Objekte d und e verlieren ihre Gueltigkeit
    c.print () ;
    return 0 ;
}
```

Das Programm produziert folgende Ausgabe:

```
Anzahl der stat_val-Instanzen: 3
Anzahl der stat_val-Instanzen: 5
Anzahl der stat_val-Instanzen: 3
```