Intensivkurs C++

Prof. Dr. Karl Friedrich Gebhardt

Auflage vom 30. September 2013

Prof. Dr. K. F. Gebhardt
Duale Hochschule Baden-Württemberg Stuttgart
Angewandte Informatik

Tel: 0711-667345-11(16)(15)(12)
Fax: 0711-667345-10
email: kfg@dhbw@stuttgart.de

Vorwort

Das vorliegende Skriptum wurde von Herrn Tobias Elpelt überarbeitet, wofür ich ihm an dieser Stelle ganz herzlich danke!

Das Skriptum ist die Arbeitsunterlage für einen viertägigen C++ Intensivkurs. Es eignet sich daher nur bedingt zum Selbststudium.

Sprachkenntnisse einer höheren Programmiersprache (z.B. C, Pascal, FORTRAN, BASIC) sind erfahrungsgemäß eine Voraussetzung, um den Stoff in vier Tagen zu erarbeiten.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel wird an Hand eines kurzen Beispiels gezeigt, wie ein C++ Programm übersetzt und zum Laufen gebracht wird. In einem zweiten Abschnitt geben wir einige Informationen über den Entwicklungsstand von C++. Schließlich werden die wesentlichen Züge objekt-orientierter Programmierung zusammengefaßt.

1.1 Hello-World-Programm

Schreiben in einen File mit Namen

```
hello.cpp
```

folgenden Code:

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

main ()
   {
   cout << "Guten Tag!\n";
   }
```

Unter Linux übersetzen Sie diesen Code mit

```
$ g++ hello.cpp
```

und lassen das Programm laufen mit:

```
$ ./a.out
```

Das Resultat sollte die Ausgabe

```
Guten Tag!
```

sein.

Unter Visual Studio.NET Command Prompt übersetzen Sie diesen Code mit

```
> cl hello.cpp
```

und lassen das Programm laufen mit:

```
> hello
```

Das Resultat sollte ebenfalls die Ausgabe

```
Guten Tag!
```

sein.

1.2 Erstes C++ Programm

Um ganz schnell einzusteigen, wollen wir ohne allzu ausführlich auf syntaktische Details einzugehen zunächst ein C++ Programm zur Berechnung des Produkts von ganzen Zahlen betrachten.

```
// produkt.cpp − Produkt von ganzen Zahlen
#include <iostream>
using namespace std;

main ()
   {
   cout << "Eingabe erste Zahl: ";
   int erste_Zahl;
   cin >> erste_Zahl;
   cout << "Eingabe zweite Zahl: ";
   int zweite_Zahl;
   cin >> zweite_Zahl;
   cout << "Das Produkt von " << erste_Zahl;
   cout << " mit " << zweite_Zahl;
   cout << " beträgt " << erste_Zahl * zweite_Zahl << " .\n";
   return 0;
   }
```

Die erste Zeile ist ein Kommentar. Die Zeichen // lassen einen Kommentar beginnen, der am Ende der Zeile aufhört. Eine zweite Möglichkeit zu kommentieren besteht in der Verwendung von C-Kommentaren zwischen den Zeichenfolgen /* und */.

Die Zeile **#include** `<iostream>` ist für den Compiler die Anweisung, die Deklarationen der Standard-Ein- und -Ausgabe einzufügen. Mit diesen Deklarationen

werden die Ausdrücke `cout ...` und `cin ...` für den Compiler verständlich. Denn in `<iostream>` sind `cout` als Standard-Ausgabe-Stream und `cin` als Standard-Eingabe-Stream definiert. Die Klammern `<>` sagen dem Präprozessor, daß er den File in einem vordefinierten, ihm bekannten Verzeichnis suchen soll.

`cout` und `cin` sind allerdings in dem **Namensraum (namespace)** `std` definiert. Daher müssen wir noch die Zeile

```
using namespace std;
```

einfügen. Stattdessen hätten wir auch den Namensraum explizit angeben können:

```
std::cout
std::cin
```

Der Operator `<<` ist überladen. Er wirkt hier so, daß er seinen zweiten Operanden, den String `"Eingabe erste Zahl: "` auf den ersten Operanden, in diesem Fall auf den Standard-Ausgabe-Stream `cout` schreibt. Ein String ist eine Folge von Zeichen zwischen doppelten Anführungsstrichen. In einem String bedeutet das Backslash-Zeichen `\` gefolgt von einem anderen Zeichen ein einzelnes spezielles Zeichen; `\n` ist das newline-Zeichen.

Jedes C++ Programm muß eine Funktion mit Namen `main` haben, mit der die Ausführung des Programms gestartet wird. Diese Funktion gibt wie jede Funktion defaultmäßig einen **int**-Wert zurück. Daher muß am Ende des Programms eine ganze Zahl zurückgegeben werden. Üblicherweise gibt man `0` zurück, wenn das Programm korrekt gelaufen ist.

Zeile 7 (Leerzeilen mitgerechnet) des Programms definiert eine Integervariable, deren Wert in der nächsten Zeile durch den Operator `>>` vom Standard-Input-Stream `cin` eingelesen wird. Jede Variable muß vor Verwendung deklariert werden. Die Deklaration kann an beliebiger Stelle erfolgen.

Die Richtung der Operatoren `>>` bzw `<<` gibt die Richtung des Datenstroms an. Die Zeilen 12, 13 und 14 zeigen, daß der Output-Operator `<<` auf sein Ergebnis angewendet werden kann. Auf diese Weise können mehrere Output-Operationen in eine Zeile geschrieben werden. Das ist möglich, weil das Ergebnis der Verknüpfung `<<` wieder ein Standard-Ausgabe-Stream ist und das Ergebnis der Verknüpfung `>>` ein Standard-Eingabe-Stream ist.

Das Resultat der Ein- bzw Ausgabeoperation ist wieder der Ein- bzw Ausgabestrom. Als “Seiteneffekt“ werden die Ströme insofern durch die Operation verändert, als Zeichen entnommen bzw aufgenommen werden.

Das Programm wird mit folgenden Kommandos übersetzt und gestartet:

Unter Linux (`$` sei der System-Prompt):

```
$ g++ produkt.cpp
$ ./a.out
Eingabe erste Zahl: 4
Eingabe zweite Zahl: 6
Das Produkt von 4 mit 6 beträgt 24 .
$
```

`a.out` ist der Default-Name für das ausführbare Ergebnis einer Übersetzung. Durch die Option `-o` kann ein Benutzername für das ausführbare Programm angegeben werden:

```
$ g++ produkt.cpp -o produkt
$ ./produkt
Eingabe erste Zahl: 5
Eingabe zweite Zahl: 3
Das Produkt von 5 mit 3 beträgt 15 .
$
```

Unter Visual Studio.NET Command Prompt (> sei der System-Prompt):

```
> cl produkt.cpp
> produkt
Eingabe erste Zahl: 4
Eingabe zweite Zahl: 6
Das Produkt von 4 mit 6 beträgt 24 .
>
```

`produkt.exe` ist der Default-Name für das ausführbare Ergebnis einer Übersetzung.

1.3 Entwicklungsstand von C++

C++ ist eine Programmiersprache mit sehr weitem Anwendungsbereich. Dieser reicht von der Programmierung eingebetteter Systeme bis zur Programmierung großer, vernetzter Systeme.

C++ wurde in den frühen achtziger Jahren von Bjarne Stroustrup bei AT&T entwickelt.

1985 kam C++ AT&T Release 1.0 heraus. Release 1.2 (1986) war die erste brauchbare Version. Release 2.x (1989) brachte Mehrfachvererbung. Templates und Exceptions kamen dann mit Release 3.0 1991.

Gewöhnlich unterscheidet man drei Anwendungsstufen von C++:

- Verwendung als **typensicheres** C (Es wird nicht mehr “alles“ übersetzt.)
- **objekt-basiertes** C++ (Klassen, Datenkapselung, Operatorenüberladung)

- **objekt-orientiertes** C++ (Vererbung)

1998 wurde C++ als Standard (ISO/IEC 14882:1998) vom Standardisierungskomitee (C++ Standard Comittee) genormt. Bis 2003 wurde die Norm überarbeitet und als Standard (ISO/IEC 14882:2003) veröffentlicht. Im August 2011 wurde der aktuelle Standard, unter dem Namen C++11 (ISO/IEC 14882:2011), verabschiedet.

1.4 Objektorientierte Programmierung

Das Schlagwort "objektorientiert..." beinhaltet üblicherweise vier Aspekte:

- Identität von Objekten
- Klassifizierung
- Polymorphismus
- Vererbung

Unter **Identität** versteht man, daß Daten als diskrete, unterscheidbare Einheiten, sogenannte Objekte gesehen werden. Ein Objekt ist z.B. ein spezieller Abschnitt in einem Text, eine spezielle Wiedergabe eines Musikstücks, das Sparkonto Nr.2451, ein Kreis in einer Zeichnung, ein Eingabekanal, ein Sensor, ein Aktor. Objekte können konkreter oder konzeptioneller Natur sein (z.B. Herstellungsanleitung, Rezeptur, Verfahrensweise). Jedes Objekt hat seine Identität. Selbst wenn die Werte aller Attribute von zwei Objekten gleich sind, können sie doch verschiedene Objekte sein. In der Datenverarbeitung werden solche Objekte häufig dadurch unterschieden, daß sie verschiedene Speicherplätze belegen.

Ein besonderes Problem für die Datenverarbeitung ist die Tatsache, daß reale Objekte i.a. eine Lebensdauer haben, die über die Programmlaufzeit hinausgeht (**Persistenz** von Objekten).

Der Zugang zur realen Welt geschieht beim objektorientierten Ansatz über Objekte, nicht über Funktionen. Für Objekte wird ein *Verhalten nach außen (Interface, Methoden)* und eine *interne Repräsentation (Datenstruktur)* durch Datenelemente definiert. Das Verhalten von Objekten ist im Laufe der Zeit sehr konstant oder kann sehr konstant gehalten werden. Wenn sich das Verhalten ändert, dann äußert sich das meistens durch eine schlichte Erweiterung des Interfaces um weitere Methoden. Alte Methoden können oft erhalten bleiben.

Eine Verhaltensänderung hat manchmal zur Folge, daß die interne Repräsentation (Datenstruktur) von Objekten geändert werden muß. Da aber das bisherige Verhalten fast immer auch mit der neuen Datenstruktur emuliert werden kann, ist die Datenstruktur von Objekten weniger konstant als ihr Verhalten. Das Verhalten von Objekten hat einen allgemeinen Charakter unabhängig von speziellen Datenverarbeitungsaufgaben.

Funktionen sind Lösungen konkreter Automatisierungsaufgaben. Diese verändern sich laufend, oder es kommen neue Aufgaben dazu. Beruhen Funktionen direkt auf der Datenstruktur, dann müssen alle Funktionen geändert werden, wenn es notwendig wird, die Datenstruktur zu ändern. Beruhen Funktionen aber auf dem Verhalten, muß an den Funktionen nichts geändert werden.

Klassifizierung bedeutet, daß Objekte mit denselben Attributen (Datenstruktur) und demselben Verhalten (Operationen, Methoden) als zu einer Klasse gehörig betrachtet werden. Die Klasse ist eine Abstraktion des Objekts, die die für die gerade vorliegende Anwendung wichtigen Eigenschaften des Objekts beschreibt und den Rest ignoriert. Die Wahl von Klassen ist letztlich willkürlich und hängt von der Anwendung ab. Jede Klasse beschreibt eine möglicherweise unendliche Menge von Objekten, wobei jedes Objekt eine **Instanz** seiner Klasse ist. Attribute und Verhalten werden in *einer* Klasse zusammen verwaltet, was die Wartung von Software wesentlich erleichtert.

Jede Methode, die für eine Klasse geschrieben wird, steht überall dort zur Verfügung, wo ein Objekt der Klasse verwendet wird. Hierin liegt die Ursache für den Gewinn bei der Software-Entwicklung. Denn der Entwickler wird dadurch gezwungen, die Methoden allgemein anwendbar und "wasserdicht" zu schreiben, da er damit rechnen muß, daß die Methode auch an ganz anderen Stellen angewendet wird, als wofür sie zunächst entworfen wurde. Der Schwerpunkt liegt auf dem Probelm**bereich** (***problem domain***) und nicht auf dem gerade zu lösenden (Einzel-)Problem.

Abstrakte Datentypen ***(data abstraction)***, **Datenkapselung** ***(information hiding)*** bedeutet, daß kein direkter Zugriff auf die Daten möglich ist. Die Daten sind nur über Zugriffsfunktionen zugänglich. Damit ist eine nachträgliche Änderung der Datenstruktur relativ leicht möglich durch Änderung der Software nur in einer lokalen Umgebung, nämlich der Klasse. Die Zugriffsfunktionen sollten so sorgfältig definiert werden, daß sie ein wohldefiniertes und beständiges Interface für den Anwender einer Klasse bilden.

Polymorphismus bedeutet, daß dieselbe Operation unterschiedliche Auswirkung bei verschiedenen Klassen hat. Die Operation "lese" könnte bei einer Klasse `Textabschnitt` bedeuten, daß ein Textabschnitt aus einer Datenbank geholt wird. Bei einer Klasse `Sparkonto` wird durch "lese" der aktuelle Kontostand ausgegeben. Bei einer Klasse `Sensor` bedeutet "lese", daß der Wert des Sensors angezeigt wird, bei einer Klasse `Aktor`, daß ein Stellwert eingegeben werden soll.

Eine spezifische Implementation einer Operation heißt **Methode**. Eine Operation ist eine Abstraktion von analogem Verhalten verschiedener Arten von Objekten. Jedes Objekt "weiß", wie es seine Operation auszuführen hat. Der Anwender von solchen Objekten muß sich nicht darum kümmern.

Vererbung ist die gemeinsame Nutzung von Attributen und Operationen innerhalb einer Klassenhierarchie. Girokonto und Sparkonto haben die Verwaltung eines Kontostands gemeinsam. Um die Wiederholung von Code zu vermeiden, können beide Arten von Konten von einer Klasse `Konto` **erben**. Oder alle Sensoren haben gewisse Gemeinsamkeiten, die an spezielle Sensoren wie Temperatursensoren, Drucksensoren usw weitervererbt werden können. Durch solch ein Design werden Code-Wiederholungen vermieden. Wartung und Veränderung von Software wird erheblich sicherer, da im Idealfall nur an *einer*

Stelle verändert oder hinzugefügt werden muß. Ohne Vererbung ist man gezwungen, Code zu kopieren. Nachträgliche Änderungen sind dann an vielen Stellen durchzuführen. Mit Vererbung wird von einer oder mehreren Basisklassen geerbt. Das andere Verhalten der erbenden Klasse wird implementiert, indem Daten ergänzt werden, zusätzliche Methoden geschrieben werden oder geerbte Methoden überschrieben ("überladen" ist hier falsch) werden. **Polymorphismus von Objekten** bedeutet, daß ein Objekt sowohl als Instanz seiner Klasse alsauch als Instanz von Basisklassen seiner Klasse betrachtet werden kann. Es muß möglich sein, Methoden dynamisch zu binden. Hierin liegt die eigentliche Mächtigkeit objekt-orientierter Programmierung.

Beispiel Meßtechnik:

Bei der Programmierung einer Prozeßdatenverarbeitung müssen Meßdaten, z.B. Temperaturen erfaßt werden.

Bei einem *funktionalen* Angang des Problems wird man versuchen, eine Funktion zur Erfassung einer Temperatur zu entwickeln, die eine Analog/Digital-Wandler-Karte anzusprechen hat, die Rohdaten eventuell linearisiert und skaliert.

Beim *objekt-basierten* Ansatz muß man zunächst geeignete *Objekte* bezüglich der Temperaturerfassung suchen. Hier bietet sich der *Temperatursensor* als Objekt an, für den die Klasse Temperatursensor definiert werden kann. Der nächste Schritt ist, die *Repräsentation* eines Temperatursensors zu definieren, d.h. festzulegen, durch welche Daten oder Attribute ein Temperatursensor repräsentiert wird. Als Datenelemente wird man wohl den aktuellen Temperaturwert, verschiedene Skalierungsfaktoren, Werte für den erlaubten Meßbereich, Eichkonstanten, Kenngrößen für die Linearisierung, Sensortyp, I/O-Kanal-Adressen oder -Nummern definieren.

Erst im zweiten Schritt wird man die Funktionen (jetzt *Methoden* genannt) zur Skalierung, Linearisierung und Bestimmung des Temperaturwerts und eventuell Initialisierung des Sensors definieren und schließlich programmieren. Ferner wird man dem Konzept der *Datenkapselung* folgend Zugriffsfunktionen schreiben, mit denen die Daten des Sensors eingestellt und abgefragt werden können.

Beim *objekt-orientierten* Denken wird man versuchen, zunächst einen allgemeinen Sensor zu definieren, der Datenelemente hat, die für jeden Sensor (Temperatur, Druck usw) relevant sind. Ein Temperatursensor wird dann alles *erben*, was ein allgemeiner Sensor hat und nur die Temperaturspezifika ergänzen. Weiter könnte der Tempertursensor relativ allgemein gestaltet werden, sodaß bei Anwendung eines NiCrNi-Thermoelements das Thermoelement vom allgemeinen Temperatursensor erbt.

Dieses Beispiel soll zeigen, daß der objekt-basierte oder -orientierte Ansatz den Systementwickler veranlaßt (möglicherweise gar zwingt), eine Aufgabe allgemeiner, tiefer, umfassender zu durchdringen. Ferner ist beim funktionalen Ansatz nicht unmittelbar klar, was z.B. beim Übergang von einem Sensor zum anderen zu ändern ist. Im allgemeinen muß man die Meßfunktion neu schreiben. Bei einer gut entworfenen Objektklasse sind nur Werte von Datenelementen zu ändern. Ferner wird es bei einer gut angelegten Klasse offensichtlich sein, welche Daten zu ändern sind.

1.5 Objektinversion

Module oder Programmteile beziehen sich häufig auf eine Datenstruktur. Betrachten wir als Beispiel ein Rechteck, dessen Daten (Länge und Breite) in einer Datenstruktur verwaltet wird. Dazu gibt es Funktionen, die mit dieser Struktur arbeiten (`flaeche` und `umfang`).

In C++ haben wir eine umgekehrte Sicht: Die Funktionen gehören zur Datenstruktur. Anstatt daß einer Funktion die entsprechende Datenstruktur als Parameter übergeben wird, wird in C++ die entsprechende Funktion der Datenstruktur (Methode oder Elementfunktion genannt) aufgerufen.

Am einfachen Beispiel des Rechtecks werden funktionale und objektbasierte Sicht in der folgenden Tabelle verglichen:

Funktionale Sicht

```
// RechteckFunktional.C

#include <iostream>

using namespace std;

struct Rechteck
    {
    double a;
    double b;
    };




void    initialisieren
    (Rechteck* r, double x,
     double y)
    {
    r->a = x;
    r->b = y;
    }


double flaeche (Rechteck r)
    {
    return r.a * r.b;
    }


double umfang (Rechteck r)
    {
    return 2 * r.a + 2 * r.b;
    }


main ()
    {
    Rechteck   g;
    initialisieren (&g, 3.8, 7.9);
    cout << "Fläche : ";
    cout << flaeche (g) << "\n";
    cout << "Umfang : ";
    cout << umfang (g) << "\n";
    return 0;
    }
```

Objektbasierte Sicht

```
// RechteckObjectbasiert .C

#include <iostream>

using namespace std;

struct  Rechteck
    {
    double a;
    double b;

    void    initialisieren
       (double x, double y);
    double flaeche ();
    double umfang ();
    };


void    Rechteck::initialisieren
    (double x, double y)
    {
    a = x;
    b = y;
    }


double Rechteck::flaeche ()
    {
    return a * b;



double Rechteck::umfang ()
    {
    return 2 * a + 2 * b;
    }


main ()
    {
    Rechteck   g;
    g.initialisieren (3.8, 7.9);
    cout << "Fläche : ";
    cout   << g.flaeche () << "\n";
    cout << "Umfang : ";
    cout << g.umfang () << "\n";
    return 0;
    }
```

1.6 Übungen

Übung Erstes Programm: Schreiben Sie das "erste" C++ Programm mit einem Editor, übersetzen Sie es und lassen Sie es laufen.

Übung class Rechteck: Ersetzen Sie im Beispiel `RechteckObjbasiert.C` das Schlüsselwort **struct** durch **class**. Übersetzen Sie das Programm.

1. Beheben Sie den Fehler.
2. Schreiben Sie eine Methode `laenge ()`
3. Schreiben Sie eine Methode `breite ()`
4. Schreiben Sie eine Methode `zeige ()`, die alle Daten (Breite, Länge, Fläche und Umfang) über das Rechteck ausgibt. Verwenden Sie dabei die oben erstellten Methoden. Wenden Sie die Methode im Hauptprogramm an.
5. Kopieren sie das Programm in einen anderen File. Ändern Sie jetzt im neuen Programm die Datenrepräsentation: Anstatt der Länge **double** `a` und der Breite **double** `b` sollen jetzt die Fläche **double** `f` und der Umfang **double** `u` verwendet werden. Führen Sie die notwendigen Korrekturen an den Methoden durch. Die Länge a und die Breite b eines Rechtecks errechnen sich aus der Fläche f und dem Umfang u folgendermaßen:

$$b = \frac{u - \sqrt{u^2 - 16f}}{4}$$

$$a = \frac{u - 2b}{2}$$

Man muß `math.h` inkludieren und die Bibliothek `m` mit `-lm` linken.

Was hat sich am Hauptprogramm durch die geänderte Datenrepräsentation geändert?

Kapitel 2

Datentypen

Dieses Kapitel zeigt die eher konventionellen Typen, Deklarationen, Ausdrücke von C++. In späteren Kapiteln wird auf die Möglichkeiten eingegangen, neue Typen, Typhierarchien, Templates und benutzerdefinierbare Operatoren zu erzeugen.

2.1 Typen und Deklarationen

Zunächst wollen wir für uns die Begriffe *Definition* und *Deklaration* klären. Eine Deklaration informiert den Compiler über den Namen und Typ eines Objekts oder einer Funktion, ohne unbedingt Details zu geben. Eine Definition – etwa einer Klasse – informiert über Details. Bei Objekten ist die Definition der Ort, wo der Compiler Speicherplatz anlegt. Bei Funktionen ist das die Implementation der Funktion (Funktionsrumpf). Deklarationen von Namen *ohne* Definition können beliebig oft vorkommen. Die *Definition* einer Klasse, eines Objekts, einer Funktion usw darf nur einmal vorkommen.

Die beiden Begriffe werden sehr lax verwendet. Das liegt daran, daß in C++ viele Statements sowohl Deklarationen als auch Definitionen sind. Jenachdem, welchen Aspekt man hervorheben will, wird der eine oder andere Begriff verwendet.

Statements, die Deklarationen *und* Definitionen sind, sind z.B.

```
int    i;
double x;
int    funk (double a) { return a; }
class A
    {
    //   Implementation der Klasse A
    };
```

Deklarationen ohne Definition sind

```
extern int i;
extern double x;
int funk (double a);          // oder
extern int funk (double a);
class A;
```

Reine Funktionsdeklarationen heißen auch Funktions*prototypen*.

Jeder Name und jeder Ausdruck hat einen Typ, der bestimmt, welche Art von Operationen damit durchgeführt werden können. Z.B. die Deklaration

```
int Anzahl;
```

spezifiziert, daß `Anzahl` vom Typ **int** ist. C++ hat verschiedene *Basis-* oder *elementare* Typen und bietet verschiedene Möglichkeiten neue Typen zu definieren.

Deklarationen und Definitionen von Objekten (Variablen, Konstanten) können irgendwo im Programmtext erscheinen. C++ erlaubt das, weil bei der Definition und Initialisierung von Variablen häufig Zwischenrechnungen nötig sind.

Ein Objekt ist nur sichtbar ab seiner Deklaration bzw Definition bis zum Ende des Blocks, indem seine Deklaration erfolgte. Der *Scope* ist der Sichtbarkeitsbereich eines Objekts oder einer Variablen. Es gibt den

1. File-Scope: *Globale* Sichtbarkeit bei Definition bzw Deklaration außerhalb aller Funktionen und Klassen.

2. Class-Scope: Sichtbarkeit innerhalb einer Klassendefinition. Die Implementation von Elementfunktionen befindet sich im Class-Scope.

3. Local-Scope: Sichtbarkeit innerhalb eines geschweiften Klammernpaars `{}` (Block).

Was im File-Scope sichtbar ist, ist auch im Class-Scope sichtbar. Was im File-Scope und im **public** Class-Scope sichtbar ist, ist auch im Local-Scope sichtbar. Im Fall des **public** Class-Scopes sind die Elemente nur unter Bezug auf ein Objekt der Klasse oder – bei statischen Elementen – auf den Klassennnamen sichtbar.

Die Auflösung des Sichtbarkeitsbereichs erfolgt von Local- bis File-Scope. D.h. bei einem Objektnamen wird zuerst nach einer Deklaration im Local-Scope gesucht, dann – falls zutreffend – im Class-Scope, schließlich im File-Scope.

Der File-Scope kann explizit durch den Scope-Operator `::` ohne Bereichsnamen angesprochen werden (`::a`).

Die Argumente einer Funktion gehören zum lokalen Scope einer Funktion.

Beispiel:

```
int n;
int n1;

class A
   {
   public:
       int  n;
       int  n2;
       void f (int k);
   };


void A::f (int k)  // k Local−Scope
   {
   int   n;        // n Local−Scope
   int   n3;       // n3 Local−Scope
   n3 = k;         // n3 von Local−Scope
   n   = n3;       // n von Local−Scope
   A::n = n3;      // n von Class−Scope
   n2 = n3;        // n2 von Class−Scope
   ::n = n3;       // n von File−Scope
   n1 = n3;        // n1 von File−Scope
   }


main ()
   {
   A  a;
   n1 = n;         // n1 und n von File−Scope
   n2 = n3;        // Fehler: n2 und n3 nicht bekannt
   a.n2 = n;       // n2 von Class−Scope
                   // n von File−Scope
   }
```

Die *Lebensdauer (Extent)* einer Variablen oder eines Objekts ist die Zeit, während der Speicher für das Objekt allokiert ist. Objekte, die mit File-Scope definiert sind, und lokale Objekte mit Spezifikation **static** haben statischen Extent. Solang das Programm läuft, ist für diese Objekte Speicher allokiert.

Objekte, die lokal und nicht **static** definiert sind, leben nur bis zum Ende des Blocks, in dem sie definiert sind.

Objekte, für die dynamisch Speicher allokiert wird, haben dynamischen Extent. Sie leben, bis sie vom Anwender zerstört werden oder bis das Programm beendet wird.

Objekte mit File-Scope *und* Spezifikation **static** sind nur im File und eventuell inkludierten Files (Übersetzungseinheit) sichtbar. Die Namen können in anderen Übersetzungseinheiten anderweitig verwendet werden.

Bemerkung: Enthält ein Block Code der folgenden Art,

```
{
```

```
// ---
static int a = 5;
a = a + 1;
// ---
}
```

dann wird die **static**-Zeile nur ein einziges Mal durchgeführt und dann überlaufen. Das kann auch in einer Klassenmethode enthalten sein.

2.1.1 Elementare Typen

Die elementaren oder Basis-Typen sind:

bool
char
short
int
long
float
double
(long double)

Mit dem ersten Typ können Boolsche Variable mit den Werten **true** oder **false** repräsentiert werden.

Die nächsten vier Typen können als **signed** oder **unsigned** spezifiziert werden. Mit diesen Typen werden ganze Zahlen repräsentiert, mit den letzten zwei Typen Gleitkommazahlen. In C++ ist der kleinste Typ **char** und die Größe aller anderen Typen wird in Einheiten der Größe von **char** angegeben. Üblicherweise ist **char** so groß (typisch ein Byte) gewählt, daß damit auf einer gegebenen Maschine ein Zeichen dargestellt werden kann. Die natürliche Größe einer **int** -Variablen erlaubt die Darstellung ganzer Zahlen. Zur Bestimmung der Größe eines Typs gibt es die Funktion **sizeof** `()`. Zwischen den Typen gilt folgende Relation:

$$1 = sizeof(char) \leq sizeof(short) \leq sizeof(int) \leq sizeof(long)$$

$$sizeof(float) \leq sizeof(double) \leq sizeof(longdouble)$$

Mehr über die Größe der Basis-Typen anzunehmen ist gefährlich.

2.1.2 Zeiger und Referenzen

Mit den folgenden Operatoren können weitere Typen erzeugt werden:

`*`	Zeiger auf
***const**	konstanter Zeiger auf
`&`	Referenz auf
`[]`	Feld von
`()`	Funktion, die entsprechenden Typ zurückgibt

Zum Beispiel:

```
char* pa;       // Zeiger auf ein Zeichen vom Typ char
char a;
char*const pa2 = &a;  // Konstanter Zeiger auf einen Typ char
                      //  initialisiert  mit Adresse von a
const char* pa3;    // Zeiger auf eine Zeichenkonstante
char& ra = a;    // ra Referenz auf ein Zeichen a vom Typ char
                 // Referenzen müssen immer initialisiert werden.
char v[10];      // Feld von 10 Zeichen
char f ();       // Funktion f gibt ein Zeichen vom Typ char zurück
```

Zeiger *(Pointer)* spielen wie in C auch in C++ Programmen eine wesentliche Rolle. Typische Anwendungen sind verkettete Listen, dynamische Speicherverwaltung und Übergabe von großen Objekten und Funktionen. Deshalb soll an dieser Stelle kurz erklärt werden, was ein Zeiger ist.

Ein Zeiger `pa` ist eine Variable, die die Adresse eines Speicherplatzes als Wert enthält. Wenn das die Adresse einer Variablen `a` ist, dann sagt man "der Zeiger `pa` zeigt auf `a`". Folgendes Bild verdeutlicht die Beziehung zwischen der Zeigervariablen und der Variablen, auf die der Zeiger zeigt.

Name einer Variablen	Adresse eines Speicherplatzes *lvalue* *(location value)* (Locationswert)	Inhalt eines Speicherplatzes *rvalue* *(read value)* (Datenwert)
	2434	
a	2435	47
d	2436	73
	2437	
	2438	
pa	2439	2435
ra	2440	2435
	2441	
	2442	

Eine Zeigervariable muß wie jede andere Variable deklariert werden:

```
char* pa;
```

Der angegebene Typ bezieht sich auf den Typ der Variablen, auf die der Zeiger schließlich zeigen soll. Der Stern `*` bedeutet, daß es sich um einen Zeiger handelt. Es spielt keine Rolle, ob und auf welcher Seite des Sterns der Zwischenraum steht:

```
char *pa;
char * pa;
char*pa;
```

Viele Programmierer verwenden die Form **char** `*pa`, um bei folgender Deklaration Mißverständnisse zu vermeiden:

```
char* pa, pc;
char *pa, pc;
```

In jedem Fall ist hier `pc` als **char**-Variable, nicht als Zeigervariable auf **char** definiert. Wir verwenden hier die Form **char*** `pa`. Zur Vermeidung von Unklarheiten sollte man für jede Variable eine Zeile spendieren.

Nach der Definition hat der Zeiger keinen Wert, d.h. er zeigt noch nirgendwohin. Man kann ihm direkt eine Adresse oder die Adresse einer Variablen vom Typ des Zeigers mit Hilfe des *Adress-Operators* `&` zuweisen.

```
pa = (char*) 2435;
pa = &a;
```

Da `2435` vom Typ **int** ist, muß man hier allerdings dem Compiler durch einen sogenannten Cast mitteilen, daß die Zahl `2435` als Adresse und als Wert eines Zeigers auf **char** aufzufassen ist.

Der Inhalt der Variablen, auf die ein Zeiger zeigt, (*Inhalt von* `pa`) ist durch den *Dereferenzierungs-Operator* `*` zugänglich (*pointer indirection*).

```
*pa = a;
d = *pa;
```

Nach diesen Statements würde `d` den Wert `47` enthalten. Diese Art der Benutzung von `*` darf nicht mit der Verwendung bei der Deklaration eines Zeigers durcheinandergebracht werden. Hier entstehen für den Anfänger die meisten Mißverständnisse bezüglich des Gebrauchs von Zeigern. Ähnliche Mißverständnisse treten beim Operator `&` auf.

Referenzen

Eine Referenz ist äquivalent zu einem konstanten Zeiger. Die Verwendung aber ist syntaktisch gleich wie bei einer normalen Variablen (*Objektsyntax* anstatt von *Zeigersyntax*).

```
// Definition :
int a;              // Variable a
int& ra = a;        // ra Referenz auf a
int*const pa = &a; // pa konstanter Zeiger   initialisiert   mit
                    // Adresse von a


// Zuweisung:
a = 5;          // a ist 5
ra = 6;         // a ist nun 6
*pa = 7;        // a ist nun 7
```

Der Vorteil von Referenzen gegenüber Zeigern wird insbesondere deutlich bei variablen Funktionsargumenten (siehe Kapitel Funktionen). Eine Referenz kann verstanden werden als ein anderer Name für dasselbe Objekt.

2.1.3 Konstanten

Konstante Werte können nicht verändert werden. Sie haben einen Typ. Beispiele für in C++ mögliche konstante Werte sind:

```
5          // int (dezimal)
0          // int
35U        // unsigned int
31L        // long int
674UL      // unsigned long int
047        // int (octal)
0x56       // int (hexadezimal)
2.7182     // double
2.7182F    // float
271.82E-2  // double
2.         // double
0.0        // double
```

Konstante Werte vom Typ **char**:

```
'a'     // druckbares Zeichen a
'5'     // druckbares Zeichen 5
'\n'    // nicht druckbar: Newline
'\t'    // Horizontal-Tab
'\v'    // Vertikal-Tab
'\a'    // Bell
'\b'    // Backspace
'\r'    // Carriage Return
'\\'    // Backslash
'\?'    // Fragezeichen
'\''    // Einfacher Anführungsstrich
'\"'    // Doppelte Anführungsstriche
'\012'  // Newline
```

Stringkonstanten haben den Typ **char*** und werden in doppelten Anführungszeichen geschrieben. (Sie sind eigentlich keine Konstanten.)

```
"Guten Tag!\n"
"a"
""              // Leerer String
```

Ein String besteht aus den zwischen den Anführungszeichen aufgeführten Zeichen *und* einem das String-Ende anzeigenden Nullzeichen `\0` (nicht `0`). Der Unterschied zwischen `'a'` und `"a"` ist, daß `'a'` vom Typ **char** ist und aus einem Zeichen `a` besteht, während `"a"` vom Typ **char*** ist und aus zwei Zeichen, nämlich `a` und dem Nullzeichen besteht.

Variable konstanten Typs (Konstanten) können mit der Spezifikation **const** definiert werden. Sie müssen bei der Definition initialisiert werden.

```
const float EULER = 2.7182;
const char KaufmannsUnd = '&';
const char* Hello = "Guten Tag";
char*const Hallo = "Guten Tag";
const char*const Hollo = "Guten Tag";
```

Konstanten können nach der Initialisierung nicht mehr verändert werden. Eine Konstante kann nur einmal definiert werden. Eine Umdefinition ist nicht möglich. Eine von C geerbte Konvention ist, Konstantennamen in Großbuchstaben zu schreiben.

Bei `Hello` kann der Inhalt des Zeigers, d.h. der String "Guten Tag" nicht mehr verändert werden, aber man kann den Zeiger `Hello` auf einen anderen String zeigen lassen. Bei `Hallo` darf der Inhalt geändert werden, der Zeiger `Hallo` darf aber auf keinen anderen String zeigen. Bei `Hollo` sind Inhalt und Zeiger konstant.

```
Hello[6] = 'D'; // Fehler
Hello = Hallo;  // o.k.
Hallo[6] = 'D'; // o.k.
Hallo = Hello;  // Fehler
Hollo[6] = 'D'; // Fehler
Hollo = Hello;  // Fehler
```

Konstante Felder können folgendermaßen initialisiert werden:

```
const char tabl[]= {'A','B','C','D','E','F'};
```

Bei Variablen ist die Initialisierung optional, wird aber sehr empfohlen.

Bemerkung: In C++ Programmen sollte für Konstanten nicht mehr **#define** ... verwendet werden, sondern wegen der Typenprüfungsmöglichkeit durch den Compiler nur noch **const**. Die Konvention der mit Großbuchstaben geschriebenen Konstanten-NAMEN sollte beibehalten werden.

Konstanten sollten in einem Implementations-File definiert werden und, falls sie anderweitig gebraucht werden, in einem Header-Filer als **extern** deklariert werden:

```
extern const double EULER;   // Header-File

const double EULER = 2.7182; // .C-File
```

2.2 Definition neuer Typen

Es gibt fünf Möglichkeiten, neue Typen zu definieren:

1. Aufzählungstyp
2. Struktur, Record
3. Klasse
4. Union, Varianten
5. Bit-Felder

1. **Aufzählungstyp:**

```
enum name
   {
   Affe,
   Baer,
   Wolf,
   Tiger,
   Schlange
   };
```

Intern wird defaultmäßig jedem Element eine ganze Zahl (hier Affe 0, Baer 1, Wolf 2, Tiger 3, Schlange 4) zugeordnet. Man kann auch explizit Werte für die Elemente angeben. Defaultmäßig wird einem Element ein Wert zugeordnet, der um Eins größer ist als der Wert des vorhergehenden Elements.

```
enum name
    {
    Affe,
    Baer = 5,
    Wolf,
    Tiger = 8,
    Schlange = 20
    };
```

Hier hätten wir die Zuordnung Affe 0, Baer 5, Wolf 6, Tiger 8, Schlange 20. Guter Programmierstil berücksichtigt diese numerischen Zuordnungen *nicht*. Insbesondere sollten diese Elemente nicht in Größer-Kleiner-Relationen verwendet werden. Dafür sind ganze Zahlen da. Da Aufzählungstypen i.a. schwer erweiterbar (Korrektur des Programms an vielen Stellen) sind, sollten sie nur dort verwendet werden, wo man ziemlich sicher ist, daß keine weiteren Elemente dazukommen. Ferner kann man nur unter Verwendung der numerischen Zuordnung über Aufzählungstypen iterieren. Daher werden Aufzählungstypen eher selten verwendet.

2. **Struktur, Record:**

```
struct name
    {
    // Datenelemente
    };
```

Mit **struct** können Strukturen oder Records definiert werden. Da Strukturen aber mit dem Typ Klasse ebensogut, jedoch mit wesentlich mehr Möglichkeiten angelegt werden können, werden wir die **struct**-Möglichkeit, Strukturen zu erzeugen, praktisch nie verwenden.

3. **Klasse:** Auf die Definition von Klassen werden wir in einem gesonderten Kapitel ausführlich eingehen.

4. **Union, Varianten:** Der Varianten-Typ erlaubt es, *einen* Speicherplatz für verschiedene Datentypen eventuell unterschiedlicher Größe anzulegen. Eine *anonyme* Union ist z.B.:

```
struct name
    {
    // ---
    union
        {
        int ganz;
        float komma;
        };
    };
```

Damit kann man Speicherplatz sparen, wenn in jeder Variablen vom Typ `name` entweder nur die Komponente `ganz` oder die Komponente `komma` verwendet wird, da `ganz` und `komma` (teilweise) den gleichen Speicherplatz belegen.

Eine andere Anwendung ist die Betrachtung einer Größe unter zwei oder mehr Gesichtspunkten. Als Beispiel wollen wir die Darstellung des Nullpointers ermitteln und dabei zeigen, daß eine Variante auch einen Namen haben kann:

```
union gz
    {
    int i
    int* p;
    };

gz g;
g.p = 0;        // p Nullpointer
int j = g.i;    // j nicht notwendig gleich Null
```

Mit Varianten oder Unions muß man vorsichtig umgehen. Anwendung empfiehlt sich nur in wenigen Spezialfällen und dann, wenn Speicher wirklich ein Problem ist.

Unions sind auch Klassen, aber mit **public** Defaultzugriff. Sie können Elementfunktionen, Konstruktoren und Destruktoren haben. Die Elemente von Unions dürfen aber nicht Klassen mit Konstanten oder Destruktoren sein. Anonyme Unions dürfen keine Elementfunktionen, Konstruktoren oder Destruktoren haben.

5. **Bit-Felder:** Es lohnt sich zunächst nicht, weiter darauf einzugehen, da die run-time-Effizienz i.a. mäßig ist. Andererseits mag es eine bequeme

Möglichleit sein, z.B. bei der Prozeßsteuerung digitale I/O-Kanäle darzustellen.

```
class Bitfeld
    {
    unsigned int kanal1 : 1;
    unsigned int kanal2_5 : 4;
    unsigend int kanal6_8 : 3;
    };
```

Mit **typedef** können schon definierten oder bekannten Typen Synonyme gegeben werden. Die Typen können dann aber immer noch unter dem alten Namen angesprochen werden, daher ist das keine Umbennung von Typen.

```
typedef alterTypName neuerTypName;
```

Redefinition eines Typnamens ist allerdings nicht möglich:

```
struct name { --- };
typedef int name; // Fehler!
```

typedef ist in zwei Fällen sinnvoll:

1. Synonymisierung der elementaren Typen, um unabhängig von einer speziellen Implementation zu werden. Wenn das eigene Programm z.B. auf ganze Zahlen angewiesen ist, die vier Byte lang sind, dann ist das auf der einen Maschine möglicherweise der Typ **int**, auf der anderen der Typ **long**. Daher wird man sich einmal entweder

   ```
   typedef int mein4ByteInt;
   ```

   oder

   ```
   typedef long mein4ByteInt;
   ```

   definieren und im eigenen Programm dann nur noch `mein4ByteInt` verwenden.

2. Synonymisierung von Zeiger-, Referenz- und Feldtypen zur besseren Lesbarkeit:

   ```
   typedef int* intZeiger;
   typedef int& intReferenz;
   typedef int zehnIntVektor[10];

   // Variablendefinition :
   intZeiger pi, pj; // pi und auch pj Zeiger auf int !
   int i;
   intReferenz j = i; // Referenz auf int, Referenzen müssen
                      // initialisiert werden.
   zehnIntVektor v;  //Definition von v als Feld von 10 int
   ```

Da diese Typen – insbesondere die Arraytypen – häufig besonderer Maßnahmen der Initialisierung, der Deinitialisierung, Überwachung der Indexgrenzen usw bedürfen, sollte in diesen Fällen eine Klasse angelegt werden.

Zeiger auf Funktionen haben eine umständliche Syntax. Hier lohnt sich ein **typedef**:

typedef int `(*fptr) ();` *// Zeiger auf int−Funktion*

Näheres über Funktionszeiger siehe im Kapitel Funktionen.

2.3 Felder und dynamische Speicherallokierung

Felder mit statischem Speicher werden folgendermaßen definiert:

double `A[12];`

Wie in C haben alle Felder Null als ihre untere Grenze. Das Feld `A[12]` ist ein Feld von 12 **double**-Elementen und hat die Elemente `A[0]` bis `A[11]`. Durch Definition geeigneter Klassen kann sich der Benutzer allerdings beliebig indizierte Felder definieren (Kapitel Templates).

Bemerkung: Durch eine Deklaration vom Typ

double `A[12];`

werden die einzelnen Elemente *nicht* initialisiert (z.B. Null gesetzt). Bei Objekten von Klassen bedeutet dies, daß der Default-Konstruktor *nicht* aufgerufen wird. Diese Methode der Anlage von Feldern ist daher eventuell nur für elementare Typen geeignet (vgl. Operator **new**).

`A` selbst ist vom Typ **double∗const**, d.h. `A` kann wie ein konstanter Zeiger verwendet werden. `A` enthält die Adresse des ersten Elements des Feldes von `A`, d.h `*A` ist äquivalent zu `A[0]`.

Name einer Variablen	Adresse eines Speicherplatzes	Inhalt eines Speicherplatzes
`A[0]`	2436	15.234
`A[1]`	2444	18.121
`A[2]`	2452	23.678
...	...	...
`A[11]`	2544	78.527
	...	
	...	
`A`	3450	2436
`p`	3454	2436
	...	

Definieren wir einen Zeiger vom Typ **double∗**, dann können wir diesem `A` zuweisen:

```
double* p;
p = A;
A = p;      // Fehler, da A konstant
```

Über `p` kann ebenfallls mit der Subskriptsyntax (`p[i]`) auf die Elemente von `A` zugegriffen werden. Der *Inhalt* von `A` ist nicht konstant und kann verändert werden, aber die *Adresse* von `A` ist konstant.

Dynamischer Speicher wird nur über Zeiger mit den Operatoren **new** und **delete** verwaltet.

```
double* A;
A = new double[12];
```

Hier wird Speicher für ein Feld von 12 **double**-Elementen angelegt. Im Unterschied zum statisch angelegten Speicher kann "12" auch eine Variable sein. Beim statischen Speicher mußte sie ein konstanter Ausdruck sein (zur Compilezeit auswertbar).

Anstatt **double** können irgendwelche Typen `Typ`, insbesondere vom Benutzer definierte Klassen verwendet werden.

```
Typ* B;
B = new Typ[n];
```

Hier wird Speicher für `n` Elemente der Größe **sizeof** `(Typ)` angelegt.

Jedes Element des Felds wird *initialisiert*. Für Klassen bedeutet das, daß der Defaultkonstruktor (Konstruktor ohne Argumente) aufgerufen wird. Voraussetzung ist daher, daß der Defaultkonstruktor existiert.

Initialisierungsparameter können *nur* angegeben werden, wenn Speicher für ein einzelnes Element angelegt wird:

```
Typ* C;
C = new Typ;
Typ* D;
D = new Typ (Initialisierungsparameter);
int* p;
p = new int (5);  // *p ist 5
```

Man beachte die Verwendung *runder* Klammern.

Ist `Typ` eine Klasse, wird der passende Konstruktor aufgerufen.

Dynamisch angelegter Speicher bleibt allokiert, bis er vom Benutzer deallokiert wird. Wenn der Benutzer nicht aufpaßt, kann das dazu führen, daß immer mehr Speicher angelegt wird, bis schließlich der zur Verfügung stehende Speicherplatz erschöpft ist, was i.a. zum Absturz des Programms führt. Um solche *Speicherlecks* zu vermeiden, sollte peinlich darauf geachtet werden, daß ein allokierter Speicherbereich wieder freigegeben wird, wenn er nicht mehr benötigt wird. Das ist möglich mit dem Operator **delete** bzw **delete** `[]` :

```
delete [] A;
delete [] B,
delete C;
delete D;
```

Bei **delete** `A` würde zwar der Speicher für das ganze Feld aufgegeben werden, aber nur für das erste Element eine Deinitialisierung (Aufruf des Destruktors) vorgenommen werden. Das bedeutet, daß der Speicher verloren geht, den die einzelnen Elemente bei ihrer Initialisierung eventuell selbst angelegt haben (und den der Destruktor freigeben würde). Das `[]`-Klammernpaar sorgt dafür, daß der Destruktur für jedes Element des Feldes aufgerufen wird. Üblicherweise ist der Destruktor so gemacht, das er den für das Objekt allokierten Speicher wieder freigibt.

Daher sollte man sich an folgende Regel halten: Wenn bei **new** eckige Klammern verwendet wurden, dann müssen sie auch bei **delete** verwendet werden. Wenn bei **new** *keine* eckigen Klammern verwendet wurden, dann dürfen auch beim **delete** keine eckigen Klammern verwendet werden.

2.3.1 Mehrdimensionale Felder

Mehrdimensionale Felder werden definiert als:

```
double B[12][5][2];
```

Angesprochen werden die Elemente von `B` mit `B[i][j][k]`, wobei `i=0...11; j=0...4; k=0...1` sein kann. `B[i,j,k]` ist syntaktisch erlaubt, ergibt aber Unsinn! Ferner darf man in C++ nicht von einer Implementation als eindimensionales Feld ausgehen!

Dynamische mehrdimensionale Felder kann man anlegen, indem man sie als eindimensionales Feld auffaßt und für den Zugriff auf die Elemente die Rechteckformel zur Index-Berechnung verwendet. Oder man geht über einen Array von Zeigern. Z.B soll das zweidimensionale Feld `D[n1][n2]` dynamisch angelegt werden:

```
double* D;
D = new double[n1 * n2];
D[i * n2 + j] = 1.1;  // Zugriff auf D[i][j]
                      // über Rechteckformel
```

(Deinitialisierung von `D`: **delete** `[] D;`)

oder

```
double** D;
D = new double*[n1];
for (int n = 0; n < n1; n++)
    {
    D[n] = new double[n2];
    }

D[i][j] = 1.1; // normaler Zugriff !
```

Deinitialisierung von `D`:

```
for (n = 0; n < n1; n++)delete [] D[n];
delete [] D;
```

In C++ wird man diese beiden – recht umständlichen – mehrdimensionalen Felder nicht in einer Anwendungsumgebung verwenden, sondern geeignete Klassen dafür definieren, die die Komplexität verbergen.

Felder als Funktionsargumente

Wenn ein Feld als Argument einer Funktion übergeben wird, dann wird nur die Anfangsadresse übergeben. Hierfür gibt es zwei Notationen:

```
void func (int dim, double* A); \\ oder
void func (int dim, double A[]);
void func2 (int dim, double B[][5]);

// Aufruf:
double a[12];
func (12, a);

double b[12][5];
func2 (12, b);
```

Die Schreibweise mit eckigen Klammern ist vorzuziehen, um zu verdeutlichen, daß es sich bei den Argumenten um Felder handelt. Bei mehrdimensionalen Feldern sind die beiden Schreibweisen für den Compiler nicht äquivalent.

2.4 Ausdrücke

Die Verknüpfung von Variablen und Konstanten mit unären oder binären Operatoren sind Ausdrücke, z.B. `-8`, `a+b` oder `(a-*p)/(5+c)`. C++ bietet eine große Anzahl von Operatoren an, auf die hier noch nicht eingegangen wird. Es seien nur einige Besonderheiten bemerkt: C++ hat einen Zuweisungs*operator* `=`, nicht ein Zuweisungsstatement. Daher können Zuweisungen an Stellen erscheinen, wo man sie nicht erwarten würde, z.B. `x = (a=6)/6` oder `a=b=c`. Letzteres bedeutet, daß zuerst die Variable `c` der Variablen `b`, dann die Variable `b` der Variablen `a` zugewiesen wird.

Typen können in Ausdrücken frei gemischt werden. C++ macht dabei alle sinnvollen Konversionen.

Da C++ keinen Datentyp für logische Variablen hat, werden logische Operationen mit dem Typ **int** durchgeführt.

Der unäre Operator `*` dereferenziert einen Zeiger (Indirektions-Operator). `*p` ist das Objekt, auf das `p` zeigt. Der unäre Operator `&` ist der Adresse-von-Operator. `&a` ist die Adresse des Objektes `a`.

Wenn der Zeiger `p` auf eine Struktur zeigt, dann werden die Komponenten `a` dieser Struktur mit `p->a` dereferenziert.

2.5 Zeichenketten

Die Standard-Bibliothek bietet den Datentyp

```
std::string
```

an. Ein `string` repräsentiert in C++ eine Zeichenkette. Wie der Name schon sagt, besteht eine Zeichenkette aus einer Reihe von Zeichen. In den meisten Fällen wird ein Zeichen durch den **char**-Datentyp dargestellt. [9, 2.13.4, 1]

```
// strtest.cpp
#include <iostream>
#include <string>

using namespace std;

main ()
    {
    string s1 = "Hallo";
    cout << s1 << endl;
    cout << "Länge von " << s1 << " ist: " << s1.length() << endl;
    string s2 = "";
    s2 = s2 + "Hal" + "lo";
    cout  << (s1 == s2) << endl;
    string s4 = s1.replace (2, 2, "lihal");
    cout << s4 << endl;
    }
```

Bevor man Funktionen der `string`-Klasse verwenden kann, muss noch eine Bibliothek eingebunden werden. Um die "`string`-Library" einzubinden wird die Compiler Anweisung **#include** `<string>` benötigt.

Nach der Deklaration von `s1` und der Initialisierung mit `"Hallo"` kann die `string`-Variable mit `cout` ausgegeben werden. Dabei können auch mehrere Zeichenketten mit `<<` verknüpft werden. Sollen hingegen die `string`-Variablen verknüpft werden, geschieht dies mit dem Operator `+`. Im Gegensatz zu C können Zeichenketten in C++ mit `==` verglichen werden. Da in unserem Beispiel `s1`

und `s2` gleich sind, wird eine `1` ausgegeben. Die `string`-Funktion `.length()` gibt die Länge der jeweiligen Zeichenkette zurück. `.replace()` ersetzt, wie der Name schon sagt, an einer bestimmten Stelle in der Zeichenkette eine bestimmte Anzahl von Zeichen, durch eine andere Zeichenkette.

Die Ausgabe sieht dann folgendermaßen aus:

```
$ ./strtest
Hallo
Länge von Hallo ist: 5
1
Halihalo
$
```

2.6 Übungen

Übung Zeiger: Erkläre, warum folgende Statements richtig bzw falsch sind. Was sind die Werte der einzelnen Variablen?

```
int* a;
int  b;
int* c;
int  d;
b = 5;
a  = b; // falsch
*a = b; // falsch, erst a = new int;
a  = &b; // richtig
*a = b; // richtig, aber unnötig
c  = &d; // richtig
*c = *a; // richtig
```

Übung sizeof (Null): Schreiben Sie ein Programm, daß folgende Ausgabe macht:

```
sizeof (0)   = 4
sizeof ('0') = 1
sizeof ('\0') = 1
sizeof ("0") = 2
```

Die Ergebnisse der **sizeof**-Funktion dürfen natürlich von Ihrer Maschine abhängen.

Kapitel 3

Kontrollstrukturen

3.1 Statements

Das einfachste Statement ist ein leeres Statement und besteht nur aus einem Semikolon:

```
;
```

Das leere Statement kann dann nützlich sein, wenn die Syntax ein Statement erfordert, wo man eigentlich keines benötigt. Das nächst komplexere Statement besteht aus einem Ausdruck und einem Semikolon:

```
a = b + c;
```

Ein *Block* ist eine möglicherweise leere Liste von Statements zwischen geschweiften Klammern:

```
{
int b = 0;
a = b + c;
b++;
}
```

Die Art der Einrückungen und Unterteilung in Zeilen spielt keine Rolle. Oben gezeigtes Beispiel ist äquivalent zu

```
{int b=0;a=b+c;b++;}
```

Ein Block ist ein Statement. Mit einem Block kann man mehrere Statements als ein einziges Statement behandeln. Der Geltungsbereich (Scope) eines im Block definierten Namens erstreckt sich vom Deklarationspunkt bis zum Ende des Blocks.

3.2 if-Statement

Das *if*-Statement besteht aus dem Schlüsselwort **if** gefolgt von einem Bedingungsausdruck in runden Klammern, einem Statement und eventuell einem **else** mit Statement.

```
if (a > 0) b = c/a;
```

oder

```
if (a > 0)
    {
    b = c/a;
    c++;
    }
else
    {
    cout << "a nicht positiv!\n";
    b = 0;
    }
```

Wenn der Bedingungsausdruck nicht Null ist, dann wird in das Statement hinter dem **if**, sonst in das Statemnt hinter dem **else** verzweigt.

Anmerkung: Es existiert auch eine Kurzfrom des *if*-Statements. Diese ist eigentlich ein Operator und hat die Form `(x ? y : z)`. Das oben genannte Beispiel würde in der Kurzform so aussehen:

```
(a > 0) ? b = c++/a : b = 0;
```

Als Einschränkung gilt, dass `y` und `z` jeweils nur ein Ausdruck sein dürfen.

Im Abschnitt 7.1 wird weiter auf den Operator eingegangen.

3.3 switch-Statement

Das *switch*-Statement testet gegen eine Menge von Konstanten:

```
char c;
// ---
switch (c)
    {
    case 'a':
        x = xa;
        break;
    case 'b':
    case 'c': x = xb; break;
    default:
        x = 0;
        break;
    }
```

Der Ausdruck `(c)` muß von ganzzahligem Typ sein (**char**, **short**, **int**, **long**).

Die **case**-Konstanten müssen alle verschieden und von ganzzahligem Typ sein. Die **default**-Alternative wird genommen, wenn keine andere paßt. Eine **default** -Alternative muß nicht gegeben werden. Die **break**-Statements werden benötigt, damit das **switch**-Statement nach Abarbeitung einer Alternative verlassen wird. Sonst würden alle folgenden Alternativen auch abgearbeitet werden. Die verschiedenen **case**-Marken dienen nur als Einsprungspunkte, von wo an der Programmfluß weitergeht. Das **switch**-Statement ist ein übersichtliches Goto!

3.4 while-Statement

Das *while*-Statement besteht aus dem Schlüsselwort **while** gefolgt von einem Bedingungsausdruck in runden Klammern und einem Statement:

```
a = 10;
while (a > 0)
    {
    b = c/a;
    a--;
    }
```

Der Bedingungsausdruck im while-Statement wird ausgewertet. Solange der Ausdruck nicht Null ist, wird das Statement hinter dem **while** immer wieder durchgeführt. Mit dem Schlüsselwort **do** kann man das auszuführende Statement auch vor das **while** setzen:

```
a = 10;
do {
    b = c/a;
    a--;
    } while (a > 0);
```

Die Bedingung wird nach Durchführung des Statements abgeprüft.

Bemerkung: In der Form **while**`(a>0)a--;` bildet das Statement hinter **while** einen eigenen Scope, d.h. es wird als

while `(a > 0) { a--; }`

interpretiert. Daher empfiehlt sich der Klarheit halber immer die Schreibweise *mit* Klammern `{}`.

3.5 for-Statement

Das *for*-Statement

```
for (int i = 0; i < 10; i++)
    {
    a[i] = i;
    b[i] = i * i;
    }
```

ist äquivalent zu

```
{
int i = 0;
while (i < 10)
    {
    a[i] = i;
    b[i] = i * i;
    i++;
    }
}
```

aber lesbarer, da die Schleifenkontrolle lokalisiert ist. Zu bemerken ist, daß der Zähler `i` am Ende der Schleife hochgezählt wird, gleichgültig, ob man `i++` oder `++i` schreibt. Die beim **while** gemachte Bemerkung gilt auch hier.

Die Variable `i` gehört zum Scope der **for**-Schleife, sodaß `i` nach der **for**-Schleife nicht mehr zur Verfügung steht und neu definiert werden kann.

3.6 goto-Statement

Mit dem *goto*-Statement

goto `Label;`

wird auf eine Zeile mit dem Label `Label` gesprungen.

`Label : statement`

`Label` kann irgendwelche Namen annehmen, z.B. `L1`, `L2`, `unten`, `weiter`.

3.7 Deklarations-Statement

Das *Deklarations*-Statement führt einen Namen in das Programm ein, z.B.:

```
double x;
float z, w; // Schreibweise nicht zu empfehlen
            // besser:
float z;
float w;
```

Das Deklarations-Statement kann an beliebiger Stelle stehen. Die Ausführung des Statements besteht darin, daß ein eventuell vorhandener initialisierender Ausdruck ausgewertet und die Initialisierung durchgeführt wird. In der Schleife

```
for (int i = 0; i < 10; i++)
    {
    int j = i;
    a[i] = j;
    b[i] = j * j;
    }
```

wird `i` zwar nur einmal initialisiert, `j` aber bei jedem Schleifendurchgang. Das ist legal, sollte aber natürlich vermieden werden.

3.8 Compiler

3.8.1 g++ (GNU-Compiler)

Programme, die das alte **for**-Scoping benutzen, müssen mit der Compileroption `-fno-for-scope` übersetzt werden. Die Option `-ffor-scope` (neues **for**-Scoping) ist Default.

3.9 Übungen

Übung Kontrollstrukturen: Das erste C++ Programm zur Berechnung des Produkts von zwei ganzen Zahlen soll folgendermaßen verändert werden:

1. Der Benutzer soll angeben, welche Verknüpfung zu machen ist (`- + * / %`). Fehleingaben sollten abgefangen werden.
2. Das Programm soll solange laufen, bis der Benutzer als Verknüpfungszeichen `~` eingibt.
3. Bei Verknüpfung `/` soll der Nenner auf Null geprüft werden. Gegebenenfalls soll eine Fehlermeldung ausgegeben werden.

Kapitel 4

Funktionen

Eine Funktion ist ein Teil eines Programms, der über einen Namen (Funktionsname) beliebig oft aufgerufen werden kann. Die Funktions*deklaration* hat folgende Form:

Rückgabetyp Funktionsname (Liste von Argumenten);

Deklaration mit Definition:

Rückgabetyp Funktionsname (Liste von Argumenten)
{
Implementation
}

```
double funk (int a, double x, char z);
```

Eine Funktion muß vor ihrer Verwendung deklariert sein. Eine Funktion darf nicht innerhalb einer anderen Funktion definiert werden. Die Liste von Argumenten muß auf jeden Fall Typangaben enthalten. Die Namen sind nicht notwendig, aber aus Gründen der Lesbarkeit zu empfehlen. Die in der Implementation verwendeten Namen müssen bei der Definition mitangegeben werden. Ein vorangestelltes **extern** macht deutlich, daß es sich um eine reine Deklaration handelt. Es kann weggelassen werden, da schon aus der Deklarations-Syntax hervorgeht, daß es sich um eine Deklaration ohne Definition handelt.

Mit Rückgabetyp ist der Typ des Rückgabewerts gemeint.

Grundsätzlich werden alle Argumente immer als Wert übergeben, sodaß keine durch die Funktion veränderte Argumente zurückgegeben werden können. Nur der Rückgabewert ist durch die Funktion zu verändern. Wenn variable Argumente benötigt werden, dann behilft man sich in C damit, daß man Argumente als Zeiger definiert und Adressen übergibt:

```
int   f (int* pn)
   {
   *pn = 7;
   return *pn;
   }

main ()
   {
   int   x = 0;
   int   y = 0;
   x = f (&y);
   }
```

Das führt zu schwer lesbarem Code. Daher gibt es in C++ die Möglichkeit Argumente als Referenzen zu definieren, die dann in der Funktionsimplementation ohne Dereferenzierungsoperator verwendet werden und die auch ohne Adreßoperator aufgerufen werden. Dasselbe Beispiel mit Referenzen sieht folgendermaßen aus:

```
int   f (int& pn)
   {
   pn = 7;
   return pn;
   }

main ()
   {
   int   x = 0;
   int   y = 0;
   x = f (y);
   }
```

Damit hat man eine ähnlich bequeme Schreibweise zur Verfügung wie z.B. in Pascal. Die Argumentspezifikation `VAR` in Pascal wird in C++ einfach durch `&` ersetzt. Große Objekte sollten aus Performanzgründen als Referenz übergeben werden. Wenn man trotzdem verhindern will, daß sie verändert werden, dann kann man das mit der Spezifikation **const** erreichen.

```
int   f (const int& pn)
   {
   // ---
   }
```

Wenn der Rückgabetyp einer Funktion eine Referenz auf ein Objekt ist, dann muß das Objekt zurückgegeben werden, *nicht* seine Adresse oder ein Zeiger darauf. Dabei wird aber tatsächlich nur eine Adresse übergeben, die auf die in der Funktion verwendete Datenstruktur zeigt, die daher so beschaffen sein muß, daß sie den Funktionskörper überlebt.

Beispiel:

```
int&  f ()
   {
   int*  pa = new int;
   *pa = 5;
   return *pa;
   }

int&  g ()
   {
   int   a;
   a = 5;
   return a;
   }
```

Bei der Funktion `f` ist zu bemerken, daß in der Funktion ein Objekt mit **new** angelegt wird, das solange lebt, bis es vom Programmierer wieder zerstört wird. Die Adresse dieses Objekts wird zurückgegeben, indem von der Syntax her das ganze Objekt, nämlich Inhalt von `pa` zurückgegeben wird.

Die Funktion `g` ist syntaktisch richtig, aber gibt die Adresse eines Objekts zurück, das automatisch nach Verlassen der Funktion aufgegeben wird.

Betrachten wir als Beispiel für die Anwendung von Funktionen folgendes Programm, mit dem verschiedene Wurzeln berechnet werden und an dem die Syntax noch einmal erklärt wird.

```
// wurzelprog.C – Berechnung der Wurzeln
// der Zahlen 1 bis 10

#include <iostream>

double wurzel (double x);
   // Deklaration der Funktion wurzel
double absval (double x);
   // Deklaration der Absolutwertfunktion
const double epsilon = 1.0E-36;
const double errorlimit = 0.00001;


main ()
   {
   double num = 1.0;
   double x;
   for (int i = 0; i < 10; i++, num++)
      {
      x = wurzel (num);
      std::cout << "Die Wurzel von " << num;
      std::cout << " ist " << x << "\n";
      }
   }
```

```
double wurzel (double x)
   // Funktion zur Berechnung der Wurzel bis auf
   // 5 signifikante Stellen genau (Newton's Methode).
   // Bei negativen Zahlen Resultat -1.
   {
   if (x < 0) return -1;
   if (x < epsilon) return 0;
   double root = x/2;
   double error = x;
   while (absval (error) > errorlimit * root)
      {
      root = (x/root + root)/2;
      error = x/root - root;
      }
   return root;
   }


double absval (double x)
   // Funktion gibt den absoluten Wert
   // der übergebenen Zahl zurück
   {
   return (x < 0) ? -x : x;
   }
```

Eine Funktion muß vor ihrer Benutzung deklariert werden. `wurzel` und `absval` sind als Funktionen deklariert, die als Übergabeparameter ein **double** nehmen und ein **double** zurückgeben. Mit der Deklaration werden Funktionen bekannt gemacht, die eventuell in einem anderen File definiert sind. Hier sind die Funktionsdefinitionen im selben File weiter unten zu finden.

Bei einem Funktionsaufruf wird jedes Argument bezüglich seines Typs überprüft (type checking). Eventuell werden Typkonversionen vorgenommen.

Die Funktionsdefinition beginnt mit dem Typ des zurückgegebenen Werts, dann folgt der Name der Funktion mit den übergebenen Argumenten in Klammern. Dahinter folgt ein Statement. Mit dem **return**-Statement wird ein Wert zurückgegeben und die Funktion verlassen.

Normalerweise haben verschiedene Funktionen verschiedene Namen. Aber wenn Funktionen ähnliche Aufgaben haben, dann kann es nützlich sein, den gleichen Namen zu verwenden. Z.B. wenn die Absolutwertfunktion auch für ganze Zahlen verwendet werden soll, dann wäre es bequem, denselben Namen zu verwenden. In C++ ist es erlaubt, denselben Namen öfter zu verwenden, sofern Anzahl oder Typ der Argumente unterschiedlich sind. Welche Funktion zu verwenden ist, kann der Compiler mit Hilfe der Typen der Argumente feststellen (*Überladung* von Funktionen).

Die *Signatur* einer Funktion wird durch den Funktionsnamen, die Anzahl und den Typ der Argumente festgelegt. Funktionen mit unterschiedlicher Signatur sind verschiedene Funktionen. Der Rückgabetyp einer Funktion gehört *nicht* zur Signatur. Die Signatur wird vom Compiler in den Namen der Funktion

integriert, was zu einer Veränderung der Namen nach der Übersetzung führt (name mangling).

Der *Typ* einer Funktion wird bestimmt durch Rückgabetyp und Anzahl und Typ der Argumente. Der Funktionsname spielt für den Typ der Funktion *keine* Rolle.

```
int absval (int x); // Deklaration der
                    // Absolutwertfunktion für int
```

Die Definition von `absval (int x)` lautet:

```
int absval (int x)
   // Funktion gibt den absoluten Wert
   // der übergebenen Zahl zurück
   {
   return (x < 0) ? -x : x;
   }
```

Wenn eine Funktion keinen Wert zurückgibt, sollte sie als **void** deklariert werden. Eine **void**-Funktion benötigt kein **return**.

```
void funktion (int a)
   {
   // ---
   }
```

Die Deklarationen `func ()` und `func (void)` sind äquivalent und bedeuten, daß keine Argumente übergeben werden. Eine variable Anzahl von Parametern wird durch drei Punkte angegeben:

```
int   func1 (...);
int   func2 (char c, ...);
```

Funktionen können Defaultargumente haben.

```
int   func1 (int a = 5);
int   func2 (int a, int b = 2, int c = 3);
int   func3 (int a, int b = 2, int c);  // Fehler
```

Nur die am weitesten rechts stehenden Argumente dürfen Defaultwerte haben. D.h., wenn in `func3` das Argument `b` einen Defaultwert hat, dann muß auch `c` einen Defaultwert haben. Die Defaultwerte erscheinen nur in der Deklaration der Funktion, nicht in der Definition (ausgenommen, daß es keine reine Deklaration gibt).

Eine Funktion mit Defaultargumenten kann folgendermaßen aufgerufen werden:

```
a = func2 (7);
a = func2 (7, 4);
a = func2 (7, 4, 9);
```

4.0.1 Scope einer Funktion

Die übergebenen Argumente gehören zum Scope der Funktion. Ihre Namen sind nur in der Funktion bekannt und der dafür angelegte Speicher besteht nur so lang, wie die Funktion läuft. In der Funktion angelegte Objekte werden nach Beendigung der Funktion deinitialisiert mit Ausnahme des Objekts, das als Funktionswert zurückgegeben wird. Dieses Objekt lebt – anonym, bis der Scope verlassen wird, in dem die Funktion aufgerufen wurde. Das ist besonders dann zu beachten, wenn große Objekte zurückgegeben werden. Außer bei **void**-Funktionen wird also bei jedem Funktionsaufruf auf jeden Fall ein Objekt angelegt.

4.0.2 Zeiger auf Funktionen

Zeiger auf Funktionen werden benötigt, wenn Funktionen als Funktionsargumente übergeben werden sollen. Die Syntax zur Definition eines Funktionszeigers ist etwas kompliziert. Die Definition muß alle Argumenttypen und den Rückgabetyp der Funktion enthalten, z.B. mit

```
int   (*pf) (char*, int, double);
```

wird `pf` als Zeiger auf eine Funktion definiert, die als Argumente ein **char***, ein **int** und ein **double** nimmt und ein **int** zurückgibt. Die Klammern um `*pf` sind wichtig, da andernfalls eine Funktion mit Namen `pf` deklariert worden wäre, die ein **int*** zurückgibt. `pf` ist vom selben Typ wie alle Funktionen, die wie

```
int   Funktionsname (char*, int, double);
```

deklariert sind. Es gibt so viele verschiedene Funktionstypen, wie es Kombinationen von Argumentlisten und Rückgabetypen gibt.

Einem Funktionszeiger wird eine Funktion zugewiesen, indem der Funktionsname mit oder ohne Adressoperator zugewiesen wird.

```
int   Funktionsname (char*, int, double);

pf = Funktionsname; // (lesbarer) oder
pf = &Funktionsname;
```

Die Funktion kann nun durch den Zeiger aufgerufen werden. Hier gibt es auch zwei alternative Formen:

```
int   a;
a = pf ("Hello", 7, 12.5); // (lesbarer) oder
a = (*pf) ("Hello", 7, 12.5);
```

Ein Funktionszeiger als Argument einer Funktion `func` sieht folgendermaßen aus:

```
int   func (int (*pf) (char*,int,double), int);

a = func (Funktionsname, 7);
```

Wenn der Funktionszeiger `pf` Rückgabetyp einer Funktion
`funk (double, int)` sein soll, dann muß `funk` folgendermaßen deklariert werden:

```
int   (*funk (double, int)) (char*, int, double);
```

Zur Vermeidung solcher Konstrukte sollte man vorher mit **typedef** Funktionstypen definieren:

```
typedef int (*pfTyp) (char*, int, double);
   // Definition des Funktionstyps pfTyp
pfTyp funk (double, int);
pfTyp pf;
int   func (pfTyp pf, int);
```

4.0.3 Inline-Funktionen

Um einen Funktionsaufruf zu sparen, werden in C häufig Macro-Definitionen verwendet. Anstatt von Macro-Definitionen sollten in C++ Inline-Funktionen verwendet werden, indem man vor die übliche Funktionsdefinition das Schlüsselwort **inline** schreibt.

```
inline int  max (int i, int j)
   {
   return i < j ? j : i;
   }
```

Der Compiler ersetzt jeden Aufruf einer Inline-Funktion durch die Implementation dieser Funktion. Inline-Funktionen sind daher so schnell wie Makrodefinitionen. Sie haben aber den Vorteil, daß sie wie Funktionen aussehen und ebenso vom Compiler überprüft werden.

Gegenüber normalen Funktionen haben Inline-Funktionen den Vorteil, daß der Funktionsaufruf gespart wird, und daß ein optimierender Compiler kontextabhängige Optimierungen durchführen kann.

Inline-Funktionen haben aber auch Nachteile. Der Code kann so stark anwachsen, daß entweder das Programm nicht mehr in den Speicher paßt, oder daß bei virtuellem Speicher der eigentliche Leistungsgewinn durch verstärktes Paging beeinträchtigt oder mehr als nivelliert wird.

Das Schlüsselwort **inline** ist für den Compiler nur ein Hinweis. Unter bestimmten Umständen (Code zu lang, Code enthält mehr als ein **return**, Rekursion, Zeiger auf Inline-Funktionen) behandelt der Compiler die Inline-Funktion als normale Funktion. Da Inline-Funktionen häufig in Header-Files stehen, die möglicherweise öfter inkludiert werden, führt das zu größerem Code und eventuell zu Schwierigkeiten beim Linken wegen mehrfacher Definition derselben Funktion. Debugger können meistens nicht mit Inline-Funktionen umgehen.

Daher wird ein sehr sparsamer Umgang mit Inline-Funktionen empfohlen.

Eine **inline**-Funktion kann nur *nach* ihrer Deklaration als **inline** verwendet werden.

4.1 Präprozessordirektiven

Der Präprozessor optimiert den Programmcode, bevor der Compiler ihn übersetzen kann. Dabei werden unter anderem Strings zusammengefasst, Kommentare und Whitespaces gelöscht. [10, 1.11] Dieser Prozess lässt sich mit den sogenannten Präprozessordirektiven manipulieren.

Die wichtigsten Direktiven sind:

```
#define KONSTANT irgend etwas

#undef KONSTANT

#include "Filename_mit_Pfad"
#include <Filename>

#ifdef  Name
// ---
#else
// ---
#endif

#ifndef Name
// ---
#endif
```

#define definiert Makros. Diese werden vom Präprozessor durch den hinter dem Makro stehenden Wert ersetzt. In unserem Fall wird das Makro "`KONSTANT`" durch "`irgend etwas`" ersetzt. Die Konvention besagt, das Makros aus Großbuchstaben bestehen. [10, 1.11.1]

#undef hebt Makros wieder auf. Dies ist allerdings optional. [10, 1.11.2]

#include inkludiert Files. Dabei wird der Programmcode einer anderen Datei an die Stelle der Direktive kopiert. Bei Verwendung von `<>` wird in voreingestellten Verzeichnissen gesucht. [10, 1.11.3]

#ifdef — **#else** — **#endif** wird für die "Bedingte Kompilierung" benötigt. Ist das Makro `Name` definiert, wird der Programmcode hinter dem **#ifdef** kompiliert, ansonsten der hinter dem **#else**. Diese Funktionalität wird meist in Zusammenhang mit plattformunabhängigem Programmcode verwendet. [10, 1.11.5]

#ifndef — **#else** — **#endif** verhält sich ähnlich wie **#ifdef**. Allerdings wird hierbei darauf geprüft, ob das Makro nicht definiert ist. Die Direktive wir oft in Header-Dateien verwendet um sicherzustellen, dass ein und dieselbe Headerdatei nur einmal inkludiert werden kann. [10, 1.11.5]

Es existieren noch weitere Präprozessordirektiven. Auf diese wird allerdings nicht mehr eingegangen, da sie im Folgenden auch nicht verwendet werden.

4.2 Programmstruktur

Ein C++ -Programm besteht normalerweise aus mehreren Sourcefiles, die jeweils Deklarationen von Typen, Funktionen, Variablen und Konstanten enthalten. Damit ein Name, der in einem anderen File definiert ist, benutzt werden kann, muß er deklariert werden, eventuell mit Spezifikation **extern**, falls die Deklaration mit einer Definition verwechselt werden kann. Die Deklaration muß mit der Definition konsistent sein.

Im allgemeinen wird dies dadurch garantiert, daß man die Deklarationen in sogenannte Header-Files schreibt, die mit der Präprozessor-Anweisung `include` in all die Files kopiert werden, die die betreffenden Deklarationen benötigen. Die meisten Compiler erlauben es nicht, daß Konstanten und Strukturen mehr als einmal definiert werden. Daher sollte man die Header-Files gegen mehrfaches Inkludieren schützen, indem man z.B. den ganzen Code des Header-Files zwischen folgende Statements setzt:

```
#ifndef meinHeader_h
#define meinHeader_h
// ---
// ---
#endif
```

`meinHeader_h` ist hier eine für den Headerfile spezifische Bezeichnung, z.B. der Filename, wobei der Punkt durch Underline ersetzt ist. Mit der Präprozessor-Anweisung `ifdef` bzw `ifndef` können in Abhängigkeit von einer Bedingung – des Bekanntseins eines Namens – Programmteile bis zum korrespondierenden `endif` ein- bzw ausgeblendet werden.

Headerfiles enthalten :

- Typdefinitionen wie **class, enum, struct**
- Funktionsdeklarationen wie **int** `funk ();`
- **inline**–Funktionsdefinitionen wie **inline int** `funk (){---};`
- Datendeklarationen wie **extern int** `zaehler;`
- Konstanten wie **const float** `pi = 3.1415;`
- Includes wie **#include** `<iostream>`

Headerfiles dürfen

- *nie* Funktionsdefinitionen (Implementationen) außer **inline**
- *nie* Datendefinitionen wie **int** `zaehler;`
- *nie* irgendwelche Initialisierungen

enthalten.

Konvention ist, daß Header-Files die Extension `.h` haben.

Als Beispiel soll hier das oben gezeigte Wurzel-Programm auf verschiedene Files aufgeteilt werden:

File: `absval.h`

```
// absval.h - Absolutwertberechnung

#ifndef absval_h
#define absval_h

double absval(double x);
   // Deklaration der Absolutwertfunktion für double
int absval(int x);
   // Deklaration der Absolutwertfunktion für int

#endif
```

File: `absval.cpp`

```
// absval.C - Absolutwertberechnung

#include "absval.h"

double absval (double x)
   // Funktion gibt den absoluten Wert
   // der übergebenen Zahl zurück
   {
   return (x < 0) ? -x : x;
   }

int absval (int x)
   // Funktion gibt den absoluten Wert
   // der übergebenen Zahl zurück
   {
   return (x < 0) ? -x : x;
   }
```

File: `wurzel.h`

```
// wurzel.h – Berechnung der Wurzel

#ifndef wurzel_h
#define wurzel_h

double wurzel (double x);
   // Deklaration der Funktion wurzel
const double errorlimit = 0.00001;

#endif
```

File: `wurzel.cpp`

```
// wurzel.C – Berechnung der Wurzel

#include "absval.h"
#include "wurzel.h"

const double epsilon = 1.0E-36;

double wurzel (double x)
   // Funktion zur Berechnung der Wurzel bis auf
   // 5 signifikante Stellen genau (Newton's Methode).
   // Bei negativen Zahlen Resultat -1.
   {
   if (x < 0) return -1;
   if (x < epsilon) return 0;
   double root = x/2;
   double error = x;
   while (absval (error) > errorlimit * root)
      {
      root = (x/root + root)/2;
      error = x/root - root;
      }
   return root;
   }
```

File: `wurzelmain.cpp`

```
// wurzelmain.C – Berechnung der Wurzeln
// der Zahlen 1 bis 10

#include <iostream>
#include "wurzel.h"

main ()
   {
   double num = 1.0;
   double x;
   for (int i = 0; i < 10; i++, num++)
      {
      x = wurzel (num);
      std::cout << "Die Wurzel von " << num;
      std::cout << " ist " << x << "\n";
      }
   }
```

Da die Konstante `errorlimit` auch im Hauptprogramm verwendet wird, mußte sie durch Definition im Header-File `wurzel.h` öffentlich gemacht werden.

Die Files werden mit folgenden Kommandos übersetzt und zu einem lauffähigen Programm `wurzelmain` zusammengebunden:

```
$ g++ -c absval.cpp
$ g++ -c wurzel.cpp
$ g++ wurzelmain.cpp absval.o wurzel.o -o wurzelmain
```

Oder unter Visual Studio:

```
> cl wurzelmain.cpp absval.cpp wurzel.cpp
```

4.3 Makefile

Um zu einem lauffähigen Programm zu kommen, müssen eventuell sehr viele und sehr lange Kommandozeilen getippt werden. Makefiles erleichtern einem hier das Leben. Exemplarisch sollen hier die wichtigsten Möglichkeiten von Makefiles am Beispiel des oben gezeigten Programms gezeigt werden.

Der Makefile hat den Namen

```
makefile
```

und das Kommando zu seiner Ausführung lautet:

```
$ make
```

oder

```
$ make Programmname
```

Ein sehr einfacher Makefile für `wurzelmain` ist:

```
wurzelmain: absval.C wurzel.C wurzelmain.C
   g++ -c absval.C
   g++ -c wurzel.C
   g++ -c wurzelmain.C
   g++ -o wurzelmain wurzelmain.o wurzel.o absval.o
```

Dieser Makefile ist zwar sehr schnell geschrieben, hat aber verschiedene Nachteile. Wenn sich z.B. der Compliler von `CC`nach `xlC` ändert, muß an vier verschiedenen Stellen eingegriffen werden. Außerdem wird bei jedem `make` jeder Programmteil neu übersetzt.

Ein besserer Makefile für `wurzelmain` ist der folgende:

```
COMPILER = g++
CFLAGS =
LDFLAGS =

PROG = wurzelmain
PROGSRC = wurzelmain.C
PROGOBJ = wurzelmain.o

APPLIB = application.a
APPLIBSRC = absval.C wurzel.C
APPLIBOBJ = absval.o wurzel.o

$(PROG): $(PROGOBJ) $(APPLIB)
   $(COMPILER) $(CFLAGS) $(LDFLAGS) -o $(PROG) $(PROGOBJ) $(APPLIB)

$(PROGOBJ): $(PROGSRC) wurzel.h
   $(COMPILER) $(CFLAGS) -c $(PROGSRC)

$(APPLIB): $(APPLIBOBJ)
   @echo "\t$(APPLIB) is now uptodate"

absval.o: absval.C absval.h
   $(COMPILER) $(CFLAGS) -c absval.C
   ar ur $(APPLIB) absval.o

wurzel.o: wurzel.C wurzel.h absval.h
   $(COMPILER) $(CFLAGS) -c wurzel.C
   ar ur $(APPLIB) wurzel.o

test:
   ./wurzelmain

clean :
   rm -f $(PROG) $(PROGOBJ) $(APPLIB) $(APPLIBOBJ) core tags
```

4.4 Einbindung von C-Funktionen

Wenn C-Funktionen verwendet werden, müssen sie durch eine Deklaration der folgenden Art bekannt gemacht werden:

```
extern "C" int cfunc (int a, char b, double x);
```

Files, die die Implementation von C-Funktionen enthalten, werden mit dem C-Compiler übersetzt. Der Makefile-Eintrag für einen C-Codefile lautet z.B.:

```
c_codefile.o: c_codefile.c
   gcc $(CFLAGS) -c c_codefile.c
   ar ur $(APPLIB) c_codefile.o
```

4.5 Übungen

Übung Funktionszeiger: Wir wollen ausprobieren, ob eine **while**-Schleife oder eine **for**-Schleife schneller ist. Dazu verwenden wir die folgenden beiden Funktionen:

```
int   fwhile (int anz)
   {
   int   i = 0;
   int   x = 0;
   while (i < anz)
      {
      x = x + i;
      i++;
      }
   return x;
   }

int   ffor (int anz)
   {
   int   x = 0;
   for (int i = 0; i < anz; i++)
      {
      x = x + i;
      }
   return x;
   }
```

Diese beiden Funktionen werden von einer Funktion `run` aufgerufen, die die folgende Form hat:

```
int   run (int anz, Typ funktion)
   {
   return funktion (anz);
   }
```

Schreibe ein Hauptprogramm, daß vom Benutzer `anz` und die Art der Funktion fordert, die verwendet werden soll. Korrigiere außerdem bei `run` die Syntax.

Übung Programmstruktur: Schreibe jede Funktion der vorhergehenden Übung in einen File. Schreibe einen Header-File. Schreibe einen Makefile.

Kapitel 5

Namensräume

Mit den Namensräumen ***(namespace)*** bietet C++ einen Mechanismus, um Daten, Funktionen, Klassen und Objekte zu gruppieren.

5.0.1 Definiton

Ein Namensraum besteht aus eine in einem Header-File definierten Schnittstelle ***(interface)*** und einer Implementation.

Schnittstelle in `Raum.h`:

```
// File: Raum.h

namespace Raum
   {
   void lichtAn ();
   void lichtAus ();
   void zeige ();
   }
```

Implementation in `Raum.cpp`:

```
// File: Raum.cpp

#include "Raum.h"
   // Man muss nicht, aber sollte das inkludieren !

#include <iostream>

namespace Raum
   {
   bool schalter = false; // an oder aus

   void lichtAn ()
      {
```

```
      schalter = true;
      }

   void lichtAus ()
      {
      schalter = false;
      }

   void zeige ()
      {
      std::cout << "Das Licht ist ";
      if (schalter) std::cout << "an";
      else std::cout << "aus";
      std::cout << "!\n";
      }

   }
```

Anwendung in `RaumApp.cpp`:

```
// File: RaumApp.cpp

#include "Raum.h"

main ()
   {
   Raum::zeige ();
   Raum::lichtAn ();
   Raum::zeige ();
   Raum::lichtAus ();
   Raum::zeige ();
   }
```

Da in C++ jede Deklaration in einem Namensraum erfolgen kann, wird dadurch das Prinzip des Information-Hiding realisierbar. Der Anwender eines Namensraums kann nur das verwenden, was in der Schnittstelle definiert ist.

Man kann die **Repräsentation** des Namensraum von der Implementation seiner Funktionen trennen:

```
// File: Raum2.cpp

#include "Raum.h"
   // Man muss nicht, aber sollte das inkludieren !

#include <iostream>

namespace Raum // Repräsentation
   {
   bool schalter = false; // an oder aus
   }
```

```
// Implementation der Funktionen von Raum:

void Raum::lichtAn ()
   {
   schalter = true;
   }

void Raum::lichtAus ()
   {
   schalter = false;
   }

void Raum::zeige ()
   {
   std::cout << "Das Licht ist ";
   if (schalter) std::cout << "an";
   else std::cout << "aus";
   std::cout << "!\n";
   }
```

5.0.2 Using-Direktive

Der Code des vorgestellten Anwendungsprogramms wirkt durch die vielen `Raum ::`-Prefixe unübersichtlich. Das kann durch Verwendung der ***Using-***Direktive vermieden werden:

```
using namespace Raum;
```

Diese Direktive kann überall gegeben werden.

Wenn wir diese Direktive außerhalb jedes Namensraums und jeder Klasse, Struktur und Funktion anbringen, dann "globalisieren" wir die Namen des Namensraums, d.h. machen sie global verwendbar. Das ist meistens eigentlich nicht erwünscht. Daher sei empfohlen, die Using-Direktive möglichst lokal zu verwenden, d.h. innerhalb eines Blocks, einer Funktion oder eines Namensraums.

Wenn eine Namensraum-Schnittstelle `N1` eine Using-Direktive für `N2` enthält, dann können die Elemente von `N2` transparent mitbenutzt werden:

```
namespace N2
   {
   int   a2;
   }

namespace N1
   {
   using namespace N2;
   int   a1;
```

```
   }

main ()
   {
   using namespace N1;
   a1 = 5;
   a2 = 3;
   std::cout << (a1 + a2);
   }
```

Man kann denselben Namen für einen Namensraum mehrfach verwenden, um etwa eine Schnittstelle für den Anwender und eine für den Implementor – eben mit gleichem Namen – zu definieren. Der Compiler muss dies auch unterstützen.

5.0.3 Aliase

Der Name eines Namensraums sollte wegen Eindeutigkeit lang und aussagekräftig sein. Das ist allerdings "lokal" lästig. Daher gibt es die Möglichkeit "Abkürzungen" oder Aliase zu definieren.

```
namespace bas = Berufsakademie_Stuttgart
```

Kapitel 6

Die C++ Klasse

Die Klasse ist ein vom Benutzer definierter Datentyp. Sie besteht aus Datenelementen möglicherweise verschiedenen Typs und einer Anzahl Funktionen, mit denen diese Daten manipuliert werden können.

Das Klassenkonzept hat folgende Leistungsmerkmale:

- Bildung neuer Typen, die so bequem zu verwenden sind wie die elementaren Typen und den Bedürfnissen des Programmierers Daten zu repräsentieren genügen.
- Kontrolle des Zugangs zu Daten: Datenstrukturen können vor dem direkten Zugriff des Benutzers geschützt werden (Datenabstraktion, Datenkapselung, Information hiding). Details der Implementation können von der Benutzeroberfläche des Typs getrennt werden.
- Initialisierung und Aufgabe von Objekten kann für den Benutzer transparent erfolgen.
- Vermeidung von Code-Wiederholungen durch Vererbungsmechanismen.
- Bildung von typunabhängigen Datenstrukturen (Template).

Die Klassendefinition besteht aus einem **Klassenkopf** ***(class head)***, der sich aus dem Schlüsselwort **class** und einem Klassennamen zusammensetzt, und einem **Klassenkörper** ***(class body)***, der in geschweiften Klammern steht. Die Klassendefinition muß mit einem Semikolon abgeschlossen werden.

```
class Klassenname
   {
   public:
   protected:
   private:
   };
```

Die Schlüsselwörter **public**, **protected** und **private** ***(access specifier)*** kontrollieren den Zugang zu den anschließend definierten Daten und Funktionen. Daten und Funktionen unter **public** können von überall im Definitionsbereich der Klasse d.h., wo ein Objekt der Klasse bekannt ist, benutzt werden, während die Klassenelemente unter **protected** und **private** nur von Elementfunktionen der Klasse benutzt werden können. Klassenelemente unter **protected** können für die Erben einer Klasse sichtbar sein, niemals aber die Elemente unter **private** . Das wird näher im Kapitel Vererbung erläutert.

Defaulteinstellung ist **private**. Wenn man sich aber an den empfehlenswerten Programmierstil hält, die öffentlich zugänglichen Daten und Funktionen zuerst zu nennen, ist man gezwungen auch den privaten Teil mit **private** zu deklarieren, was das Klassendesign übersichtlicher macht. (Bei **struct** ist Defaulteinstellung **public**.)

Dem Modell der Datenabstraktion oder Datenkapselung entspricht es, alle Daten im **private**-Teil zu deklarieren und sogenannte **Zugriffsfunktionen** ***(access functions)***, mit denen die Daten manipuliert werden, im **public**-Teil zu deklarieren. Die in einer Klasse definierten Funktionen heißen **Methoden** oder **Elementfunktionen** ***(member functions)***.

Beispiel Klausurnote:

Als Beispiel wollen wir uns die Klasse `Klausurnote` anschauen.

Eine Klausurnote ist ein so komplexes Objekt, daß wir es mit einem elementaren Typ nicht adäquat darstellen können. Klausurnoten können nur Werte zwischen 1,0 und 5,0 annehmen und sie müssen auf Zehntel gerundet sein. Ferner gibt es für eine Klausurnote eine verbale Darstellung (sehr gut, gut usw). Damit all dies gewährleistet ist, bilden wir einen neuen Typ `Klausurnote`, indem wir eine entsprechende Klasse definieren:

```
class Klausurnote
   {
   public:
      int   set (char* Note);
      char* getNumerisch ();
      char* getVerbal ();
      void  druckeNumerisch ();
      void  druckeVerbal ();
   private:
      char note[3];
   };
```

Es ist Konvention, den Klassennamen mit einem Großbuchstaben beginnen zu lassen. Als Datenmodell für die Note wählen wir einen Array von drei **char** , um das deutsche Dezimalkomma darstellen zu können. Als Interface dieser Klasse bieten wir fünf Funktionen (Methoden) an, mit denen der Benutzer der Klasse die Note belegen (`set`), lesen (`getNumersich`, `getVerbal`) und auf dem Bildschirm ausgeben kann (`drucke...`). Da wir uns noch nicht mit Vererbung beschäftigen, gibt es hier keinen **protected**-Teil.

Die Benutzung dieser Klasse im Anwenderprogramm würde etwa folgendermaßen aussehen:

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

using namespace std;

void main ()
   {
   Klausurnote bio;
   bio.set ("2,3");
   bio.druckeVerbal ();
   char buf[256];
   cout << "Bitte Note eingeben: "; cin >> buf;
   bio.set (buf);
   bio.druckeNumerisch ();
   }
```

Es wird das Objekt `bio` vom Typ `Klausurnote` angelegt. Erst jetzt wird Speicher für ein Klassenobjekt angelegt. Die Methoden des Objekts `bio` werden über den Dereferenzierungs-Operator "`.`" angesprochen. Wenn das Objekt über einen Zeiger angesprochen wird, dann wird der Derefenzierungsoperator "`->`" verwendet.

```
Klausurnote bio;
Klausurnote* pBio;
pBio = &bio;
pBio->set("2,3");
```

In objektorientierter Terminologie bedeutet der Aufruf einer Methode das Senden einer Botschaft an ein Objekt mit der Wirkung, daß für das Objekt diese Methode aufgerufen wird.

Die Methoden sind deklariert, aber noch nicht definiert. Normalerweise werden Methoden außerhalb der Klassendefinition definiert. Der Bezug zur Klasse wird durch Angabe des Namens der Klasse über den sogenannten ***Scope-***Operator `::` hergestellt:

```
// set.cpp – Zugriffsfunktion für Klausurnote

#include <iostream>
#include "Klausurnote.h"

using namespace std;

int Klausurnote::set (char* Note)
   {
   int n = -1;
   while (Note[++n] != '\0');
   if ( (n > 3) || (n == 0) || (n > 1 && Note[1] != ','))
```

```
    {
    cout << '\"' << Note << '\"';
    cout << " ist keine gültige Note!\n";
    return -1;
    }
  switch (Note[0])
    {
    case '5':
       if (n == 3 && Note[2] != '0')
         {
         cout << '\"' << Note << '\"';
         cout << " ist keine gültige Note!\n";
         return -1;
         }
    case '4': case '3': case '2': case '1':
       note[0] = Note[0];
       note[1] = ',';
       if (n == 3) note[2] = Note[2];
       else note[2] = '0';
       return 0;
    default:
       cout << '\"' << Note << '\"';
       cout << " ist keine gültige Note!\n";
       return -1;
    }
  }
```

Bemerkung 1: Eine Elementfunktion kann andere Elemente der Klasse ohne Dereferenzierung ansprechen. Z.B. `note` kann direkt benutzt werden. Doch wie wäre so etwas zu dereferenzieren? Mit jeder Klasse ist implizit der sogenannte **this**-Zeiger definiert, der auf das Objekt selbst zeigt:

`Klausurnote*const this;`

Der Zeiger ist konstant, d.h. sein Inhalt kann verändert werden, nicht aber seine Adresse, die bei der Definition eines Objekts der Klasse als die Adresse des Objekts initialisiert wurde. Somit könnte man statt `note` auch `this->note` verwenden, was hier nicht nötig ist, da für den Compiler klar ist, welches `note` gemeint ist. **this** ist ein Schlüsselwort von C++. Daher muß und kann **this** nicht explizit deklariert werden.

Der **this**-Zeiger wird für Methoden benötigt, die als Ergebnis die eigene Klasse zurückgeben (vgl. Abschnitt **this**-Zeiger).

Bemerkung 2: Die Manipulation von gekapselten Daten beruht auf Funktionsaufrufen. Ein extensiver Gebrauch dieses Konzeptes kann zu beträchtlichen Leistungseinbußen führen, da ein Funktionsaufruf wesentlich mehr Rechenzeit kostet als der direkte Code. In C konnte man sich mit Macrodefinitionen behelfen. In C++ gibt es die viel sicherere Möglichkeit der ***Inline***-Funktion, wobei der Compiler die Möglichkeit einer Typenüberprüfung hat. Methoden, die direkt in der Klassendefinition definiert werden, sind automatisch **inline**. Das empfiehlt sich

häufig für kurze `set-` und `get`-Funktionen. In unserem Beispiel könnte `druckeNumerisch ()` als Inline-Funktion implementiert werden:

```
#include <iostream>

class Klausurnote
   {
   public:
      int   set (char *Note);
      char* getNumerisch ();
      char* getVerbal ();
      void druckeNumerisch ()
         {
         cout << note[0] << note[1] << note[2];
         }
      void druckeVerbal ();
   private:
      char note[3];
   };
```

Außerhalb der Klassendefinition muß die Funktion als **inline** definiert werden, was wir an der Funktion `druckeNumerisch` zeigen:

```
inline void Klausurnote::druckeNumerisch ()
   {
   cout << note[0] << note[1] << note[2];
   }
```

Die meisten Compiler machen es von der Länge der Funktion abhängig, ob bei einer als **inline** deklarierten Funktion wirklich der Code an die entsprechende Stelle kopiert wird oder ein Funktionsaufruf verwendet wird (vgl Kapitel Funktionen). Eine **inline**-Funktion kann nur *nach* ihrer Deklaration als **inline** (Implementation) verwendet werden.

Bemerkung 3: Die Daten einer Klasse sind zugänglich über sogenannte Zugriffs- oder Access-Funktionen, z.B. `set..` und `get...`. Die `get`-Funktionen verändern häufig nicht das Objekt und könnten daher auch auf konstante Objekte angewendet werden. Aber für konstante Objekte ist jeglicher Methodenzugriff verboten, es sei denn die Methode wird durch Anhängen von **const** an die Argumentenliste explizit als eine Methode erklärt, die das Objekt unverändert läßt. Z.B. könnte `getNumerisch` folgendermaßen deklariert werden:

```
char* getNumerisch () const;
```

`getNumerisch` kann nun auch auf konstante Objekte angewendet werden.

Bemerkung 4: Ein Objekt, d.h. seine Elementfunktionen können auf die privaten Daten eines anderen Objekts *derselben* Klasse zugreifen.

Bemerkung 5: Auf den Scope-Operator `::` kann immer dann verzichtet werden, wenn klar ist, auf welche Klasse eine Funktion oder ein Datenelement sich bezieht.

Bemerkung 6: Wir unterscheiden **Anwender** ***(Client)*** und **Implementor** einer Klasse. Der Anwender ist derjenige, der die Klasse benutzt. Er sieht nur die Klassendefinition und kann die öffentlichen Teile **(Schnittstelle**, ***Interface)*** dieser Definition verwenden. Nachträgliche Änderungen der Klasse durch den Implementor sollten die Definition einer Klasse nicht berühren, da dies mindestens eine Rekompilation oder gar Änderung des Anwender-Codes bedeutet. Daher sollten auch Inline-Funktionen nur verwendet werden, wenn die Performanz es wirklich erfordert. Denn Änderung von Inline-Funktionen bedeutet immer Rekompilation von Anwender-Code.

Der Implementor muß damit rechnen, daß der Anwender alles verwendet, was im **public** Teil angeboten wird. Um für spätere Änderungen offen zu sein, sollte der Implementor nicht zu Spezifisches, insbesondere bezüglich der Datenstruktur, anbieten.

Änderung der Datenstruktur einer Klasse hat immer Rekompilation des Anwender-Codes zur Folge. Um dieses Problem kommt man nur herum, wenn man die Klasse in eine **äußere** und **innere** Klasse aufteilt, wobei die äußere Klasse nur einen Zeiger auf die innere Klasse hat. Die innere Klasse ist nur für den Implementor sichtbar, der dort auch Datenstrukturänderungen vornehmen kann, ohne den Anwender zur Rekompilation zu zwingen. Auch aus Geheimhaltungsgründen mag diese Vorgehensweise interessant sein.

6.1 Konstruktoren und Destruktoren

6.1.1 Konstruktoren

Wenn wir ein Objekt der Klasse `Klausurnote` anlegen, dann wird es – bisher jedenfalls – nicht initialisiert. Die Initialisierung von Datentypen sollte i.a. einerseits nicht vergessen werden, andererseits höchstens ein einziges Mal erfolgen.

Man möchte im wesentlichen zwei Dinge garantieren:

- Automatische, einmalige Initialisierung von Variablen
- Automatische Allokierung und Deallokierung von Speicher

Mit den normalen Access-Methoden kann dies nicht garantiert werden. Daher gibt es für Klassen spezielle Initialisierungsfunktionen, die sogenannten **Konstruktoren**. Ein Konstruktor wird bei der Definition eines Objekts automatisch aufgerufen.

Der Konstruktor ist eine Funktion, die *denselben* Namen wie die Klasse hat, *keinen* Rückgabewert liefert und nur bei der Definition eines Objekts der Klasse aufgerufen wird. Eine Klasse kann mehrere Konstruktoren haben, die zwar denselben Namen tragen, sich aber durch die Anzahl und Art der Parameter unterscheiden, d.h. unterschiedliche Signatur haben.

```
class Klassenname
   {
   public:
      Klassenname (int, int, int); // Konstruktor
      Klassenname (double);        // Konstruktor
      Klassenname ();              // Konstruktor
   protected:
   private:
   };
```

In unserem Notenbeispiel wollen wir bei Initialisierung eines Notenobjekts diesem Objekt entweder die zwar ungültige Note "0,0" zuordnen oder eine vom Benutzer bei der Definition des Notenobjekts angegebene korrekte Note zuordnen. Dazu definieren wir zwei Konstruktoren:

```
class Klausurnote
   {
   public:
      Klausurnote ();
      Klausurnote (char* Note);
      int   set (char *Note);
      char* getNumerisch ();
      char* getVerbal ();
      void druckeNumerisch ()
         {
         cout << note[0] << note[1] << note[2];
         }
      void druckeVerbal ();
   private:
      char note[3];
   };
```

```
#include "Klausurnote.h"

Klausurnote::Klausurnote ()
   {
   note[0] = '0';
   note[1] = ',';
   note[2] = '0';
   }

Klausurnote::Klausurnote (char* Note)
   {
   if (set (Note) == -1)
      {
      note[0] = '0';
      note[1] = ',';
      note[2] = '0';
      }
   }
```

Die Definition von Objekten sieht nun folgendermaßen aus:

```
Klausurnote bio; // Verwendung des ersten Konstruktors

Klausurnote phy = Klausurnote ("2,3"); // Verwendung des
                                       // zweiten Konstruktors

Klausurnote che ("2,3"); // Verwendung des zweiten Konstruktors,
                         // aber kürzer und üblicher

Klausurnote mat = "2,3"; // Initialisierung , keine Zuweisung
                         // Verwendung des zweiten Konstruktors,
```

Anonyme Objekte einer Klasse können erzeugt werden, indem man einfach einen der Konstruktoren aufruft. Wenn z.B. der Rückgabewert einer Funktion vom Typ `Klausurnote` ist, kann das etwa folgendermaßen aussehen:

```
Klausurnote korrigiere ()
   {
   // ---
   if (Fehler == viele) return Klausurnote ("5,0");
   // ---
   }
```

Damit die Verwendung der Konstruktoren eindeutig bleibt, dürfen zwei Konstruktoren niemals die gleiche Signatur haben.

Anstatt zwei Konstruktoren zu definieren, hätte einer mit einem Defaultargument genügt:

```
Klausurnote (char* Note = "0,0");
```

Defaultargument bedeutet, daß der angegebene Wert als Argument genommen wird, wenn kein Argument übergeben wird. (In unserem Beispiel suggerieren wir aber dem Anwender, daß er die Note "0,0" übergeben kann, und wir können ihm eine solche Eingabe auch nicht verbieten bzw als Fehleingabe erkennen.)

Bei mehreren Argumenten können ab einem gewissen Argument alle weiteren weggelassen werden, sofern dort Defaultwerte angegeben sind. Man kann Argumente nicht selektiv weglassen. Der Konstruktor

```
Klassenname (int A = 12, int B = 7, int C = 1);
```

kann z.B. so verwendet werden:

```
Klassenname objekt (34);
```

Dabei wird `A` mit `34`, `B` mit `7` und `C` mit `1` übergeben.

Die verschiedenen überladenen Konstruktoren sollten sich ähnlich verhalten. Sie sollten *konsistente* Objekte kreieren. Ansonsten müßten komplizierte Unterscheidungen bei anderen Elementfunktionen getroffen werden. Wenn z.B. ein Konstruktor Speicher allokiert, dann sollten alle anderen Konstruktoren dieser Klasse auch Speicher allokieren, da sonst z.B. der Destruktor erst prüfen müßte, ob Speicher angelegt ist, ehe er Speicher deallokiert.

6.1.2 Default-Konstruktor

Ein Konstruktor ohne Argumente oder ein Konstruktor, bei dem alle Argumente einen Defaultwert haben, heißt *Default-Konstruktor*. Der Default-Konstruktor wird vom Compiler *nur* dann automatisch erzeugt, wenn es keine vom Klassen-Implementor definierte Konstruktoren gibt.

6.1.3 Copy-Konstruktor

Der Konstruktor, der als Argument ein Objekt derselben Klasse nimmt, heißt Copy-Konstruktur.

```
Klassenname (const Klassenname& x);
```

Das Argument des Copy-Konstruktors muß immer als Referenz deklariert werden. Denn vor ihrer schließenden Klammer ist eine Klasse nicht definiert, aber der Name der Klasse ist bekannt. Das hat zur Folge, daß (nichtstatische) Objekte der Klasse innerhalb derselben Klasse nicht definiert werden können. Es können nur Zeiger- und Referenztypen definiert werden. Wenn der Copy-Konstruktor nicht vom Klassen-Implementor definiert wird, dann wird er vom Compiler erzeugt, (auch wenn es andere Konstruktoren gibt).

6.1.4 Destruktor

Wenn ein Objekt seinen Scope verläßt, sind häufig gewisse Aufräumarbeiten zu erledigen. Z.B. muß allokierter Speicher wieder deallokiert werden. Damit dies automatisch und für den Benutzer transparent passiert, kann ein Destruktor

definiert werden. Der Destruktor wird automatisch aufgerufen, wenn ein Objekt seinen Lebensbereich verläßt. Der Name des Destruktors ist vorgegeben, nämlich als der Klassenname mit vorangestellter Tilde ~ :

```
~Klassenname
```

```
class Klassenname
   {
   public:
      Klassenname (int, int, int); // Konstruktor
      Klassenname (double);        // Konstruktor
      Klassenname ();              // Konstruktor
      ~Klassenname();              // Destruktor
   protected:
   private:
   };
```

Der Destruktor hat niemals Argumente. Sinnvolle Destruktoren werden wir noch kennenlernen. In unser Beispiel bauen wir einen eher sinnlosen Destruktor zur Demonstration der Syntax ein:

```
Klausurnote::~Klausurnote ()
   {
   std::cout << "Die Note ";
   druckeNumerisch ();
   std::cout << " wird jetzt deinitialisiert!\n";
   }
```

6.1.5 Bemerkungen

1. Wenn irgendein Konstruktor vom Implementor definiert wurde, dann gibt es den Defaultkonstruktor *nicht, wenn* ihn der Implementor nicht definiert.

2. Den Copy-Konstruktor dagegen gibt es immer; entweder den vom Implementor definierten oder einen Default-Copy-Konstruktor, der aber immer eine "shallow" Kopie macht. Da der Copy-Konstruktor häufig implizit – etwa bei Übergabe von Funktionsargumenten – aufgerufen wird, sollte man ihn definieren oder verbieten. Man kann ihn verbieten, indem man ihn im privaten Teil der Klasse deklariert und nicht implementiert.

3. Ohne hinreichenden Grund sollten Default-Konstruktor, Copy-Konstruktor und Destruktor für eine Klasse *nicht* definiert werden, da der Compiler automatisch diese Funktionen zur Verfügung stellt. Das ist *eine* in der Literatur vertretene Meinung.

4. Neuerdings wird die **kanonische Form** einer Klasse empfohlen: In einer Klasse sollte immer der Default-Konstruktor, der Copy-Konstruktor, der Zuweisungsoperator und der Destruktor definiert werden.

   ```
   class Klassenname
      {
   ```

```
public:
   Klassenname ();
   Klassenname (const Klassenname& x);
   Klassenname& operator = (const Klassenname& x);
   ~Klassenname ();
};
```

5. Initialisierung bedeutet, daß ein Konstruktor aufgerufen wird. Eine Initialisierung kann nur genau einmal für ein Objekt durchgeführt werden.

   Zuweisung bedeutet, daß der Zuweisungsoperator aufgerufen wird. Einem Objekt kann beliebig oft zugewiesen werden. (Das Objekt muß natürlich vor der Zuweisung existieren.)

   Bei Initialisierung sollte der Deutlichkeit halber eine Konstruktoraufruf-Syntax verwendet werden:

   ```
   int   i (5);  // anstatt von: int i = 5;

   for (int j (0); j < 10; j++);
   ```

   oder

   ```
   class A
      {
      public:
         A (int x, int y);
      private:
         int n;
         int m;
         int k;
      };

   A::A (int x, int y)
      : n (x),
        m (y),
        k (5)
      {
      // anstatt von: n = x;
      // anstatt von: m = y;
      // anstatt von: k = 5;
      }
   ```

6. Objekte *derselben* Klasse haben gegenseitig Zugriff auf die **private** und **protected** Klassenelemente.

7. Speicherplatzverbrauch: Beim Anlegen eines Objekts einer Klasse wird *nur* für die Datenelemnete Speicher angelegt. Der Speicherplatzbedarf eines Objekts ist unabhängig von der Anzahl der Methoden der Klasse.

6.2 Friends

Der Zugang zu den privaten Daten einer Klasse nur über `get`-Funktionen ist sehr sicher, kann aber für Funktionen, die intensiv mit den privaten Daten *verschiedener* Klassen arbeiten sehr ineffektiv werden. Außerdem kann der Export der Datenrepräsentation über `set` und `get`-Funktionen so detailliert werden, daß die Datenrepräsentation schwer änderbar wird. Wenn man im Beispiel `Klausurnote` Methoden wie

```
char getNotenZahlVorDemKomma () { return note[0]; }
```

zur Verfügung stellen würde, dann wäre eine Reimplementation der Datenrepräsentation als **int** von `10` bis `50` nur unschön möglich.

Als Beispiel betrachten wir die Funktion `durchschnitt`, die aus `n` Noten, die in einem Feld `noten[]` übergeben werden, eine Durchschnittsnote errechnen soll.

```
double durchschnitt (int n, Klausurnote noten[]);
```

Diese Funktion wollen wir nicht zu einem Element der Klasse `Klausurnote` machen, da wir sie ohne Referenz auf ein spezielles Notenobjekt aufrufen wollen. Sie wird implementiert:

```
#include "Klausurnote.h"

double  durchschnitt (int n, Klausurnote noten[])
   {
   double  s = 0.0;
   for (int i (0); i < n; i++)
      {
      s = s + noten[i].note[0] - '0'
         + (noten[i].note[2] - '0') / 10.0;
      }
   return s / n;
   }
```

`durchschnitt` greift auf das private Element `note` zu, was der Compiler verbieten wird. Es gibt aber die syntaktische Möglichkeit, einer Funktion den direkten Zugriff auf die privaten Daten einer Klasse zu erlauben, indem man sie in der betreffenden Klasse als **friend** deklariert.

```
friend double durchschnitt (int n, Klausurnote noten[]);
```

Es spielt keine Rolle, wo – ob im privaten oder öffentlichen Teil – die **friend**-Funktion deklariert wird. Die **friend**-Funktion hat Zugriff auf die privaten Elemente einer Klasse, muß diese aber dereferenzieren.

Auch Elementfunktionen einer Klasse können Freunde einer anderen Klasse sein

```
class A
   {
   // ---
   void f();
   };

class B
   {
   friend void A::f();
   // ---
   };
```

Wenn alle Elementfunktionen einer Klasse Freunde einer anderen Klasse werden sollen, was durchaus üblich ist, dann kann die ganze Klasse zum Freund erklärt werden.

```
class B
   {
   friend class A;
   // ---
   };
```

Das **friend**-Konzept sollte möglichst sparsam verwendet werden, da es das Konzept der Datenkapselung verletzt. Daher empfiehlt es sich, `friends` noch vor dem **public**-Teil der Klassendefinition zu deklarieren.

Funktionen, die **friend** mehr als einer Klasse sind, sollten vermieden werden, da durch solch eine Funktion die betroffenen Klassen gekoppelt werden.

friend-Funktionen werden nicht vererbt (siehe Kapitel Vererbung).

6.3 Klassen-Scope-Operator ::

Es gibt Fälle, wo der Name eines Klassenelements unterschieden werden muß von anderen Namen. Häufig möchte man in einer z.B. `set`-Funktion denselben Namen als Parameter verwenden, der auch in der Klasse verwendet wird:

```
class A
   {
   public:
      void setm (int m) { A::m = m; }
      // oder besser
      void setm (int m) { this->m = m; }
   private:
      int m;
   };
```

Das Element `m` der Klasse wird durch den Operator `::` oder durch den **this**-Zeiger von dem Funktionsargument `m` unterschieden.

Ein Name mit vorangestelltem Operator :: ohne Klassenbezug bezieht sich immer auf einen global definierten Namen.

```
int m;

class A
   {
   public:
      int vergleiche_m () { return ::m == m; }
   private:
      int m;
   };
```

Das ist insbesondere dann nützlich, wenn man gleiche Namen für Elementfunktionen und globale Systemfunktionen hat und die Systemfunktionen in der Klasse auch noch verwenden will.

```
class Rechner
   {
   public:
      void pow (double x, double y);
   // ---
   };

void Rechner::pow (double x, double y)
   {
   double z = ::pow (x,y);
   cout << "Die Potenz von " << x << " mit " << y;
   cout << " beträgt: " << z << "\n";
   }
```

6.4 Verschachtelte Klassen

Klassendefinitionen können verschachtelt werden, d.h. innerhalb einer Klasse können weitere Klassen **public**, **protected** oder **private** deklariert werden.

```
class A
   {
   public:
      // ---
      class B
         {
         public:
            int m;
         };
   private:
      // ---
      class C
         {
         public:
            int m;
         private:
            int n;
         };
   };
```

Die Namen der Klassen `B` und `C` stehen nur im Scope von Klasse `A` zur Verfügung. Solche Konstrukte werden daher i.a. nur für sehr kleine, unbedeutende Klassen verwendet. Größere Klassen sollten eher separat deklariert werden und dann als **friend** in der Klasse `A` geführt werden. Die Verschachtelung hat aber den Vorteil, daß die Anzahl der globalen Namen niedrig gehalten wird.

Außerhalb des Scopes von Klasse `A` kann `B` als `A::B` verwendet werden:

```
A::B b;
```

`C` kann nicht außerhalb von `A` verwendet werden, da es **private** ist.

6.5 Statische Klassenmitglieder

Eine Klasse ist ein Typ und jedes Objekt einer Klasse hat eine eigene Kopie der Datenelemente der Klasse. Aber es kann sein, daß manche Klassen so implementiert werden sollen, daß alle Objekte einer Klasse dasselbe Datenelement benutzen. Das können z.B. gemeinsame Zähler etwa für die Anzahl der Objekte dieser Klasse sein oder generell Daten, auf die jedes Objekt zugreift und die sich mit der Zeit ändern, sodaß es nicht möglich wäre, eine Konstante daraus zu machen. Um solche Daten als Klassenmitglieder zu verwalten gibt es das Schlüsselwort **static**. Statische Datenelemente heißen auch **Klassenvariable** ***(class variable)***.

```
class A
   {
   // ---
   static int zaehler;
   static int feld[3];
   // ---
   };
```

Durch die **static**-Deklaration wird erreicht, daß `zaehler` und das Feld `feld` für alle Objekte der Klasse `A` nur einmal angelegt wird. Innerhalb des Scopes einer Klasse kann ein statisches Datenmitglied wie jedes andere Element direkt mit seinem Namen angesprochen werden. Außerhalb des Scopes einer Klasse gibt es zwei Möglichkeiten sofern das statische Element **public** ist.

```
A  a;
a.zaehler = 5;  // Zugriff über ein Objekt
A::zaehler = 6; // Zugriff über Klassenname
                // Definition eines Objektes ist daher
                // nicht unbedingt erforderlich
```

Damit kann man die Anzahl der globalen Variablen beträchtlich reduzieren. Die Initialisierung von statischen Variablen erfolgt außerhalb der Klassendefinition typischerweise im `.cpp`-File (*nicht* im Header-File) der entsprechenden Klasse zusammen mit den Definitionen der nicht-**inline**-Elementfunktionen.

```
int A::zaehler = 0;
int A::feld = { 5, 3, 9 };
```

Statische Klassenelemente müssen initialisiert werden, da sie sonst nicht angelegt werden. Die Deklaration von statischen Elementen einer Klasse ist nicht zu verwechslen mit der Definition von statischen Objekten:

```
static A a;
```

Statische im Gegensatz zu automatischen Objekten bleiben ab ihrer Definition bis zum Ende des Programms erhalten, auch wenn etwa der Block verlassen wird, in dem sie definiert wurden.

Auch Elementfunktionen können als **static** deklariert werden (**Klassenmethoden**, ***class methods)***. Das macht dann Sinn, wenn die Funktion nur auf statische Daten zugreift, sodaß sie auch ohne Referenz auf ein Objekt der Klasse aufgerufen werden kann.

```
class A
   {
   public:
      static void setZaehler (int i) { zaehler = i; }
      static int getZaehler () { return zaehler; }
      // ---
   private:
      static int zaehler;
      // ---
   };

// ---
int i = A::getZaehler ();
A::setZaehler (5);

// oder
A a;
i = a.getZaehler ();
a.setZaehler (5);
```

Für statische Elementfunktionen ist der **this**-Zeiger nicht definiert. Ein statisches Datenmitglied kann als Defaultargument einer Elementfunktion erscheinen, wenn es vor der Elementfunktion definiert wurde. Ferner kann ein statisches Objekt einer Klasse Element der eigenen Klasse sein.

6.6 Konstanten und Klassen

Leider ist es nicht erlaubt Konstanten innerhalb von Klassen zu definieren (Angabe des Wertes innerhalb der Klasse), um damit etwa Felder zu initialisieren oder weil Konstanten häufig einen engen Bezug zu einer Klasse haben. (Die Deklaration von Konstanten ist allerdings möglich.)

```
class A
   {
   static const int bufgroesse = 50; // Fehler
   char buf[bufgroesse];
   };
```

Konstanten müssen außerhalb der Klasse definiert werden:

```
const int  bufgroesse = 50;

class A
   {
   char buf[bufgroesse];
   };
```

Das hat aber den Nachteil, daß `bufgroesse` keine Beziehung zur Klasse `A` hat. Für andere Puffer mit anderen Puffergrößen müssen dann immer wieder neue Namen für `bufgroesse` erfunden werden.

```
const int  bufgroesseA = 50;
const int  bufgroesseB = 70;

class A
   {
   char buf[bufgroesseA];
   };

class B
   {
   char buf[bufgroesseB];
   };
```

Das ist sehr lästig. Der Ausweg über eine statische Konstante, die außerhalb der Klasse initialisiert wird, scheitert auch:

```
class A
   {
   static const int bufgroesse;
   char buf[bufgroesse];
   };

const int A::bufgroesse = 50;
```

Hier wird `buf` falsch (mit `0`) initialisiert, da `bufgroesse` später initialisiert wird.

Für **int**-Konstanten gibt es eine Art Workaround, indem die Konstante als Element eines anonymen **enum** deklariert wird, da **enum**-Elemente innerhalb der Klassendefinition mit einem **int**-Wert initialisiert werden dürfen.

```
class A
   {
   enum {bufgroesse = 50};
   char buf[bufgroesse];
   };
```

Bemerkung: Die **static** definierte Konstante wird für alle Objekte der Klasse nur einmal angelegt. Bei einem nicht **static** definierten konstanten Klassenelement, wird dieses für *jedes* Objekt angelegt. Es kann dann im Konstruktor für jedes Objekt verschieden definiert werden:

```
class A
   {
   const int iii;
   A (int j) : iii (j) {}
   };
```

Wenn eine Elementfunktion für ein konstantes Objekt aufgerufen werden soll, dann muß diese Funktion als **const** deklariert werden.

```
class A
   {
   public:
      int   f () const;
      // ---
   };
```

Dieses **const** gehört zur Signatur der Funktion. D.h. `f` kann mit und ohne **const** definiert werden, falls z.B. das Verhalten für konstante Objekte anders sein soll. Natürlich sollten nur die Elementfunktionen als **const** definiert werden, die sicher auf konstante Objekte anwendbar sind. In dem Fall *sollten* diese Funktionen auch als **const** definiert werden.

Nur Elementfunktionen können als **const** deklariert werden. Globale und auch **friend**-Funktionen können nicht als **const** deklariert werden.

6.7 this-Zeiger

Der Zeiger **this** ist ein Zeiger, der auf das Objekt selbst zeigt. Er ist immer definiert, sodaß Elementfunktionen diesen Zeiger verwenden können. Er wird benützt, um Klassenelemente eindeutig anzusprechen.

```
class A
   {
   int   i;
   void  f (A a)
      {
      this->i = a.i; // eventuell klarer für Leser
      i = a.i;       // ist äquvalent
      }
   };
```

Häufiger wird der **this**-Zeiger verwendet, um das eigene Objekt als Funktionswert zu liefern.

```
class A
   {
   int   i;
   A& f ()
      {
      // ---
      return *this;
      }
   A* g ()
      {
      // ---
      return this;
      }
   };
```

Ein typisches Beispiel ist der Zuweisungsoperator.

Bemerkung: Der **this**-Zeiger eines *variablen* Objekts der Klasse `A` ist vom Typ `A*const`. Der **this**-Zeiger eines *konstanten* Objekts der Klasse `A` ist vom typ **const** `A*const`. Will man z.B. ein Datenelement `i` eines konstanten Objekts vom Typ `A` verändern (z.B. `i = 567`), dann muß man den **this**-Zeiger von **const** `A*const` nach `A*const` casten:

```
( (A*const)this)->i = 567;
```

6.8 Beispiel Zweidimensionale Vektoren

Ein einfaches Beispiel für die Bildung einer Klasse sind zweidimensionale Vektoren. Die Datenrepräsentation des Vektors sind zwei Zahlen vom Typ **double** `x1` und `x2`.

Header-File: `ZweiDimVek.h`

```
// ZweiDimVek.h

#ifndef ZweiDimVek_h
#define ZweiDimVek_h

class ZweiDimVek {
   public:
      ZweiDimVek ();          // Default-Konstruktor
      ZweiDimVek (double a, double b); // Konstruktor
      ZweiDimVek (const ZweiDimVek& v); // Copy-Konstruktor
      double  getX1 () const { return x1; }
      double  getX2 () const { return x2; }
      void  setX1 (double a) { x1 = a; }
      void  setX2 (double a) { x2 = a; }
      double  skalarProdukt (const ZweiDimVek& v2) const;
      double  betrag () const;
```

```
    void drucke () const;
  private:
    double x1;  double x2;
  };

#endif
```

Implementations-File: `ZweiDimVek.cpp`

```
#include <iostream>
#include <cmath>
#include "ZweiDimVek.h"

ZweiDimVek::ZweiDimVek ()
  {
  x1 = 0.0;
  x2 = 0.0;
  }

ZweiDimVek::ZweiDimVek (double a, double b)
  {
  x1 = a;
  x2 = b;
  }

ZweiDimVek::ZweiDimVek (const ZweiDimVek& v)
  {
  x1 = v.x1;
  x2 = v.x2;
  }

double ZweiDimVek::skalarProdukt (const ZweiDimVek& v2) const
  {
  return x1 * v2.x1 + x2 * v2.x2;
  }

double ZweiDimVek::betrag () const
  {
  return sqrt (x1 * x1 + x2 * x2);
  }

void ZweiDimVek::drucke () const
  {
  std::cout << '(' << x1 << ", " << x2 << ")\n";
  }
```

Anwendungs-File: `apl.cpp`

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

```
using namespace std;

main ()
   {
   ZweiDimVek v;
   ZweiDimVek v1 (2.0, 2.0);
   ZweiDimVek v2 (2.0, 3.0);
   ZweiDimVek v3 (v1);
   cout << "v: "; v.drucke ();
   cout << "v1: "; v1.drucke ();
   cout << "v2: "; v2.drucke (); cout << "v3: "; v3.drucke ();
   cout << "Das Skalarprodukt v1*v2 ist ";
   cout << v1.skalarProdukt (v2) << "\n";
   cout << "Der Betrag von v3 ist " << v3.betrag () << "\n";
   return 0;
   }
```

6.9 Beispiel Zeichenkette

Oft werden wir Strings benötigen. Die Strings sind in C++ wie in C sehr primitiv und fehlerträchtig implementiert. Das wollen wir durch Definition einer Stringklasse etwas verbessern.

Die Klasse soll als Namen die deutsche Bezeichnung für Strings haben: `Zeichenkette`. Unser String kann im Prinzip alle 256 ASCII-Zeichen enthalten. Daher müssen wir von der C-Konvention, daß ein String durch `'\0'` beendet wird, abgehen. Stattdessen werden wir uns die Anzahl der Bytes im String in einem besonderen Datenelement `anzZeich` der Klasse merken. Die einzelnen Zeichen des Strings werden in einem dynamischen **char**-Feld `z` verwaltet. Somit sieht unsere Stringklasse zunächst folgendermaßen aus.

```
class Zeichenkette
   {
   public:
   private:
      int   anzZeich;
      char* z;
   };
```

Dem Prinzip der Datenkapselung folgend, werden die Daten im privaten Teil der Klasse geführt. Die Daten werden belegt, wenn wir ein Objekt vom Typ Zeichenkette anlegen. Dabei wollen wir einen normalen C-String oder eine unserer Zeichenketten zur Initialisierung angeben können. Damit dies möglich ist, müssen wir für die verschiedenen Fälle Konstruktoren anbieten.

```
// Zeichenkette.h
#ifndef Zeichenkette_h
#define Zeichenkette_h
```

```
#include <iostream>

using namespace std;

class Zeichenkette
   {
   public:
      Zeichenkette ()                        // Defaultkonstruktor
         : anzZeich (0)
         {
         z = new char[1];
         }
      Zeichenkette (const char* C_String);
      Zeichenkette (const Zeichenkette& Z); // Copykonstruktor
      ~Zeichenkette () { delete [] z; }   // Destruktor
      void zeige () const; // Gibt String auf stdout aus mit
                           // Kommentar und Zeilenvorschub.
   private:
      int   anzZeich;
      char* z;
   };
#endif

// Zeichenkette.cpp
#include <iostream>
#include "Zeichenkette.h"

using namespace std;

Zeichenkette::Zeichenkette (const char* C_String)
   {
   // zähle Zeichen in C_String
   int   i = 0;
   while (C_String[i] != '\0') i++;
   anzZeich = i;
   if (anzZeich > 0)
      {
      // Kopiere String
      z = new char[anzZeich];
      for (i = 0; i < anzZeich; i++) z[i] = C_String[i];
      }
   else z = new char[1]; // ,damit leere Zeichenketten
                         // bezüglich Zerstörung
                         // konsistent sind.
   }

Zeichenkette::Zeichenkette (const Zeichenkette& Z)
   {
   if (Z.anzZeich > 0)
      {
```

```
      anzZeich = Z.anzZeich;
      // Kopiere String
      z = new char[anzZeich];
      for (int i = 0; i < anzZeich; i++) z[i] = Z.z[i];
      }
   else
      {
      anzZeich = 0;
      z = new char[1];
      }
   }

void Zeichenkette::zeige () const
   {
   cout << "Zeichenkette: " << '"';
   if (anzZeich > 0)
      for (int i = 0; i < anzZeich; i++) cout << z[i];
   cout << '"' << " hat ";
   cout << anzZeich << " Zeichen.\n";
   }

// Testprogramm für Klasse Zeichenkette
#include "Zeichenkette.h"

main ()
   {
   Zeichenkette a;
   Zeichenkette b ("Hello World");
   Zeichenkette c (b);
   a.zeige ();
   b.zeige ();
   c.zeige ();
   return 0;
   }
```

6.10 Übungen

Übung Konstruktoren der Klasse `Rechteck`: Ersetzen Sie die in der Klasse `Rechteck` definierte Methode `initialisieren ()` durch einen Konstruktor (Siehe Übung des Kapitels "Einleitung").

Übung Methoden Klausurnote:

1. Schreibe für die Klasse `Klausurnote` einen Konstruktor, der ein **double** als Argument zur Initialisierung der Note nimmt, und wende ihn an.

2. Schreibe für die Klasse `Klausurnote` einen Copy-Konstruktor und wende ihn an.

3. Schreibe für die Klasse `Klausurnote` die Methode `plus`, mit der zwei Klausurnoten addiert werden, wobei das geschnittene Mittel gebildet wird $(3,3+3,8=3,5)$.

   Anwendung:

   ```
   Klausurnote a, b, c;
   c = a.plus (b);
   ```

4. Schreibe für die Klasse `Klausurnote` die **friend**-Funktion `plusFr`, mit der zwei Klausurnoten addiert werden, wobei das geschnittene Mittel gebildet wird.

   Anwendung:

   ```
   Klausurnote a, b, c;
   c = plusFr (a, b);
   ```

   Kann diese Funktion auch so

   ```
   c = plusFr (1.6, 2.8);
   ```

   angewendet werden? Und wenn ja, warum? Ausprobieren!

5. Schreibe für die Klasse `Klausurnote` die Funktion `plusFr` als Methode `plus2` der Klasse, aber auch mit zwei Argumenten vom Typ der Klasse.

   Anwendung:

   ```
   Klausurnote a, b, c;
   c = ?;
   ```

6. Schreibe die Methode `plus2` so, daß das aufrufende Objekt das Resultat der Addition ist und daß das Resultat als Referenz zurückgegeben wird.

7. Zähle die instanziierten Klausurnotenobjekte mit:

   (a) Alle jemals angelegten Objekte.

   (b) Nur die, die gerade noch angelegt sind.

   (c) Schreibe die statische Methode, die die Zählerstände auf dem Bildschirm ausgibt.

   (d) Wenn ein Objekt angelegt oder aufgegeben wird, dann gib die Adresse des Objekt aus.

Übung Konstruktoren: Die Notenangabe "`2,3`" ist sehr umständlich und damit fehleranfällig. Wir wollen das absichern, indem wir auch Notenangaben wie `2.3` und `2,3` (ohne Anführungsstriche) erlauben. Schreibe dafür geeignete Konstruktoren und u.U. Überladungen der Elementfunktion `set`.

Übung Zeichenkettenlänge: Implementiere die Methode `laenge ()`, die die Länge einer Zeichenkette zurückgibt.

Übung Zeichenkettenkonstruktor (a): Implementiere einen Konstruktor, der eine Zeichenkette mit einem **char** initialisiert.

Übung Zeichenkettenkonstruktor (b): Implementiere einen Konstruktor, der eine Zeichenkette mit n gleichen **char** initialisiert. Wie kann man (a) und (b) zu einem Konstruktor machen?

Übung Zeichenkettenmethode `enthaelt`: Implementiere die Methode `enthaelt (Zeichenkette& z)`, die angibt, wie häufig die Zeichenkette `z` in einer Zeichenkette enthalten ist.

Übung Komplexe Zahlen: Definiere eine Klasse `Komplex` zur Darstellung von komplexen Zahlen.

Kapitel 7

Operatoren

In diesem Kapitel werden wir die von C++ zur Verfügung gestellten Operatoren betrachten, mit denen Daten verändert und verknüpft werden können. Im ersten Abschnitt werden die Eigenschaften der vordefinierten Operatoren behandelt. Im zweiten Abschnitt wird gezeigt, wie man die Bedeutung dieser Operatoren verändern kann.

7.1 Operatoren und ihre Hierarchie

Operatoren können aus ein bis drei Zeichen bestehen, z.B. `+` oder `++` . *Unäre* Operatoren wirken auf *ein* Datenelement. Sie sind *rechts*-assoziativ, d.h. `-~a` wird als `-(~a)` nicht als `~(-a)` interpretiert.

Die *binären* Operatoren verknüpfen *zwei* Datenelemente und sind alle – mit Ausnahme der Zuweisungsoperatoren – *links*-assoziativ.

`a + b + c + d` bedeutet `( (a + b)+ c) + d` .

Aber `a = b = c = d = 1` bedeutet `a = (b = (c = (d = 1)))` .

`a *= b += c = d -= 1` bedeutet `a *= (b += (c = (d -= 1)))` .

Es gibt einen einzigen *ternären* Operator (`x ? y : z`).

In der folgenden Tabelle sind alle vordefinierten Operatoren mit abnehmender Präzedenz aufgeführt. Operatoren in einem Kasten haben gleiche Präzedenz. Die Präzedenzregeln sind so gemacht, daß sie weitgehend "natürlich" funktionieren. Welche Komponente bei binären Operatoren zuerst ausgewertet wird, ist unbestimmt:

```
int   a = 0;
(a *= 2) == a++;
```

Nach diesen Statements kann `a` entweder gleich `1` oder `2` sein. Nur die Operatoren "`&&`", "`||`" und "`,`" garantieren, daß der linke Ausdruck zuerst ausgewertet wird.

`::`	Scope Resolution	*Klasse*`::`*Element*
`::`	global	`::`*Name*
`.`	Elementselektion	*Objekt*`.`*Element*
`->`	Elementselektion	*Zeiger* `->`*Element*
`[]`	Subskript	*Zeiger*`[`*Ausdr*`]`
`()`	Funktionsaufruf	*Ausdr*`(`*Ausdr_Liste*`)`
`()`	Wertkonstruktion	*Typ*`(`*Ausdr_Liste*`)`
`sizeof`	Größe eines Objekts	`sizeof` *Ausdr*
`sizeof`	Größe eines Typs	`sizeof` *Typ*
`++`	Nachinkrementierung	*Variable* `++`
`++`	Vorinkrementierung	`++`*Variable*
`--`	Nachinkrementierung	*Variable* `--`
`--`	Vorinkrementierung	`--`*Variable*
`~`	Komplement (bitweis)	`~`*Ausdr*
`!`	logisches Nicht	`!`*Ausdr*
`-`	unäres Minus	`-`*Ausdr*
`+`	unäres Plus	`+`*Ausdr*
`&`	Adresse von	`&`*Variable*
`*`	Dereferenzierung	`*`*Ausdr*
`new`	erzeuge (allokiere)	`new` *Typ*
`delete`	vernichte (deallokiere)	`delete` *Zeiger*
`delete[]`	vernichte Feld	`delete[]` *Zeiger*
`()`	Cast (Typ Konversion)	`(`*Typ*`)`Ausdr
`.*`	Elementselektion	*Objekt*`.*`*Zeiger_auf_Element*
`->*`	Elementselektion	*Zeiger* `->*`*Zeiger_auf_Element*
`*`	Multiplikation	*Ausdr* `*` *Ausdr*
`/`	Division	*Ausdr* `/` *Ausdr*
`%`	Modulo	*Ausdr* `%` *Ausdr*
`+`	Addition	*Ausdr* `+` *Ausdr*
`-`	Subtraktion	*Ausdr* `-` *Ausdr*

`<<`	Linksverschiebung (bitweis)	*Ausdr* `<<` *Ausdr*
`>>`	Rechtsverschiebung (bitweis)	*Ausdr* `>>` *Ausdr*
`<`	kleiner als	*Ausdr* `<` *Ausdr*
`<=`	kleiner oder gleich als	*Ausdr* `<=` *Ausdr*
`>`	größer als	*Ausdr* `>` *Ausdr*
`>=`	größer oder gleich als	*Ausdr* `>=` *Ausdr*
`==`	gleich	*Ausdr* `==` *Ausdr*
`!=`	ungleich	*Ausdr* `!=` *Ausdr*
`&`	bitweises Und	*Ausdr* `&` *Ausdr*
`^`	bitweises exklusives Oder	*Ausdr* `^` *Ausdr*
`|`	bitweises inklusives Oder	*Ausdr* `|` *Ausdr*
`&&`	logisches Und	*Ausdr* `&&` *Ausdr*
`||`	logisches inklusives Oder	*Ausdr* `||` *Ausdr*
`? :`	bedingter Ausdruck (ternär)	*Ausdr* `?` *Ausdr* `:` *Ausdr*
`=`	einfache Zuweisung	*Variable* `=` *Ausdr*
`*=`	Multiplikation und Zuweisung	*Variable* `*=` *Ausdr*
`/=`	Division und Zuweisung	*Variable* `/=` *Ausdr*
`%=`	Modulo und Zuweisung	*Variable* `%=` *Ausdr*
`+=`	Addition und Zuweisung	*Variable* `+=` *Ausdr*
`-=`	Subtraktion und Zuweisung	*Variable* `-=` *Ausdr*
`<<=`	Linksverschiebung und Zuweisung	*Variable* `<<=` *Ausdr*
`>>=`	Rechtsverschiebung und Zuweisung	*Variable* `>>=` *Ausdr*
`&=`	bitweises Und und Zuweisung	*Variable* `&=` *Ausdr*
`^=`	bitweises excl. Oder und Zuweisung	*Variable* `^=` *Ausdr*
`|=`	bitweises incl. Oder und Zuweisung	*Variable* `|=` *Ausdr*
`,`	Komma	*Ausdr* `,` *Ausdr*

Eine ausführliche Beschreibung aller Operatoren ist Sache eines Handbuchs, wie sie im Buch von Stroustrup zu finden ist. Einige Operatoren dürften bekannt sein, einige werden wegen ihrer speziellen Funktion in anderen Kapiteln erklärt. Daher begnügen wir uns hier mit kurzen Bemerkungen zu einigen eher C bzw C++ spezifischen Eigenschaften.

Der Operator "`->`" in `p->a` ist eine Abkürzung für `(*p).a` .

`a++` innerhalb eines Ausdrucks bedeutet, daß der Ausdruck erst mit dem ursprünglichen Wert von `a` ausgewertet wird und dann `a` um `1` erhöht wird. Bei `++a` innerhalb eines Ausdrucks wird `a` erst um `1` erhöht, und dann der Ausdruck mit dem erhöhten Wert von `a` ausgewertet. Entsprechendes gilt für den Operator "`--`" . Moderne Compiler sind allerdings so gut, daß man auf diese manchmal schwer verständliche Schreibweise zugunsten einer ausführlicheren Schreibweise verzichten kann. Sie ist eigentlich nur noch praktisch bei **for**-Schleifen (**for** `(i = 0; i < 5; i++)`). Dort ist es übrigens gleichgültig, ob `i++` oder `++i` verwendet wird.

Bei der In(De)krementierung von Zeigern ist zu beachten, daß in Einheiten der Größe des Typs, auf den der Zeiger zeigt, in(de)krementiert wird.

Der Operand von "`~`" muß ganzzahlig sein. Das Resultat ist das bitweise Komplement des Operanden.

C++ kennt den boolschen-Typ **bool**. Auf diesen Datentyp können die logischen

Operatoren "`!`", "`||`" und "`&&`" angewendet werden.

Alternativ dazu kannn **int** verwendet werden, wobei `0` logisch Null und alles andere logisch Eins ist. Der Operator "`!`" (logisches Nicht) liefert ein Resultat, dessen Typ **int** ist und dessen Wert `1` ist, wenn der Wert des arithmetischen Operanden `0` war, und `0` sonst.

Die Division "`/`" wird bei ganzzahligen Größen ganzzahlig durchgeführt.
`11 / 3` ergibt `3` .
`11 mod 3` muß als `11 % 3` geschrieben werden und ergibt `2` .

`a << b` bedeutet, daß `a` um `b` Bit nach links (nicht zirkular) verschoben wird. Dabei werden die rechts entstehenden Bit mit Nullen belegt. Entsprechendes gilt für die Rechtsverschiebung. (Bemerkung: Bei negativem **int** werden Einsen von links nachgeschoben. Sicherheitshalber sollte man die Shiftoperatoren nur auf **unsigned** anwenden.)

Die Operatoren vom Typ `a *= b` sind Abkürzungen für `a = a * b` .

Durch Komma getrennte Ausdrücke werden von links nach rechts abgearbeitet. Die Seiteneffekte des linken Ausdrucks werden durchgeführt, ehe der rechte Ausdruck ausgewertet wird. Ergebnis des ganzen Ausdrucks ist der Typ und Wert des rechten Ausdrucks.

7.2 Überladung von Operatoren

C++ gibt dem Anwender die Möglichkeit, eigene Typen in Form von Klassen zu definieren. Um auch diese Typen mit den gewohnten Operatoren manipulieren zu können, gibt es die Möglichkeit, Operatoren zu *überladen*.

Für die in der folgenden Tabelle aufgeführten 40 Operatoren können neue Bedeutungen definiert werden.

`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`	`~`	`!`
`=`	`<`	`>`	`+=`	`-=`	`*=`	`/=`	`%=`	`^=`	`&=`
`|=`	`<<`	`>>`	`>>=`	`<<=`	`==`	`!=`	`<=`	`>=`	`&&`
`||`	`++`	`--`	`->*`	`,`	`->`	`[]`	`()`	`new`	`delete`

Es ist nicht möglich, die Präzedenz und die Syntax dieser Operatoren zu ändern. Man kann also keinen binären Operator `++` definieren oder den Operator `^` unär definieren. Auch ist es *nicht* möglich, über diese 40 Operatorzeichen hinaus neue Operatorzeichen zu definieren wie z.B. für das Potenzieren den Operator `**` . Dafür gibt es nur die Möglichkeit, einen anderen binären Operator zu überladen oder besser die Funktionsaufruf-Notation zu verwenden.

Der Name einer Operatorfunktion besteht aus dem Schlüsselwort **operator** und dem Operator selbst, z.B. **operator** `-` . Jede Operatorfunktion muß mindestens ein Klassenargument haben. Eine Operatorfunktion kann explizit oder durch schlichte Verwendung des Operators aufgerufen werden.

operator `- (a);`
oder
`-a`

Eine Operatorfunktion kann entweder als Element einer Klasse oder global definiert werden. Wenn die Operatorfunktion Element einer Klasse ist, dann muß das *erste* Argument weggelassen werden, weil dann das erste Argument (linke Seite bei einem binären Operator) immer der **this**-Zeiger der betreffenden Klasse ist. Z.B. für die Klasse `Zeichenkette` kann der Operator "`!`" (, der `1` zurückgibt, wenn die Zeichenkette leer ist, sonst `0`,) als Element der Klasse `Zeichenkette` folgendermaßen definiert werden:

```
int   Zeichenkette::operator ! () const
   {
   return anzZeich == 0;
   }
```

Hier muß das erste (und einzige) Argument entfallen, da es automatisch das Objekt ist, mit dem der Operator aufgerufen wird.

Definiert man die Operatorfunktion **operator** `!` *global*, dann sieht die Definition folgendermaßen aus:

```
int   operator ! (const Zeichenkette& z)
   {
   return z.laenge () == 0;
   }
```

oder

```
int   operator ! (const Zeichenkette& z)
   {
   return z.anzZeich == 0;
   }
```

Bei der zweiten Möglichkeit muß aber der Operator als **friend** der Klasse `Zeichenkette` definiert werden, da die Implementation der Operatorfunktion auf private Elemente der Klasse `Zeichenkette` zugreift.

Die Aufrufsyntax ist in beiden Fällen entweder

```
Zeichenkette a;
// ---
--- !a ---
```

oder

operator `!` `(a);` *// für den global definierten Operator*

`a.`**operator** `! ();` *// für den als Methode definierten Operator*

Wie entscheidet man, ob ein Operator global zu definieren ist oder als Elementfunktion? Bei einem Operator als Elementfunktion *muß* der linke Operand ein Objekt der betreffenden Klasse sein. Wenn daher der linke Operand auch ein Objekt anderen Typs sein darf, dann muß die Operatorfunktion global definiert

werden. Wenn dann private Elemente der Klasse bei der Implementation verwendet werden, muß die Operatorfunktion als **friend** der betreffenden Klasse definiert werden.

Daraus ergibt sich folgende Regel: Unäre Operatorfunktionen sollten Element der Klasse sein, da das einzige Argument eines unären Operators ohnehin ein Objekt der Klasse sein muß. Binäre Operatorfunktionen werden mit Ausnahme der Zuweisungsoperatoren, des Subskriptoperators "`[]`", des Calloperators "`()`" und des Zeigeroperators "`->`" global definiert, da das erste Argument häufig von anderem Typ als dem der betreffenden Klasse sein kann.

Anstatt Operatorfunktionen für jede Typkombination zu deklarieren, sollte die Klasse mit entsprechenden Konversionskonstruktoren ausgestattet werden. Betrachten wir als Beispiel den Operator "`+`" der Klasse `Zeichenkette`, der zwei Zeichenketten zu einer neuen Zeichenkette konkatenieren soll:

```
#include "Zeichenkette.h"

Zeichenkette operator + (
   const Zeichenkette& a,
   const Zeichenkette& b)
   {
   Zeichenkette  c;
   c.anzZeich = a.anzZeich + b.anzZeich;
   if (c.anzZeich > 0)
      {
      delete [] c.z; c.z = new char[c.anzZeich];
      for (int i = 0; i < a.anzZeich; i++) c.z[i] = a.z[i];
      for (int i = 0; i < b.anzZeich; i++) c.z[i + a.anzZeich] = b.z[i];
      }
   return c;
   }
```

Innerhalb der Klasse `Zeichenkette` muß dieser Operator als **friend** deklariert werden, da auf private Elemente der Klasse zugegriffen wird.

```
class Zeichenkette
   {
   friend Zeichenkette operator + (
      const Zeichenkette& a,
      const Zeichenkette& b);
   // ---
```

Wenn wir anstatt einer `Zeichenkette` einen C-String übergeben wollen, dann müssen wir keinen zweiten und dritten Operator "`+`" schreiben mit den Signaturen:

```
Zeichenkette operator + (
   const char* C_String,
   const Zeichenkette& Z)
   {---}
Zeichenkette operator + (
   const Zeichenkette& Z,
   const char* C_String)
   {---}
```

Das ist unnötig, weil wir schon einen Konversionskonstruktor

```
Zeichenkette (const char* C_string)
```

geschrieben haben, der immer dort automatisch aufgerufen wird, wo ein **char*** angeboten wird, aber eine `Zeichenkette` erwartet wird. Da also der **operator** `+` als Argumenttypen `Zeichenkette` erwartet, wird ein angebotener C-String durch den Konversionkonstruktur in eine `Zeichenkette` gewandelt.

Wenn wir nun auch noch einzelne **char** vor oder hinter eine `Zeichenkette` hängen wollen, dann müssen wir nur einen entsprechenden Konstruktor schreiben.

```
#include "Zeichenkette.h"

Zeichenkette::Zeichenkette (char c)
   {
   anzZeich = 1;
   z = new char[anzZeich];
   z[0] = c;
   }
```

Nun sind Aufrufe wie

```
Zeichenkette a ("Wie gehts");
(':' + a + '\?').zeige ();
```

möglich. Aber

```
(':' + "Gut" + '.').zeige (); // Fehler
```

funktioniert nicht, weil es für den Compiler keinen Grund gibt, den geklammerten Ausdruck in eine `Zeichenkette` zu wandeln. Hier muß die Konvertierung erzwungen werden:

```
(Zeichenkette (':') + "Gut" + '.').zeige (); // funktioniert
```

Aus Performanzgründen kann es gelegentlich sinnvoll sein, Operatoren für spezielle Typenkombinationen zu schreiben, um den Konstruktoraufruf zu vermeiden.

Um auch Zuweisungen `a = b` für Zeichenketten schreiben zu können, müssen wir noch einen Zuweisungsoperator "`=`" für Zeichenketten definieren. Für jede

Klasse ist zwar ein Default-Zuweisungsoperator definiert, der alle in der Klasse definierten Datenelemente kopiert.

Bei Zeiger-Klassenelementen bedeutet das allerdings, daß nur die Adresse kopiert wird, nicht aber der Inhalt der Adresse, insbesondere nicht ein ganzes Feld, wenn der Zeiger auf ein Feld zeigt. Daher muß für Klassen, die mit dynamischem Speicher arbeiten, ein Zuweisungsoperator geschrieben werden, der dies berücksichtigt. Die Klasse `Zeichenkette` hat einen dynamischen Speicherbereich und der Zuweisungsoperator könnte folgendermaßen aussehen:

```
#include "Zeichenkette.h"

Zeichenkette& Zeichenkette::operator = (
   const Zeichenkette& b)
   {
   if (this == &b) return *this;
   delete [] z;
   if ( (anzZeich = b.anzZeich) > 0)
      {
      z = new char[anzZeich];
      for (int i = 0; i < anzZeich; i++) z[i] = b.z[i];
      }
   else z = new char[1];
   return *this;
   }
```

Der binäre Operator "`=`" wird als Element der Klasse `Zeichenkette` definiert, weil der linke Operand sinnvollerweise immer den Typ `Zeichenkette` hat. Anstatt von **void** wird eine Referenz auf `Zeichenkette` zurückgegeben, damit man Ausdrücke wie `a=b=c="Guten Tag"` für Zeichenketten schreiben kann. Denn diese Ausdrücke werden explizit folgendermaßen aufgelöst:

```
a.operator = (b.operator = (c.operator = ("Guten Tag")))
```

Das bedeutet, daß der **return**-Wert als vom Typ Zeichenkette weiterverwendet werden soll.

Die Implementation des Operators "`=`" überprüft als erstes, ob die beiden Zeichenketten gleich sind. Wenn ja, muß nichts getan werden. Das ist nicht nur effizient, sondern auch notwendig, da das Statement **delete** `[] z` bei Identität der Zeichenketten gerade die Daten unzugänglich macht, die zu kopieren wären. Wir haben für die Lösung des allgemein schwierigen Problems der *Objektidentität* eine effektive Methode gewählt, die aber in bestimmten Fällen bei Mehrfachvererbung schief gehen kann.

Die Implementation des Zuweisungs-Operators für eine Klasse `A` sollte i.a. folgendermaßen aussehen:

```
A& A::operator = (const A& b)
   {
   if (this != &b)
      {
```

```
    // ---
    }
  return *this;
  }
```

7.2.1 Zusammenfassung Operatorensyntax

Die wichtigsten syntaktischen Möglichkeiten werden in der folgenden Klasse `A` zusammengefaßt, wobei `Ti` irgendwelche andere Typen sind.

```
class A
  {
  friend T1 operator binär (const A& a, const T2& b);
  friend T1 operator binär (const T2& b, const A& a);
  public:
    T1 operator unär ();
    T1 operator binär (const T2& b);
    A& operator Zuweisung (const A& b);
    T1 operator [],(),-> (const T2& b);
  };
```

Die folgenden Abschnitte behandeln einige Operatoren genauer bzw Operatoren, deren Syntax etwas ungewöhnlich ist.

7.2.2 Inkrement- und Dekrementoperatoren

Die Operatoren `--` und `++` können vor und hinter dem Operanden verwendet werden. Die Syntax der Operatorfunktion kann dies nicht unterscheiden. Daher gibt es die Möglichkeit die beiden Formen durch ein **int** als Argument zu unterscheiden.

```
class K
  {
  public:
    K& operator -- ();    // prefix
    K& operator -- (int); // postfix
    K& operator ++ ();    // prefix
    K& operator ++ (int); // postfix
  // ---
  };
```

7.2.3 Für Klassen vordefinierte Operatoren

Außer Operator "`=`" haben auch die Operatoren "`&`" und "`,`" eine vordefinierte Bedeutung für Klassen. Um Mißbrauch durch den Anwender einer Klasse

zu vermeiden, mag es nötig sein, diese Operatoren dem Klassenanwender unzugänglich zu machen. Dies ist möglich, indem man diese Operatoren **private** macht, ohne daß ihnen neue Bedeutungen gegeben werden müßten.

```
class K
   {
   // ---
   private:
      void operator = (const K& k);
      void operator & ();
      void operator , (const K& k);
      // ---
      K (const K& k); // Copy-Konstruktor
   };
```

Damit auch **friend**-Funktionen und Elementfunktionen dieser Klasse nicht versehentlich diese Operatoren verwenden, dürfen sie nur deklariert, nicht aber definiert werden. Bei versehentlicher Verwendung meldet der Linker einen Fehler.

Wenn man den Zuweisungsoperator verbietet, sollte man wahrscheinlich auch den Copy-Konstruktor verbieten, was mit derselben Technik gemacht wird.

7.2.4 Konversionsoperatoren

Bisher haben wir Konstruktoren als Methode zur Typenkonversion kennengelernt:

```
class K
   {
   public:
      K (T&);
      // ---
   };
```

Durch den hier deklarierten Konstruktor wird der Typ `T` zum Typ `K` konvertiert. Mit der Konstruktormethode kann man aber nicht `K` in einen Typ `T` konvertieren, ohne die Definition des Typs `T` zu verändern. Insbesondere geht das nicht, wenn `T` ein elementarer Typ ist. Hier bietet der Cast-Operator "`()`" eine Möglichkeit,

```
K::operator T ();
```

wobei `T` ein Typname ist. Dieser Operator konvertiert Typ `K` nach Typ `T`. Z.B. soll die Konversion einer Zeichenkette zum Typ **char** das erste Zeichen einer Zeichenkette liefern oder, falls die Zeichenkette leer ist, das Zeichen `'\0'`.

```
#include "Zeichenkette.h"

Zeichenkette::operator char ()
   {
   char c = '\0';
   if (anzZeich != 0) c = z[0];
   return c;
   }
```

Es sei bemerkt, daß beim Cast-Operator *kein* Rückgabetyp angegeben wird. Denn der Rückgabetyp ist schon als Typ `T` vermerkt.

Die Aufrufsyntax ist vielfältig:

```
Zeichenkette z ("Hallo");
char c;
c = (char)z; \\ Cast-Notation oder
c = z;       \\ implizite Notation oder
c = char (z); \\ Funktionscast-Notation
```

Aus Sicherheitsgründen sollten Konversionsoperatoren restriktiv definiert werden. Man sollte sogar eher explizite Konversionsfunktionen definieren:

```
T  K::Tof () {---}
char Zeichenkette::charof () {---}
```

Die beiden Konvertierungen von Typ `K` nach Typ `T` und umgekehrt läßt der Compiler koexistieren. Erst wenn diese Konversionen wirklich verwendet werden, kann es Fehlermeldungen wegen Ambiguität geben. Daher sollten solche ambigen Definitionen vermieden werden. Im Beispiel `Zeichenkette` ist nun der Ausdruck

```
Zeichenkette a ("Hallo");
char c = 'H';
--- a + c ---
```

ambig, weil der Compiler nicht weiß, ob er nun `a` in **char** oder `c` in `Zeichenkette` wandeln soll. Solche Ambiguitäten müssen dann durch einen expliziten Cast geklärt werden.

7.2.5 Subskript-Operator

Die Funktion **operator** `[]` kann verwendet werden, um Subskripten von Klassenobjekten eine Bedeutung zu geben. Zum Beispiel wollen wir die einzelnen Elemente einer Zeichenkette zugänglich machen. Da das erste Argument des Subskript-Operators sicherlich ein Objekt der betreffenden Klasse ist, wird der Operator als Element der Klasse `Zeichenkette` definiert.

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

char& Zeichenkette::operator [] (int i)
   {
   if (i < 0 || i >= anzZeich)
      {
      cerr << "Index i = " << i;
      cerr << " nicht im Bereich.\n";
      }
   return z[i];
   }
```

Die Aufruf-Syntax ist wie gewohnt:

```
Zeichenkette a ("Hallo");
char c;
c = a[2];
a[1] = 'e';
```

Der Rückgabewert dieses Operators wurde als Referenz auf **char** definiert, damit es möglich ist, mit der Subskript-Syntax Zeichenkettenelemente zu verändern. Das hat allerdings zur Folge, daß der Operator so nicht auf konstante Zeichenketten anwendbar ist. Um das zu erreichen, überlädt man den Operator weiter:

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

char Zeichenkette::operator [] (int i) const
   {
   if (i < 0 || i >= anzZeich)
      {
      cerr << "Index i = " << i;
      cerr << " nicht im Bereich.\n";
      }
   return z[i];
   }
```

Da mit diesem Operator die Zeichenkette nicht verändert werden kann, kann er auch für konstante Objekte zugelassen werden.

7.2.6 Funktionsaufruf-Operator

Die Notation `Ausdruck (Ausdruck_Liste)` wird als eine binäre Operation mit dem Operator `()` , dem linken Operanden `Ausdruck` und dem rechten Operanden `Ausdruck_Liste` interpretiert. Die `Ausdruck_Liste` wird dabei wie eine Funktionsargumentenliste überprüft und ausgewertet.

Angewendet wird dieser Operator hauptsächlich als Iterator, als Subskriptoperator für mehrdimensionale Felder oder als Substring-Operator.

Die letztere Anwendung wird hier als Beispiel für die Klasse `Zeichenkette` gezeigt. Als Elementfunktion wird in der Klassendefinition

```
Zeichenkette operator () (const int i, const int j) const;
```

deklariert. Die Implementation sieht folgendermaßen aus:

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

Zeichenkette Zeichenkette::operator () (int i, int j) const
   {
   Zeichenkette a;
   if (i < 0 || i > j || j >= anzZeich)
      cerr << "Indizes sind nicht im Bereich!\n";
   else
      {
      a.anzZeich = j - i + 1;
      delete [] a.z; a.z = new char[a.anzZeich];
      for (int k = 0; k < a.anzZeich; k++) a.z[k] = z[i + k];
      }
   return a;
   }
```

Die Aufrufsyntax ist z.B.

```
Zeichenkette a ("Guten Tag");
a (3, 7).zeige ();
```

7.2.7 Operatoren new und delete

Die Operatoren **new** und **delete** können global und für jede Klasse überladen werden. Wichtigste Anwendung ist das Debuggen von Speicherproblemen. Die Syntax lautet z.B.:

```
void* operator new (size_t s)
   {
   cerr << "Das ist mein eigenes new für die Klasse X !" << endl;
   return ::operator new (s);
   }

void operator delete (void* p)
   {
   cerr << "Das ist mein eigenes delete für die Klasse X !" <<
       endl;
   ::operator delete (p);
   }
```

Hier wird am Ende durch den Scope-Operator `::` das global definierte **new** bzw **delete** aufgerufen. Das geht nur dann gut, wenn diese Definitionen in einer Klasse stehen. Denn dann wird **new** und **delete** nur für diese Klasse überladen, d.h. werden aufgerufen, wenn ein Objekt der betreffenden Klasse dynamisch angelegt oder aufgegeben wird. **new** und **delete** sind implizit statische Funktionen und können daher auch nicht-virtuell sein.

Wenn das globale **new** bzw **delete** überladen werden soll, dann kann man natürlich *nicht* das globale **new** bzw **delete** aufrufen, sondern muß z.B. ein `malloc` bzw `free` verwenden.

```
int   cerrSchonAngelegt (0);

void* operator new (size_t s)
   {
   if (cerrSchonAngelegt == 1)
      cerr << "Das ist mein eigenes globales new !" << endl;
   return malloc (s);
   }

void  operator delete (void* p)
   {
   if (cerrSchonAngelegt == 1)
      cerr << "Das ist mein eigenes globales delete !" << endl;
   free (p);
   }

main ()
   {
   cerr << "irgendwas" << endl;
   cerrSchonAngelegt = 1;
   ---
   }
```

7.2.8 Operatoren ->

Die Überladung des Dereferenzierungs-Operators *(member selection operator)* `->` erlaubt die Konstruktion von ***smart Pointers***, mit denen Delegation, Referenz-Zählung ***reference-counting*** und "Vererbung auf Objektbasis" realisiert werden kann.

Wenn der Operator `->` überladen wird, dann wird er als ein unärer Postfix-Operator aufgefaßt, auf dessen Resultat wieder der Operator `->` angewendet wird. Ist dieser *nicht* überladen, dann wird dabei der normale binäre Operator verwendet. Ist dieser wiederum überladen, wird er wieder als unärer Postfix-Operator verwendet.

Also: Falls bei einem Ausdruck `b->m ()` der Operator `->` in der Klasse `B` von `b` überladen ist, dann wird
`(b->)->m ()`

ausgeführt, wobei das Resultat von `(b->)` entweder ein Zeiger einer Klasse sein muß, bei der es die Methode `m ()` gibt, oder ein Objekt einer Klasse sein muß, bei der `->` wieder geeignet überladen ist. In dem Fall würde dann
`( (b->)->)->m ()`
ausgeführt werden.

Das folgende Beispiel wird den Mechanismus verdeutlichen:

```
// deref.h
#include <iostream>

using namespace std;

class A
   {
   public:
      A () :  m (5) {}
      int   m;
      void  f () { cout << "A::f : m = " << m << endl; }
   };

class B
   {
   public:
      A  a;
      A* operator -> ()
         {
         cout << "B gibt seinen Senf dazu und addiert 2" << endl;
         a.m = a.m + 2;
         return &a;
         }
   };

class C
   {
   public:
      B  b;
      B& operator -> ()
         {
         cout << "C gibt seinen Senf dazu und addiert 3" << endl;
         b.a.m = b.a.m + 3;
         return b;
         }
   };
```

Der Operator muß als Elementfunktion definiert werden und hat als Rückgabewert

1. entweder einen Zeiger auf ein Objekt einer beliebigen Klasse

2. oder eine Referenz auf ein Objekt oder ein Objekt einer Klasse, bei der der Operator `->` überladen ist.

Über Objekte der Klassen `B` oder `C` können nun mit der Zeigersyntax Methoden der Klasse `A` aufgerufen werden, wobei jeweils zunächst der in der Überladung des Operators definierte Code durchgeführt wird. Das folgende Beispiel zeigt verschiedene Anwendungen der oben definierten Klassen:

```
// deref.cpp
#include "deref.h"

using namespace std;

main ()
   {
   A  aa;
   cout << "aa.f () : ";
   aa.f ();
   cout << endl;

   B  bb;
   cout << "bb->f () : ";
   bb->f ();
   cout << endl;

   C  cc;
   cout << "cc->f () : ";
   cc->f ();
   cout << endl;

   return 0;
   }
```

Als Ergebnis erscheint auf dem Bildschirm:

```
aa.f () : A::f : m = 5

bb->f () : B gibt seinen Senf dazu und addiert 2
A::f : m = 7

cc->f () : C gibt seinen Senf dazu und addiert 3
B gibt seinen Senf dazu und addiert 2
A::f : m = 10
```

Anwendungen:

- Delegation: Die Ausführung der Methode `f` wird von der Klasse `C` an Klasse `B` delegiert (modifiziert weitergereicht), von der Klasse `B` an die Klasse `A` delegiert.

- Vererbung auf Objektbasis: Wenn das Objekt der Klasse `A` nicht in `B` angelegt wird, sondern als Referenz oder Zeiger geführt wird, sodaß es eine eigene Existenz hat, dann könnten verschiedene Objekte der Klasse `B` sich auf *dasselbe* `A`-Objekt beziehen oder davon "erben". Der Code von `B` lautete dann:

```
class B
   {
   public:
      B (A& a) : a (a) {}
      A& a;
      A* operator -> ()
         {
         cout << "B gibt seinen Senf dazu und addiert 2" <<
             endl;
         a.m = a.m + 2;
         return &a;
         }
   };
```

Damit kann eine `uses-a`-Beziehung beinahe wie eine Vererbungsbeziehung behandelt werden.

- Referenz-Zählung ***(smart pointers)*** ist eine komplizierte Anwendung, wobei `A` die Aufgabe hätte, die Anzahl der Referenzen durch `B`-Objekte auf sich mitzuzählen. Dazu wird ein **Handle-Representation-Pattern** verwendet.

Im einzelnen funktioniert das folgendermaßen: Gegeben sei ein Klasse `R` *(representation)*, für die wir Referenzzählung durchführen wollen. Dazu schreiben wir die Smart-Pointer-Klasse `HR` *(handle for representation)*:

```
#include <iostream>
class R { };

class HR
   {
   private:
      R* rep;
      int* zaehler;
   public:
      R* operator -> () { return rep; }
      HR (R* rep) : rep (rep), zaehler (new int (1)) {}
      HR (const HR& hr) : rep (hr.rep), zaehler (hr.zaehler)
         {
         (*zaehler)++;
         }
      HR& operator = (const HR& hr)
         {
         if (rep == hr.rep) return *this;
         if (--(*zaehler) == 0)
            {
            delete rep;
            delete zaehler;
            }
         rep = hr.rep;
         zaehler = hr.zaehler;
         (*zaehler)++;
         return *this;
         }
      ~HR () { if (--(*zaehler) == 0) { delete rep; delete zaehler; }}
      void operator () () const
         {
         std::cout << "Zählerstand: " << *zaehler << "\n";
         }
   };
```

Wenn diese Technik – empfohlenermaßen! – angewendet wird, dann sollte man noch den Abschnitt “Handle Classes“ im Buch von Stroustrup [1] zu Rate ziehen, wo noch weitere Möglichkeiten diskutiert werden. Insbesondere lohnt es sich, ein Template zu schreiben.

7.3 Beispiel Zweidimensionaler Vektor

Für das einfache Beispiel zweidimensionialer Vektor wurden einige Operatoren implementiert.

Header-File: `ZweiDimVek.h`

```
// ZweiDimVek.h
```

```
#ifndef ZweiDimVek_h
#define ZweiDimVek_h

#include <iostream>

class ZweiDimVek
   {
   friend double operator * (
      const ZweiDimVek& v1,
      const ZweiDimVek& v2);
   friend std::ostream& operator << (
      std::ostream& s,
      const ZweiDimVek& v);
   public:
      ZweiDimVek ();         // Default−Konstruktor
      ZweiDimVek (double a, double b); // Konstruktor
      ZweiDimVek (ZweiDimVek& v); // Copy−Konstruktor
      double  getX1 () const { return x1; }
      double  getX2 () const { return x2; }
      void  setX1 (double a) { x1 = a; }
      void  setX2 (double a) { x2 = a; }
      double  betrag () const;
      void  drucke () const;
      ZweiDimVek operator - () const;
   private:
      double  x1;
      double  x2;
   };

#endif
```

Implementations-File: `ZweiDimVek.cpp`

```
// ZweiDimVek.cpp

#include <iostream>
#include <cmath>
#include "ZweiDimVek.h"

using namespace std;

ZweiDimVek::ZweiDimVek ()
   {
   x1 = 0.0;
   x2 = 0.0;
   }

ZweiDimVek::ZweiDimVek (double a, double b)
   {
   x1 = a;
```

```
   x2 = b;
   }

ZweiDimVek::ZweiDimVek (ZweiDimVek& v)
   {
   x1 = v.x1;
   x2 = v.x2;
   }

double operator * (
   const ZweiDimVek& v1,
   const ZweiDimVek& v2)
   {
   return v1.x1 * v2.x1 + v1.x2 * v2.x2;
   }

double ZweiDimVek::betrag () const
   {
   return sqrt (x1 * x1 + x2 * x2);
   }

void ZweiDimVek::drucke () const
   {
   cout << '(' << x1 << ", " << x2 << ")\n";
   }

ostream& operator << (
   ostream& s,
   const ZweiDimVek& v)
   {
   s << '(' << v.x1 << ", " << v.x2 << ')';
   return s;
   }

ZweiDimVek ZweiDimVek::operator - () const
   {
   ZweiDimVek v;
   v.x1 = -x1;
   v.x2 = -x2;
   return v;
   }
```

Anwendungs-File: `apl.cpp`

```
// apl.cpp – Anwendungsprogramm für ZweiDimVek

#include <iostream>
#include "ZweiDimVek.h"

using namespace std;
```

```
main ()
   {
   ZweiDimVek v;
   ZweiDimVek v1 (2.0, 2.0);
   ZweiDimVek v2 (2.0, 3.0);
   ZweiDimVek v3 (v1);
   cout << "v: "; v.drucke ();
   cout << "v1: "; v1.drucke ();
   cout << "v2: "; v2.drucke ();
   cout << "v3: "; v3.drucke ();
   cout << "Das Skalarprodukt v1*v2 ist " << v1 * v2 << "\n";
   cout << "Der Betrag von v3 ist " << v3.betrag () << "\n";
   cout << "Minus von v3 ist " << -v3 << "\n";
   return 0;
   }
```

7.4 Übungen

Übungen zur Operatorensyntax:

```
short int   a;
short int   b;
a = 5;
b = a++;
```

Was ist a? Was ist b?

```
b = ~a;
```
Was ist b?

```
b = !a
```
Was ist b?

```
b = a << 3
```
Was ist b?

```
b = a >> 2
```
Was ist b?

```
a = 6; b = 5;
```
Was ist `a == b`?
Was ist `a & b`?
Was ist `a ^ b`?
Was ist `a | b`?
Was ist `a && b`?
Was ist `a || b`?
Was ist `a-- != b ? a : b`?
`a = 6, b = a, b++;` Was ist b?
`b = 5, b = a += b, b == a;` Was ist b?

`a = 6, b = 5, b = (a += b, b == a);` Was ist `b`?

Übungen zur Klausurnote:

Schreibe für die Klasse `Klausurnote` die Operatoren:

1. `+` : wirkt wie die Methode `plus`

   Anwendungsbeispiele für Operator `+` sind:

   ```
   Klausurnote a (3.5), b("2,7"), c;
   c = a + b;
   c = a + 2.7;
   c = 2.7 + a;
   c = 3.5 + 2.7;
   c = a + 3.5 + 2.7;
   c = 3.5 + 2.7 + a;
   ```

   Funktioniert das alles? Warum? Warum nicht?

2. `=` und `+=`

3. `*` : bedeutet nicht Note `*` Note, sondern **double** `*` Note oder Note `*` **double**.

4. Castoperator nach **double**

5. Funktionsaufrufoperator **operator** `()(int x, int y)` belegt Note mit `x ,y` . Schreibe auch entsprechenden Konstruktor.

6. `==` `<` `>` `>=` `<=`

7. **new** und **delete**

Übung Operatoren für Zeichenketten:

Definiere für die Klasse `Zeichenkette` die Operatoren
“`=`“, “`+`“, “`!=`“, “`==`“, “`+=`“, “`<`“, “`<=`“, “`>`“, “`>=`“,
“`<<`“ (`a << 12` soll bedeuten, daß die Zeichenkette `a` rechts auf insgesamt 12 Zeichen mit Leerzeichen aufgefüllt wird bzw bis auf 12 Zeichen abgeschnitten wird, *linksbündiges Padding.*),
“`>>`“ (`a >> 12` soll bedeuten, daß die Zeichenkette `a` links auf insgesamt 12 Zeichen mit Leerzeichen aufgefüllt wird bzw bis auf 12 Zeichen *rechts* abgeschnitten wird, *rechtsbündiges Padding.*),
“`||`“ (`a || 12` soll bedeuten, daß die Zeichenkette `a` in einem Feld der Länge 12 zentriert wird, wobei rechts und links mit Leerzeichen aufgefüllt wird bzw die Zeichenkette auf 12 Zeichen abgeschnitten wird.),
“`-=`“ und “`-`“ (`a -= x` soll bedeuten, daß die Teilzeichenkette `x` in `a` gelöscht wird, “`-`“ entsprechend.),
“`/`“ (“`a / x`“ gibt als Resultat an, wie oft die Teilzeichenkette `x` in `a` enthalten ist.),
“`%`“ (“`a % x`“ gibt als Resultat an, wo die Teilzeichenkette `x` in `a` beginnt.),
“`|`“ (“`a | 4`“ setzt Einfügepunkt vor Zeichen `4`, gibt wieder `a` zurück.),

"`*`" ("`a * b`" fügt Zeichenkette `b` in `a` am Einfügepunkt ein und gibt Resultat zurück.).

Bemerkung: Im allgemeinen ist es günstiger einen binären Operator `æ` unter Verwendung des entsprechenden Operators `æ=` zu implementieren.

Bemerkung: Der in dieser Übung vorgeschlagene extensive Gebrauch von Operatorüberladung ist i.a. nicht zu empfehlen. Überladung macht Programme schwer lesbar, wenn sie nicht intuitiv verständlich ist.

Übung Cast-Operator für Zeichenkette: Schreiben Sie für die Klasse `Zeichenkette` einen Cast-Operator, der die `Zeichenkette` in einen C-String wandelt. (Der Operator sollte eine Kopie des Strings liefern.)

Übung Klasse `Vektor`: Schreiben Sie eine Klasse `Vektor` für n-dimensionale Vektoren von **double**-Elementen mit dem Konstruktor `Vektor (int n, double x = 0)` und den Operatoren `[] + - * ()`, wobei der Operator `()` den Vektor vernünftig darstellen soll.

Übung zum Operator `()`: Schreibe einen Funktionsaufruf-Operator für die Klasse `Zeichenkette`, der ab einer Postion in der Zeichenkette nach einem Zeichen vom Typ **char** sucht und dessen Position zurückgibt oder, falls nicht gefunden, `-1` liefert.

Kapitel 8

Vererbung

Durch das Konzept der Klasse mit Operatorüberladung bietet C++ schon sehr mächtige Möglichkeiten. Der Vererbungsmechanismus ist eine elegante Möglichkeit, Code wiederzuverwenden. Durch Vererbung kann die Code-Redundanz wesentlich verringert werden.

Mit den heutzutage zur Verfügung stehenden Editoren ist die Wiederverwendung einmal geschriebenen Codes scheinbar kein Problem, da beliebige Mengen Text beliebig oft kopiert werden können. Damit handelt man sich aber ein großes Redundanzproblem ein. Häufig müssen Programmteile leicht modifiziert werden, oder der Code muß an neue Gegebenheiten angepaßt werden, was dann Änderungen an sehr vielen Stellen in einem oder vielen Programmen nachsichzieht.

Der Vererbungs- und Ableitungsmechanismus für Klassen ist eine Möglichkeit, Strukturen und Funktionen wiederzuverwenden in Strukturen, die sich nur leicht voneinander unterscheiden, insbesondere bei Strukturen, die in einer "Ist-ein" Beziehung stehen.

Ein Angestellter *ist eine* Person. Eine Dampflokomotive *ist eine* Lokomotive. Eine Lokomotive *ist ein* Fahrzeug. Ein Girokonto *ist ein* Konto. Ein Quadrat *ist ein* Rechteck. Das letzte Beispiel ist problematisch, da ein Quadrat ein Rechteck nicht immer substituieren kann. Denn wie soll eine Methode, die nur die Länge des Rechtecks ändert, auf ein Quadrat angewendet werden?

Der Vererbungsmechanismus in C++ dient im wesentlichen folgenden Zielen:

- Spezialisierung: Erzeugung neuer Typen durch Anfügen von zusätzlichen Eigenschaften an alte Typen (Angestellter – Person)
- Erzeugung neuer Typen durch Einschränkung alter Typen (Quadrat – Rechteck)
- Generalisierung: Definition einer gemeinsamen Schnittstelle für verschiedene Typen

8.1 Syntax des Vererbungsmechanismus

Die Syntax für den Vererbungsmechanismus ist einfach. `A` sei eine Klasse, von der die Klasse `B` erben soll. Dann sieht die Definition von `B` folgendermaßen aus:

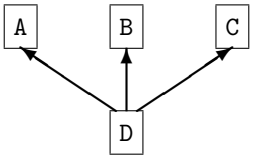
```
class B : A
   {
   // ---
   };
```

`A` heißt **Basisklasse**, **Obertyp** oder **Superklasse**, von der `B` *abgeleitet* wird oder von der `B` *erbt*. `B` heißt **Untertyp** oder **Subklasse**. Ein Objekt von `B` enthält ein Objekt von `A` als *Subobjekt*. Graphisch wird dies dargestellt durch:

`B` kann wiederum Basisklasse sein: **class** `C : B {---};` , sodaß *Klassenhierarchien* erstellt werden können.

Eine Klasse kann sich von mehreren Klassen ableiten **(Mehrfachvererbung, *multiple inheritance*)**:

```
class D : A, B, C
   {
   // ---
   };
```



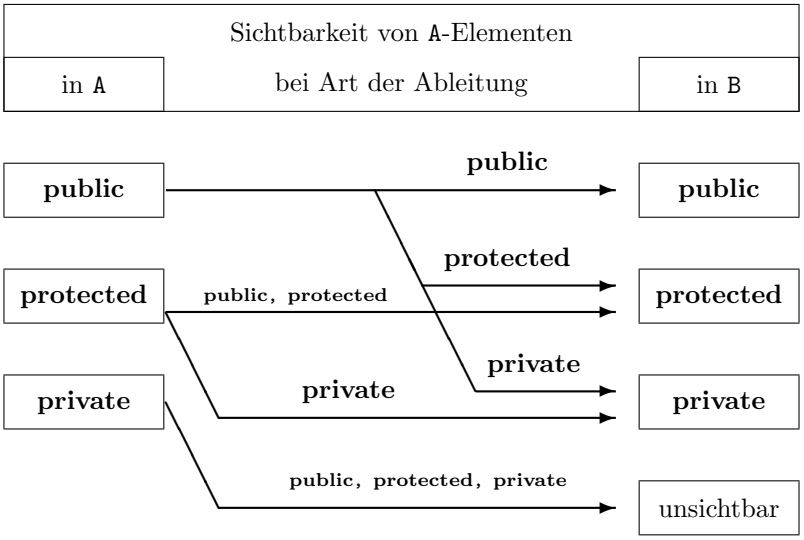
8.1.1 Zugriffsrechte

Eine Klasse kann **public**, **protected** und **private** Elemente haben. Elementfunktionen der Klasse haben auf alle Elemente derselben Klasse Zugriff. Globale Funktionen oder Elementfunktionen anderer Klassen können nur auf **public** Elemente (*globale Sichtbarkeit*) zugreifen.

Eine Klasse `B` kann auf drei Arten von einer Klasse `A` abgeleitet werden:

```
class B : public A {---};
class B : protected A {---};
class B : private A {---};
```

Je nach Art der Ableitung werden die Mitglieder von `A` verschiedene Sichtbarkeit in `B` haben.



Private Elemente einer Basisklasse können niemals außerhalb der Basisklasse sichtbar gemacht werden. Ein Vererbungsmechanismus kann die Sichtbarkeit immer nur einschränken oder höchstens erhalten. Der Default-Ableitungsmechanismus ist **private**. Um Mißverständnisse auszuschließen, sollte die Ableitungsart *immer* spezifiziert werden, da z.B. bei

```
class C : public A, B {---};
```

von `B` privat (d.h. Default) abgeleitet wird. Klarer ist die ausführliche Schreibweise

```
class C : public A, private B {---};
```

oder, wenn **public** gemeint war:

```
class C : public A, public B {---};
```

Bemerkung: Der **protected**-Mechanismus bedeutet eine relativ enge Kopplung zwischen Basisklasse und abgeleiteter Klasse. Daher sollte man davon nur vorsichtig Gebrauch machen. Anstatt die privaten Elemente einer Basisklasse **protected** zu deklarieren, sollte man eher **inline** `set-` und `get`-Funktionen verwenden.

Anstatt des **protected**-Mechanismus können wir auch den **friend**-Mechanismus verwenden, um nur speziellen Klassen den Zugang zu privaten Elementen zu erlauben.

Zugriffsdeklarationen

Mit **Zugriffs- oder Access-**deklarationen kann die Sichtbarkeit von geerbten Elementen noch genauer eingestellt werden, indem das Element in der erbenden Klasse nocheinmal deklariert wird unter expliziter Nennung der Klasse. Das ursprüngliche Sichtbarkeitsniveau darf dabei nicht geändert werden. Damit können **private** geerbte Elemente für den Anwender wieder sichtbar gemacht werden.

Beispiele:

1. ```
   class A
     {
     public:
       int   f () { return x = 5; }
       int   g () { return x = 6; }
       int   y;
     protected:
       int   x;
     };

   class B : private A
     {
     public:
       A::f; // Access−Declaration
       A::y; // Access−Declaration
       int   h () { return g (); }
     };
   ```

   Durch die Access-Deklarationen werden `f` und `y` in `B` wieder öffentlich und sind von außen zugreifbar. Der Rest von `A` bleibt nach außen unsichtbar.

   ```
   main ()
     {
     B  b;
     b.f ();
     // b.g ();  // Fehler: g private
     b.y = b.h ();
     }
   ```
   ```

2.
```
class A
   {
   protected:
      int   x;
   };
class B : private A
   {
   public:
      A::x;    // Fehler
   protected:
      A::x;    // o.k.
   private:
      A::x;    // Fehler
   };
```

Das ursprüngliche Sichtbarkeitsniveau kann nicht geändert werden.

8.1.2 Virtuelle Funktionen

Einem Zeiger vom Typ "Zeiger auf eine Basisklasse" kann die Adresse einer von dieser Basisklasse abgeleitete Klasse zugewiesen werden ("Up-Cast").

```
class A {---};

class B : public A {---};

A* pa;
B  b;
pa = &b;
```

Über den Zeiger `pa` sind eigentlich nur Elemente des `A`-Teils von `B` zugänglich. Das Resultat der Ausführung des folgenden Programms

```
#include <iostream>

using namespace std;

class A
   {
   public:
      void f (char* s) { cout << s << "f kommt von A.\n";}
   };

class B : public A
   {
   public:
      void f (char* s) { cout << s << "f kommt von B.\n";}
      void g (char* s) { cout << s << "g kommt von B.\n";}
   };

main()
   {
   B  b;
   A* pa;
   B* pb;
   pa = &b;
   pb = &b;
   pa->f ("pa->f: ");
   pb->f ("pb->f: ");
// pa->g ("pa->g: ");// übersetzt nicht,
                     // da g nicht Element von A
   pb->g ("pb->g: ");
   return 0;
   }
```

ist

```
pa->f: f kommt von A.
pb->f: f kommt von B.
pb->g: g kommt von B.
```

Andererseits wäre es wünschenswert, daß mit `pa->f ()` das `f` von `B` ausgeführt wird im Hinblick darauf, verschiedene verwandte Klassen über ein gemeinsames, durch die Basisklasse definiertes Interface anzusprechen ("Polymorphismus von Objekten"). Dazu müßte es einen Mechanismus geben, der bei dem Aufruf `pa ->f ()` dafür sorgt, daß der Compiler in der abgeleiteten Klasse nach einer Funktion mit demselben Namen `f` sucht, um diese dann aufzurufen.

Dies kann tatsächlich erreicht werden, indem man in der Basisklasse die Funktion `f` als **virtual** erklärt.

```
class A
   {
   public:
      virtual void f (char* s) {---};
   };
```

Damit wird dem Compiler angedeutet, daß es möglicherweise in abgeleiteten Klassen andere Versionen von `f` gibt. Das Resultat des obigen Programmlaufs ist jetzt:

```
pa->f: f kommt von B.
pb->f: f kommt von B.
pb->g: g kommt von B.
```

Wenn allerdings keine andere Version einer virtuellen Funktion in einer abgeleiteten Klasse gefunden wird, dann wird die Version der Basisklasse genommen.

Nur nicht-statische Elementfunktionen können **virtual** gemacht werden.

Eine als **virtual** definierte Funktion bleibt **virtual**, auch wenn in abgeleiteten Klassen das Schlüsselwort **virtual** nicht verwendet wird. **virtual** wird beliebig weit vererbt.

Wenn abgeleitete Klassen eine virtuelle Funktion redefinieren, müssen sie auch alle anderen virtuellen Funktionen mit gleichem Namen (aber unterschiedlicher Signatur) redefinieren! Es gibt sonst die Warnung, daß die nicht implementierten Funktionen von den implementierten verdeckt werden.

8.1.3 Abstrakte Klassen

Eventuell ist eine Basisklasse dermaßen allgemein, daß für eine Methode keine vernünftige Implementation geschrieben werden kann. In dem Fall kann man eine solche Methode als *rein* virtuell durch "Initialisierung" mit Null erklären.

```
virtual void  f () = 0;
```

Eine Klasse mit mindestens einer rein virtuellen Funktion ist eine *abstrakte* Klasse, da davon keine Objekte angelegt werden können.

Eine Klasse, die sich von einer abstrakten Klasse ableitet und die nicht alle rein virtuellen Funktionen der Basisklasse implementiert, ist ebenfalls abstrakt.

Der wesentliche Sinn abstrakter Klassen ist, daß Schnittstellen definiert werden können, ohne daß man sich um die Implementationsdetails kümmern muß.

Eine rein virtuelle Funktion kann trotzdem noch in der Basisklasse definiert werden. Sie kann aber nur mit expliziter Referenz auf die Basisklasse aufgerufen werden.

```
#include <iostream>

using namespace std;

class A
   {
   public:
      virtual void  f () = 0;
   };

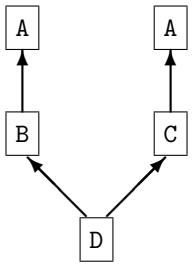
void A::f ()
   {
   cout << "f kommt von A.\n";
   }

class B : public A
   {
   public:
      void f () { A::f ();}
   };

main ()
   {
   B  b;
   b.f ();
   }
```

8.1.4 Virtuelle Vererbung

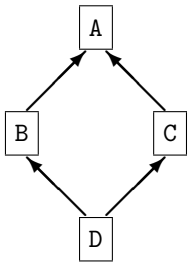
Bei Mehrfachvererbung kann es vorkommen, daß diesselbe Basisklasse öfter als einmal in einer abgeleiteten Klasse enthalten ist. Betrachten wir z.B. den folgenden Vererbungsbaum:



`D` erbt von `B` und `C`, die ihrerseits jeweils von `A` erben. Es ist also durchaus erlaubt mehrere Subobjekte `A` zu haben. Ambiguitäten beim Zugriff auf Elemente von `A` müssen durch explizite Nennung der Basisklasse aufgelöst werden.

```
class A { public: int a; };
class B : public A {---};
class C : public A {---};
class D : public B, public C
   {
   public:
      void f ()
         {
         a = 5;      // Fehler: Zugriff ambig
         B::a = 5;   // Zugriff auf A−Objekt von B
         C::a = 6;   // Zugriff auf A−Objekt von C
         }
   };
```

Aber manchmal ist es sinnvoll, daß das A-Objekt nur einmal vorkommt, wenn etwa `B` und `C` dieselbe in `A` verwaltete Information benötigen. Der Vererbungsgraph müßte folgendermaßen aussehen:



Mit der Spezifikation **virtual** beim Vererbungsmechanismus wird erreicht, daß nur *ein* Objekt für die Basisklasse angelegt wird. Oder: Jede Basisklasse, die bei Vererbung als nicht **virtual** spezifiziert wird, wird ein eigenes Objekt haben.

```
class A { public: int a; };
class B : virtual public A {---};
class C : virtual public A {---};
class D : public B, public C
   {
   public:
      void f ()
         {
         a = 5;  // überflüssig, da es nur ein
         a = 6;  // A−Objekt gibt
                 // Zugriff nicht ambig
         }
   };
```

8.2 Konstruktoren und Destruktoren

Die Konstruktoren der Basisklassen werden automatisch aufgerufen, wobei die Reihenfolge in der Vererbungshierarchie von oben nach unten geht. Die Destruktoren werden in umgekehrter Reihenfolge aufgerufen. Wenn Argumente an einen Konstruktor einer Basisklasse zu übergeben sind, dann muß der Konstruktor bei der Definition des Konstruktors der abgeleiteten Klasse explizit aufgerufen werden. Die Syntax lautet z.B.

```
D::D (int x, double y)
   : B (x), C (y)
   {
   // ---
   }
```

In diesem Beispiel ist angenommen, daß `B` einen Konstruktor `B (int)` und `C` einen Konstruktor `C (double)` hat. Mit dieser Syntax können auch Datenelemente von Klassen initialisiert werden – vorzugsweise, da die Elemente sofort mit dem richtigen Wert initialisiert werden. Ansonsten wird der Wert erst nach einer Defaultinitialisierung zugewiesen. Wenn `D` ein Datenelement **int** `z` hat,

```
class D : B, C
   {
   public:
      D (int x, double y, int z);
      // ---
   private:
      int   z;
   };
```

dann kann die Konstruktorimplementation lauten:

```
D::D (int x, double y, int z)
   : B (x), C (y), z (z)
   {
   // ---
   }
```

anstatt von

```
D::D (int x, double y, int z)
   : B (x), C (y)
   {
   D::z = z;
   // ---
   }
```

Zu bemerken ist, daß `z (z)` vom Compiler richtig verstanden wird! Der Versuch explizit den Namenskonflikt aufzulösen mit `D::z (z)` würde mißverstanden werden als ein Versuch, eine Elementfunktion `z` zu definieren.

Die Reihenfolge der Initialisierung einer Klasse ist folgendermaßen: Zunächst werden die Basisklassenkonstruktoren aufgerufen in der Reihenfolge, wie sie abgeleitet wurden. Dann werden die Konstruktoren der Datenelemente der Klasse aufgerufen in der Reihenfolge, wie sie definiert sind. Zum Schluß wird der Konstruktor der Klasse aufgerufen.

Z.B.:

```
class A { --- };
class B : A { --- };
class C { --- };
class D : C, B
   {
   public:
      D ();
      C  x;
      B  y;
   };
```

Reihenfolge des Aufrufs von Konstruktoren bei `D d`:

1. Konstruktor von `C` für Basisklasse `C` von Objekt `d`
2. Konstruktor von `A` für Basisklasse `A` von Objekt `d`
3. Konstruktor von `B` für Basisklasse `B` von Objekt `d`
4. Konstruktor von `C` für Klassenelement `x`
5. Konstruktor von `A` für Basisklasse `A` von Klassenelement `y`
6. Konstruktor von `B` für Klassenelement `y`
7. Konstruktor von `D` für Objekt `d`

Die Destruktoren werden in umgekehrter Reihenfolge aufgerufen.

Wenn eine Klasse eine virtuelle Funktion enthält, dann sollte sie auch einen virtuellen Destruktor haben, damit bei einem Pointer `p` vom Typ dieser Klasse **delete** `p` weiß, welcher Destruktor aufzurufen ist, falls `p` auf ein Objekt einer abgeleiteten Klasse zeigt.

Wenn ein Konstruktor oder ein Destruktor virtuelle Funktionen aufruft, dann nimmt er immer die in der *eigenen* Klasse definierte Überladung der virtuellen Funktion. Das verdeutlicht das folgende Beispiel:

// CtorDtor.cpp

```
#include <iostream>

using namespace std;

class A
   {
   public:
      A () { cout << "A-Konstruktor    : "; f (); }
      ~A () { cout << "A-Destruktor     : "; f (); }
```

```
    virtual void  f () { cout << "f kommt von A." << endl; }
    void meth () { cout << "A-meth : "; f (); }
  };

class B : public A
  {
  public:
    B () { cout << "B-Konstruktor    : "; f (); }
    virtual ~B () { cout << "B-Destruktor    : "; f (); }
    void f () { cout << "f kommt von B." << endl; }
  };

class C : public B
  {
  public:
    C () { cout << "C-Konstruktor    : "; f (); }
    ~C () { cout << "C-Destruktor    : "; f (); }
    void f () { cout << "f kommt von C." << endl; }
  };

main ()
  {
  {
  cout << endl;
  A  a;
  cout << "a.meth () : "; a.meth ();
  }
  {
  cout << endl;
  C  c;
  cout << "c.meth () : "; c.meth ();
  }
  {
  cout << endl;
  A* pa = new C;
  cout << "pa->meth (): "; pa->meth ();
  delete pa;
  }
  {
  cout << endl;
  B* pb = new C;
  cout << "pb->meth (): "; pb->meth ();
  delete pb;
  }
  }
```

Die Ausführung dieses Programms hat folgendes Resultat:

```
A-Konstruktor     : f kommt von A.
```

```
a.meth () : A-meth : f kommt von A.
A-Destruktor      : f kommt von A.

A-Konstruktor     : f kommt von A.
B-Konstruktor     : f kommt von B.
C-Konstruktor     : f kommt von C.
c.meth () : A-meth : f kommt von C.
C-Destruktor      : f kommt von C.
B-Destruktor      : f kommt von B.
A-Destruktor      : f kommt von A.

A-Konstruktor     : f kommt von A.
B-Konstruktor     : f kommt von B.
C-Konstruktor     : f kommt von C.
pa->meth (): A-meth : f kommt von C.
A-Destruktor      : f kommt von A.

A-Konstruktor     : f kommt von A.
B-Konstruktor     : f kommt von B.
C-Konstruktor     : f kommt von C.
pb->meth (): A-meth : f kommt von C.
C-Destruktor      : f kommt von C.
B-Destruktor      : f kommt von B.
A-Destruktor      : f kommt von A.
```

8.3 Statische und dynamische Bindung

Nicht-virtuelle Funktionen werden statisch gebunden, virtuelle dynamisch. Ein Zeiger oder eine Referenz hat einen statischen und einen dynamischen Typ. Der statische Typ eines Zeigers oder einer Referenz ist der Typ, der bei seiner Definition angegeben wird.

```
A* pa;  // statischer Typ von pa ist A
        // dynamischer Typ von pa ist A

B  b;
pa = &b; // dynamischer Typ von pa ist nun B
```

Zeigern und Referenzen können aber u.U. Adressen von Objekten anderen Typs, insbesondere von abgeleiteten Klassen zugewiesen werden. Durch solch eine Zuweisung wird nur der dynamische Typ verändert. Dieser Unterschied zwischen statischem und dynamischem Typ wird deutlich bei der unterschiedlichen Behandlung von virtuellen und nicht-virtuellen Funktionen und bei Defaultparametern.

Defaultparameter werden *immer* – auch bei virtuellen Funktionen (!) – statisch gebunden. Wenn man daher Defaultparameter in abgeleiteten Klassen ändert,

führt das zu sehr unerwartetem Verhalten. Daher sollte man nie Defaultparameter redefinieren. Folgendes Beispiel illustriert das:

```
#include <iostream>

using namespace std;

class A
   {
   public:
      virtual void drucke (int x = 5) = 0;
   };

class B : public A
   {
   public:
      void drucke (int x = 6)
         {
         cout << "x = " << x << "\n";
         }
   };

main ()
   {
   A  *pa;
   B  b;
   pa = &b;
   pa->drucke ();
   b.drucke ();
   return 0;
   }
```

Laufzeiteffizienz ist der Grund für diese merkwürdige Spracheigenschaft.

8.4 Implementation von objektorientiertem Design in C++

In diesem Abschnitt zeigen wir, wie das Resultat einer objektorientierten Analyse bzw Design in C++ implementiert werden kann.

Das folgende ist nur eine kurze Betrachtung aus der Sicht der Sprache. D.h. wir fragen uns, was bedeutet wohl öffentliches oder privates Erben usw.

8.4.1 "Ist-ein" – Beziehung

Öffentliches (**public**) Erben bedeutet eine "Ist-ein"-Beziehung (`is-a`).

```
class B : public A {---};
```

`B` ist ein `A`. Alles, was für `A` zutrifft, trifft auch für `B` zu. Die Objekte von `B` sind eine Teilmenge der Objekte von `A`. Objekte von `B` können Objekte von `A` voll ersetzen *(Substitutionsprinzip)*.

Nicht-virtuelle Funktionen von `A` sind Funktionen, deren Schnittstelle *und* deren Implementation geerbt werden sollen. Es ist daher nicht vernünftig, nicht-virtuelle Funktionen zu überschreiben, da es dann zu unterschiedlichem Verhalten kommt, jenachdem ob ein Zeiger bzw eine Referenz auf `A` oder `B` verwendet wird.

Bei virtuellen Funktionen hingegen soll nur die Schnittstelle geerbt werden. Wenn diese Funktionen noch rein virtuell sind, dann wird die erbende Klasse zu einer eigenen Implementation dieser Funktionen gezwungen. Damit kann verhindert werden, daß ungewollt eine Standardimplementation verwendet wird. Die Standardimplementation kann aber trotzdem bereitgestellt werden.

8.4.2 "Ist-fast-ein" – Beziehung

Die Beziehung "Ist-fast-ein" (`is-like-a`) ist dadurch charakterisiert, daß die Teilmengen-Beziehung der "Ist-ein"-Beziehung

```
B is-like-an A
```

insofern nicht gilt, als die Objekte der links stehende Klasse `B` *nicht* vollständig Objekte der Klasse `A` subtituieren können. Die Klasse `B` ist häufig eingeschränkt (Quadrat ist ein eingeschränktes Rechteck), bietet aber auch zusätzliche Methoden an.

Diese Beziehung kann man durch *private* Vererbung implementieren, wobei ja alle Methoden und Datenelemente des Elters `A` für das Kind `B` zunächst verboten werden. Durch Access-Deklarationen kann dann ein Teil wieder öffentlich gemacht werden. Daher sprechen wir auch von einer partiellen privaten Vererbung.

```
class B : private A
  {
  public:
    // Access−Deklarationen
  };
```

8.4.3 "Hat-ein" – Beziehung

Die "Hat-ein"-Beziehung (`B` hat ein `A`) wird auch als "Ist-implementiert-mit"-Beziehung, ***Layering, Containment, Embedding,*** **Einbettung** bezeichnet (`has-a`). Ein Objekt vom Typ `A` ist Komponente von einem Objekt vom Typ `B`. `B` heißt auch Komponentengruppe.

Die "Hat-ein" Beziehung wird implementiert, indem

- entweder ein Objekt von `A` ein Datenelement von `B` wird

- oder eine Referenz oder ein Zeiger von `A` ein Datenelement von `B` wird. Diese Möglichkeit hat zwar den Nachteil, daß die Existenz des Objekts extra verwaltet werden muß, aber den großen Vorteil, daß der Polymorphismus ausgenutzt werden kann, wenn `A` in – oder vorzugsweise am Anfang – einer "Ist-ein"-Vererbungshierarchie steht.

  D.h. Erben von `B` können `A*` oder `A&` mit Erben von `A` belegen oder initialisieren.

```
class A {---};

class B
   {
   private:
      A  a;
   };
```

Eine andere Möglichkeit ist private Erblichkeit zu verwenden:

```
class B : private A {---};
```

Da man mit einem `B`-Objekt global nicht auf die Methoden von `A` zugreifen kann, ist dies sicherlich keine "Ist-ein"-Beziehung. Bei privater Vererbung wird nur die Implementation von `A` geerbt. Ist `B` Mitglied einer anderen – etwa "Ist-ein" – Klassenhierarchie, dann hat man sofort mit Mehrfachvererbung zu tun, auf deren Probleme noch einzugehen ist. Daher ist die Möglichkeit, die "Hat-ein"-Beziehung über private Erblichkeit zu realisieren, nicht zu empfehlen.

Private Erblichkeit ist aber nur schwer zu umgehen, wenn `B` auf **protected** Elemente von `A` zugreifen muß oder wenn virtuelle Funktionen von `A` redefiniert werden müssen.

Da die Komponenten (`A`) einer Komponentengruppe (`B`) ganz zu `B` gehören, müssen sie bei einer Kopie eines Objekts vom Typ `B` auch vollständig kopiert werden. Es müssen also ein Copy-Konstruktur und ein Zuweisungsoperator implementiert werden, die bezüglich der Komponenten der "Hat-ein"-Beziehung eine "tiefe" Kopie machen. Das ist gewährleistet, wenn die Komponenten als Objekte und *nicht* als Zeiger oder Referenz angelegt werden.

8.4.4 "Benutzt-ein" – Beziehung

Wenn Objekte einer Klasse `B` die Dienste von Objekten einer anderen Klasse `A` benutzen und die Objekte von `A` nicht als Teile betrachtet werden können, spricht man von einer "Benutzt-ein" – Beziehung (`uses-a`).

Implementieren kann man dies, indem man einen *Zeiger* oder eine *Referenz* auf ein Objekt der Klasse `A` als Datenelement in der Klasse `B` führt. Das `A`-Objekt ist also unabhängig vom Objekt der Klasse `B` lebensfähig. Verwendet man eine Referenz, dann muß eine Zuweisung in den Konstruktoren von `B` erfolgen. D.h. das `A`-Objekt muß vor dem entsprechenden `B`-Objekt existieren und das `A`-Objekt kann nicht durch ein anderes (ohne Kopie) ausgetauscht werden.

```
class A {---};

class B
   {
   private:
      A* pa;
      A& ra;
   };
```

Es ist auch möglich, daß eine Methode von `B` sich ein Objekt der Klasse `A` kreiert und dann dessen Methoden verwendet.

Beim Copy-Konstruktor und beim Zuweisungsoperator der Klasse `B` muß darauf geachtet werden, daß bezüglich der "benutzten" Objekte nur "flache" Kopien gemacht werden, d.h. daß nur der Zeiger auf das benutzte Objekt, nicht das Objekt selbst kopiert wird. Da bei einer Referenz eine flache Zuweisung nicht möglich ist, *muß* der Zuweisungsoperator normalerweise *verboten* werden. (Denn Referenzen können nicht umgebogen werden.).

8.4.5 Andere Beziehungen

Beziehungen, die nicht in eine der oben genannten Kategorien passen, werden in den meisten Fällen – insbesondere, wenn die Beziehung Eigenschaften und Verhalten hat, – durch Definition einer eigenen Klasse implementiert. Diese Klasse enthält dann Zeiger oder Behälter von Zeigern, die auf die Partner der Beziehung zeigen.

Zusätzlich können – aus Effizienzgründen – Zeiger auf die Beziehung in den Partnerklassen verwaltet werden.

8.4.6 Botschaften

In der Objektorientierung spielen Botschaften *messages* an Objekte von anderen Objekten eine große Rolle. In C++ wird das Senden einer Botschaft an ein anderes Objekt dadurch implementiert, daß eine Methode des die Botschaft empfangenden Objekts aufgerufen wird.

8.4.7 Mehrfachvererbung

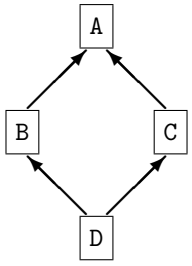
Mehrfachvererbung ist umstritten. Smalltalk kennt Mehrfachvererbung nicht. Unter C++ hat man als erstes mit dem Problem der Ambiguität zu tun, wenn derselbe Name von verschiedenen Klassen geerbt wird. Man ist gezwungen explizit den Bezug anzugeben, was für den Anwender einer Klasse bedeutet, daß er eine genaue Kenntnis der Klassenhierarchie haben muß.

Das zweite Problem ist, daß eine tiefer liegende Klasse `A` mehrfach geerbt wird. Das kann verhindert werden, indem `A` von allen Klassen virtuell geerbt wird. Aber dazu muß eventuell das Interface dieser Klassen umgeschrieben werden,

was für die Erstellung von Bibliotheken nicht akzeptabel ist. Der Ausweg, generell nur virtuell zu erben, hat Speicher- und Laufzeitkosten zur Folge.

Ein weiteres, schlimmeres Problem bei virtueller Vererbung ist, daß die Argumente zur Initialisierung virtueller Basisklassen in der *letzten* abgeleiteten Klasse angegeben werden müssen.

Betrachten wir noch einmal folgenden Erblichkeitsgraphen:



Wenn `A` eine virtuelle Funktion `f` definiert, die von `B`, nicht aber von `C` redefiniert wird, dann ist der Aufruf

```
D  d;
d.f ();
```

ambig, wenn `A` nicht-virtuell von `B` und `C` geerbt wird. Der Aufruf `d.f ()` bedeutet `d.B::f ()`, wenn `A` virtuell von `B` und `C` geerbt wird. Das kann mit folgendem Programm getestet werden:

```
#include <iostream>

using namespace std;

class A
   {
   public:
      virtual void drucke ()
         {
         cout << "Drucke von A.\n";
         }
   };

class B : virtual public A
   {
   public:
      void drucke ()
         {
         cout << "Drucke von B.\n";
         }
   };

class C : virtual public A
   {
```

```
   public:
   };

class D : public B, public C
   {
   public:
   };

main ()
   {
   A* pa;
   D  d;
   pa = &d;
   pa->drucke ();
   d.drucke ();
   return 0;
   }
```

Downcasting heißt den Zeiger einer Basisklasse in einen Zeiger einer abgeleiteten Klasse zu casten. Downcasting von einer virtuell geerbten Basisklasse auf eine abgeleitete Klasse ist nicht erlaubt. Das ist nicht so schlimm, da man sowieso Downcasting vermeiden sollte.

Mehrfachvererbung verliert etwas von ihrer Komplexität, wenn man auf virtuelle Basisklassen verzichten kann. Das geht dann leicht ohne Speicherkosten, wenn man abstrakte Basisklassen ohne Datenelemente und Konstruktoren, aber mit einem virtuellen Destruktor verwendet.

Zusammenfassung: Man vermeide Mehrfachvererbung.

8.5 Komplexe Zahlen

Die Datenrepräsentation von komplexen Zahlen kann wie bei zweidimensionalen Vektoren durch zwei Zahlen vom Typ **double** erfolgen (`x + i y`). Addition, Subtraktion, Betragsberechnung, Negierung werden wie bei zweidimensionalen Vektoren kodiert.

Hier bietet es sich beim Entwurf der Klasse `Komplex` zunächst einmal alles von der Klasse `ZweiDimVek` zu erben. Dann müssen Methoden wie zum Beispiel "konjugiert komplex" ergänzt werden. Andere Funktionen wie z.B. das Produkt müssen überschrieben werden, da sie anders kodiert werden.

Da wir Zugang zu den privaten Elementen von `ZweiDimVek` haben wollen, müssen wir auch in diese Klasse eingreifen und **private** zu **protected** abändern. (Der bessere Weg wäre aber gewesen, die Zugriffsfunktionen zu verwenden.)

Header-File: `ZweiDimVek.h`

```
// ZweiDimVek.h

#ifndef ZweiDimVek_h
```

```
#define ZweiDimVek_h

#include <iostream>

using namespace std;

class ZweiDimVek
   {
   friend double operator * (
      const ZweiDimVek& v1,
      const ZweiDimVek& v2);
   friend ostream& operator << (
      ostream& s,
      const ZweiDimVek& v);
   public:
      ZweiDimVek ();         // Default-Konstruktor
      ZweiDimVek (double a, double b); // Konstruktor
      ZweiDimVek (ZweiDimVek& v); // Copy-Konstruktor
      double  getX1 () const { return x1; }
      double  getX2 () const { return x2; }
      void  setX1 (double a) { x1 = a; }
      void  setX2 (double a) { x2 = a; }
      double  betrag () const;
      void  drucke () const;
      ZweiDimVek operator - () const;
   protected:
      double  x1;
      double  x2;
   };

#endif
```

Header-File: `Komplex.h`

```
// Komplex.h

#ifndef Komplex_h
#define Komplex_h

#include "ZweiDimVek.h"

class Komplex : public ZweiDimVek
   {
   friend Komplex operator * (
      const Komplex& z1,
      const Komplex& z2);
   public:
      Komplex ();          // Default-Konstruktor
      Komplex (double a, double b); // Konstruktor
      Komplex (Komplex& z); // Copy-Konstruktor
```

```
    void drucke () const;
    Komplex operator ~() const; // konjugiert komplex
  };

#endif
```

Implementations-File: `Komplex.cpp`

```
// Komplex.cpp

#include <iostream>
#include <cmath>
#include "Komplex.h"

using namespace std;

Komplex::Komplex () : ZweiDimVek () {}

Komplex::Komplex (double a, double b)
  : ZweiDimVek (a, b)
  { }

Komplex::Komplex (Komplex& z)
  : ZweiDimVek (z)
  { }

Komplex operator * (
  const Komplex& z1,
  const Komplex& z2)
  {
  Komplex z;
  z.x1 = z1.x1 * z2.x1 - (z1.x2 * z2.x2);
  z.x2 = z1.x1 * z2.x2 + z1.x2 * z2.x1;
  return z;
  }

void Komplex::drucke () const
  {
  cout << x1 << " + i" << x2 << "\n";
  }

Komplex Komplex::operator ~() const
  {
  Komplex z;
  z.x1 = x1;
  z.x2 = -x2;
  return z;
  }
```

Anwendungs-File: `apl.cpp`

```
// apl.cpp – Anwendungsprogramm für Komplex

#include <iostream>
#include "Komplex.h"

using namespace std;

main ()
   {
   Komplex z;
   Komplex z1 (2.0, 2.0);
   Komplex z2 (2.0, 3.0);
   Komplex z3 (z1);
   cout << "z: "; z.drucke ();
   cout << "z1: "; z1.drucke ();
   cout << "z2: "; z2.drucke ();
   cout << "z3: "; z3.drucke ();
   cout << "Das Produkt z1*z2 ist " << z1 * z2 << "\n";
   cout << "Der Betrag von z3 ist " << z3.betrag () << "\n";
   cout << "Minus von z3 ist " << -z3 << "\n";
   cout << "Konjugiertkomplex von z3 ist " << ~z3 << "\n";
   return 0;
   }
```

8.6 Übungen

Übung Quadrat – Rechteck: Erweitern Sie die Übung des Kapitels "Einleitung" um die Klasse `Quadrat`, wobei möglichst viel Code wiederverwendet werden soll.

Übung Zugriffsrechte:

Ersetze im unten angegebenen Programm `Art_der_Vererbung` nacheinander durch **private**, **protected** und **public** und übersetze. Wo schimpft der Compiler?

```
class A
   {
   public:
      int   x;
   protected:
      int   y;
   private:
      int   z;
   };

class B : Art_der_Vererbung A
   {
   public:
```

```
      void f ()
         {
         x = 1;
         y = 1;
         z = 1;
         }
   };

main ()
   {
   B  b;
   b.x = 1;
   b.y = 1;
   b.z = 1;
   }
```

Übung Virtuelle Funktionen:

Probiere die verschiedenen Möglichkeiten an dem oben gegebenen Beispiel durch.

Kapitel 9

Templates

`T` sei ein beliebiger Typ (elementarer Typ oder Klasse). Die unten definierte Klasse `Feld` stellt ein Feld von Objekten des Typs `T` zur Verfügung mit beliebigen oberen und unteren Indexgrenzen. Die Indexgrenzen werden bei Operationen auf diesem Feld überprüft.

```
class Feld
   {
   public:
      Feld (int unterIndexgrenze = 0, int obereIndexgrenze = 0);
      virtual ~Feld ();
      T& operator [] (int Index);
      const T& operator [] (int Index) const;
   private:
      int   uIg;
      int   oIg;
      T* p;
      void check (int& Index) const;
   };
```

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

using namespace std;

Feld::Feld
   (
   int unterIndexgrenze,
   int obereIndexgrenze
   )
   :  uIg (unterIndexgrenze),
      oIg (obereIndexgrenze)
   {
   if (oIg < uIg)
      {
      cerr << "Fehler: Obere Indexgrenze ist ";
      cerr << "kleiner als untere Indexgrenze!\n";
      oIg = uIg;
      }
   p = new T[oIg - uIg + 1];
   if (p == 0)
      {
      cerr << "Fehler: Speicher konnte nicht ";
      cerr << "allokiert werden!\n";
      }
   }

Feld::~Feld () { delete [] p; }

void Feld::check (int& Index) const
   {
   if (Index < uIg || Index > oIg)
      {
      cerr << "Fehler: Feldindex ";
      cerr << "außerhalb der Grenzen!\n";
      if (Index > oIg)
         {
         Index = oIg;
         cerr << "Obere Grenze " << oIg;
         cerr << " wird verwendet.\n";
         }
      else
         {
         Index = uIg;
         cerr << "Untere Grenze " << uIg;
         cerr << " wird verwendet.\n";
         }
      }
   }
```

```
T& Feld::operator [] (int Index)
   {
   check (Index);
   return p[Index - uIg];
   }

const T& Feld::operator [] (int Index) const
   {
   check (Index);
   return p[Index - uIg];
   }
```

Wenn `Feld` ein Feld von **double** sein soll, dann müssen wir `T` durch **double** ersetzen. Wenn wir ein Feld für Objekte der Klasse `Zeichenkette` benötigen, müssen wir den Code für `Feld` duplizieren und `T` durch `Zeichenkette` ersetzen.

Vervielfältigung von fast identischem Code ist sehr schwer zu warten. Wenn z.B. die Indexcheck-Routine verändert werden soll, dann muß man alle Kopien von `Feld` durchgehen.

Es wäre daher schön, wenn man im Code von `Feld` einen variablen Typ `T` einsetzen könnte bzw wenn Typ `T` ein Argument der Klasse `Feld` wäre. C++ bietet dafür die Möglichkeit, den Code der Klasse `Feld` als ***Template*** **(generischer oder parametrisierter Datentyp)** aufzufassen, indem man vor die Klassendefinition von `Feld`

```
template <class T>
```

schreibt:

```
template <class T>
class Feld {---};
```

Ansonsten kann der Code der `Feld`-Definition unverändert bleiben.

Die Köpfe der Elementfunktionen von Feld müssen folgendermaßen ergänzt werden:

```
template <class T> Feld<T>::Feld (
// ---

template <class T> Feld<T>::~Feld () { delete [] p; }
// ---

template <class T> T& Feld<T>::operator [] (int Index)
// ---

template <class T> T& Feld<T>::operator [] (int Index) const
// ---
```

```
template <class T> void Feld<T>::check (int& Index) const
// ---
```

Das Template-Konzept ist insbesondere für Behälterklassen oder Container-Klassen wie Felder, Listen, Mengen, Stacks und assoziative Felder geeignet.

Obwohle es **<class T>** heißt, muß T keine Klasse sein, sondern kann auch ein elementarer Typ sein. Ferner kann auch eine durch Komma getrennte Liste von Typen übergeben werden:

```
template <class T1, class T2, class T3>
class ABC {---};
```

Ein Feld für **double** wird nun folgendermaßen angelegt:

```
Feld<double> fd (-12, 12);
```

oder für Objekte der Klasse Zeichenkette

```
Feld<Zeichenkette> fZ (1, 100);
```

Die Parameter von **template** können außer Typen auch konstante Ausdrücke, Strings und Funktionsnamen sein, z.B. **int z** oder **char* s** oder **int f (double x)**.

9.1 Funktionstemplate

Mit dem Template-Konzept können auch globale Nicht-Elementfunktionen definiert werden:

```
template <class T> void funk (---) {---}
```

Da solche Funktionen meistens sinnvoller als Elementfunktionen eines Typs deklariert werden, gibt es kaum vernünftige Anwendungen für Funktionstemplates.

Als Beispiel wird hier eine Funktion definiert, die zwei Objekte desselben Typs vertauscht.

Headerfile:

```
#ifndef vertausch_h
#define vertausch_h

template <class T> void vertausch (T& a, T& b);

#include "vertausch.C"

#endif
```

Definitionsfile:

```
template <class T> void vertausch (T& a, T& b)
   {
   T  temp;
   temp = a;
   a = b;
   b = temp;
   }
```

Anwendung:

```
double  a = 3;
double  b = 4;
vertausch (a, b);
```

9.2 Compiler

Es gibt zwei Compilationsmodelle:

- ***inclusion model:*** Die Implementation der Templatefunktionen wird in den Headerfile inputiert, wobei Templates nicht getrennt übersetzt werden.
- ***separation model:*** Die Templates werden getrennt übersetzt. Das ist zwar schwerer zu implementieren und fordert längere Compilezeiten für die einzelnen Files, erlaubt aber eine bessere Trennung von Schnittstelle und Implementation, entspricht dem Standardmodell der Sprache (getrennte Compilation) und erlaubt schnellere Compilation von großen Programmen.

9.2.1 CC unter HPUX

Der Headerfile muß die Extension `.h` haben. Der Implementationsfile muß denselben Namen wie der Headerfile, aber die Extension `.C` haben. Der Implementationsfile darf den Header-File nicht inputieren.

9.2.2 xlC unter AIX

Der Headerfile muß die Extension `.h` haben. Der Implementationsfile muß denselben Namen wie der Headerfile, aber die Extension `.c` haben. Der Implementationsfile darf den Header-File nicht inputieren.

9.2.3 g++ (GNU-Compiler)

Der Headerfile muß am Ende den Implementationsfile inputieren. z.B.:

```
#include "Stack.C"
```

Der Implementationsfile darf den Header-File nicht inputieren.

Übersetzt wird nur der Applikationfile in zwei Stufen. Z.B.:

```
g++ -frepo -c apl.C
g++ apl.o
```

`-frepo` darf nur beim Übersetzen, nicht beim Linken angegeben werden.

9.3 Übungen

Übung Stack

1. Schreiben Sie eine Template-Klasse `Stack` (Stack von beliebigen Typen), wobei der zu verwaltende Typ ein erster Templateparameter und die Stackgröße ein zweiter Templateparameter ist.

   Die Klasse `Stack` soll zwei Methoden haben: `push` und `pop`, deren Rückgabewerte `0` oder `1` sind, jenachdem ob die Operation erfolgreich war.

2. Implementieren Sie `push` und `pop` als Überladungen der Operatoren `<<` und `>>`.

3. Schreiben Sie einen Operator der einen Stack nach **int** casted, sodaß man `0` erhält, wenn der Stack entweder ganz voll oder ganz leer ist, `1` sonst. Dann kann man nämlich Code wie z.B.
   while `(stack << x);`
   schreiben. (Dieser Cast-Operator ist natürlich gefährlich, weil er nicht zwischen voll und leer unterscheidet, und daher für die Praxis nicht zu empfehlen. Als Übung ist das allerdings recht nett.)

Kapitel 10

Exception Handling

Mit *Exception Handling* bietet C++ eine Möglichkeit zur Fehlerbehandlung. Die Idee ist, daß eine Funktion, die einen Fehler selbst nicht behandeln kann, eine Exception wirft, in der Hoffnung, daß eine andere Funktion den Fehler behandeln kann.

Dabei spielen die Schlüsselworte **try**, **throw** und **catch** eine Rolle. Die Syntax ist folgendermaßen:

```
// ---
try
   {
   // ---
   throw Objekt;
   // ---
   }
   catch (Typ_of_Objekt x)
      {
      // ---
      }
```

oder

```
void f ()
   {
   // ---
   if (Fehler) throw Objekt; // vom Typ
                             // Typ1 oder Typ2 oder Typ3
   // ---
   }

void ff ()
   {
   // ---
   try
      {
      // ---
      --- f () ---
      // ---
      }
      catch (Typ1 x)
         {
         // behandle Fehler von der Art Typ1
         }
      catch (Typ2 x)
         {
         // behandle Fehler von der Art Typ2
         }
      catch (Typ3& x)
         {
         // behandle Fehler von der Art Typ3
         }
      // weiter
      // ---
   }
```

Wenn innerhalb des **try**`{---}`-Blocks ein Objekt geworfen wird, dann wird nach einem **catch**`(---){---}`-Block gesucht, dessen Argumenttyp der Typ des geworfenen Objekts ist. Wenn solch ein **catch** gefunden wird, dann wird der dazugehörige Block abgearbeitet. Nur *ein* **catch**-Block wird abgearbeitet. Die folgenden **catch**-Blöcke werden übersprungen und es geht bei `weiter` weiter. Als Argument von **catch** kann auch `...` verwendet werden (**catch** `(...)`), womit jede Art von Objekt gefangen wird.

Im **catch** kann man den Typ auch als Referenz (`Typ3&`) deklarieren. Das hat den Vorteil, daß das geworfene Objekt nicht kopiert wird. Allerdings sollte es dann außerhalb des **try** Blocks definiert worden sein und nicht "weiter" geworfen werden.

Das **catch** `(...)` verträgt sich mit anderen **catch**es, muß allerdings an letzter Stelle stehen, da es sonst alle geworfenen Objekte abfangen würde.

try-catch kann man verschachteln. Jeder **catch**-Block ist ein Exception-Handler. Wird kein geeigneter **catch**-Block gefunden, wird nach einem eventuell

darüberliegenden **try-catch**-Konstrukt gesucht. Wird überhaupt kein geeignetes **catch** gefunden wird das Programm abgebrochen.

Durch ein **throw** werden alle Objekte deinitialisiert (Aufruf von Destruktoren), die zwischen **try** und **throw** erzeugt worden sind. Wenn in einem Konstruktor eine Exception auftritt, wird das Objekt nicht erzeugt und damit dann auch kein Destruktor aufgerufen. Dynamisch angelegter Speicher wird nur aufgeräumt, wenn sich ein Destruktor (oder der **catch**-Block) darum kümmert.

Bezüglich der Sichtbarkeit von Objekten in **try**-Blöcken, **catch**-Blöcken und außerhalb gelten die üblichen Regeln.

Hier ein lauffähiges Beispiel zur Demonstration der Syntax:

```
#include <iostream>

using namespace std;

class Excep
   {
   public:
      Excep (int i)
         : Zahl (i) { }
      int   Zahl;
   };

void f ()
   {
   int   i;
   cout << "Eingabe Nummer des Excep-Objekts <bisher nur 5> :";
   cin >> i;
   if (i == 5) throw Excep (5);
   }

void ff ()
   {
   try
      {
      f ();
      }
      catch (Excep x)
         {
         cerr << "Excep Nr. " << x.Zahl << " wurde geworfen.\n";
         }
   }

main ()
   {
   while (1)
      {
      ff ();
      }
```

```
   }
```

Wenn in **catch** der Fehler zwar behandelt wurde, dann soll u.U. der Fehler an einen höheren **try–catch**-Konstrukt weitergereicht werden. Das ist möglich durch ein **throw** ohne Argumente im **catch**-Block.

```
catch (Excep x)
   {
   // ---
   throw;
   }
```

Damit der Anwender einer Funktion weiß, ob die Funktion eventuell ein **throw** macht, kann dies spezifiziert werden mit folgender Syntax:

`void f1 () throw (E1, E2, E5);` // Die Funktion `f1` kann die Exceptions `E1`, `E2` und `E5` werfen.

`void f2 () throw ();` // Die Funktion `f2` kann *keine* Exception werfen.

`void f3 ();` // Die Funktion `f3` kann *alle möglichen* Exceptions werfen.

Die Wirkung ist so, daß der Compiler dies nicht überprüft, weil er es allgemein wohl nicht kann. Zur Laufzeit wird ein nicht spezifierter **throw** weitergereicht und führt eventuell zum Abruch des Programms.

Man kann Fehler gruppieren, indem man die Fehler von einem allgemeinen Fehlertyp ableitet:

```
class E {---};       // allgemeiner Fehlertyp
class E1 : E {---};
class E2 : E {---};
class E3 : E {---};

try
   {
   // ---
   }
   catch (E2) {---}   // fängt Fehlertyp E2
   catch (E) {---}    // fängt Fehlertyp E, E1, E3
// ---
```

Es kommt auf die Reihenfolge an: Wenn **catch** `(E)` vor **catch** `(E2)` steht, dann fängt dieses alle Sohn-Objekte, also auch die Objekte vom Typ `E2` ab.

10.1 Behandlung von UNIX-Systemfehlern

Unter UNIX generieren Betriebssystemfehler eines von 32 Signalen. Dieses Signal wird dem fehlerverursachenden Prozeß geschickt. Für die meisten Signale stellt das Betriebssystem Default-Behandlungsroutinen zur Verfügung.

Die Systemfunktion `signal` bitetet die Möglichkeit, jedem Signal eine neue Behandlungsroutine zuzuweisen. In unserem unten angegebenen Beispiel heißt die Routine `signalBehandler`. In dem Beispiel sind zwei Klassen definiert `Signal` und `BehandeltesSignal`. Objekte der Klasse `Signal` werden geworfen, wenn ein entsprechendes Signal vom Betriebssystem generiert wurde. Die Klasse `BehandeltesSignal` verwaltet für jedes Signal die Behandlungsroutine.

Beispiel:

```
// Signal.h

#ifndef Signal_h
#define Signal_h

class Signal
   {
   public:
      Signal (int sig);
      char* name ();
      void behandle ();

   private:
      int   sig;
      char sigName[8];
   };

#endif
```

```
// Signal.C

#include "Signal.h"
#include <signal.h>
#include <iostream>
#include <strstream>

using namespace std;

Signal::Signal (int sig)
   : sig (sig)
   {
   ostrstream buf (sigName, 8);
   switch (sig)
      {
      case SIGSEGV : buf << "SIGSEGV" << '\0'; break;
      default : buf << "unbek." << '\0'; break;
      }
   }

char* Signal::name ()
   {
   return sigName;
```

```
   }

void Signal::behandle ()
   {
   cerr << "Betriebssystemsignal " << name () << " (" << sig << ")
        wurde";
   cerr << " geschickt!" << endl;
   }

// BehandeltesSignal.h

#ifndef BehandeltesSignal_h
#define BehandeltesSignal_h

#include "Signal.h"

extern "C" void eigenerSignalBehandler (int sig);

class BehandeltesSignal
   {
   public:
      BehandeltesSignal (int sig);
      ~BehandeltesSignal ();

   private:
      int   sig;
      void (*alterBehandler) (int sig);
   };

#endif

// BehandeltesSignal.C

#include "BehandeltesSignal.h"
#include <signal.h>

void eigenerSignalBehandler (int sig)
   {
   throw Signal (sig);
   }

BehandeltesSignal::BehandeltesSignal (int sig)
   :  sig (sig)
   {
   alterBehandler = signal (sig, eigenerSignalBehandler);
   }

BehandeltesSignal::~BehandeltesSignal ()
   {
   signal (sig, alterBehandler);
```

```
   }

// apl.C – Anwendung UNIX–Signal–Behandlung

#include "Signal.h"
#include "BehandeltesSignal.h"
#include <signal.h>

int   main ()
   {
   BehandeltesSignal segmentation (SIGSEGV);
   int*  p = (int*)0xffffffff;
   int   i;
   try
      {
      i = *p;
      }
      catch (Signal& sig)
         {
         sig.behandle ();
         }
   return 0;
   }
```

10.2 Compiler

10.2.1 CC unter HPUX

Programme, die Exceptions benutzen, müssen mit der Compileroption `+eh` übersetzt werden.

10.2.2 g++ (GNU-Compiler)

Programme, die Exceptions benutzen, müssen mit den Compileroptionen `-fhandle-exceptions -frtti` übersetzt werden.

10.3 Übungen

Übung zum Exception-Handling:

1. Übersetze den File eh.C und laß das Programm laufen. Versuche zu verstehen, was passiert.

2. Wirf auch mal ein anderes Objekt vom Typ `Excep` (z.B. Nr. 7).

3. Reiche das `Excep`-Objekt Nr. 99 so weiter, daß das Programm endlich abgebrochen wird.
4. Schreibe drei Erben `E1`, `E2` und `E3` von `Excep` und wende sie an, wobei Fehler gruppiert werden sollen.
5. Definiere eine neue Klasse, die als Exception geworfen wird.
6. Spezifiziere, welche Exception die Funktionen des Programms jeweils werfen.
7. Gib anstatt einer Zahl einen Buchstaben ein. Warum läuft der Bildschirm voll? Fange das durch Ausnahmebehandlung ab (vgl. Kapitel Streams).

Übung Index außerhalb der Grenzen:

Im Beispiel `Zeichenkette` wurden bisher bei den Operatoren `[]` und `()` mögliche Bereichsverletzungen der Indizes nicht abgefangen. Behandle den Fehler mit Exception-Handling so, daß der Benutzer einen neuen Index eingeben muß.

Kapitel 11

Streams

C++ selbst bietet keine Sprachmittel für Ein- und Ausgabe. Einerseits kann man einfach die I/O-Funktionen von C verwenden, indem man `<stdio.h>` inputiert.

```
#include <stdio.h>

main ()
   {
   printf ("Guten Tag\n");
   return 0;
   }
```

Andererseits wird üblicherweise eine C++ Bibliothek für Ein- und Ausgabe zur Verfügung gestellt, die auf dem Konzept der Konversion von typisierten Objekten in **Zeichenfolgen** *(streams)* und umgekehrt aufgebaut ist. Dieses Konzept ist wesentlich eleganter und insgesamt leichter zu erlernen.

Wesentliche Vorteile des Stream-Konzepts sind Typensicherheit und die leichte Einbindbarkeit in vom Benutzer definierte Klassen.

Zur Benutzung der Klassen dieser Bibliothek muß `<iostream>` inputiert werden. Drei Stream-Objekte werden dort schon zur Verfügung gestellt:

`cin` für Standardeingabe vom Typ `istream`
`cout` für Standardausgabe vom Typ `ostream`
`cerr` für Standardfehlerausgabe vom Typ `ostream`

Als Handbuch empfiehlt sich das Buch von Teale. [8]

11.1 Ausgabe

In der Stream-Bibliothek wird der Ausgabeoperator `<<` und der Eingabeoperator `>>` für die elementaren Typen definiert:

```
double  x = 5.6;
int   a = 3;
cout << "a=";
cout << a;
cout << '\n';
cout << x;
```

Diese binären Operatoren sind so definiert, daß sie als erstes Argument ein Streamobjekt nehmen und eine Referenz auf das erste Argument zurückgeben. Damit ist es möglich zu konkatenieren:

```
cout << "a=" << a << '\n';
```

Dieses Statement wird folgendermaßen abgearbeitet:

```
((cout << "a=") << a) << '\n';
```

An den Ausgabe-Stream wird durch dieses Statement schließlich die Zeichenfolge `a=3'\n'` angefügt.

Die Klasse `ostream` für den Ausgabe-Stream ist etwa folgendermaßen definiert:

```
class ostream : public virtual ios
   {
   // ---
   public:
      ostream& operator << (char);
      ostream& operator << (char*);
      ostream& operator << (int);
      ostream& operator << (double);
   // ---
   };
```

Für vom Benutzer definierte Typen können diese Operatoren überladen werden, was wir am Beispiel der `Zeichenkette` zeigen:

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

ostream& operator << (ostream& s, const Zeichenkette& a)
   {
   for (int i = 0; i < a.anzZeich; i++) s << a.z[i];
   return s;
   }
```

Dieser Operator muß als **friend** in der Klasse `Zeichenkette` definiert werden, da er auf private Elemente von `Zeichenkette` zugreift. Der Operator kann folgendermaßen angewendet werden:

```
Zeichenkette z ("a=");
Zeichenkette ez ('\n');
int   a = 3;
cout << z << a << ez;
```

11.2 Eingabe

Für die Eingabe gibt es die Klasse `istream`, die für die elementaren Typen den Operator `>>` zur Verfügung stellt.

```
class istream : public virtual ios
   {
   // ---
   public:
      istream& operator >> (char&);
      istream& operator >> (char*);
      istream& operator >> (int&);
      istream& operator >> (double&);
   // ---
   };
```

Der Eingabeoperator überliest *Whitespace* (Blank, Carriage Return, Linefeed, Formfeed, Newline, Tab) und versucht dann möglichst viele der folgenden Zeichen in ein Objekt vom Typ des zweiten Arguments zu konvertieren. Strings werden bis zum nächsten Whitespace eingelesen. Wenn dies fehlschlägt, weil etwa zu früh `EOF` (End-of-File) erreicht wird oder eine Konversion nicht möglich ist, wird der Bedingungsausdruck `(cin >> a)` Null. Die Statements

```
char a;
while (cin >> a);
```

lesen von Standardinput bis `EOF`.

Natürlich kann auch der Input-Operator für eigene Typen überladen werden: (Zeichenketten sollen bis zum nächsten `'\t'` (Tab) oder `'\n'` (Newline) eingelesen werden.)

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

void Zeichenkette::addBuf (int groesse, char* buf)
   {
   char* p;
   p = new char[anzZeich + groesse];
   for (int i = 0; i < anzZeich; i++) p[i] = z[i];
   for (int i = 0; i < groesse; i++) p[anzZeich + i] = buf[i];
   delete [] z;
   z = p;
   anzZeich = anzZeich + groesse;
```

```
   }

istream& operator >> (istream& s, Zeichenkette& a)
   {
   char c;
   int  i = 0;
   a.anzZeich = 0;
   char buf[256];
   while (s.get (c) && c != '\t' && c != '\n')
      {
      buf[i] = c;
      i++;
      if (i == 256)
         {
         a.addBuf(256, buf);
         i = 0;
         }
      }
   a.addBuf (i, buf);
   return s;
   }
```

Da wir auch Blanks lesen wollten, mußten wir hier die `istream`-Elementfunktion `get` verwenden, die im nächsten Abschnitt besprochen wird.

11.3 Elementfunktionen von `iostream`

Mit den Element-Funktionen `get ()` und `put ()` der iostream-Klasse können alle Zeichen, auch Whitespace einzeln ein- und ausgegeben werden. Die Zeichenausgabe-Funktion ist definiert als `put (char c)`. Für die Eingabe-Funktion gibt es zwei Formen:

1. `istream& get (char& c)` : `c` ist das eingelesene Zeichen und der Funktions-Rückgabewert wird in einem Bedingungsausdruck Null, wenn kein Zeichen mehr eingelesen werden kann, d.h. wenn End-of-File `EOF` angetroffen wurde.

2. **int** `get ()` : Hier wird das eingelesene Zeichen einschließlich `EOF` als **int** zurückgegeben. Der Wert von `EOF` ist meistens `-1`.

Die beiden folgenden Beispiele zeigen, wie von Standardinput jedes Zeichen gelesen wird, bis das Dateiende erreicht ist, und wie das Gelesene auf Standardoutput ausgegeben wird:

```
#include <iostream>

using namespace std;

main ()
   {
   char c;
   while (cin.get (c)) cout.put (c);
   return 0;
   }
```

oder

```
#include <iostream>

using namespace std;

main ()
   {
   int c;
   while ( (c=cin.get () != EOF) cout.put (c);
   return 0;
   }
```

Bemerkung: **while** `(cin.get (c))` ist möglich, obwohl es keinen Cast von `istream&` nach **int** gibt. Denn **while** `(a)` ist äquivalent zu **while** `(a != 0)`. Der Operator "`!=`" ist aber in der `istream`-Klasse so definiert, daß er auf der einen Seite ein **int** auf der anderen Seite ein `istream&` nehmen kann und ein **int** zurückgibt.

In C++ wird die Ausgabe mit `cout` gepuffert. Daher funktioniert die Fehlersuche mit `cout` nur, wenn man die Ausgabe direkt nach Ausführung des Ausgabe-Statments erzwingt, indem man sie *flushed*:

```
cout << "Guten Tag Welt!\n" << flush;
```

oder

```
cout << "Guten Tag Welt!\n";
cout.flush ();
```

Auch ein `cin` bewirkt ein Flushen des Ausgabepuffers. `printf` wird ungepuffert ausgegeben. Bei einer gemischten Verwendung von `cout` und `printf` erscheinen unabhängig von der Reihenfolge wahrscheinlich erst die `printf`-Ausgaben, dann die `cout`-Ausgaben, falls durch Flushen nicht etwas anderes erzwungen wird.

Mit der Elementfunktion

```
istream& getline (char* Puffer, int MaxZeichen, char Begrenzer = '\n');
```

kann man höchstens `MaxZeichen - 1` Zeichen oder höchstens bis zum Zeichen `Begrenzer` lesen. Die Zeichen werden in den `Puffer` mit einem `'\0'` am Ende geschrieben. Der `Begrenzer` wird nicht in den `Puffer` geschrieben, aber aus

dem Stream entfernt, sodaß das nächste lesbare Zeichen hinter dem `Begrenzer` im Stream steht.

Mit der Funktion

```
putback (char c)
```

kann das zuletzt gelesene Zeichen in den Eingabe-Strom zurückgestellt werden. Das Verhalten ist nur dann definiert, wenn nur genau ein Zeichen und zwar das gelesene Zeichen zurückgestellt wird. Eine Modifikation des Eingabe-Stroms ist mit `putback` nicht möglich.

Die Funkion

```
int   peek ()
```

gibt das nächste Zeichen (oder `EOF`) zurück ohne es aus dem Eingabe-Strom zu extrahieren.

11.4 Streamzustände

Auf die Streamzustände kann über Methoden der Klasse `ios`, die eine Basisklasse von `ostream` und `istream` ist, zugegriffen werden.

```
class ios
   {
   public:
      int   eof () const; // end of file  angetroffen
      int   fail () const; // nächste Operation geht schief,
                           // soweit war aber  alles  in  Ordnung
      int   bad () const; // Stream defekt
      int   good () const; // nächste Operation wird wohl funktionieren
   // ---
   };
```

Falls der Kommentar hinter der Statusfunktion *nicht* zutrifft, dann gibt die Funktion `0` zurück, sonst etwas ungleich `0` .

Ein Stream kann mit

```
cin.clear (ios::goodbit);
```

wieder repariert werden. Wenn z.B. bei der Eingabe anstatt eines erwarteten **int** ein **char** gegeben wurde, dann gerät der Eingabestrom in einen fehlerhaften Zustand. Jeder weitere Versuch, von diesem Eingabestrom zu lesen, schlägt fehl, wenn der Eingabestrom nicht in einen fehlerfreien Zustand gebracht wird. Solch ein Fehler kann z.B. folgendermaßen abgefangen werden:

```
int   i;
char  c;
// ...
while (!(cin >> i))
   {
```

```
cin.clear (ios::goodbit);
cin.get (c); // entferne fälschlichen char
}
```

11.5 File-I/O

Um von Files zu lesen, muß zusätzlich `<fstream>` inputiert werden. Dort werden die Klassen `fstream`, `ifstream` und `ofstream` definiert und zur Verfügung gestellt.

#include `<fstream>`

using namespace `std;`

```
fstream f;   // Definition eines Ein– und Ausgabestroms
ifstream fi; // Definition eines Eingabestroms
ofstream fo; // Definition eines Ausgabestroms
ofstream fe; // Definition eines Ausgabestroms
```

Diese File-Streams können auf Standardein- und ausgabe umgelenkt werden.

```
fi.attach (0); // Umlenkung auf stdin
fo.attach (1); // Umlenkung auf stdout
fe.attach (2); // Umlenkung auf stderr
```

Die Verknüpfung mit einem File (öffnen eines Files) erfolgt durch die Methode `open`. Dabei wird ein Filename (Pfadname) als C-String und ein Modus angegeben:

`in` lesen

`out` schreiben

`app` appendieren

```
fi.open (Inputfilename, ios::in);
fo.open (Outputfilename, ios::out);
f.open (filename, ios::in | ios::out);
fe.open (errorfile, ios::app);
```

if `(!fi) { --- }` *// open fehlgeschlagen*

Ein nicht mehr benötigter File sollte geschlossen werden.

```
f.close ();   // File schließen
```

Anlegen und Öffnen eines Filestreams ist auch in einem Schritt möglich, da es einen Konstruktor mit den Argumenten der Methode `open` gibt:

```
fstream f2 (filename2, ios::in | ios::out);
```

Die Übergabe eines Files als Funktionsargument ist nur als Referenz möglich, weil der Copy-Konstruktor (und auch der Zuweisungsoperator) verboten wurde.

```
int funk (ostream& fo, ---);
int funk (istream& fi, ---);
```

11.5.1 Manipulation der Position im File – Random Access

Mit der Methode

`seekp (streampos p, seek_dir d = ios::beg)` (oder `seekg`)

können beliebige Lese- oder Schreib-Positionen in einem File gesetzt werden. Der erste Parameter vom Typ `streampos`, das als **long** definiert ist, gibt den Byte-Offset vom zweiten Parameter an, der die Werte

```
ios::beg // Anfangsposition (Default)
ios::cur // aktuelle Position
ios::end // End-of-File-Position
```

annehmen kann. Der erste Parameter kann auch negativ sein.

Der File muß entweder zum Lesen oder zum Lesen und Schreiben geöffnet werden.

```
fstream f ("filename", ios::in);
fstream f ("filename", ios::in | ios::out);
```

Die Methode `tellp ()` (oder `tellg`) gibt die aktuelle Position zurück als Byte-Offset zum Fileanfang.

Das folgende Beispiel-Programm vertauscht in einem File das 13. Byte mit dem 17. Byte.

```
#include <fstream>

using namespace std;

main ()
  {
  fstream f ("dummy", ios::in | ios::out);
  char hilf1, hilf2;
  f.seekp (12, ios::beg); // nun cur 13. Byte
  f >> hilf1;             // nun cur 14. Byte
  f.seekp (3, ios::cur); // nun cur 17. Byte
  f >> hilf2;             // nun cur 18. Byte
```

```
f.seekp (-1, ios::cur); // nun cur 17. Byte
f << hilf1;           // nun cur 18. Byte
f.seekp (12, ios::beg); // nun cur 13. Byte
f << hilf2;           // nun cur 14. Byte
f.close ();
}
```

11.6 Formatierung

Mit der `ostream`-Elementfunktion `width (int)` wird die minimale Feldbreite für die (und nur die) nächste Outputoperation eines Strings oder einer numerischen Größe vorgegeben.

```
cout.width (5);
cout << '=' << 123 << ',';
```

ergibt

```
=  123,
```

Wenn mehr Zeichen als durch `width` angegeben zu drucken sind, werden diese trotzdem ausgegeben, um heimliche Fehler zu vermeiden. Ein unschöner Ausdruck ist besser als ein falscher Ausdruck. Man beachte, daß das `width` nicht auf die Ausgabe von **char** wirkt, sondern nur auf Strings und numerische Größen.

Mit der Funktion `fill (char)` kann ein Füllzeichen spezifiziert werden.

```
cout.width (5);
cout.fill ('.');
cout << '=' << 123 << ',';
```

ergibt

```
=..123,
```

Mit `setf` können verschiedene Formatierungen gesetzt werden. Z.B. zur Darstellung von Integern:

```
cout.setf (ios::dec, ios::basefield);
cout.setf (ios::oct, ios::basefield);
cout.setf (ios::hex, ios::basefield);

cout.setf (ios::showbase);
```

bewirkt, daß die Basis angezeigt wird (Octal durch eine führende `0`, Hexadezimal durch ein führendes `0x`).

Die Bündigkeit innerhalb eines Felds wird durch

```
cout.setf (ios::left, ios::adjustfield);
cout.setf (ios::right, ios::adjustfield);
cout.setf (ios::internal, ios::adjustfield);
```

gesetzt, wobei bei `internal` die Füllzeichen zwischen Vorzeichen und den Wert gehen.

Gleitkommazahlen-Ausgabe wird gesteuert mit

```
cout.setf (ios::scientific, ios::floatfield);
cout.setf (ios::fixed, ios::floatfield);
cout.setf (0, ios::floatfield); // Default
```

und

```
cout.precision (8);
```

sorgt dafür, daß Gleitkommazahlen mit acht Stellen ausgegeben werden. Default ist sechs. Bei `fixed` und `scientific` sind damit die Stellen hinter dem Dezimalpunkt gemeint.

```
cout.setf (ios::showpoint);
```

sorgt dafür, daß der Dezimalpunkt und mindestens eine Stelle hinter dem Dezimalpunkt gedruckt wird.

11.6.1 Verwendung von Manipulatoren

Um Manipulatoren verwenden zu können, muß `<iomanip>` inputiert werden. Manipulatoren werden direkt in den Stream geschrieben:

```
cout << hex << 12345 << 12345 << oct << 12345;
```

Die wichtigsten Manipulatoren sind:

`oct` : oktale Ausgabe

`hex` : hexadezimale Ausgabe

`dec` : dezimale Ausgabe

`flush` : flushed die Ausgabe

`endl` : addiert ein `'\n'` und ein `flush`

`setw (int w)` : Breite `w` des Ausgabefeldes wird gesetzt (Gilt nur für die nächste numerische oder String-Ausgabe. Andere Manipulatoren gelten bis sie überschrieben werden.)

`setfill (char f)` : Füllzeichen `f` wird gesetzt

`setprecision (int p)` : Anzahl der Dezimalstellen

`ws` : entfernt Whitespace aus Eingabe

11.7 String-Streams

Die Stream-Bibliothek unterstützt auch I/O-Operationen auf Zeichenfeldern (Strings). Dazu muß `<strstream>` inputiert werden. Dort werden die Klassen `ostrstream` und `istrstream` zur Verfügung gestellt.

Nach der Definition

```
ostrstream buf;
```

kann `buf` jetzt wie ein Stream verwendet werden:

```
buf << 1234 << '\n';
```

Mit der Methode `str ()` kann auf das Zeichenfeld eines `ostrstream` zugegriffen werden. Sie gibt einen Zeiger darauf zurück.

```
char* p = buf.str ();
```

Ab jetzt kann aber nichts mehr in den Stream `buf` geschrieben werden. `str ()` friert `buf` ein. Ein `'\0'` wird *nicht* angehängt. Wenn `str ()` aufgerufen wurde, muß der Benutzer dafür sorgen, daß der Speicher wieder freigegeben wird durch ein **delete** `p` , wenn der Inhalt von `p` nicht mehr gebraucht wird.

Die Methode `pcount ()` liefert als Resultat die Anzahl der Character im `ostrstream`.

Bemerkung: `buf` kann auch direkt mit einem C-String initialisiert werden:

```
char string[80];
ostrstream buf (string, 80);
```

`istrstream` wird u.a. verwendet um einen String in einen Zahlenwert zu verwandeln:

```
int   Wert;
char* s = "123456";
int   anzZeich = 6;
istrstream a (s, anzZeich);
a >> Wert;
```

Folgende Zeile ist ein Beispiel für eine anonyme Verwendung von `istrstream`.

```
istrstream (s, anzZeich) >> Wert;
```

Achtung: Die Stream-Bibliothek `<strstream>` ist veraltet und wird möglicherweiße in zukünftigen Versionen von C++ nicht mehr zur Verfügung stehen. Es wir empfohlen anstatt dessen die Stream-Bibliothek `<sstream>` zu verwenden. Grundsätzlich bietet die `<sstream>`-Bibliothek ähnliche Funktionalitäten wie die `<strstream>`-Bibliothek, allerdings existiert unter anderem die Methode `pcount ()` nicht.

11.8 Übungen

Übung zur Klausurnote:

1. Verwende String-Streams für die Zahl-String Konversionen in der Klasse Klausurnote.
2. Überlade den Operator `<<` so, daß eine Note in der Form
   `befriedigend (3,2)`
   ausgegeben wird.
3. Überlade den Operator `>>` so, daß eine Note in der Form
   `3,2`
   eingelesen wird. Kann man gleichzeitig auch in der Form
   `3.2`
   einlesen?

Übung Filekomprimieren:

Der File `STROM` enthält neben allen möglichen anderen Zeichen vor allem Nullen. Dieser File soll nach folgendem Algorithmus komprimiert werden: Jede Folge von bis zu zehn Nullen soll ersetzt werden durch `0x`, wobei `x` die Anzahl der Nullen Modulo 10 ist. Bei 10 Nullen ist `x` gleich `0`. Also `0000` wird ersetzt durch `04` und `0000000000` durch `00` und `0` durch `01` und `00000000000000` durch `0004`.

Schreibe ein Programm, daß von Standardinput liest und nach Standardoutput schreibt.

Schreibe ein Programm, daß von File `STROM` liest und nach File `dummy` schreibt.

Übung zum Stack:

1. Überladen Sie die Operatoren `<<` und `>>` der Klasse `Stack`, sodaß ganze Stacks ausgegeben oder eingelesen werden können. (Beim Einlesen Abbruch mit `<Strg> D` (`EOF`))
2. Wenden Sie diese Operatoren an, um einen Stack auf einen String zu schreiben bzw von einem String zu lesen.

Kapitel 12

Referenzzählung – *Reference Counting*

Als C++ Programmierer hat man – durch Konstruktoren und Destruktoren – die volle Kontrolle über die Art und Weise, wie ein Objekt erzeugt oder zerstört wird.

Diese Möglichkeiten können genutzt werden

- die Leistung zu steigern,
- den Speicherplatz besser auszunutzen
- und die Allokation von benötigtem Speicher zu garantieren ***(smart pointers)***,

indem vermieden wird, Speicher mit identischem Inhalt zu kopieren.

Der i.a. benutzte Mechanismus heißt ***representation sharing*** oder ***reference counting***. D.h. anstatt Speicher zu kopieren, wird mitgezählt, wie oft der Speicher benötigt wird. Wenn der Speicher nicht mehr benötigt wird, wird er aufgegeben. Das lohnt sich besonders für Klassen, die dynamisch Speicher allokieren, z.B. eine Klasse die Strings oder Zeichenketten beliebiger Länge verwaltet.

Wir werden vier Möglichkeiten des *reference counting* vorstellen, die unterschiedlich kompliziert zu implementieren bzw zu verallgemeinern sind.

Als Beispiel betrachten wir eine Stringklasse `Zhket` (Zeichenkette), die im ersten Abschnitt vorgestellt wird.

12.1 Beispiel Klasse Zhket

```
class Zhket
   {
   friend Zhket  operator + (const Zhket& a, const Zhket& b);

   public:
      Zhket ();
      Zhket (const char* C_String);
      Zhket (const Zhket& zk);
      Zhket&  operator = (const Zhket& zk);
      ~Zhket ();
      void zeige () const;

   private:
      int   anz;
      char* z;
   };


Zhket::Zhket () : anz (0), z (new char [1]) {}

Zhket::Zhket (const char* C_String)
   :  anz (0)
   {
   while (C_String[anz++] != '\0');
   if (anz > 0)
      {
      z = new char[anz];
      for (int i (0); i < anz; i++)
         z[i] = C_String[i];
      }
   else z = new char[1];
   }


Zhket::Zhket (const Zhket& zk)
   :  anz (zk.anz)
   {
   if (anz > 0)
      {
      z = new char[anz];
      for (int i (0); i < anz; i++)
         z[i] = zk.z[i];
      }
   else z = new char[1];
   }
```

```
Zhket&  Zhket::operator = (const Zhket& zk)
   {
   if (this != &zk)
      {
      anz = zk.anz;
      if (anz > 0)
         {
         delete z; z = new char[anz];
         for (int i (0); i < anz; i++) z[i] = zk.z[i];
         }
      }
   return *this;
   }

Zhket::~Zhket () { delete z; }
```

```
void Zhket::zeige () const
   {
   cout << "Zeichenkette: " << '"';
   if (anz > 0)
   for (int i (0); i < anz; i++) cout << z[i];
   cout << '"' << " hat ";
   cout << anz << " Zeichen.\n";
   }
```

```
Zhket operator + (const Zhket& a, const Zhket& b)
      {
      Zhket  c;
      c.anz = a.anz + b.anz;
      if (c.anz > 0)
         {
         delete [] c.z; c.z = new char[c.anz];
         for (int i (0); i < a.anz; i++) c.z[i] = a.z[i];
         for (int i (0); i < b.anz; i++) c.z[i + a.anz] = b.z[i];
         }
      return c;
      }
```

```
main ()
   {
   Zhket a;
   Zhket b ("Hello World");
   Zhket c (b);
   a.zeige ();
   b.zeige ();
   c.zeige ();
   return 0;
   }
```

Wir bemerken, daß in dieser "normalen" Version beim Kopieren einer Zeichenkette neuer Speicher angelegt wird. Bei der Zuweisung wird alter Speicher erst freigegeben, dann neuer wieder angelegt.

12.2 *Handle Class* Idiom

Beim *Handle Class* Idiom erfolgt der Zugriff auf die Zeichenketten-Klasse (Repräsentation der Zeichenkette) über ein *handle*. Damit das Anwendungsprogramm nur wenig geändert werden muß, bekommt das *handle* den Namen der ursprünglichen Stringklasse `Zhket`. Das ergibt insgesamt folgende Vorgehensweise:

1. `Zhket` wird in `ZhketRep` umbenannt. Alles in `ZhketRep` wird **private**, damit man von außen nicht mehr zugreifen kann.

2. In `ZhketRep` wird ein Zähler `count` geführt, der die Anzahl der Referenzen auf ein Objekt von `ZhketRep` verwaltet. An der Implementation der Methoden von `ZhketRep` ändert sich nichts.

3. Eine neue Klasse `Zhket` wird definiert, die als Datenelement nur einen Zeiger auf `ZhketRep` führt.

4. Alle Methoden von `ZhketRep` müssen in `Zhket` so implementiert werden, daß sie im wesentlichen an `ZhketRep` delegieren. Die Konstruktoren und der Destruktor verwalten zusätzlich den Referenzzähler. Damit `Zhket` auf die Elemente von `ZhketRep` zugreifen kann, muß `Zhket` Freund von `ZhketRep` werden.

Im folgenden ist das Ergebnis gelistet:

```
class Zhket;

class ZhketRep
   {
   friend class Zhket;
   friend ZhketRep  operator + (const ZhketRep& a, const ZhketRep& b);
   friend Zhket  operator + (const Zhket& a, const Zhket& b);
   private:
      ZhketRep ();
      ZhketRep (const char* C_String);
      ZhketRep (const ZhketRep& zk);
      ZhketRep&  operator = (const ZhketRep& zk);
      ~ZhketRep ();
      void zeige () const;
      int   anz;
      char* z;
      int   count;
   };



ZhketRep::ZhketRep () : anz (0), z (new char [1]) {}

ZhketRep::ZhketRep (const char* C_String)
   :  anz (0)
   {
   while (C_String[anz++] != '\0');
   if (anz > 0)
      {
      z = new char[anz];
      for (int i (0); i < anz; i++)
         z[i] = C_String[i];
      }
   else z = new char[1];
   }



ZhketRep::ZhketRep (const ZhketRep& zk)
   :  anz (zk.anz)
   {
   if (anz > 0)
      {
      z = new char[anz];
      for (int i (0); i < anz; i++)
         z[i] = zk.z[i];
      }
   else z = new char[1];
   }
```

```
ZhketRep&  ZhketRep::operator = (const ZhketRep& zk)
   {
   if (this != &zk)
      {
      anz = zk.anz;
      if (anz > 0)
         {
         delete z; z = new char[anz];
         for (int i (0); i < anz; i++) z[i] = zk.z[i];
         }
      }
   return *this;
   }

ZhketRep::~ZhketRep () { delete z; }
```

```
void ZhketRep::zeige () const
   {
   cout << "Zeichenkette: " << '"';
   if (anz > 0)
   for (int i (0); i < anz; i++) cout << z[i];
   cout << '"' << " hat " << anz;
   cout << " Zeichen (count = " << count << ").\n";
   }
```

```
ZhketRep operator + (const ZhketRep& a, const ZhketRep& b)
      {
      ZhketRep  c;
      c.anz = a.anz + b.anz;
      if (c.anz > 0)
         {
         delete [] c.z; c.z = new char[c.anz];
         for (int i (0); i < a.anz; i++) c.z[i] = a.z[i];
         for (int i (0); i < b.anz; i++) c.z[i + a.anz] = b.z[i];
         }
      return c;
      }
```

```
class Zhket
   {
   friend Zhket  operator + (const Zhket& a, const Zhket& b);

   public:
      Zhket ();
      Zhket (const char* C_String);
      Zhket (const Zhket& zk);
      Zhket&  operator = (const Zhket& zk);
      ~Zhket ();
      void zeige () const;

   private:
      ZhketRep*  rep;
   };



Zhket::Zhket ()
   : rep (new ZhketRep)
   {
   rep->count = 1;
   }

Zhket::Zhket (const char* C_String)
   :  rep (new ZhketRep (C_String))
   {
   rep->count = 1;
   }



Zhket::Zhket (const Zhket& zk)
   :  rep (zk.rep)
   {
   rep->count++;
   }



Zhket&  Zhket::operator = (const Zhket& zk)
   {
   if (this != &zk)
      {
      if (--rep->count == 0) delete rep;
      rep = zk.rep;
      rep->count++;
      }
   return *this;
   }
```

```
Zhket::~Zhket ()
   {
   if (--rep->count == 0) delete rep;
   }


void Zhket::zeige () const
   {
   rep->zeige ();
   }


Zhket operator + (const Zhket& a, const Zhket& b)
     {
     Zhket c;
     *c.rep = *a.rep + *b.rep;
     return c;
     }


main ()
   {
   Zhket a;
   Zhket b ("Hello World");
   Zhket c (b);
   a.zeige ();
   b.zeige ();
   c.zeige ();
   return 0;
   }
```

Die Handle-Klasse kümmert sich um das Management. Die Repräsentationsklasse um die Details der Zeichenkettenmanipulation.

Beim Kopieren und Zuweisen einer Zeichenkette wird nur das Handle kopiert, nicht die Repräsentation der Zeichenkette. D.h. beim Kopiern und Zuweisen entsteht keine neue Zeichenkette. Wenn eine Zeichenkette verändert wird, muß dafür gesorgt werden, daß eine "tiefe" Kopie angelegt wird. Das sei beispielsweise an dem Operator `[]` gezeigt, der den elementweisen Zugriff auf die Zeichenkette erlaubt.

```
char& ZhketRep::operator [] (int i)
   {
   if (i >= anz) i = anz - 1;
   if (i < 0) i = 0;
   return z[i];
   }
```

```
char& Zhket::operator [] (int i)
   {
   if (rep->count > 1)
      {
      Zhket neu;
      *neu.rep = *rep;
      rep->count--;
      rep = neu.rep;
      rep->count++;
      }
   return (*rep)[i];
   }
```

12.3 *Counted Pointers* Idiom

Ein Nachteil des oben angegebenen Idioms ist, daß die Signaturen aller Konstruktoren und Methoden in beiden Klassen (Repräsentation und Handle) dupliziert werden müssen.

Wenn wird die Überladung des Operators `->` ausnützen, dann müssen nur noch einige Konstruktoren, der Destruktor und überladenene Operatoren dupliziert werden.

Die Handle-Klasse wird dann zu einer Klasse von Objekten, die wie Zeiger ***(counted pointers)*** behandelt werden. Diese Zeiger sind insofern bequem, als automatisch dafür Speicher angelegt und wieder aufgegeben wird.

Unser Beispiel wird mit diesem Idiom zu:

An der Repräsentationsklasse `ZhketRep` ändert sich überhaupt nichts.

In der Klasse `Zhket` gibt es folgende Änderungen: Die Methode `zeige ()` muß nicht mehr definiert werden. Dafür wird der Operator `->` überladen:

```
class Zhket
   {
   friend Zhket  operator + (const Zhket& a, const Zhket& b);
   public:
      Zhket ();
      Zhket (const char* C_String);
      Zhket (const Zhket& zk);
      Zhket&  operator = (const Zhket& zk);
      ~Zhket ();
      char& operator [] (int i);
      char operator [] (int i) const;
//    ZhketRep*   operator -> ();
      const ZhketRep* operator -> () const;
   private:
      ZhketRep*  rep;
   };
```

```
char& Zhket::operator [] (int i)
   {
   if (rep->count > 1)
      {
      Zhket neu;
      *neu.rep = *rep;
      rep->count--;
      rep = neu.rep;
      rep->count++;
      }
   return (*rep)[i];
   }
```

```
char Zhket::operator [] (int i) const
   {
   return (*rep)[i];
   }
```

```
#ifdef false
ZhketRep*  Zhket::operator -> ()
   {
   if (rep->count > 1)
      {
      Zhket neu;
      *neu.rep = *rep;
      rep->count--;
      rep = neu.rep;
      rep->count++;
      }
   return rep;
   }
#endif
```

```
const ZhketRep* Zhket::operator -> () const
   {
   return rep;
   }
```

```
main ()
   {
   const Zhket b ("Hello World");
   Zhket c (b);
   cout << "const Zhket b (\"Hello World\");" << endl;
   cout << "Zhket c (b);" << endl;
   cout << "b    : "; b->zeige (); cout << "c  : "; c->zeige ();
   cout << "b[1] : " << b[1] << endl;
   cout << "b    : "; b->zeige (); cout << "c  : "; c->zeige ();
   cout << "c[1] : " << c[1] << endl;
   cout << "b    : "; b->zeige (); cout << "c  : "; c->zeige ();
   c[1] = 'a'; cout << "c[1] = 'a';" << endl;
   cout << "b    : "; b->zeige (); cout << "c  : "; c->zeige ();
   Zhket d;
   d = b + c; cout << "b + c : "; d->zeige ();
   return 0;
   }
```

Bei dem Operator `->` muß man sich allerdings entscheiden, ob man ihn für alle Methoden der Repräsentationsklasse oder nur für die **const** Methoden zuläßt. Die erste (mit **#ifdef false #endif** ausgeklammerte) Alternative ist nicht zu empfehlen, da dabei (durch i.a. zu häufiges Kopieren der Repräsentation) die Zeigernatur verloren geht. Durch Zulassung des Operators `->` nur für **const** Methoden wird man allerdings gezwungen, die nicht-**const** Methoden in der Zeigerklasse ebenfalls zu implementieren.

Diese Überlegungen werden hinfällig, wenn man die Zeigerklasse wie normale Zeiger verwenden will, d.h. wenn die Repräsentation bei einem Methodenzugriff nie kopiert werden soll. Dann sollte der Operator `->` in der Zeigerklasse als

```
ZhketRep*  Zhket::operator -> ()const;
```

deklariert werden.

12.4 Referenzzählung für Klassen

Manchmal ist es nicht möglich, eine Klasse zu verändern. Wenn man hier Referenzzählung implementieren möchte, dann muß man eine zusätzliche Repräsentationsklasse einführen, die den Zähler und einen Zeiger auf die ursprüngliche Klasse verwaltet. Diese zusätzliche Klasse wird innerhalb der Klasse Zeigerklasse definiert, da sie nur für die Zeigerklasse interessant ist.

In unserem Beispiel sei die ursprüngliche Klasse `Zhket`, die wir nicht verändern dürfen, aber deren Referenzen wir zählen wollen.

`ZhketPtr` sei die Zeigerklasse und `ZhketRep`, die zusätzliche Repräsentationsklasse.

Die Implementation dieser Klassen ist wie folgt:

```
class Zhket
   {
   friend Zhket  operator + (const Zhket& a, const Zhket& b);
   public:
      Zhket ();
      Zhket (const char* C_String);
      Zhket (const Zhket& zk);
      Zhket&  operator = (const Zhket& zk);
      ~Zhket ();
      char& operator [] (int i);
      char  operator [] (int i) const;
      void  zeige () const;
   private:
      int   anz;
      char* z;
      int   count;
   };
```

```
class ZhketPtr
   {
   friend ZhketPtr  operator + (const ZhketPtr& a, const ZhketPtr& b);
   public:
      ZhketPtr ();
      ZhketPtr (const char* C_String);
      ZhketPtr (const ZhketPtr& zk);
      ZhketPtr&  operator = (const ZhketPtr& zk);
      ~ZhketPtr ();
      char& operator [] (int i);
      char  operator [] (int i) const;
      Zhket*  operator -> () const;
      void  z () const { cout << "(count = " << rep2->count << ") "; }

   private:
      class ZhketRep
         {
         public:
            ZhketRep (Zhket* rep)
               :  rep (rep)
               {}
            int   count;
            Zhket*  rep;
         };
      ZhketRep*  rep2;
   };
```

```
ZhketPtr::ZhketPtr ()
   : rep2 (new ZhketRep (new Zhket))
   {
   rep2->count = 1;
   }

ZhketPtr::ZhketPtr (const char* C_String)
   :  rep2 (new ZhketRep (new Zhket (C_String)))
   {
   rep2->count = 1;
   }



ZhketPtr::ZhketPtr (const ZhketPtr& zk)
   :  rep2 (zk.rep2)
   {
   rep2->count++;
   }



ZhketPtr&  ZhketPtr::operator = (const ZhketPtr& zk)
   {
   if (this != &zk)
      {
      if (--rep2->count == 0)
         {
         delete rep2->rep;
         delete rep2;
         }
      rep2 = zk.rep2;
      rep2->count++;
      }
   return *this;
   }



ZhketPtr::~ZhketPtr ()
   {
   if (--rep2->count == 0)
      {
      delete rep2->rep;
      delete rep2;
      }
   }
```

```
ZhketPtr operator + (const ZhketPtr& a, const ZhketPtr& b)
     {
     ZhketPtr c;
     *c.rep2->rep = *a.rep2->rep + *b.rep2->rep;
     return c;
     }


char& ZhketPtr::operator [] (int i)
   {
   if (rep2->count > 1)
     {
     ZhketPtr neu;
     *neu.rep2->rep = *rep2->rep;
     rep2->count--;
     rep2 = neu.rep2;
     rep2->count++;
     }
   return (*rep2->rep)[i];
   }


char ZhketPtr::operator [] (int i) const
   {
   return (*rep2->rep)[i];
   }


#ifdef false
Zhket*  ZhketPtr::operator -> ()
   {
   if (rep->count > 1)
     {
     ZhketPtr neu;
     *neu.rep2->rep = *rep2->rep;
     rep2->count--;
     rep2 = neu.rep2;
     rep2->count++;
     }
   return rep2->rep;
   }
#endif


Zhket*  ZhketPtr::operator -> () const
   {
   return rep2->rep;
   }
```

```
main ()
   {
   const ZhketPtr b ("Hello World");
   ZhketPtr c (b);
   cout << "const ZhketPtr b (\"Hello World\");" << endl;
   cout << "ZhketPtr c (b);" << endl;
   cout << "b: "; b.z (); b->zeige (); cout << "c: "; c.z (); c->zeige ();
   cout << "b[1]: " << b[1] << endl;
   cout << "b: "; b.z (); b->zeige (); cout << "c: "; c.z (); c->zeige ();
   cout << "c[1]: " << c[1] << endl;
   cout << "b: "; b.z (); b->zeige (); cout << "c: "; c.z (); c->zeige ();
   c[1] = 'a'; cout << "c[1] = 'a';" << endl;
   cout << "b: "; b.z (); b->zeige (); cout << "c: "; c.z (); c->zeige ();
   ZhketPtr d;
   d = b + c; cout << "b + c : "; d.z (); d->zeige ();
   return 0;
   }
```

Die Zeigerklasse kann individuell an die Erfordernisse der referenzierten Klasse angepaßt werden. Allerdings muß für jede Klasse relativ viel Code geschrieben werden. Das folgende Template ist weniger mächtig, dafür aber ohne zusätzlichen Code anwendbar.

12.4.1 Template eines Referenzzählers

Da die zu referenzierende Klasse, im folgenden `Rep` nicht verändert wird und beinahe nur ein Parameter in der Zeigerklasse, im folgenden `CountPtr` ist, bietet sich ein Template an, wobei die zu referenzierende Klasse der variable Typ ist.

Da die möglicherweise vielen unterschiedlichen Konstruktoren von `Rep` nicht allgemein darstellbar sind, muß bei der Initialisierung von `CountPtr` ein Zeiger auf `Rep` übergeben werden. Wenn `Rep` nicht mehr referenziert wird, dann wird ein **delete** auf diesen Zeiger ausgeführt. Daher müssen die Objekte von `Rep` mit **new** initialisiert werden, sicherheitshalber beim Konstruktoraufruf für ein `CountPtr`-Objekt. In der folgenden Implementation wird eine eventuelle Nicht-Initialisierung nur durch eine Fehlermeldung abgefangen.

Der Code für `CountPtr` lautet folgendermaßen:

```
template <class Rep>
class CountPtr
   {
   public:
      CountPtr (Rep* rep);
      CountPtr (const CountPtr<Rep>& zk);
      CountPtr&  operator = (const CountPtr<Rep>& zk);
      ~CountPtr ();
      Rep* operator -> () const;
      Rep& operator * () const;

      void z () const { cout << "(count = " << rep2->count << ") "; }
         // nur zur Diagnose

   private:
      class CountPtrRep
         {
         public:
            CountPtrRep (Rep* rep)
               : rep (rep)
               {}
            int   count;
            Rep* rep;
         };
      CountPtrRep* rep2;
   };
```

```
template <class Rep>
CountPtr<Rep>::CountPtr (Rep* rep)
   :  rep2 (new CountPtrRep (rep))
   {
   rep2->count = 1;
   }
```

```
template <class Rep>
CountPtr<Rep>::CountPtr (const CountPtr<Rep>& zk)
   :  rep2 (zk.rep2)
   {
   rep2->count++;
   }
```

```
template <class Rep>
CountPtr<Rep>& CountPtr<Rep>::operator = (const CountPtr<Rep>& zk)
   {
   if (this != &zk)
      {
      if (--rep2->count == 0)
         {
         delete rep2->rep;
         delete rep2;
         }
      rep2 = zk.rep2;
      rep2->count++;
      }
   return *this;
   }
```

```
template <class Rep>
CountPtr<Rep>::~CountPtr ()
   {
   if (--rep2->count == 0)
      {
      delete rep2->rep;
      delete rep2;
      }
   }
```

```
template <class Rep>
Rep* CountPtr<Rep>::operator -> () const
   {
   return rep2->rep;
   }
```

```
template <class Rep>
Rep& CountPtr<Rep>::operator * () const
   {
   return *rep2->rep;
   }
```

```
main ()
   {
   const CountPtr<Zhket> b (new Zhket ("Hello World"));
   CountPtr<Zhket> c (b);
   cout << "const CountPtr<Zhket> b (new Zhket (\"Hello World\"));" << endl;
   cout << "CountPtr<Zhket> c (b);" << endl;
   cout << "b: "; b.z (); b->zeige (); cout << "c: "; c.z (); c->zeige ();
   cout << "(*b)[1]: " << (*b)[1] << endl;
   cout << "b: "; b.z (); b->zeige (); cout << "c: "; c.z (); c->zeige ();
   cout << "(*c)[1]: " << (*c)[1] << endl;
   cout << "b: "; b.z (); b->zeige (); cout << "c: "; c.z (); c->zeige ();
   (*c)[1] = 'a'; cout << "(*c)[1] = 'a';" << endl;
   cout << "b: "; b.z (); b->zeige (); cout << "c: "; c.z (); c->zeige ();
   CountPtr<Zhket> d (new Zhket (*b + *c));
   cout << "*b + *c : "; d.z (); d->zeige ();
   return 0;
   }
```

Kapitel 13

Reguläre Ausdrücke

Möchte man in Texten nach einzelnen Strings oder Mustern suchen, sind Reguläre Ausdrücke unverzichtbar. Ab der Version C++11 sind Reguläre Ausdrücke im Standard definiert. Dies bedeutet die Standard-Bibliothek unterstützt diese Funktionalität.

Um Reguläre Ausdrücke Verwenden zu können muss `<regex>` inkludiert werden. Reguläre Ausdrücke werden von gcc ab der Version 4.8.1 unterstützt. Unterstützt der verwendete Compiler bzw. die verwendete Standard-Bibliothek dies nicht, kann alternativ die "Boost"-Bibliothek verwendet werden, indem `<boost/regex.hpp>` eingebunden wird. Zusätzlich muss beim Kompilieren gegen die "Boost"-Bibliothek gelinkt werden z.B. `g++ regex.cpp /usr/lib/libboost_regex.a`.

Im Folgenden werden Grundkenntnisse von Regulären Ausdrücken vorausgesetzt. Weitere Informationen dazu sind unter anderem bei Wikipedia zu finden (https://de.wikipedia.org/wiki/Regulärer_Ausdruck).

13.1 Die Reguläre Ausdrücke Klasse

Die Grundlage der Funktionalität bildet die Klasse `regex`. Mit ihr lassen sich die Suchmuster definieren.

```
static const regex wort(R"([A-Za-z][a-z]+)");
static const regex zahl(R"([0-9]+)");
```

Hier werden zwei einfache Suchmuster erstellt. Der erste Reguläre Ausdrück repräsentiert ein Wort, der zweite repräsentiert eine Zahl. Das `R` vor dem String, der das Suchmuster enthält, bewirkt, dass Sonderzeichen nicht escaped werden müssen.

Die "Boost"-Bibliothek kennt die `R`-Funktionalität nicht. Deshalb muss, wenn die "Boost"-Bibliothek verwendet wird, das vorangestellte `R` weggelassen werden und Sonderzeichen wie `\` escaped werden.

Das definierte Suchmuster kann auf drei verschiedene Weisen eingesetzt werden. Es kann dafür verwendet werden, einen String auf die exakte Übereinstimmung zu testen. Ferner kann man es dafür verwenden ein Suchmuster in einem Text zu finden oder zu ersetzen. In den folgenden Abschnitten werden die drei Möglichkeiten vorgestellt.

13.2 Übereinstimmung

Möchte man einen Text oder eine Zeichenkette auf die exakte Übereinstimmung mit einem Suchmuster prüfen, wird die Funktion `regex_match( --- )` verwendet.

```
string zeichenkette = "Hallo";
string satz = "Hallo Welt!";

if(regex_match(zeichenkette, wort))
   cout << "Übereinstimmung" << endl;
else
   cout << "keine Übereinstimmung" << endl;

if(regex_match(satz, wort))
   cout << "Übereinstimmung" << endl;
else
   cout << "keine Übereinstimmung" << endl;
```

Hier wird das oben definierte Suchmuster `wort` auf die Strings `zeichenkette` und `satz` angewendet. Es ist zu erwarten, dass das Suchmuster zwar auf `zeichenkette` zutrifft, aber nicht auf `satz`.

13.3 Suche

Um überprüfen zu können ob sich ein Suchmuster in einem Text befindet, verwendet man die Funktion `regex_search( --- )`.

```
string pincode = "123456";

if(regex_search(satz, wort))
   cout << "Enthalten" << endl;
else
   cout << "nicht Enthalten" << endl;

if(regex_search(pincode, wort))
   cout << "Enthalten" << endl;
else
   cout << "nicht Enthalten" << endl;
```

Nun wird das oben definierte Suchmuster `wort` auf die Strings `satz` und `pincode` angewendet. Hierbei trifft `satz` nun auf das Suchmuster zu, allerdings nicht `pincode`.

13.4 Ersetzen

Reguläre Ausdrücke können nicht nur zum Suchen, sondern auch zum Ersetzen verwendet werden. Dies passiert mit der Funktion `regex_replace( --- )`. Das folgende Beispiel zeigt wie.

```
string text = satz + " Mein Pincode lautet: " + pincode;

cout << "Vorher:" << "\n" << text << endl;

text = regex_replace(text, zahl, "****");

cout << "Nachher:" << "\n" << text << endl;
```

Das oben definierte Suchmuster `zahl` wird durch den String `****` ersetzt. Die Ausgabe sieht dann so aus:

```
Vorher:
Hallo Welt! Mein Pincode lautet: 123456
Nachher:
Hallo Welt! Mein Pincode lautet: ****
```

13.5 Ergebnisse

Um zum Beispiel auf die Ergebnisse der Suche zugreifen zu können, werden diese in Container gespeichert. Es existiert ein Iterator um durch die Ergebnisse laufen zu können.

13.5.1 Container

Es ist möglich einen Container zu definieren um die gefundenen Strings darin zu speichern. Dieser wird beim Aufruf der `regex_`- Funktion mit übergeben. Ein Container speichert immer nur ein Ergebnis der Suche.

```
smatch res;

regex_search(text, res, zahl);

if(!res.empty())
   cout << res.str() << endl;
```

Mit `!res.empty()` kann man feststellen, ob die Suche erfolgreich war und ob der Container nicht leer ist. Ist dies der Fall, kann man mit `res.str()` das Ergebnis als String aus dem Container heraus holen.

13.5.2 Iterator

Mit dem Iterator kann man durch die einzelnen Ergebnisse der Suche laufen. Diese werden wiederum in Container gespeichert. Der folgende Programmcode zeigt, wie man mit dem Iterator umgeht.

```
sregex_iterator begin(text.begin(), text.end(), wort);
sregex_iterator end;

while(begin != end)
   {
   cout << (*begin).str() << endl;
   ++begin;
   }
```

Zuerst werden zwei Iteratoren deklariert, aber nur einer initialisiert. Der nicht initialisierte Iterator dient dazu, um festzustellen ob das Ende erreicht wurde. Bei der Initialisierung des Iterators, wird ein Zeiger auf den Anfang und das Ende des Strings übergeben. Ferner wird noch das gesuchte Muster angegeben. Nun kann in einer Schleife über die Ergebnisse iteriert werden. Dazu wird um den Iterator auf das nächste Ergebnis zu setzen der Iterator inkrementiert.

13.6 Beispiel

Mit Regulären Ausdrücken kann man sehr einfach Worte und Zahlen eines Textes zählen. Im hier genannten Beispiel wir ein Abschnitt aus dem Text “Die Brueder Wright“ verwendet, welcher im Projekt Gutenberg veröffentlicht wurde.

```
#include <boost/regex.hpp>
#include <iostream>
#include <string>

using namespace std;
using namespace boost;

int main () {
   int count_wort = 0;
   int count_zahl = 0;

   string text;

   static const regex wort("([A-Za-z][a-z]+)");
   static const regex zahl("([0-9]+)");

   text = "Die Wrights führen ihren Stammbaum bis in das 14. Jahrhundert zurück.\
          Viele hervorragende Leute, deren Namen auch in der Geschichte verewigt\
          sind, haben der Familie angehört. Von grossmütterlicher Seite stammen\
          sie aus Holland, wo die ersten Aufzeichnungen bei Lord Afferden Ende des\
          14. Jahrhunderts beginnen und bis in die heutige Zeit vollständig\
          fortgeführt sind. Die Nachkommen des Lords wanderten später nach Amerika\
          aus und siedelten sich um das Jahr 1650 in Long Island an. Die\
          Grossmutter Katherine Reeder war verwandt mit dem Gouverneur Andrew H.\
          Reeder, der in Kansas im Jahre 1854 die Zügel der Regierung inne hatte.\
          Väterlicherseits können die Vorfahren zurückgeführt werden bis zu John\
          Wright, der im Jahre 1538 das Gut Kelvedon Hall im Kreise Essex in\
          England erwarb. Sein und seiner Frau Olive Nachkomme im vierten Grade,\
          Samuel Wright, wanderte im Jahre 1630 nach Amerika aus und siedelte sich\
          6 Jahre später als Farmer in Springfield in Massachusetts an. Hier wurde\
          er bald zum Diakon der ersten puritanischen Kirche und später zum\
          Pfarrer der Gemeinde erwählt. Nach segensreichem Wirken entschlief er\
          sanft im Jahre 1665 zu Northampton. Seine Nachkommen blieben in\
          Neu-England und manche berühmten Leute sind aus ihnen hervorgegangen. Zu\
          nennen sind Edmond Freeman, Reverend Joshua Moody, Reverend John\
          Russell, John Otis und John Porter in Windsor. Durch den letzten sind\
          die Wrights verwandt geworden mit dem berühmten amerikanischen General\
          Ulysses S. Grant und mit dem Präsidenten Grover Cleveland; ferner mit\
          dem bekannten General Joseph Warren in Bunkerhill. Der Grossvater\
          Wrights, Silas Wright, war Senator der Stadt New York und später\
          Gouverneur des Staates New York. Er besass umfangreiche Güter, um deren\
          Bewirtschaftung er sich selbst kümmerte. Seine Kinder wurden gleichfalls\
          zu Landleuten erzogen. Er starb in New York im Jahre 1847.";

   sregex_iterator it_wort(text.begin(), text.end(), wort);
   sregex_iterator it_zahl(text.begin(), text.end(), zahl);
   sregex_iterator it_end;

   for(;it_wort != it_end; ++it_wort)
      count_wort++;

   for(;it_zahl != it_end; ++it_zahl)
      count_zahl++;

   cout << "Der Text hat " << count_wort << " Worte und " << count_zahl << " Zahlen." << endl;
}
```

Literaturverzeichnis

[1] Bjarne Stroustrup, "The C++ Programming Language", Addison-Wesley

[2] Bjarne Stroustrup, "Die C++ Programmiersprache", Addison-Wesley

[3] Andrew Koenig und Bjarne Stroustrup, "The Annotated C++ Language Standard", Addison-Wesley

[4] Stanley B. Lippman, "C++ Primer", Addison-Wesley

[5] Scott Meyers, Ëffekive C++", Addison-Wesley

[6] Scott Meyers, Ëffekiv C++ programmieren", Addison-Wesley

[7] James Coplien, Ädvanced C++ Programming Styles and Idioms", Addison-Wesley

[8] Steve Teale, "C++ IOStreams Handbook", Addison-Wesley

[9] ISO/IEC 14882:1998, "Programming languages - C++", INTERNATIONAL STANDARD

[10] Jürgen Wolf, "C++ von A bis Z", Galileo Press