Programmieren in C++

Vorlesung im Sommersemester 1997

Peter Pfahler
Raum: F2.311
Telefon: 606688
Email: peter@uni-paderborn.de

1 Übersicht

1. C++ ist nicht C: Geschichte, Sprachkonzepte und ein Beispiel
2. Klassen
   - Das `class`-Konstrukt
   - Konstruktoren und Destruktoren
   - Freunde
3. Andere Datentypen im Überblick
4. Anweisungen und Ausdrücke
5. Überladen von Funktionen und Operatoren
6. Klassenschablonen
7. Klassenhierarchie und Vererbung
   - Abgeleitete Klassen
   - Abstrakte Klassen
   - Mehrfache Vererbung
8. Objektorientierter Programmentwurf:
   50 Ways to Improve Your Programs
9. Java: Überblick und Vergleich

2 C++ ist nicht C: Geschichte, Sprachkonzepte und ein Beispiel

```
// This may look like C code,
// but it is really -*- C++ -*-
```

Gnu g++ header files.

2.1 Das erste (und wichtigste) C++ -Beispiel

```
// hello.cc
#include <iostream.h>

void main(void)
{ cout << "Hello World!\n";
}
```

Übersetzung mit SUN oder GNU Compiler:

```
CC -o hello hello.cc
```

oder

```
g++ -o hello hello.cc
```

2.2 Geschichte

1980 Bjarne Stroustrup, AT&T Bell Laboratories
"C mit Klassen" (wie in SIMULA67)

1983 Entstehung des Names " C++ "

1985 Erste kommerzielle Portierungen (Mainframes, PC)

1986 B. Stroustrup: "The C++ Programming Language"

1990 Einsetzung einer ANSI Standardisierungsgruppe
(Vorschlag Exception Handling, Templates)

1995 Im April wird die erste Version des Normentwurfs
veröffentlicht

1996 Im Dezember wird die 2. Version des Normentwurfs
veröffentlicht

Beziehung zu C:

`As close as possible, but not closer.`

ANSI C++ -Komitee

2.3 Sprachkonzepte

C++ ist eine imperative Programmiersprache, die verschiedene Stile des Programmierens unterstützt, bzw. ermöglicht.

2.3.1 Programmierstil Prozedurales Programmieren

Entscheide, welche Prozeduren Du willst.
Verwende den besten Algorithmus, den Du kennst.

Beispiel:

```
double sqrt(double arg)
{ // Code to compute the square root
}

void main(void)
{ double root2 = sqrt(2);
  ...
}
```

2.3.2 Programmierstil Modulares Programmieren

Entscheide, welche Module Du brauchst.
Zerlege das Programm so,
daß die Daten in Module eingekapselt sind.

Beispiel:

Definition der Benutzungsschnittstelle:

```
// Schnittstellen-Datei "stack.h"
void push(char);
char pop(void);
const int stack_size = 100;
```

Implementierung:

```
#include "stack.h"
static char s[stack_size];
static char* top=s;

void push(char c)
{ // Ueberlauf pruefen und push
}

char pop(void)
{ // Unterlauf pruefen und pop
}
```

2.3.3 Programmierstil Datenabstraktion

Entscheide, welche Typen Du brauchst.
Stelle eine komplette Menge von Operationen für jeden Typ zur Verfügung (*Abstrakter Datentyp ADT*).

Beispiel:

```
class stack_id { /* ... */ };

stack_id create_stack(int size);
void push(stack_id, char);
char pop(stack_id);
void destroy_stack(stack_id);
```

Ohne Unterstützung der Programmiersprache unterscheiden sich solche benutzerdefinierten Typen in wesentlichen Aspekten von den eingebauten Typen (z.B. Wertzuweisung, Operatoren).

In C++ können sich Benutzertypen so verhalten wie eingebaute:

```
class complex
{ double re,im;
  public:
  complex(double r, double i) { re=r; im=i;}
  complex(double r)           { re=r; im=0;}
  friend complex operator+(complex, complex);
     ...
};
complex operator+(complex a, complex b)
{return complex(a.re + b.re, a.im + b.im);}
```

kann wie folgt benutzt werden:

```
void f()
{ complex a = 2.3;
  complex b = 1 + a;
  complex c = b + complex(1.5, 2.5);
  c = c + a;
}
```

2.3.4 Objektorientierte Programmierung

Probleme bei der Datenabstraktion: Wartung, Anpassung, Erweiterung.

Beispiel:

```
enum kind{ circle, rectangle, ...};
class shape {
      point center;  // Mittelpunkt;
      kind k;        // Form;
      public:
      point where() { return center;}
      void draw();
      };

void shape::draw()
{ switch (k)
  { case circle :    // Kreis zeichnen;
                     break;
    case rectangle : // Rechteck zeichnen;
                     break;
    ...
  }
}
```

Bei der Erweiterung um geometrische Formen, muß an vielen Stellen im Programmtext geändert werden. Dazu müssen insbesondere alle Programmquellen vorhanden sein.

Objektorientierte Programmierung löst dieses Problem durch den Vererbungsmechanismus:

```
class shape {
      point center;  // Mittelpunkt;
      public:
      point where() { return center;}
      virtual void draw();
      };
class circle : public shape {
      void draw();
      };
void circle::draw()
{ // Zeichne einen Kreis;;
}
```

Entscheide, welche Klassen Du brauchst.
Stelle für jede Klasse eine komplette Menge von Operationen zur Verfügung.
Mache Gemeinsamkeiten durch Vererbung explizit.

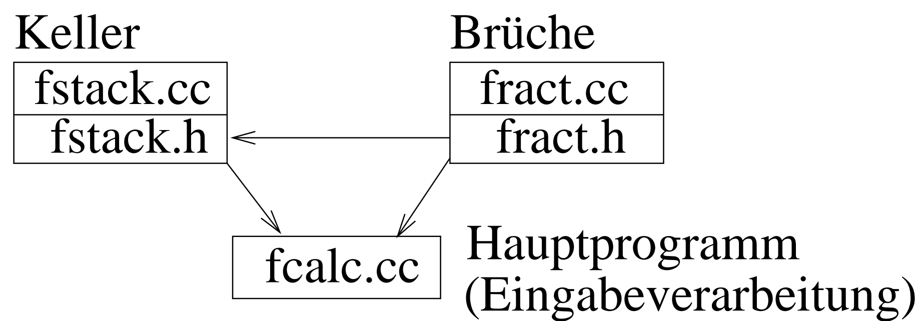
2.4 Das erste C++ -Projekt

Implementierung eines UPN-Rechners für Brüche:

Beispiel:

```
1 8 / 5 8 / + 1 - =
-1/4

6 2 3 / * =
4
```

Programmstruktur



2.4.1 `fstack`: Ein einfacher Keller

```
// fstack.h: a simple stack

#include "fract.h"    // class fraction

void            push(fraction val);  // Deklaration
fraction        pop(void);           // Deklaration
```

! Zeilenkommentar `//`.
! Funktionsprototypen

```
// fstack.cc: a simple stack

#include <iostream.h>
#include "fract.h"
#include "fstack.h"

const int       STACKSIZE = 20;
fraction        stack[STACKSIZE];
int             stackptr = 0;

void push(fraction val)
{   if (stackptr <= STACKSIZE - 1)
        stack[stackptr++] = val;
    else cerr << "Stack Overflow.\n";
} /* push */

fraction pop()
{   if (stackptr > 0)
        stackptr--;
    else cerr << "Stack Underflow.\n";
    return stack[stackptr];
} /* pop */
```

! Konstanten statt `#define`.

! Benutzung von `iostream`.

2.4.2 Die Klasse fraction

```
// fract.h: class "fraction"

#ifndef _FRACT_H
#define _FRACT_H
#include <iostream.h>

class fraction
{   int                fnum;
    int                fden;
    public:
    int num() {return fnum;}       // access fnum
    int den() {return fden;}       // access fden

    void print()                   // output
    {   if (fden != 1)
           cout << fnum << '/' << fden << "\n";
        else cout << fnum << "\n";
    } /* print */

    fraction(int n = 0, int d = 1) // constructor
    {   fnum = n;
        fden = d;
    } /* fraction */
};
```

```
// operators:
fraction operator + (fraction a, fraction b);
fraction operator - (fraction a, fraction b);
fraction operator * (fraction a, fraction b);
fraction operator / (fraction a, fraction b);
#endif
```

- **!** Klasse mit “private” und “public” Komponenten.
- **!** Element-Funktionen.
- **!** Konstruktoren.
- **!** Default-Parameter.
- **!** Definition überladener Operatoren.
- **!** Schutz vor Mehrfach-Inklusion.

```
// fract.cc: Fraction Operations

#include "fract.h"

int gcd(int a, int b)
{   if (a < 0) return gcd(-a, b);
    if (a == 0) return b;
    if (b < a)  return gcd(b, a);
    return gcd(b % a, a);
} /* gcd */

fraction reduce(fraction f)
{   fraction r;
    int g = gcd(f.num(), f.den());
    if (g != 0) r = fraction(f.num()/g, f.den()/g);
    return r;
} /* reduce */
```

```
fraction operator + (fraction a, fraction b)
{   fraction r (a.num()*b.den()+b.num()*a.den(),
                a.den()*b.den());
    return reduce(r);
}
fraction operator - (fraction a, fraction b)
{   fraction r (a.num()*b.den()-b.num()*a.den(),
                a.den()*b.den());
    return reduce(r);
}
fraction operator * (fraction a, fraction b)
{   fraction r (a.num()*b.num(), a.den()*b.den());
    return reduce(r);
}
fraction operator / (fraction a, fraction b)
{   fraction r (a.num()*b.den(), a.den()*b.num());
    return reduce(r);
}
```

! Verwendung von Member-Funktionen.
! Initialisierung mit Default-Konstruktor.
! Initialisierung mit explizitem Konstruktor.

2.4.3 `fcalc.cc`: Das Hauptprogramm

```
// fcalc.cc: a stack calculator

#include <iostream.h>
#include "fstack.h"
#include "fract.h"



int
main()
{
    char            inpch;
    int             num;

    cin >> inpch;
    while (inpch != '=')
    {
        if ('0' <= inpch && inpch <= '9')
        {
            cin.putback(inpch);
            cin >> num;
            push(num);
        } else
```

```
            switch (inpch)
            {
            case '+':
                { push(pop() + pop());
                  break;
                }
            case '-':
                { fraction right_operand = pop();
                  push(pop() - right_operand);
                  break;
                }
            case '*':
                { push(pop() * pop());
                  break;
                }
            case '/':
                { fraction right_operand = pop();
                  push(pop() / right_operand);
                  break;
                }
            default:
                { cout << "Input Error: "
                       << inpch << ".\n";
                  return 1;
                }
            } /* switch */
        cin >> inpch;
    } /* while */
    pop().print();
    return 0;
}
```

2.4.4 Ein Makefile für den Bruch-Rechner

```
CCC = g++

# object files:
O = fstack.o fcalc.o fract.o

# resulting executable:
R = fcalc

$(R) : $(O)
	$(CCC) $O -o $(R)

# dependencies:
fstack.o: fract.h fstack.h
fcalc.o: fstack.h fract.h
fract.o: fract.h
```

3 Klassen

`A class is a user-defined type.`

C++ Reference Manual

Arten von C++ Klassen

- `class`: Elemente standardmäßig `private`.
- `struct`: Elemente standardmäßig `public`.
- `union`: Elemente standardmäßig `public`, keine Vererbung.

Äquivalente Begriffe aus der OO-Welt

- Klasse: benutzerdefinierter C++ Typ.
- Objekt, Klasseninstanz: Wert von einem solchen Typ.
- Attribut: Datenelement.
- Methode: Elementfunktion (Member Function).
- Botschaft senden: Aufruf einer Elementfunktion.

Klassen sind voll in die Sprache integriert:

- Klassen können Elementtypen von Arrays sein.
- Klassenobjekte können an Funktionen übergeben werden.
- Klassenobjekte können von Funktionen zurückgegeben werden.
- Pointer können auf Klassenobjekte zeigen.
- Klassen können verschachtelt sein.
- Speicher kann dynamisch für Klassenobjekte allokiert werden.

C++ Klassen sind mächtiger als C Structures.
Sie unterstützen ADT und OOP durch

- Möglichkeiten der Zugriffsbeschränkungen (`public, private, protected, friend`).
- Funktionen als Klassenelemente ("Member Functions").
- Die Möglichkeit, Objekte zu initialisieren und wegzuräumen ("Konstruktor", "Destruktor").
- Vererbung mit abgeleiteten Klassen und virtuellen Funktionen.

3.1 Klassendefinition

```
struct date
{ int day, month, year;
};
```

Definiert den Typ `date`.

Objektdefinition

```
date today = {3,5,1993};
date heute = today;
```

(In C:

```
typedef struct { int day, month, year;} date;
```

)

Bevor ein Bezeichner in einem C++ -Programm verwendet werden darf, muß er deklariert worden sein:

- **Deklarationen** machen Bezeichner bekannt und liefern Typinformation.
- **Definitionen** liefern die vollständige Beschreibung dessen, wofür der Bezeichner steht.

Viele Deklarationen sind gleichzeitig Definitionen:

```
char ch;
int i = 77;
float real (complex *c) {return c->re;}
enum beer {Carlsberg, Tuborg};
```

aber es gibt auch “reine” Deklarationen

```
extern int stcktop;
date getdate();
complex add(complex, complex);
struct user;
```

Derart deklarierte Bezeicher müssen anderswo definiert sein.

Ein C++ -Programm muß für jeden verwendeten Bezeichner **genau eine Definition** enthalten. Für einen Bezeichner kann es aber **mehrere Deklarationen** geben (solange sie im Typ übereinstimmen).

Eine **Klassendefinition** ist mit der schließenden geschweiften Klammer abgeschlossen.

Eine Klasse gilt als **deklariert** nach Abschluß des Klassenkopfes, so daß der Klassenname schon innerhalb der Klassendefinition verwendet werden kann (z.B. für Zeiger auf Elemente des eigenen Typs). Natürlich sind direkte oder indirekte Rekursionen verboten ("Klasse enthält sich selbst").

Für verzeigerte rekursive Datentypen ist die **Vorausdeklaration** nützlich:

```
struct date;


struct date_list
{ date d;
  date_list * next;
};

struct date
{ int day, month, year;
};
```

3.2 Zugriffskontrolle

"Data Hiding" zur Unterstützung modularen Programmierens:

```
class X { int a;  // private by default
        };        // don't forget the ";"

struct X { int a; // public by default
         };
```

Die Standard-Zugriffsbeschränkung wird geändert durch Zugriffsspezifikationen:

- `public`: öffentlich zugreifbar für alle.
- `private`: zugreifbar für Elementfunktionen, Konstruktoren, Destruktoren und Freunde der eigenen Klasse.
- `protected`: bedeutet `public` für abgeleitete Klassen und `private` für den Rest des Programms.

Verletzung der Zugriffserlaubnis meldet der Compiler:

```
class date
{ int day, month, year;
};
...
date today;
cout << today.day;
```

g++:

```
member day is a private member of class date.
```

CC:

```
cannot access date::day: private member.
```

Verbot des Zugriffs auf `date, month, year` erlaubt Umstellung der Implementierung:

```
class date
{ long julian;
};
```

ohne die Anwendung zu tangieren.

3.3 Elementfunktionen ("Member Functions")

Menge der Operationen, die ein Anwender mit Objekten der Klasse ausführen kann.

Die Brauchbarkeit einer Klasse hängt von der Vollständigkeit und der Effektivität ihrer Elementfunktionen ab.

```
class date
{ int day, month, year;
public:
  void print(void);     // Deklaration
};

void date::print(void)  // Definition
{   cout << day << "." << month
         << "." << year << "\n";
}
```

`::` gibt an, daß `print` Element des Gültigkeitsbereichs von `date` ist ("scope resolution operator").

Elementfunktionen dürfen nur außerhalb der Klasse definiert werden, wenn sie innerhalb deklariert worden sind.

```
class date
{ int day, month, year;
public:
  void print(void);
};

void date::einbruch(void)  // Fehler
{   day = day+1;
}
```

Compiler:

```
date::einbruch() is not a member of date.
Compilation failed.
```

3.4 Inline-Elementfunktionen

Hinweis an den Compiler: Statt Aufruf der Funktion setze den Code der Funktion ein.
Vorteil: Geschwindigkeit.
Nachteil: Größerer Code.

Es gibt zwei Möglichkeiten:

```
class date
{ int day, month, year;
public:
  void print(void)        // Definition
  { cout << day << "." << month
         << "." << year << "\n"; }
};
```

oder

```
class date
{ int day, month, year;
public:
  void print(void);       // Deklaration
};

inline void date::print(void)
{   cout << day << "." << month
         << "." << year << "\n";
}
```

Tip: Definition außerhalb der Klasse kann leichter in `inline` und zurück verwandelt werden.

Elementfunktionen dürfen natürlich Parameter haben. Hier ein vollständiges Programm:

```
#include <iostream.h>

class date
{   int day, month, year;
  public:
    void set(int d, int m, int y);
    // besser: Konstruktor (spaeter)
    void print();
    void advance(int);
};

int dayspermonth(int m, int y)
{   static int dpm[] = {0,31,28,31,30,31,30,
                        31,31,30,31,30,31};
    if (m != 2) return dpm[m];
    else return 28 + (y%4 == 0)
                   - (y%100 == 0)
                   + (y%400 == 0);
} /* dayspermonth */

void date::set(int d, int m, int y)
    {   day=d;
        month=m;
        year=y;
    } /* date::set */
```

```
void date::advance(int n)
{ int d;
  day+=n;
  while (day > (d = dayspermonth(month,year)))
         {   day -= d;
             if (++month > 12)
             {   month = 1;
                 year++;
             } /* if */
         } /* while */
} /* date::advance */

void date::print()
{    cout << day << "." << month
          << "." << year << "\n";
} /* date::print */

int main(void)
{   date today;
    today.set(3,5,1993);
    today.advance(235);
    today.print();
} /* main */
```

3.5 Der `this`-Zeiger

Innerhalb einer Elementfunktion bezeichnet das Schlüsselwort **`this`** einen Zeiger auf das Objekt, für das die Funktion aufgerufen wurde.

Beispiel:

Berechne Datum +n Tage ohne Veränderung des Originals:

```
class date
{   int day, month, year;
  public:
    ...
    date add(int);
};

date date::add(int n)
{   date r = *this;
    r.advance(n);
    return r;
} /* date::add */

...
    date xmas = today.add(235);
    xmas.print();
...
```

3.6 Konstante Elementfunktionen

Ein konstantes C++ Objekt darf nicht verändert werden:

```
const char blank = ' ';
...
blank = '\t';     // Fehler!
```

Genauso dürfen für konstante Klassenobjekte nur Elementfunktionen aufgerufen werden, die das Objekt nicht verändern. Solche Elementfunktionen müssen als `const` markiert werden:

```
class date
{   int day, month, year;
  public:
    ...
    int getday(void) const
    { return day;}
    int changeyear(int y)
    { year = y;}
};


    ...
    const date fix = today;
    cout << fix.getday();     // OK.
    fix.changeyear(2000);     // Fehler!
```

Compiler:

```
non-const memberfunction date::changeyear
called for const object.
Compilation failed.
```

Experiment:

```
...
int changeyear (int y) const  // Luege
{ year = y; }
...
```

CC:

```
error: assignment to const type.
```

g++:

```
warning: assignment to read-only member year.
```

3.7 Statische Klassenelemente

Daten- und Funktionselemente einer Klasse können statisch deklariert werden. Ein solches Element existiert dann nur einmal. Alle Objekte einer Klasse teilen es sich ("Klassenspezifische Daten und Funktionen").

```
class date
{   int day, month, year;
  public:
    static int printcount;
    static void print_printcount();
    void print();
};

void date::print()
{    cout << day << "." << month
          << "." << year << "\n";
     printcount++;
}

// Die static-Deklaration ist keine Definition.
// Die erfolgt hier:

int date::printcount = 0;

void date::print_printcount()
{ cout << printcount << "Daten gedruckt";
}
```

Ein statisches Datenobjekt ist ein separates Objekt. Es existiert auch, wenn kein Objekt seiner Klasse existiert.

Benutzung (objektgebunden oder nicht):

```
int main(void)
{   date today;
    ...
    cout << today.printcount;
    cout << date::printcount;
    today.print_printcount();
    date::print_printcount();
} /* main */
```

Vorteil gegenüber globalen Variablen und Funktionen:

Das statische Klassenelement ist global für die Elemente seiner Klasse, beeinflußt den Rest des Programms aber nicht (keine Namenskonflikte).

3.8 Freunde

Es gibt Situationen, wo man Nicht-Elementfunktionen Zugriff auf die privaten Daten gewähren möchte. Dazu dient das `friend`-Konstrukt:

```
class ich
{ private:
    ...
  public:
    ...
    // eine Funktion als Freund:
    friend int vertrauensvoll(void);

    // eine Elementfunktion als Freund:
    friend void andereklasse::elemfu(int);

    // eine ganze Klasse von Freunden:
    friend class meinfreund;
};
```

Durch den Zugriff auf private Daten können Freunde gefährlich sein.

3.9 Konstruktoren

Beginn der Lebenszeit einer Variablen:

1. Reserviere Speicherplatz für die Variable.
2. Rufe einen **Konstruktor** auf, der die Variable initialisiert.

Ende der Lebenszeit einer Variablen:

1. Rufe einen **Destruktor** auf, der notwendige Aufräumungsarbeiten ausführt.
2. Gib den Speicher für die Variable wieder frei.

3.9.1 Primitive Datentypen

Für primitive Datentypen sind Konstruktoren und Destruktoren sehr einfach. Der Compiler kennt sie und wendet sie automatisch an. Initialisierung kann auf drei verschiedenen Arten spezifiziert werden:

```
initializer:
   '=' assignment_expression |
   '=' '{' initializer_list '}' |
   '(' expression_list ')' .
```

C++ Reference Manual

Die “= expression”-Notation stammt aus C:

```
int i = 7;
complex z = complex(3.4,6);
char *p = "hallo\n";
```

Die “= initializer_list”-Notation ist ebenfalls aus C. Sie dient zur Initialisierung von Arrays und Structures:

```
struct studi
{ char *name;
  long matrnr;
} st_liste[] = {"Adam", 3567890,
                "Eva",  3678901};
```

Die “(expression_list)”-Notation stammt aus Simula. Wird meist für nicht-arithmetische Typen genommen:

```
date today(25,4,1994);
```

Die Notationen 1 und 3 sind austauschbar (Geschmacksache):

```
int i(7);
date today = date(25,4,1994);
complex z(3.4,6);
```

Bei fehlender Initialisierung werden

- Statische Variablen und statische Klassenelemente mit 0 von geeignetem Typ initialisiert.
- Automatische und Registervariablen werden nicht initialisiert (Gefahr!).

3.9.2 Konstruktoren für Klassen

Müssen vom Klassen-Designer zur Verfügung gestellt werden. Konstruktoren haben den selben Namen wie die Klasse. Sie haben keinen Ergebnistyp und dürfen auch kein Ergebnis liefern.

```
class date
{   int day, month, year;
 public:
    date(int, int, int); // Konstruktor-Dekl.
    void print();
};
// Konstruktor-Definition:
date::date(int d, int m, int y)
{ // Initialisierung durch Wertzuweisung:
  day = d; month = m; year = y; }

...
date xmax(24,12,1997);       // Simula-Stil
date today = date(17,4,1997); // C-Stil
```

Wie Elementfunktionen darf der Konstruktor auch innerhalb der Klassendeklaration definiert werden.

Eine andere Möglichkeit für die Initialisierung der Klassenelemente:

```
class date
{   int day, month, year;
 public:
    date(int, int, int); // Konstruktor-Dekl.
    void print();
};
// Konstruktor-Definition:
date::date(int d, int m, int y)
: day(d), month(m), year(y) {}
```

Diese **Initialisierungsliste** initialisiert die Klassenelemente mit deren Konstruktor. Im Rumpf des Konstruktors ist in unserem Beispiel nichts mehr zu tun.

Wertzuweisung und Initialisierung sind zwei sehr verschiedene Dinge (mehr dazu später).

Es darf beliebig viele Konstruktorfunktionen geben. Sie müssen in Anzahl und/oder Typ der Parameter unterscheidbar sein.

```
class date
{   int day, month, year;
 public:
    date(int, int, int); // Konstruktor-Dekl.
    date(int);           // Konstruktor-Dekl.
    void print();
};

// Konstruktor-Definitionen:
date::date(int d, int m, int y)
: day(d), month(m), year(y) {}

date::date(int y)
: year(y), day(1), month(1) {}

...
date newyear(1999);
```

3.9.3 Der Default-Konstruktor

Ein **Default-Konstruktor** ist ein Konstruktor ohne Parameter:

```
class date
{   int day, month, year;
 public:
    date() { day=1; month=1; year=1956;}
};
```

Jede Klasse sollte einen Default-Konstruktor haben.

Der Default-Konstruktor wird aufgerufen für

- Nicht initialisierte Objekte:

```
date tomorrow;
```

- Arrays von dynamisch allokierten Objekten:

```
date* urlaub = new date[29];
```

- Objekte, die Elemente von Klassen sind:

```
class geburtstag
{ char *name;
  date geboren;
  ...
}
```

wenn sie nicht von deren Konstruktoren explizit konstruiert werden.

Wenn eine Klasse überhaupt keinen Konstruktor hat, wird der Default-Konstruktor vom Compiler generiert:

```
 class date
 {   int day, month, year;
 };
 ...
date irgendwann; // OK, aber unschoen!
```

Existiert jedoch ein Konstruktor, muß auch der Default-Konstruktor angegeben werden (wenn er benutzt wird):

```
class date
{   int day, month, year;
  public:
    date(int d, int m, int y)
         : day(d),month(m) ,year(y) {};
};
...
date irgendwann;    // FEHLER!
```

COMPILER:

```
Too few arguments for constructor ‘date’
Compilation failed
```

3.9.4 Der Copy-Konstruktor

Der **Copy-Konstruktor** hat folgende Aufgaben:

- Initialisierung von Objekten durch Kopieren anderer Objekte:

```
date today(25,4,1994);
date heute = today;
```

- Value-Parameter-Übergabe von Klassenobjekten:

```
void remember(date d);
...
remember(today);
```

- Wertrückgabe von Objekten:

```
date nextparty()
{ ...
  return libori;
}
```

Ein Copy-Konstruktor für Klasse `X` hat die Form

```
X::X(const X&) // konstanter Referenzparameter
```

Wenn kein Copy-Konstruktor angegeben wird, wird er vom Compiler generiert. Der generierte Copy-Konstruktor erzeugt elementweise Kopien der Datenelemente des Objekts.

Ein selbstgeschriebener Copy-Konstruktor

```
    date(const date& original)
    {  day=original.day;
       month=original.month;
       year=original.year;
       cout << "Handmade Copy Constructor\n";
   } /* date */
```

Eigene Copy-Konstruktoren werden in der Regel nötig, wenn die Objekte dynamische Daten enthalten.

Beispiel:

```
class student
{   char *name;
    int matrnr;
  public:
    student(char *, int);   // Konstruktor
    void gross(void);
    void print();
};
void student::gross(void)
{ *name = *name - ' ';
}
void student::print()
{  cout << name << " " << matrnr << "\n";
}
...
student erika("mustermann",3567890);
student clone = erika;
erika.gross();
clone.print(); // Schreibt "Mustermann"
```

$\Longrightarrow$ Elementweises Kopieren genügt nicht. Copy-Konstruktor selbst schreiben (z.B. mit `strcpy`).

3.10 Initialisierung $\Longrightarrow \Longleftarrow$ Wertzuweisung

Beispiel:

```
class date
{   int day, month,year;
  public:
    date(int d, int m, int y)
    : day(d), month(m), year(y) {}
};

class geburtstag
{   char *name;
    date geb;
   public:
    geburtstag(char *n, date d)
    { name = n;
      geb = d;
    }
};

int main(void)
{  geburtstag ich("Peter",date(1,4,1971));
} /* main */
```

Ablauf der Konstruktion von Objekt `ich`:

1. Initialisiere mit Konstruktor `geburtstag`
   dazu:
   (a) Initialisiere die Datenelemente der Reihe nach
   dazu:
      i. Gibt es einen Konstruktor-Aufruf für `name`?
      Nein. $\Longrightarrow$ nimm Default-Konstruktor für `char*`.
      ii. Gibt es einen Konstruktor-Aufruf für `geb`?
      Nein. $\Longrightarrow$ nimm Default-Konstruktor für `date`.
      Den gibt es nicht $\Longrightarrow$ FEHLER!

Die Klasse **geburtstag** sollte besser so aussehen:

```
class geburtstag
{   char *name;
    date geb;
  public:
    geburtstag(char *n, date d)
    : name(n), geb(d) { }
};
```

Ablauf der Konstruktion von Objekt `ich`:

1. Initialisiere mit Konstruktor `geburtstag`
   dazu:
   (a) Initialisiere die Datenelemente der Reihe nach
       dazu:
       i. Gibt es einen Konstruktor-Aufruf für `name`?
          Ja. $\Longrightarrow$ führe `name(n)` aus.
       ii. Gibt es einen Konstruktor-Aufruf für `geb`?
          Ja. $\Longrightarrow$ führe `geb(d)` aus.

       dann:
   (b) Führe den Konstruktor `geburtstag` aus (Hier leer).

3.11 Destruktoren

Der Destruktor ("es kann nur einen geben") einer Klasse `X` heißt `~X`. Er wird jedesmal aufgerufen, wenn ein Objekt der Klasse `X` gelöscht wird:

- Statische Variablen: Am Ende des Programms.
- Automatische Variablen: Am Ende ihrer Funktion.
- Dynamische Objekte: Beim `delete`-Aufruf.

Der Destruktor hat keinen Ergebnistyp und keine Parameter.

Beispiel:

```
class string
{   char *s;
  public:
    string(char *p) : s(p)
      { cout << "create " << s << "\n";}
    ~string()
      {cout << "destroy " << s << "\n";}
};

string g("global");

int main()
{   string s1("hans");
    string s2("otto");
} /* main */
```

erzeugt die Ausgabe

```
create global
create hans
create otto
destroy otto
destroy hans
destroy global
```

Destruktoren werden zum Freigeben dynamisch allokierten Speichers verwendet:

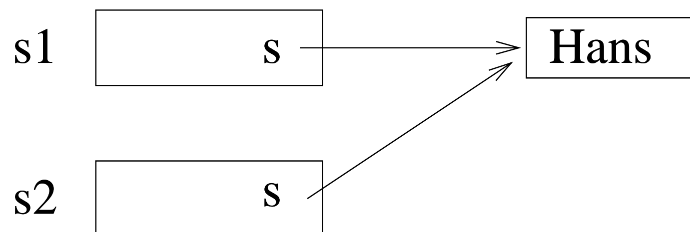
```
class string
{   char *s;
  public:
    string(char *p)
    { s = new char[strlen(p)+1];
      strcpy(s,p);
      cout << "create " << s << "\n";}
    ~string()
    { cout << "destroy " << s << "\n";
      delete [] s;
    }
};
```

Bei fehlenden Benutzer-definierten Copy-Konstruktoren kann es Probleme geben:

```
int main()
{   string s1("Hans");
    string s2=s1;
} /* main */
```

erzeugt folgende Ausgabe:

```
create Hans
destroy Hans
destroy Hans    // "Hans" ist schon freigegeben
```



3.12 Vordefinierte Klassen: Die C++ I/O-Bibliothek

Die Ein/Ausgabe ist nicht in der Sprache definiert, sondern Bestandteil der Standardbibliothek.

	ANSI C	**C++**

Stream I/O	#include <stdio.h>
	printf(...)
	scanf(...)
		#include <iostream.h>
		cout << X;
		cin >> X;

File I/O	#include <stdio.h>
	FILE out;
	out=fopen("o","w");
	fprintf(out, ...)
	FILE inp;
	inp=fopen("i","r");
	fscanf(inp, ...)
		#include <fstream.h>
		ofstream out("o");
		
		out << X;
		ifstream inp("i");
		
		inp >> X;

String I/O	#include <stdio.h>
	sprintf(...)
	
	sscanf(...)
		#include <strstream.h>
		char t[80];
		ostrstream otxt(t,80);
		otxt << X;
		istrstream itxt(t,80);
		itxt >> X;

3.12.1 Stream-Ausgabe

Drei Output-Stream-Objekte sind definiert:

`cout`: Standardausgabe.

`cerr`: Ungepufferte Fehlerausgabe.

`clog`: Gepufferte Fehlerausgabe.

Die Ausgabe erfolgt mit dem **Einfügeoperator `<<`**, der für die Standardtypen vordefinert ist. Z. B.:

```
ostream& operator<<(char);
ostream& operator<<(unsigned char);
ostream& operator<<(const char*);
ostream& operator<<(int);
ostream& operator<<(long);
ostream& operator<<(double);
ostream& operator<<(float);
ostream& operator<<(unsigned int);
ostream& operator<<(unsigned long);
ostream& operator<<(void*);
ostream& operator<<(short);
ostream& operator<<(unsigned short)
```

Für formatierte Ausgabe stehen sog. **Manipulatoren** zur Verfügung, die ebenfalls durch << in den Ausgabe-Stream eingefügt werden. Dazu "#include <iomanip.h>".

Wichtige Ausgabe-Manipulatoren	
Manipulator	**Bedeutung / Benutzung**
hex	Hex-Ausgabe `cout << hex << 17;`
dec	Dezimal-Ausgabe `cout << dec << i;`
oct	Oktal-Ausgabe `cout << oct << 100;`
endl	Newline und Flush `cout << endl;`
flush	Flush `cout << flush;`
setw(int)	Feldweite `cout << setw(8);`
setfill(int)	Füllbuchstabe `cout << setfill('.');`
setiosflags(long)	Setzen von Flags `cout << setiosflags(ios::right);`
resetiosflags(long)	Rücksetzen von Flags `cout << resetiosflags(ios::dec);`
setprecision(int)	Fließpunktpräzision `cout << setprecision(6);`

Einige wichtige ios-Flags (Ausgabe)	
Flag	**Bedeutung**
`ios::left`	Linksbündige Ausgabe
`ios::right`	Rechtsbündige Ausgabe
`ios::scientific`	Wissenschaftliches Float-Format
	z.B. `-1.23E+04`
`ios::fixed`	Festpunkformat für Float
	z.B. `123.45`

Beispiel:

```
#include <iostream.h>
#include <iomanip.h>
struct buch
{ char *kapitel;
  int seite;
} inhalt[4] = { "Einleitung", 1,
                "Grundlagen", 11,
                "Neuigkeiten", 102,
                "Zusammenfassung",102
              };
void druckeinhalt()
{ cout << setfill('.');
  for (int kap=0; kap<4 ; kap++)
  { cout << setiosflags(ios::left) << setw(20);
    cout << inhalt[kap].kapitel;
    cout << resetiosflags(ios::left) << setw(4);
    cout << inhalt[kap].seite << endl;
  }
}
```

liefert die Ausgabe:

```
Einleitung.............1
Grundlagen............11
Neuigkeiten..........102
Zusammenfassung......102
```

Selbstdefinierte Einfügeoperatoren <<

Die erforderliche Signatur ist

```
ostream& operator << (ostream& OS, classtype& E)
```

oder

```
ostream& operator << (ostream& OS, classtype E)
```

Beispiel:

```
#include <iostream.h>
class date
{   int day, month, year;
  public:
    date(int d, int m, int y)
        : day(d), month(m), year(y) {};
    friend ostream& operator << (ostream&, date&);
};

ostream& operator << (ostream& OS, date& D)
{ OS << D.day << '.' << D.month << '.' << D.year;
  return OS;
}

int main(void) /*  */
{   date today(3,5,1993);
    cout << today;
}
```

3.12.2 Stream-Eingabe

Ein Input-Stream-Objekt ist definiert:

`cin`: Standardeingabe.

Die Eingabe erfolgt mit dem **Extraktionsoperator `>>`**, der für die Standardtypen vordefiniert ist.
Der Extraktionsoperator betrachtet "Whitespace" (Leerzeichen, Tabulatoren, Zeilenende) als Abgrenzung zwischen Eingabesymbolen und nicht als eigenständige Symbole.

Das Programm

```
char z;
while (cin >> z)
   cout << z << ", "
```

erzeugt aus der Eingabe

```
ab
 c
```

die Ausgabe

```
a, b, c,
```

```
#include <iostream.h>
#include <string.h>
const int puffergroesse = 28;
char puffer[puffergroesse];
int main(void)
{ int anzahl = 0, max = 1, laenge;
  char laengstes[puffergroesse];
  while (cin >> puffer)
  { laenge = strlen(puffer);
    anzahl++;
    if (laenge > max)
    { max = laenge;
      strcpy(laengstes, puffer);
    } /* if */
  } /* while */
  cout << "Anzahl der Woerter: " << anzahl << endl;
  cout << "Laengstes Wort: " << laengstes << endl;
} /* main */
```

ergibt mit der Eingabe

```
Wo immer im Mittelalter innerhalb der von Deutschen
besiedelten Landstriche sich Zuege feinerer
Gesittung und Geselligkeit zeigten, stammten sie
aus dem Morgenland oder aus dem Erbe der roemischen
Kultur.
```

die Ausgabe

```
Anzahl der Woerter: 29
Laengstes Wort: Geselligkeit
```

Wichtige Eingabe-Manipulatoren	
Manipulator	**Bedeutung / Benutzung**
hex	Hex-Eingabe `cin >> hex >> i;`
dec	Dezimal-Eingabe `cin >> dec >> i;`
oct	Oktal-Eingabe `cin >> oct >> i;`
setw(int)	Feldweite `cin >> setw(24);`
setfill(int)	Füllbuchstabe `cin >> setfill(' ');`
setiosflags(long)	Setzen von Flags `cin >> setiosflags(ios::fixed);`
resetiosflags(long)	Rücksetzen von Flags `cin >> resetiosflags(ios::dec);`
setprecision(int)	Fließpunktpräzision `cin >> setprecision(12);`

Einige wichtige ios-Flags (Eingabe)	
Flag	**Bedeutung**
`ios::skipws`	Überlese Whitespace
`ios::scientific`	Wissenschaftliches Float-Format z.B. `-1.23E+04`
`ios::fixed`	Festpunkformat für Float z.B. `123.45`

Mit `setw(int)` lässt sich verhindern, daß Eingabepuffer überlaufen. Obiges Programm hätte die Strings besser so eingelesen:

```
...
cin >> setw(puffergroesse); // lies hoechstens
while ( cin >> puffer) // puffergroesse Zeichen
{ ...
```

Weitere Eingabeoperationen

Die Klasse `istream` stellt außer dem Operator `>>` noch andere nützliche Funktionen zur Verfügung, z. B.

```
istream& get(char& c);
istream& get(char* p, int n, char delim = '\n');
istream& putback(char c);
int get();
int peek();
```

- `get(Z)` liest ein einzelnes Zeichen und speichert es in Z (auch Whitespace !).
- `get(puffer, grenze, trennzeichen)` liest eine Folge von höchstens `grenze` - 1 Zeichen und speichert sie in `puffer`. Wird das Ende der Datei erreicht oder das `trennzeichen` gelesen, werden weniger als `grenze` -1 Zeichen geliefert.
- `putback(c)` schickt das Zeichen `c` in den Eingabestrom zurück.
- `get()` liest ein Zeichen und liefert es als Ergebnis zurück (auch EOF).
- `peek()` liefert das nächste Zeichen ohne es einzulesen (auch EOF).

Benutzerdefinierte Extraktions-Operatoren `>>` lassen sich ebenfalls definieren.
Die erforderliche Signatur ist

```
istream& operator >> (istream& IS, classtype& E)
```

Solche Eingabefunktionen sind in der Regel komplizierter als Ausgabefunktionen, da sie im Fehlerfall die Statusbits des Eingabestroms setzen müssen.

3.12.3 Ein/Ausgabe von Dateien

Öffnen eines Ausgabe-Files und Verbindung mit einer Datei:

```
ofstream outp("outfilename");
```

Öffnen eines Eingabe-Files und Verbindung mit einer Datei:

```
ifstream inp("infilename");
```

Öffnen eines Ein/Ausgabe-Files und Verbindung mit einer Datei:

```
fstream iof("filename", mode);
```

Test, ob das Öffnen der Datei erfolgreich war:

```
ifstream eingabe("inp");
if (!eingabe)
{    cerr << "Cannot open file inp\n");
     exit(1);
}
```

Ein File-Objekt läßt sich auch erst kreieren und später öffnen:

```
ifstream eingabe;     // Default-Konstruktor
eingabe.open("inp"); // Oeffnen
...
eingabe.close();     // Schliessen
```

Der I/O-Modus kann auch explizit angegeben werden, z. B.:

```
ifstream inp("infile", ios::in);
ofstream outp("outfile", ios::out);
```

Einige wichtige I/O-Modi	
Flag	**Bedeutung**
`ios::app`	Anhängen von Daten an Ausgabe-File
`ios::out`	Ausgabe (mit Überschreiben)
`ios::in`	Eingabe
`ios::nocreate`	Öffnet nur Dateien, die existieren
`ios::noreplace`	Öffnet nur Dateien, die nicht existieren, es sei denn zum Anhängen.

Neben den Operatoren `<<` und `>>` können auch die Funktionen `get()` und `put()` benutzt werden.

Beispiel: Filecopy-Programm

```
#include <stdlib.h>   // fuer exit()
#include <fstream.h>

int main(int argc, char **argv)
{ if (argc < 3)
  { cerr << "Usage: filecopy inp outp\n"; exit(0);}
  ifstream inp(argv[1]);
  if (!inp)
  { cerr << "Cannot open " << argv[1]; exit(1);}
  ofstream outp(argv[2]);
  if (!outp)
  { cerr << "Cannot open " << argv[2]; exit(1);}

  char c;          // jetzt geht's los:
  while (inp.get(c) && outp) outp.put(c);
} /* main */
```

3.12.4 Ein/Ausgabe von und in Strings

- Internes Erzeugen von Ausgabetexten.
- Stringverarbeitung und Manipulation.

Zwei vordefinierte Klassen:

- `ostrstream`: Schreiben in Strings.
- `istrstream`: Lesen aus Strings.

Schreiben in Strings
Beispiel:

```
const int buflen = 128;
char buf[buflen];

char *greet(char *name)
{ // stream oeffnen und mit "buf" verbinden:
  ostrstream s(buf,buflen);
  s << "Hello " << name << "!\n";
  return buf;
}
int main(void)
{ cout << greet("Krause");
} /* main */
```

Lesen aus Strings

Beispiel:

```
const int buflen = 128;
char buf[buflen] = "120     Mueller   17.25";

int main(void)
{ int num;
  char name[20];
  float dollar;
  istrstream s(buf,buflen);
  s >> num >> name >> dollar;
  cout << "Num: " << num
       << " Name: " << name
       << " Dollar: " << dollar << endl;
} /* main */
```

4 Grundtypen und abgeleitete Typen

4.1 Die Grundtypen von C++

- Ganzzahlige Typen
  - char
  - short int
  - int
  - long int
- Fließkomma-Typen
  - float
  - double
  - long double

Die Regeln für die Speicherplatzgröße:

$$
\begin{aligned}
1 &= \text{sizeof(char)} \\
&<= \text{sizeof(short int)} \\
&<= \text{sizeof(int)} \\
&<= \text{sizeof(long int)}
\end{aligned}
$$

$$
\begin{aligned}
&\text{sizeof(float)} \\
<= &\text{ sizeof(double)} \\
<= &\text{ sizeof(long double)}
\end{aligned}
$$

und

char	hat mindestens	8 Bit
short int	hat mindestens	16 Bit
long int	hat mindestens	32 Bit

Vorzeichen für ganzzahlige Typen

unsigned	char	
unsigned	short int	
unsigned	int	
unsigned	long int	
signed	char	
signed	short int	// = short int
signed	int	// = int
signed	long int	// = long int

char ist abhängig von der Maschine vorzeichenbehaftet oder nicht
$\Longrightarrow$ char, signed char und unsigned char sind verschiedene Typen.

4.1.1 Implizite Typumwandlung

Die Grundtypen können in Wertzuweisungen und Ausdrücken frei kombiniert werden $\Longrightarrow$ Information kann verloren gehen.

Beispiel:

```
int i = 256 + 255;
char ch = i;
int wieviel = ch;
```

$$\texttt{wieviel} = \begin{cases} -1 & falls\ char\ signed\ (VAX) \\ 255 & falls\ char\ unsigned\ (68k) \end{cases}$$

Die implizite Typumwandlung wird durch recht komplexe Regeln beschrieben. Festgelegt werden

- Ausweitung (*Integral Promotion*): Die Umwandlung von `char`, `short int` und Aufzählungselementen nach `int`.
- Integer-Konvertierung: Die Umwandlung von `int` in andere ganzzahlige Typen.
- Konvertierungen zwischen `float` und `double`.
- Konvertierungen zwischen Fließkommazahlen und ganzen Zahlen.

Angewandt werden diese Regeln z.B. in der Operandenanpassung für arithmetische Ausdrücke.

Beispiel:

```
float x = 8.0;
x *= 1.0 / 2.0    // x == 4.0
x *= 1.0 / 2      // x == 2.0
x *= 1 / 2.0      // x == 1.0
x *= 1 / 2        // x == 0.0
```

4.1.2 Explizite Typumwandlung

Zwei mögliche Notationen:

- C-Cast-Notation: `d = (double) a;`
- Funktionale Notation: `d = double(a);`

Die funktionale Notation geht nur bei Typen mit einfachem Namen, also nicht bei:

```
char *p = (char *)0777;
```

Abhilfe durch `typedef`:

```
typedef char* charptr;
char *p = charptr(0777);
```

4.1.3 Benutzer-definierte Umwandlung

Umwandlung mit Konstruktoren

Ziel: Umwandlung in einen Klassentyp.
Konstruktor-Anwendung ist auch eine explizite Typumwandlung:

```
date today = date(17,5,1993);
```

Mit einstelligen Konstruktoren (bzw. mehrstelligen mit Default-Werten für den zweiten bis letzten Parameter) bekommen wir auch implizite Typumwandlung:

```
class date
{   int julian;
  public:
    date(int d);
    date(char *datestring);
}
void f(date d)
{ date d1 = 17503;
  date d2 = "12.3.1986";
  d1 = 2001;    // date tmp = 2001; d1 = tmp
  f(4000);      // date tmp = 4000; f(tmp)
}
```

Umwandlung mit Operatoren

Ziel: Umwandlung vom Klassentyp in einen anderen Typ.
Form: `X::operator T()`
definiert eine Umwandlung von `X` nach `T`. Diese wird vom Compiler implizit angewandt:

```
class tiny
{    char val;
  public:
     tiny(int i)
     { if (i<0 || i > 63)
       error("Out of Range\n");
       else val = i;
     }
     operator int()
     { return val;}
 };
 tiny t1 = 5;
 tiny t2 = 62;
 int i = t2 - t1;
```

Vorsicht mit
explizten und impliziten
Typumwandlungen.

4.2 Konstanten

Es gibt

- Literale Konstanten
  - Integer-Konstanten
  - Fließkomma-Konstanten
  - Textzeichen-Konstanten
  - String-Konstanten
- Benannte Konstanten
  - Als `const` deklarierte Werte
  - Elemente von Aufzählungstypen
  - Array- und Funktionsnamen

4.2.1 Integer-Konstanten

Dezimal:	`0 1234 1234567890123456789`
Oktal:	`012 0123`
Hexadezimal:	`0x0 0x2a 0xffff`

Der Typ einer Integer-Konstanten ist abhängig von der Größe des Wertes:
Dezimal-Konstanten: `int`, sonst `long int`, sonst `unsigned long int`.

Hex- und Oktalkonstanten: `int`, sonst `unsigned int`, sonst `long int`, sonst `unsigned long int`.
Vorzeichenbehaftung und Länge können auch explizit angegeben werden:

```
void f(int);
void f(long int);
void f(unsigned int);
void f(unsigned long int);
...
f(3)   // ruft f(int)
f(3L)  // ruft f(long)
f(3U)  // ruft f(unsigned int)
f(3LU)  // ruft f(unsigned long int)
```

4.2.2 Fließkomma-Konstanten

Form: `1.2` `.23` `44.` `1.2e10` `456.789e-19`

Typ: stets `double` außer wenn mit Suffix `f` versehen wie

`3.145159f`

4.2.3 Textzeichen-Konstanten

Form: `'a'` `'7'`

Typ: `char`

Einige Sonderformen	
Zeichen	**Schreibweise**
newline	`'\n'`
horizontal tab	`'\t'`
alert	`'\a'`
backslash	`'\\'`
single quote	`'\''`
double quote	`'"'`
integer 0	`'\0'`
octal number	`'\ooo'` (1-3 Ziffern)
hex number	`'\xhhh'` (1-3 Ziffern)

4.2.4 Strings

Form: `"this is a string\n"`

Typ: `char`-Array angemessener Länge.

Ein String endet mit `'\0'`:
$\Longrightarrow$ `sizeof("hallo") == 6`

(im Ggs. zu `strlen("hallo") == 5`)

Übrigens:

```
"this is not a string
 but a syntax error"
```

aber

```
char text[] = "So lassen sich auch richtig"
              "lange String-Literale angeben.";
```

4.2.5 Benannte Konstanten

Deklaration mit dem Schlüsselwort `const`:

```
const float betrag = 3.60;
```

Eine Konstante darf nicht verändert werden.
$\Longrightarrow$ Sie **muß** initialisert werden.

Zuweisung der Adresse einer Konstanten an einen Zeiger ist verboten:

```
float *fp = &betrag;   // FEHLER!
```

Aber man kann Zeiger auf `const`-Objekte definieren:

```
// Zeiger auf Konstantes Float:
const float *fp = &betrag;
*fp = 0.0;                 // FEHLER!
float xx;
fp = &xx;                  // OK
```

Ein konstanter Zeiger auf ein `float` sieht so aus:

```
float *const fp = &xx;
fp = 0;                    // FEHLER!
*fp += 1.5;                // OK
```

Aufzählungen:

```
enum {rund, eckig, platt};
```

entspricht

```
const int rund =  0;
const int eckig = 1;
const int platt = 2;
```

Eine Aufzählung kann auch benannt werden:

```
enum see {pader, lippe, diemel, nord};
```

`see` ist ein neuer Typ. Implizite Umwandlung nach `int` ist definiert, aber nicht umgekehrt.

Die Werte von Aufzählungselementen können auch angegeben werden:

```
enum iomode { out=100, in=2, fast_in,
              inout=0x1f, fast_inout=not_impl}
```

Nicht angegebene Werte sind um 1 größer als ihr Vorgänger, falls sie einen haben, sonst 0;

Aufzählungen können auch lokal in Klassen definiert werden.

4.3 Abgeleitete Typen

Aus Grundtypen und Benutzer-definierten Typen können mit den **Deklarationsoperatoren** neue Typen abgeleitet werden:

`*`	Zeiger
`[]`	Array
`&`	Referenz
`()`	Funktion

Die Deklaration soll dabei die spätere Anwendung widerspiegeln:

```
int a[10];              i = a[3];
int *ptr;               i = *ptr;
int f(int);             i = f(99);
```

Verwirrung entsteht oft dadurch, daß

`*` und `&`	Präfix-Operatoren und
`[]` und `()`	Postfix-Operatoren

sind und unterschiedliche Präzedenzen haben:

```
int *v[10];    // Array aus 10 int-Zeigern
int (*v)[10];  // Zeiger auf Array aus 10 ints
```

`Die meisten Menschen merken sich einfach die Gestalt der häufigsten Typen.`

B. Stroustrup

4.3.1 Der Typ `void`

Syntaktisch: wie ein Grundtyp.

Aber: `void`-Objekte gibt es nicht!

Zweck:

- Funktionen ohne Ergebnis: `void f();`
- Pointer auf Unbekanntes: `void *vp;`

Einer Variablen vom Typ `void*` können beliebige Zeiger zugewiesen werden. Sie darf jedoch nicht dereferenziert werden.
Zum Zugriff muß die **explizite** Typumwandlung benutzt werden:

```
void *f(unsigned int);
...
int *iptr = (int*) f(10);
```

4.3.2 Zeiger-Typen

`Für die meisten Typen T ist T* ein Zeiger auf T.`

B. Stroustrup

Ein `T*` kann die Adresse eines Objekts vom Typ `T` enthalten.
Für Arrays und Funktionen ist die Notation komplizierter:

```
int *p;          // Zeiger auf int
char **cpp;      // Zeiger auf char-Zeiger
int (*vp)[7];    // Zeiger auf ein int-Array
int (*fp)(t)     // Zeiger auf Fkt. "int f(t)"
```

Die elementare Operation auf Zeigern ist die **Dereferenzierung** mit dem Operator `*`:

```
char c = 'P';
char *cp = &c;     // Addresse von c
char c2 = *cp;     // c2 = 'P'
class date
{public: long julian;} *dp;
cout << (*dp).julian;   // Klammern!
cout << dp->julian;     // dto. aber schoener!
```

Arithmetik mit Zeigern

Erlaubt sind

- Zeiger + ganze Zahl
- Zeiger - ganze Zahl
- Zeiger - Zeiger

Voraussetzung: Der Zeiger zeigt auf ein "Datenobjekt", d.h. nicht auf `void` oder eine Funktion.

```
int strlen(char *p)
{ int l=0;
  while (*p++) l++;
  return l;
}
```

oder

```
int strlen(char *p)
{ char *pos = p;
  while (*pos++);
  return (pos-p)+1;
}
```

aber nicht

```
// Ein Zeiger auf "strlen":
int (*fp)(char*) = strlen;
fp++;          // VERBOTEN!
```

4.3.3 Arrays

Für einen Typ `T` ist `T[size]` ein Array aus `size` Elementen vom Typ `T`, indiziert von `0` bis `size-1`:

```
float v[100];     // 100 floats
int d[24][80];    // 24 80-elementige int-Arrays
char *texte[50];  // 50 char-Zeiger
date urlaub[29];  // 29 Urlaubstage
```

Die Länge eines Arrays muß nicht angegeben werden, wenn ein **Initialisierer** spezifiziert ist:

```
int v1[] = {1, 2, 3, 4};        // 4 Elemente
char c[] = {'a', 'b', 'c'};     // 3 Elemente
```

Sonderform für `char`-Arrays:

```
char c[] = "abc";   // 4 Elemente !!!
```

Vorsicht: Eine entsprechende Zuweisung ist nicht definiert:

```
char c[4];
c = "abc";    // FEHLER
```

Ist der Initialisierer zu kurz, wird mit Nullen von entsprechendem Typ aufgefüllt:

```
int a[5] = {1,2,3}; // {1,2,3,0,0}
char c[5] = "abc"   // {'a','b','c','\0','\0'}
float y[4][2] = {{1,2},    // |1,2|
                 {3,4},    // |3,4|
                 {5,6}};   // |5,6|
                           // |0,0|
```

4.3.4 Zeiger und Arrays

Der Name eines Arrays kann als Zeiger auf das erste Element verwendet werden.

```
char text[] = "hallo";
char *p = text;
```

ist dasselbe wie

```
char text[] = "hallo";
char *p = &text[0];
```

Wird oft in Funktionsaufrufen ausgenutzt:

```
char text[] = "C++ is Fun";
int length = strlen(text);
```

übergeben wird der Zeiger auf `text[0]`
$\Longrightarrow$ Wert-Parameter-Übergabe (*call-by-value*) mit Arrays geht nicht!

Arithmetik auf Zeigern hängt vom Typ ab, auf den gezeigt wird: Wenn `p` vom Typ `T*` auf ein Element eines `T`-Arrays zeigt, liefert `p+1` die Adresse des nächsten Elementes.

```
int main()
{
    double d[2];
    double* dp1 = &d[0];
    double* dp2 = &d[1];

    cout << "Zeigerdifferenz: " << dp2 - dp1;
    cout << " Adressendifferenz: "
         << long(dp2) - long(dp1);
}
}
```

liefert (Sun4)

```
Zeigerdifferenz: 1 Adressendifferenz: 8
```

Das Ergebnis der Pointer-Arithmetik ist undefiniert:

- Wenn bei Zeiger/Int-Arithmetik das Array verlassen wird.
- Wenn bei Zeiger/Zeiger-Subtraktion die Zeiger in verschiedene Arrays zeigen.

4.4 Referenzen

Eine Referenz ist ein alternativer Name für ein Objekt:

```
int i = 42;
int& r = i; // r und i stehen fuer Dasselbe.
int x = r;  // x = 42;
r = r - 1;  // i = 41;
```

Der Wert einer Referenz kann nicht verändert werden
$\Longrightarrow$ eine Referenz **muß** initialisiert werden.
Der Initialisierer einer Referenz muß ein L-Wert (ein zuweisbares Objekt) sein:

```
double &r = 3.14;  // FALSCH: kein L-Wert
```

Ausnahme: Referenzen auf Konstante:

```
const double &r = 3.14;  // OK.
```

entspricht der Sequenz

```
double temp = 3.14;
double &r = temp;
```

Referenzen als Parameter: Call-by-Reference

Funktionen, die ihre Parameter verändern:

```
void inc(int& i)
{ i++;}
int main()
{ int x = 1;
  inc(x);   // x = 2;
}
```

Zeitersparnis bei der Parameterübergabe:

```
class bank
{ kunden viele[10000];
  ...
};
void pruefe(bank b);
```

Diese Parameterübergabe benutzt den Copy-Konstruktor.
Besser:

```
// ein Referenzargument,
// das nicht veraendert wird:
void pruefe(const bank &b);
```

Referenzen als Funktionsergebnis

Ebenfalls zur Vermeidung von Kopien:

```
bank& zahlezinsen()
{ ...
}
```

Vorsicht: Solche Funktionen können links von Wertzuweisungen stehen:

```
int& index(int *ip, int i)
{ return ip[i];
}
int a[] = {1,2,3};
int main()
{ index(a,1) = 17; // a = {1,17,3};
}
```

Wenn dies nicht beabsichtigt ist:

```
const int& index(int *ip, int i)
{ return ip[i];
}
```

Vorsicht: **niemals** Referenzen auf lokale Objekte zurückgeben:

```
string& anhaengen(string& s1, string& s2)
{ string tmp = s1;
  tmp += s2;
  return tmp;
}
```

Solche Objekte sind nach Verlassen der Funktion nicht mehr da!

4.4.1 Funktionen

Eine **Funktionsdeklaration** enthält

- den Namen
- den Ergebnistyp
- die Anzahl und Typen der Parameter

```
int strlen(char *);
void printdate(date);
void stop(void);
```

Als Hilfe für den Leser darf die Funktionsdeklaration Parameternamen enthalten:

```
int inset(myset menge, int suchmich);
```

Jede verwendete Funktion muß irgendwo (genau einmal) definiert werden. Eine **Funktions-Definition** ist eine Funktions-Deklaration mit Rumpf:

```
void swap(int*, int*);
...
void swap(int *p, int *q)
{ int tmp = *p;
  *p = *q;
  *q = tmp;
}
```

Alle Parameter, die im Rumpf benutzt werden, müssen einen Namen haben. $\Longrightarrow$ Alle die nicht benutzt werden, brauchen keinen Namen.

Inline-Funktionen

Wie Elementfunktionen und Konstruktoren dürfen Funktionen auch `inline` definiert werden:

```
inline void printexpr(int val)
{ cout << "(" << val << ")";
```

`Inline` ist eleganter und **sicherer** als cpp-Makros:

```
#define mul(x,y) (x*y)
...
cout << mul(6+1,4)    // SURPRISE!
```

Parameterübergabe

Beim Aufruf werden die formalen Wert-Parameter mit den aktuellen Parametern **initialisiert** ($\Rightarrow$ Copy-Konstruktor). Die Typen werden abgeglichen durch standardmäßige und Benutzer-definierte Typumwandlungen.
Die formalen Referenz-Parameter werden zu Referenzen auf die aktuellen Parameter.

Call-by-Value $\Rightarrow\Leftarrow$ Call-by-Reference

```
void f(int i, int& k)
{ i++;  // veraendert lokales i
  k++;  // veraendert den akt. Parameter
}
```

Call-by-Reference wegen der möglichen Seiteneffekte vermeiden. Wenn wegen Effizienz eingesetzt: verwende `const`-Argumente:

```
void f(const large& arg)
{ ...
}
```

Referenzübergabe für

- Literale
- Konstanten
- Argumente, die Typumwandlung benötigen

ist nur an formale Parameter der Art `const T&` möglich:

```
float sqrt(const float&);  // Wurzel
int main()
{ float r;
  double d;
  r = sqrt(2.0f); // Ref. auf temp=2.0f
  r = sqrt(r);    // Ref. auf r
  r = sqrt(d);    // Ref. auf temp=(float)d
```

Funktionsergebnisse

Funktionen, die nicht als `void` deklariert sind, **müssen** ein Funktionsergebnis liefern.

```
int fac(int n)
{ if (n==0)
    return 1;
    else return n*fac(n-1);
}
```

Die Semantik der Ergebnisrückgabe entspricht (wie die der Argumentübergabe) der der **Initialisierung**.
Der Typ des `return`-Ausdrucks und der Typ der Funktion werden abgeglichen durch standardmäßige und Benutzer-definierte Typumwandlungen.

Default-Argumente

Funktionen und Konstruktoren dürfen Default-Argumente haben, die beim Aufruf weggelassen werden können:

```
void printdate(date d, int kurz=1);
{ cout << d.day << ".";
  if (kurz)
     cout << d.month << ".";
  else cout << monthname(d.month) << ".";
  cout << d.year << ".";
}
...
printdate(date(17.5.1993),1); // kurz
printdate(date(17.5.1993));   // dasselbe
printdate(date(17.5.1993),0); // lang
```

Default-Argumente sind nur für die "hinteren" Funktionsparameter möglich.

```
int print(int basis = 10, int value); // FEHLER!
```

Vorsicht vor Syntaxfehlern:

```
void sonicht(char*=0);  // Syntax-Fehler!
             // *= ist Zuweisungsoperator
```

Beliebig viele Argumente

Wenn Anzahl und Typen der Argumente nicht exakt angegeben werden können, werden Auslassungspunkte `...` als letztes Element der Parameterliste benutzt:

```
int printf(const char*, ...);
```

Die Funktion muß Informationen über Anzahl und Art der Parameter erhalten. Im Falle von `printf` geschieht dies über den 1. Parameter (*Format-String*):

```
printf("Hello Number %d\n",7);
```

Der Compiler kann keine Typen prüfen und anpassen (`char` und `short` werden zu `int`, `float` wird zu `double`).
In `<stdarg.h>` stehen Zugriffsmakros für nicht-spezifizierte Parameter zur Verfügung.
Ein gut entworfenes Progamm sollte solche Funktionen jedoch möglichst nicht benutzen.

4.4.2 Zeiger auf Funktionen

Es gibt zwei zulässige Operationen auf Funktionen:

- Aufrufen: `()`
- Adresse bestimmen: `&`

Der aus der Adressbestimmung resultierende Zeiger kann zum Aufruf der Funktion benutzt werden:

```
void error(char *p);
void (*fuptr)(char*) = &error;
...
(*fuptr)("Du Pfeife"); // ruft error() auf
fuptr("Selber"); // dto. mit impl. Konvertierung
```

Die Klammern sind **nötig**, da `()` höhere Priorität hat als `*`.

Die Deklaration des Zeigers enthält Argumentliste und Ergebnistyp. Diese müssen mit der Signatur der Funktion exakt übereinstimmen:

```
void (*fuptr)(char*);
int f1(char*);
void f2(int*);
...
fuptr = &f1;   // FEHLER: Ergebnistyp
fuptr = &f2    // FEHLER: Parametertyp
```

4.5 Sonstiges

4.5.1 Bitfelder

Möglichkeit für die low-level-Programmierung zur Nachbildung von Statusflags, Device-Registern u. ä.
Erlaubt innerhalb von Klassen:

```
struct statreg
{ unsigned int enable : 1;
  unsigned int page : 3;
  unsigned int : 1;        // frei
  unsigned int mode : 2;
  unsigned int : 4;        // frei
  unsigned int access : 1;
  unsigned int length : 1;
  unsigned int non_resident : 1;
} *sr;
sr->access = 0;
```

Als Typ erlaubt: int und unsigned int.
Mischung mit normalen struct-Komponenten ist möglich.
Verboten: Arrays, Adressbildung, Feldgrössen >= sizeof(int).

4.6 Wichtige Unterschiede zu ANSI C

- In C++ **muß** eine Funktion vor der Benutzung deklariert worden sein.
- In C++ **muß** die Signatur von Funktionen vollständig angegeben werden (*Prototype*).
- `const`-Variablen **müssen** in C++ initialisiert werden.
- In C++ **muß** ein `void`-Zeiger vor Zuweisung an einen anderen Zeiger explizit konvertiert werden.
- `char`-Arrays, die mit Strings initialisiert werden, müssen in C++ groß genug für die abschliessende Null sein.
- In C++ ist `sizeof(char)` = 1 und die Größe eines Aufzählungstyps nicht unbedingt = `sizeof(int)`.
- In C++ können Aufzählungen lokal definiert werden.
- In C++ können Deklarationen und Anweisungen gemischt werden.
- In C++ definiert eine Klassendefinition den Namen der Klasse als Typnamen.
- C++ erlaubt Referenztypen.
- C++ erlaubt `inline`-Funktionen.
- C++ erlaubt überladene Funktionen.

5 Anweisungen und Ausdrücke

5.1 Anweisungen

```
statement:
     labeled-statement
     expression-statement
     compound-statement
     selection-statement
     iteration-statement
     jump-statement
     declaration-statement
```

Hauptunterschied zu C ist das `declaration-statement`.

5.1.1 Benannte Anweisung (Labeled Statement)

```
int main()
{ int ende=9999;
  goto ende;
  cout << "hallo\n";
  ende:  cout << "ende " << ende << endl;
}
```

Der Gültigkeitsbereich eines Labels ist die Funktion, in der es definiert ist. Es kann nur in `goto`-Anweisungen benutzt werden (auch vor seiner Definition). Label-Namen kollidieren nicht mit anderen Bezeichnern.

5.1.2 Ausdrucksanweisung (Expression Statement)

```
expression-statement:
    expression ';' |
    ';' .
```

Wertzuweisung und Funktionsaufruf sind Ausdrucksanweisungen (siehe “Ausdrücke”). Die leere Anweisung ist nützlich für leere Schleifenrümpfe und Labels am Blockende.

5.1.3 Blöcke (Compound Statements)

```
compound-statement:
    '{' statement-list '}' |
    '{' '}'.
```

Dient dazu, mehrere Anweisungen dort zu verwenden, wo eine Anweisung erlaubt ist. Besitzt eigenen Gültigkeitsbereich.

5.1.4 Auswahlanweisung (Selection Statement)

```
selection-statement:
   'if' '(' expression ')' statement |
   'if' '(' expression ')' statement
        'else' statement |
   'switch' '(' expression ')' statement .
```

Die `if`-Anweisung

Erlaubt als `expression` sind:

- Arithmetische Typen
- Zeigertypen
- Klassentypen, für die eine Typumwandlung nach Arithmetik- oder Zeigertyp definiert ist.

Die `switch`-Anweisung

Erlaubt als `expression` sind:

- Ganzzahlige Typen
- Klassentypen, für die eine Typumwandlung in einen ganzzahligen Typ definiert ist.

Anweisungen im Rumpf dürfen markiert sein mit ein oder mehreren Labeln der Form:

```
case constant-expression:
```

Alle `constant-expressions` haben verschiedene ganzahlige Werte. Es darf ein Label geben der Form:

```
default:
```

Es wird angesprungen, wenn keine der anderen Fallmarken zutrifft.

5.1.5 Schleifen (Iteration Statements)

```
iteration_statement:
    'while' '(' expression ')' statement |
    'do' statement 'while '(' expression ')' |
    'for' '(' init-stat ';'
              expropt ';'
              expropt ')' statement .
init-stat:
    expression-statement |
    declaration-statement .
expropt :
    expression |
    .
```

Die `while`-Schleife

Erlaubt als `expression` sind:

- Arithmetische Typen
- Zeigertypen
- Klassentypen, für die eine Typumwandlung nach Arithmetik- oder Zeigertyp definiert ist.

Der Test findet statt vor der Ausführung des Rumpfes.

Die `do`-Schleife

Erlaubt als `expression` sind:

- Arithmetische Typen
- Zeigertypen
- Klassentypen, für die eine Typumwandlung nach Arithmetik- oder Zeigertyp definiert ist.

Der Test findet statt nach der Ausführung des Rumpfes.

```
int zaehler =0;
char c;
do { cin >> c;
     zaehler++;
   } while (c != '\n');
```

Die `for`-Schleife

Erlaubt als erste `expression` sind:

- Arithmetische Typen
- Zeigertypen
- Klassentypen, für die eine Typumwandlung nach Arithmetik- oder Zeigertyp definiert ist.

Der Test (erste `expression`) findet statt vor der Ausführung des Rumpfes. Danach wird die zweite `expression` ausgewertet.
Die Initialisierungsanweisung kann eine Deklarationsanweisung sein:

```
for (int i = 1; i <= MAX; i++)
    ...
```

Der Gültigkeitsbereich so definierter Objekte ist auf die for-Schleife beschränkt.

```
int main()  // Schreibe "01230"
{ int i = 0;
  for (int i = 0 ; i < 4 ; i ++)
    cout << i << endl;
  cout << i << endl;
}
```

5.1.6 Unbedingte Sprünge (Jump Statements)

```
jump-statement:
    'break' ';' |
    'continue' ';' |
    'return' expropt ';' |
    'goto' identifier .
```

Wenn Sprünge einen Gültigkeitsbereich verlassen, werden für alle dort konstruierten Objekte (deklarierte und temporäre) die Destruktoren aufgerufen.

Die `break`-Anweisung

erlaubt in `switch` und Schleifen. Verläßt kleinsten umschliessenden `switch` oder Schleife.

Die `continue`-Anweisung

nur erlaubt in Schleifen. Verzweigt zum Ende des Rumpfes der kleinsten umschliessenden Schleife.

Die `return`-Anweisung

Rückkehr aus einer Funktion.
`return` ohne Ausdruck erlaubt in:

- Funktion mit Ergebnistyp `void`.
- Konstruktoren
- Destruktoren

Sonst ist ein Return-Ausdruck erforderlich. Erreichen des Funktionsendes ohne `return` ist äquivalent zur Ausführung einer `return`-Anweisung ohne Ausdruck und daher in Funktionen mit Ergebnistyp verboten.

Die `goto`-Anweisung

Unbedingter Sprung zur mit dem angegebenen Label markierten Anweisung. Nur innerhalb einer Funktion.

5.1.7 Deklarations-Anweisung (Declaration Statement)

```
declaration-statement:
    declaration.
```

Deklarationen können beliebig im Anweisungsteil plaziert werden. Verdeckt eventuelle Deklarationen gleichen Namens aus äußeren Blöcken für den Rest des Blockes, in der sie steht.

```
int main() // gib "(i,j)=(2,1)" aus
{ int i = 1;
  { int j = i;
    int i = 2;
    cout << "(i,j)=(" << i << "," << j << ")\n";
  }
}
```

Im Gegensatz zu C ist es in C++ nicht erlaubt, über eine Deklarationsanweisung mit Initialisierung zu springen, es sei denn sie steht in einem Block, der als Ganzes übersprungen wird:

```
    ...
    goto lx;      // FEHLER: überspringt Dekl.
    ...
ly: X a = 0;      // Konstruktion von a
    ...
lx: goto ly;      // OK: Destruktion von a
```

5.2 Ausdrücke

Ein Ausdruck ist eine Folge von **Operatoren** und **Operanden**. Er kann einen Wert liefern und Seiteneffekte verursachen.
Im Allgemeinen ist die Reihenfolge, in der Operanden ausgewertet werden, undefiniert:

```
k = i + f(i++);  // Wert von k undef.
i = v[i++];      // Wert von i undef.
```

Die Bedeutung eines Ausdrucks ergibt sich aus den Regeln für

- die Präzedenz von Operatoren
- die Assoziativität von Operatoren

`*f(i)` bedeutet `*(f(i))`, da `()` höhere Präzendenz hat als `*`. Es ist also die Dereferenzierung eines Funktionsergebnisses.

`c=x!=0` bedeutet `c = (x != 0)`, da die Vergleichsoperationen höhere Präzedenz haben als die Wertzuweisung.

`a=b=c=0` heißt `a = (b = (c = 0))`, da die Wertzuweisung rechtsassoziativ ist.

`a-b-c` bedeutet `(a - b) - c`, da die Subtraktion linksassoziativ ist.

5.2.1 Übersicht über die Operatoren

<table>
<tr><th>Op</th><th>Bedeutung</th><th>Typ</th><th>Assoz.</th><th>Präzedenz</th></tr>
<tr><td>,</td><td>Komma-Operator</td><td>bin</td><td>links</td><td>1</td></tr>
<tr><td>=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>+=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>-=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>*=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>/=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>|=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>^=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>&=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>%=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td><<=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>>>=</td><td>Wertzuweisung</td><td>bin</td><td>rechts</td><td>2</td></tr>
<tr><td>? :</td><td>Bedingung</td><td>tern</td><td>rechts</td><td>3</td></tr>
<tr><td>||</td><td>Log. Oder</td><td>bin</td><td>links</td><td>4</td></tr>
<tr><td>&&</td><td>Log. Und</td><td>bin</td><td>links</td><td>5</td></tr>
<tr><td>|</td><td>Bitweises Oder</td><td>bin</td><td>links</td><td>6</td></tr>
<tr><td>^</td><td>Bitweises XOR</td><td>bin</td><td>links</td><td>7</td></tr>
<tr><td>&</td><td>Bitweises Und</td><td>bin</td><td>links</td><td>8</td></tr>
<tr><td>==</td><td>Gleichheit</td><td>bin</td><td>links</td><td>9</td></tr>
<tr><td>!=</td><td>Ungleichheit</td><td>bin</td><td>links</td><td>9</td></tr>
<tr><td><</td><td>Kleiner</td><td>bin</td><td>links</td><td>10</td></tr>
<tr><td><=</td><td>Kleiner gleich</td><td>bin</td><td>links</td><td>10</td></tr>
<tr><td>></td><td>Größer</td><td>bin</td><td>links</td><td>10</td></tr>
<tr><td>>=</td><td>Größer gleich</td><td>bin</td><td>links</td><td>10</td></tr>
<tr><td><<</td><td>Links-Shift</td><td>bin</td><td>links</td><td>11</td></tr>
<tr><td>>></td><td>Rechts-Shift</td><td>bin</td><td>links</td><td>11</td></tr>
</table>

Op	Bedeutung	Typ	Assoz.	Präzedenz
+	Addition	bin	links	12
-	Subtraktion	bin	links	12
*	Multiplikation	bin	links	13
/	Division	bin	links	13
%	Modulo	bin	links	13
->*	Member-Zeiger	bin	links	14
.*	Member-Zeiger	bin	links	14
++	Pre-Inkrement	un	rechts	15
--	Pre-Dekrement	un	rechts	15
*	Dereferenzierung	un	rechts	15
&	Adresse	un	rechts	15
+	Unäres Plus	un	rechts	15
-	Unäres Minus	un	rechts	15
!	Logisches Not	un	rechts	15
~	Bitweises Not	un	rechts	15
++	Post-Inkrement	un	rechts	15
--	Post-Dekrement	un	rechts	15
(type)	Cast	bin	rechts	15
new	Allokation	un/bin	rechts	15
delete	Deallokation	un/bin	rechts	15
[]	Indizierung	bin	links	16
()	Funktionsaufruf	bin	links	16
()	funktionaler Cast	bin	links	16
.	Selektion	bin	links	16
->	Zeigerselektion	bin	links	16
sizeof	Größe	un	rechts	16
::	Scope Resolution	un/bin	links	17

5.2.2 Einige ausgewählte Operatoren

:: Scope Resolution

Es gibt vier Arten von Gültigkeitsbereichen:

- Local: Namen, die in Blöcken deklariert sind und formale Funktionsparameter.
- Function: Labels gelten innerhalb der Funktion, in der sie deklariert sind.
- Class: Namen von Klassenelementen.
- File: Namen, die außerhalb aller Blöcke und Klassen definiert sind.

Der binäre :: Operator erlaubt Zugriff auf verdeckte Klassenelemente:

```
class X
{ public:
    static int xval;
    int getval();
};
X::xval = 0;
int X::getval() {return xval;}
```

Der unäre `::` Operator erlaubt Zugriff auf Namen im File-Gültigkeitsbereich, selbst wenn sie verdeckt sind:

```
int g = 0;
int f(int g)
{ if (g)              // lokales g
    return g;
  else return ::g;  // globales g
}
```

`sizeof`: Größenbestimmung

liefert die Größe in Bytes seines Operanden. Der Operand ist:

- ein Ausdruck (wird nicht ausgewertet)
- ein geklammerter Typname.

`sizeof` ist nicht anwendbar auf Funktionen, Bitfelder, `void` und Arrays ohne Dimensionsangabe.
Auf eine Referenz angewendet, liefert `sizeof` die Größe des Objekts, auf das die Referenz verweist:

```
class X
{ ...
};
X obj;
X& ref = obj;
const int i = sizeof ref; // == sizeof(X)
```

Für Arrays liefert `sizeof` die Gesamtzahl der Bytes im Array:

```
int arr[100];
const int i = sizeof arr;
```

Beispiel:

```
int i=99;
double d = 2.0;

int main()
{ cout << "int: " << sizeof(int) << endl;
  cout << "i: " << sizeof i << endl;
  cout << "i++: " << sizeof i++ << endl;
  cout << "d=d+i: " << sizeof(d=d+i) << endl;
  cout << "wert(i): " << i << endl;
  cout << "wert(d): " << d << endl;
  return 0;
}
```

liefert

```
int: 4
i: 4
i++: 4
d=d+i: 8
wert(i): 99
wert(d): 2
```

`new`: Speicherallokation

Der Operator `new` bewirkt

1. Allokation freien Speicherplatzes für das Objekt.
2. Initialisierung des Objektes.
3. Rückgabe eines geeigneten Zeigers auf das Objekt.

```
date *d;
d = new date;
```

ruft den Default-Konstruktor von `date` auf.

```
date *d;
d = new date(5,6,1993);
```

ruft den `date(d,m,y)`-Konstruktor auf.

Arrays werden wie folgt allokiert:

```
char *p = new char[n];
date *urlaub = new date[29];
```

Dabei wird der Default-Konstruktor für jedes Array-Element aufgerufen. Dynamisch allokierte Arrays können nicht explizit initialisert werden!

```
class elem
{ int i;
public:
 elem():       i(0) { cout <<i<<' ';}
 elem(int v): i(v) {cout <<i<<' ';}
 ~elem()            { cout <<'~'<<i<<' ';}
};

elem e1;
elem e2(1);
elem e3[2];
elem e4[2] = {elem(2), elem(3)};

elem *p1 = new elem;
elem *p2 = new elem(4);
elem *p3 = new elem[2];
// Initialisierte Arrays gehn nicht!
```

erzeugt folgende Ausgabe:

```
0 1 0 0 2 3 0 4 0 0 ~3 ~2 ~0 ~0 ~1 ~0
```

Man sieht: Dynamisch allokierte Objekte werden nicht automatisch zerstört und deallokiert.

Die Initialisierung wird nur bei erfolgreicher Allokation (Rückgabewert von `new` ungleich 0) ausgeführt.

Das Ergebnis von `new` sollte immer überprüft werden.

```
elem *p = new elem[n];
if (!p)
{ cerr << "New failed (elem[n]) \n";
  exit 1;
}
```

Beispiel:

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

class vector
{   double *co;
    int dim;
  public:
    vector(int d);
    double& operator [] (int d);
};

vector::vector(int d) : dim(d)
{ co = new double[dim];
  if (!co)
  { cerr << "New failed\n";
    exit (1);
  }
}

double& vector::operator[] (int d)
{ if (d >= 0 && d < dim)
    return co[d];
  else { cerr << "index out of range\n";
         exit(1);
       }
}
```

`delete`: Speicher-Recycling

Der Operator `delete` bewirkt

1. Aufruf des Destruktors für das Objekt.
2. Freigabe des Speichers für das Objekt.

```
date *d = new date;
...
delete d;  // Zerstoeren und Freigeben
```

`delete` darf nur auf Zeiger angewandt werden, die durch `new` erzeugt wurden. Sonst ist das Verhalten undefiniert. `delete` auf den Nullpointer angewandt ist harmlos.

```
int i;
int *p = &i;
delete p;  // FEHLER !!!
p = new int[100];
p = p + 1;
delete p;  // FEHLER !!!
```

Arrays werden mit `delete []` recycled. Diese Form führt dazu, daß der Destruktor für jedes Element des Arrays aufgerufen wird:

```
class elem
{ int i;
public:
  elem() : i(0)       {cout <<i<<' ';}
  ~elem()             {cout <<'~'<<i<<' ';}
};

int main()
{ elem *p3 = new elem[6];
  delete [] p3;
}
```

liefert:

```
0 0 0 0 0 0 ~0 ~0 ~0 ~0 ~0 ~0
```

Beispiel:

```
// Ein dynamisch wachsender Keller
const int stksize = 5;
const int stkgrow = 5;

class stack
{
private:
  int *stkref;
  int *stkend;
  int *stkptr;
public:
  stack(int size = stksize);
  ~stack()                 {delete [] stkref;}
  int push(int elem);
  int pop()                { return *--stkptr;}
  int empty() const        { return stkref == stkptr;}
};
stack::stack(int size)
{ stkref = new int[size];
  if (!stkref)
    { cerr << "No more Memory\n";
      exit(2);
    }
  stkend = stkref+size;
  stkptr = stkref;
}
```

```
int stack::push(int elem)
{ if (stkptr >= stkend)
    { int *snew = new int[stkend-stkref+stkgrow];
      for (int *p = stkref; p < stkend; p++)
        snew[p-stkref] = *p;
      stkend = snew + (stkend-stkref)+stkgrow;
      stkptr = snew + (stkptr-stkref);
      delete [] stkref;
      stkref = snew;
    }
  return *stkptr++ = elem;
}
```

.* und ->*: Zeiger auf Klassenelemente

Funktionen können über Zeiger indirekt aufgerufen werden:

```
void print(char *s)
{ cout << s << endl;
}
void (*fp)(char*) = &print;
(*fp)("hallo");
```

Dies ist auch für Elementfunktionen möglich. Dazu muß der Klassennamen angegeben werden:

```
class X
{ ....
  public:
  void print (char*s);
};
void (X::*fp)(char*) = &X::print;
```

Zum indirekten Aufruf brauchen wir ein Objekt der Klasse X:

```
X a;
(a.*fp)("string1");     // Operator .*
X *b;
(b->*fp)("string2");    // Operator ->*
```

Zeiger auf `public` Datenelemente von Klassen sind auch möglich:

```
struct A
{ int a;
  int b;
};
int A::*iptr = &A::a;
A var;
var.*iptr = 12;
A *ptr;
A::*iptr = &A::b;
ptr->*iptr = 13;
```

6 Überladen von Funktionen und Operatoren

Überladen von Funktionen erspart das Erfinden neuer Funktionsnamen für Funktionen, die ähnliches tun:

```
int max(int a, int b);
int max(const int* ptr, int b);
int max(const intlist& l);
```

Überladen von Operatoren ermöglicht es, benutzerdefinierte Typen (fast) wie eingebaute Typen zu behandeln:

```
complex c1, c2;
...
c1 += c2;
cout << c1;
```

6.1 Überladen von Funktionen

Überladene Funktionen müssen sich in Anzahl und/oder Typ der Argumente unterscheiden. Das Funktionsergebnis ist **kein** Unterscheidungsmerkmal.
Folgendes sind **keine** überladenen Funktionen:

```
int max(int i, int k);
int max(int a, int b);
```

Das ist eine Redeklaration der gleichen Funktion (erlaubt).

```
int max(int, int);
unsigned int max(int, int);
```

Das ist eine Redeklaration der gleichen Funktion (verboten, da Ergebnistypen verschieden).

```
typedef char* string;
int suche(char*, string);
int suche(char*, char*);
```

Typedef führt keinen neuen Typ ein. Das ist also die Redeklaration der gleichen Funktion (erlaubt).

Zur **Identifikation** einer überladenen Funktion geht der Compiler wie folgt vor (für einen Parameter):

1. Nimm die Funktion, bei der der Argumenttyp genau paßt:

```
double sqrt(double);  // <--- diese
float sqrt(float);
cout << sqrt(7.777);
```

2. Wenn eine einzige Funktion paßt, nachdem `char`, `unsigned char`, `short` nach `int`, `unsigned short` entweder nach `int` oder nach `unsigned int` (falls `sizeof(short)` $<$ `sizeof(int)`) und `float` nach `double` umgewandelt wurde, oder triviale Umwandlungen wie `T` nach `T&` oder `T` nach `const T` angewendet wurden, nimm diese:

```
int inc(int);  // <--- diese
float inc(float);
cout << inc('x');
```

3. Wenn eine einzige Funktion nach den anderen Standard-Umwandlungen paßt, nimm diese:

```
int testptr(char *);
int testptr(void *);   // <--- diese
list *lp;
cout << testptr(lp);
```

4. Wenn eine einzige Funktion nach benutzerdefinierten Umwandlungen paßt, nimm diese:

```
class cmplx
{ cmplx(double);
  ...
};
double max(cmplx,cmplx); // <--- diese
cmplx max(char *list);
cout << max(2,3);
```

Gibt es mehrere Funktionsargumente, muß eine Funktion gefunden werden, die in einem Argument besser paßt als alle anderen und in den anderen Argumenten mindestens genauso gut.

Hier wird die Sache recht kompliziert $\Longrightarrow$

Man sollte Situationen vermeiden, wo man nicht mehr intuitiv weiß, was geschieht: Don't be too clever!

6.2 Überladen von Operatoren

Ziel der Einführung benutzerdefinierter überladener Operatoren war es, C++ erweiterbar zu machen.

Erweiterbar heißt **nicht** änderbar:

- Man kann keine neuen Operatoren erfinden.
- Man kann die Operatoren für Standardtypen nicht umdefinieren.
- Man kann auch deren Präzedenz. Assoziativität und Stelligkeit nicht ändern.

Alle Operatoren können überladen werden **außer**:

`.` hat schon eine vordefinierte Bedeutung für alle Klassen.

`.*` dto.

`::` beeinflußt Namensräume des Compilers.

`?:` erschien nicht lohnend.

`sizeof` muß vom Compiler statisch auswertbar sein.

Die Operatoren für eingebaute Typen dürfen nicht umdefiniert werden
$\Longrightarrow$ jeder benutzerdefinierte Operator muß mindestens ein Argument vom Typ Klasse (oder Referenz darauf) haben (oder als Methode definiert sein):

```
typedef char* string;
string operator+(string, string);
```

ist nicht erlaubt.

Operatoren können auf zwei Arten benutzt werden:

```
class date
{ ...
  date operator+(int);
};
date today(14,6,1993);
date tomorrow = today + 1; // Infix-Notation
date thedayaftertomorrow = today.operator+(2);
                           // Funktions-Notation
```

Identitäten, wie sie für Operatoren eingebauter Typen existieren:

```
++a   ≡  a+=1   ≡  a=a+1
v->m  ≡  (*v).m  ≡  v[0].m
```

müssen für selbstdefinierte Operatoren **nicht** gelten.

Es ist jedoch eine gute Idee, solche Äquivalenzen beizubehalten: *Prinzip der geringsten Überraschung.*

Benutzerdefinierte Operatoren sind entweder

- Methoden (Elementfunktionen) oder
- Funktionen (meistens Freunde der Klasse).

Einige Operatoren müssen jedoch Methoden sein:

`=` Wertzuweisung

`()` Funktionsaufruf

`[]` Indizierung

`->` Memberzugriff über Zeiger

Status	**Syntax**	**tatsächlicher Aufruf**
Methode	`Op X`	`X.operator Op()`
Methode	`X Op`	`X.operator Op()`
Methode	`X Op Y`	`X.operator Op(Y)`
Funktion	`Op X`	`operator Op(X)`
Funktion	`X Op`	`operator Op(X)`
Funktion	`X Op Y`	`operator Op(X,Y)`

Da sie auf diese Art nicht unterscheidbar wären, gibt es für Inkrement `++` und Dekrement `--` eine Sonderregel:

Funktion	`operator ++(T)`	Pre-Inkrement
Funktion	`operator ++(T,int)`	Post-Inkrement
Elementfunktion	`operator ++()`	Pre-Inkrement
Elementfunktion	`operator ++(int)`	Post-Inkrement

Analoges gilt für `--`.

```
class date
{   int jul;
  public:
    date(int i) :jul(i) {}
    date operator ++()  // Prefix
    { jul++;
      return *this;
    }
    date operator ++(int) // Postfix
    { date before= *this;
      jul++;
      return before;
    }
    void print()
    {  cout << jul << endl;
    }
}; // class date

int main()
{ date today(14141414);

  today++.print();   // schreibt 14141414
  ++today.print();   // ERROR: void operand for ++
  (++today).print(); // schreibt 14141416
}
```

6.2.1 Ein vollständiges Beispiel

Implementierung einer String-Klasse:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#define TEST(text) {clog << text << endl;}

void checkptr(void *p)
{ if (!p)    { cerr << "new() failed\n";
               exit(99);}
}

class string
{ char *s;
  int len;
public:
  string();
  string(char*);
  string(const string&);
  ~string() {delete [] s;}
  string& operator = (const string&);
  string& operator += (const string&);
  friend string operator + (const string&,
                            const string&);
  friend ostream& operator << (ostream&,
                               const string&);
  friend istream& operator >> (istream&, string&);
};
```

```
string::string() : len(0), s(0)
{ TEST("default constructor");
}
string::string(char *str)
{ TEST("constructor");
  s=new char[1 + (len=strlen(str))];
  checkptr(s);
  strcpy(s,str);
}
string::string(const string& ori)
{ TEST("copy constructor");
  s=new char[1+(len=ori.len)];
  checkptr(s);
  strcpy(s,ori.s);
}
string& string::operator = (const string& right)
{ TEST("operator =");
  if (this != &right)
  { delete [] s;
    s=new char[1+(len=right.len)];
    checkptr(s);
    strcpy(s,right.s);
  }
  return *this;
}
```

```
string& string::operator += (const string& right)
{ TEST("operator +=");
  char *news = new char[1+(len += right.len)];
  checkptr(news);
  strcpy(news,s);
  strcat(news,right.s);
  delete [] s;
  s = news;
  return *this;
}
string operator + (const string& left,
                   const string& right)
{ TEST("operator +");
  string result=left;
  result+=right;
  return result;
}
ostream& operator << (ostream& os,
                      const string& str)
{ TEST("operator <<");
  os << str.s;
  return os;
}
istream& operator >> (istream& is, string& str)
{ TEST("operator >>");
  char buf[128];
  is >> buf;
  str.s=new char[1+(str.len=strlen(buf))];
  checkptr(str.s);
  strcpy(str.s,buf);
  return is;
}
```

```
int main()
{ string anf("Peter"), ende;
  cin >> ende;
  ende = ende + ".";
  anf += " " + ende;
  cout << anf << endl;
}
```

erzeugt mit Eingabe “Hase”:

```
constructor
default constructor
operator >>
Hase
constructor
operator +
copy constructor
operator +=
copy constructor
operator =
constructor
operator +
copy constructor
operator +=
copy constructor
operator +=
operator <<
Peter Hase.
```

6.2.2 Überladen der Wertzuweisung

Der Wertzuweisungsoperator `=` hat eine Default-Definition (elementweises Kopieren) für alle Klassen. Er ist der einzige Operator, der nicht an Unterklassen vererbt wird (später).

```
class window
{   char data[4096];
  public:
    window()
    { for (int i=0; i < 4096; i++)
            data[i]=0;
    }
    operator = (const window &w)
    { if (this != &w)
          for (int i= 0 ; i < 4096; i++)
             data[i] = w.data[i];
    }
};
...
window shell1, shell2;
shell2 = shell2;
```

Die anderen Wertzuweisungsoperatoren (`+=`, `*=`, usw. haben keine vordefinierte Bedeutung. Sie sollten definiert werden, wenn die zugrundeliegenden Operatoren definiert werden (z.B. kein `+` ohne das dazugehörende `+=`).

6.2.3 Überladen der Indizierung

```
class string
{ ...
  char& operator [] (const int);
}

char& string::operator [](const int index)
{ if (index<0 || index >= len)
    { cerr << "string index out of range\n";
      exit(1);
    }
  return s[index];
}

string anf;
anf[0] = 'X';
```

Das zweite Argument (der Index), darf natürlich von beliebigem Typ sein (assoziative Arrays).

6.2.4 Überladen des Elementzugriffs

Der Operator `->` wird als unärer Postfix-Operator überladen, d.h. `x->m` wird interpretiert als

```
(x.operator ->())->m;
```

Das bedeutet, daß `->` entweder

- einen Zeiger auf eine Klasse oder
- ein Objekt einer Klasse, für die wiederum `->` definiert ist

zurückgeben muß.

Der Ablauf für `x->m` sieht wie folgt aus:

1. Wenn `x` ein Zeiger auf eine Klasse ist, wende den normalen Elementzugriff (`x->m`) an.

2. Wenn `x` ein Objekt einer Klasse ist, rufe den Operator `->` für diese Klasse auf. Wenn der nicht existiert $\Rightarrow$ Fehler.

Wende 1. und 2. rekursiv auf das Ergebnis an.

Anwendung: *Delegation* von Aufgaben an andere Klassen:

```
class mystring
{ private: char *s;
  public: mystring(char *s);
          char *getmystring() {return s;}
};

mystring::mystring(char *c)
{ s = new char[strlen(c)+1];
  strcpy(s,c);
}

class mystring_op
{ private: mystring *s;
  public: mystring* operator -> ();
};

mystring* mystring_op::operator ->()
{ if (!s)
    s = new mystring("undef");
  return s;
}

int main()
{ mystring_op a;
  cout << a->getmystring() << endl;
}
```

6.2.5 Überladen des Funktionsaufrufs

Der Funktionsaufrufsoperator `()` ist ein binärer Operator mit einer Ausdrucksliste als zweites Element (evtl. leer).

```
class string
{ ...
  string operator() (int position, int length);
}

string string::operator() (int pos, int length)
{ TEST("operator ()");
  if (pos <0 || pos > len)
    { cerr << "position out of range\n";
      exit(1);
    }
  string result;
  result.s = new char[1+(result.len=length)];
  for (int i=0; i<length && (pos+i)<len; i++)
    result.s[i] = s[pos+i];
  result.s[i]='\0';
  return result;
}
```

Anwendung: Iteratoren

```
class string
{ ...
  friend class string_iterator;
}
class string_iterator
{ int currelem;
  string *str;
public:
  string_iterator(string& whatstring)
  { str = &whatstring;
    currelem=0;
  }
  char operator () (void)
  { if (currelem == str->len)
        return 0;
    else return str->s[currelem++];
  }
};
int main()
{ string s("This is my string");
  string_iterator next(s);
  char c;
  while (c=next())
    cout << c;
}
```

6.2.6 Überladen von new und delete

Die Operatoren `new` und `delete` sind standardmäßig für alle Klassen definiert. Man kann sie überladen, wenn man eine spezielle Speicherverwaltung benötigt. Die Standardversion ist dann unter `::new` bzw. `::delete` erreichbar.

```
#include <stddef.h>
class string
{ ...
  static int newsize;
  void* operator new(size_t size)
  { newsize+=size;
    void *p =  ::new string;
    return p;
  }
};
int string::newsize = 0;
...
string* sp=new string("Pader");
cout << *sp << " size: "
     << string::newsize << endl;
```

schreibt

```
Pader size: 8
```

6.2.7 Warnung

Überladen von Operatoren scheint verlockend, da man insbesondere bei kleinen Beispielen hübsche Effekte erzielen kann. Man sollte es jedoch überlegt einsetzen:

- Programme können für andere unverständlich werden.

  Beispiel: Überladen des Operators `!` zur Ausgabe von Warnmeldungen:

  ```
  if (!success)
     !"computation failed";
  ```

- Die Präzedenz stimmt oft nicht, so daß Klammerung notwendig wird ⇒ Fehleranfällig.

  Beispiel: Überladen von `^` als Exponentiation.

  ```
  a ^ b + c
  ```

  ist nicht das, was man erwartet.

- Einführung von Operatoren für eine Klasse muß **vollständig** und **konsistent** sein. Das ist gar nicht einfach.

7 Schablonen

Schablonen erlauben es, C++ -Konstrukte zu parametrisieren, insbesondere mit **Typen**.
Schlagworte: Generizität, Parametrische Polymorphie

Es gibt zwei Arten von Schablonen:

Funktionsschablonen (*Function Templates*) ermöglichen parametrisierbare Funktionsdefinitionen (*generische Funktionen):*

```
template <class T> void myprint(T elem)
{ cout << elem << endl;
}
```

Klassenschablonen (*Class Templates*) bieten die Möglichkeit, Klassendefinitionen zu parametrisieren:

```
template <class T> class stack
{   int size;
    int top;
    T* data;
  public:
    stack(int s);
    void push(T elem);
    T pop();
}
```

7.1 Funktionsschablonen

```
template <class typ>           /* Deklaration */
typ min(typ a, typ b);
...
template <class argtype>       /* Definition */
argtype min(argtype a, argtype b)
{ return (a<b)?a:b;
}
```

Funktionsschablonen deklarieren/defininieren noch **keine** Funktionen. Dies geschieht erst, wenn im Programm zur Schablone passende Deklarationen oder Anwendungen auftreten:

```
int min(int, int);
int main()
{  double f = 99;
   // min(double, double)
   cout << min(8.7, f);
   int i;
   // min(int*, int*)
   cout << min(&i, (int*) &f);
}
```

Der Compiler hat die Aufgabe, für jede benötigte Ausprägung der Funktion Code zu erzeugen. In unserem Beispiel sind das zwei Versionen von `min()`.

CC generiert:

```
_min__Fdd      // double, double
_min__FPiPi    // int*, int*
```

Das Problem ist nun, daß Anwendungsstelle und Funktionsschablonendefinition i.A. nicht in der gleichen Datei liegen (getrennte Übersetzung).
Der Compiler kennt bei der Übersetzung der Schablonenanwendung deren Definition nicht und kann daher nicht den Code für benötigte Version erzeugen.
Dies geht erst zur Linkzeit, wenn alle Programmkomponenten zusammen kommen.

Zur Zeit erwarten viele Compiler, daß die Schablonendefinition an der Anwendungstelle verfügbar ist (z.B. über `#include`). Der Normentwurf (Dezember 1996) schlägt eine Kennzeichnung von Schablonen mit `export` vor. Exportierte Schablonen brauchen an der Anwendungsstelle nur noch deklariert zu werden. .

Für die Benutzung von Funktionsschablonen gelten folgende Bedingungen:

- Es gibt genau eine Schablonendefinition für eine Funktion
- Alle Schablonenparameter müssen in der Parameterliste der Funktion verwendet werden.
- Es gibt keine Typkonvertierungen bei der Zuordnung aktueller Parameter zu Schablonenparametern.

  Beispiel:

```
int i;
unsigned int j;
cout << min(i,j);     // Fehler: min braucht
                         2 gleiche Typen
```

- Als Schablonen-Parameter sind auch Übersetzungszeit-Konstanten erlaubt:

  Beispiel:

```
template <const X&x, int i>
void f() { ... }
```

- Weitere normale Funktionsparameter sind erlaubt:

  Beispiel:

```
template <class c1, class c2>
c1 min(c1 a, c2 b, char *text);
```

  Auch für diese Parameter: keine automatische Konvertierung.

- Der Ergebnistyp der Funktion trägt nicht zur Identifikation der Schablonenfunktion bei:

  Beispiel:

```
template <class c1>
c1 min(c1 a, c1 b);
int i,j;
float k;
i = min(i,j);   /* int-minimum */
k = min(i,j);   /* int-minimum */
```

- `extern`, `inline` und `static` sind möglich

```
template <class c>
inline int check(c arg);
```

- Überladen von Funktionsschablonen ist möglich. Die Definitionen unterscheiden sich dann in Zahl und/oder Typisierung der Argumente.

  <u>**Beispiel:**</u>

```
template <class c>
c fu (c arg1, double d);
template <class c>
c fu (double d, c arg1);
template <class c>
int fu (c arg1, double d);   // FEHLER
```

- Auch Überladen mit konkreten Funktionen ist möglich:

  <u>**Beispiel:**</u>

```
template <class c>
c min(c arg1, c arg2)
{ return (a<b)?a:b; }

// die Schablone ist fuer strings nicht gut:
// also:

char* min(char* a, char* b)
{ return (strcmp(a,b) < 0)? a : b;
}
```

Liegen Schablonen- und konkrete Versionen einer Funktion vor, entscheidet der Compiler wie folgt:

1. Existiert eine konkrete Funktion mit **exakt** passenden Parametern, nimm diese.
2. Sonst: Existiert eine Schablone mit exakt passenden Parametern, nimm diese.
3. Sonst: Wenn eine konkrete Version mit impliziten Typanpassungen in Frage kommt, nimm diese (Standardregel für überladene Funktionen)

Beispiel:

```
template <class c>
c fu(c,c);                  // Schablone
char* fu(char*, char*)      // konkrete Fkt.1
int fu(int, int)            // konkrete Fkt.2

fu("x","y");                // konkrete Fkt.1
fu(4,7);                    // konkrete Fkt.2
fu(3.4, 5.6);               // Schablone(double)
fu(4,6.0);                  // konrete Fkt. 2
```

7.2 Klassenschablonen

Ziel: Definition generischer Klassen

Beispiel:

```
template <class T, int size> class array
{ private : T inhalt[size];
  public:   array(T);
            void print();
            int getsize() {return size;}
};
template <class T, int size>
array<T,size>::array(T elemval)
{ for (int i = 0; i < size; i++)
    inhalt[i] = elemval;
}
template <class T, int size>
void array<T,size>::print()
{ for (int i = 0; i < size; i++)
    cout << inhalt[i] << " ";
}

int main()
{ array<char,10> carray('p');
  carray.print();
  array<double,7> darray(12.0);
  darray.print();
}
```

- Eine Deklaration einer Schablonenklasse hat die folgende Form:

```
template <class T, int size> class array;
```

- Die Klasse heißt

```
array<T, size>
```

  Das wird gebraucht z.B. in der Definition der Elementfunktionen:

```
// der einstellige Konstruktor
array<T,size>::array(T elemval)
```

- Elementfunktionen, Konstruktoren und Destruktoren, die außerhalb der Schablonenklassen-Definition geschrieben werden, sind als Schablonenfunktionen zu formulieren:

```
template <class T, int size>
void array<T,size>::print()
{ for (int i = 0; i < size; i++)
    cout << inhalt[i] << " ";
}
```

- Auch normale Funktionen, die Schablonenklassen als Parametertypen haben, werden zu Schablonenfunktionen:

```
template <class T, int size>
// z.Z. nur fuer g++
void printsize(array<T,size> a)
{ cout << a.getsize();
}
```

- Um Objekte und Funktionen mit Schablonenklassenparametern zu kreieren muß angegeben werden:
  - für Typparameter: der konkrete Typ
  - für andere Parameter: konstante Ausdrücke, deren Typ exakt mit der Schablone übereinstimmt.

```
void printsize(array<char,100> a)
{ cout << "noe";
}

int main()
{ array<double,7> darray(12.0);
  array<char,100> carray(' ');
  printsize(carray);
}
```

Ein vollständiges Beispiel

```
#include <iostream.h>

template <class T>      // declaration tset
class tset;             // (forward)

template <class T>      // definition elem
class elem
{ friend class tset<T>; // tset uses these linked
private:                // lists to represent sets
  T val;                // the element value
  elem* next;           // pointer to next list elem
public:
  elem() : val((T)0), next(0) {}   // default constr.
  elem(T v, elem* n);   // constructor
  ~elem();              // destructor
};

template <class T>   // definition tset
class tset
{ private: elem<T>* objects;   // element list
  public: tset();              // default constructor
          ~tset();             // destructor
          int in(T e);         // member test
          void add(T e);       // adds an element
          void printset(void); // output set
};
```

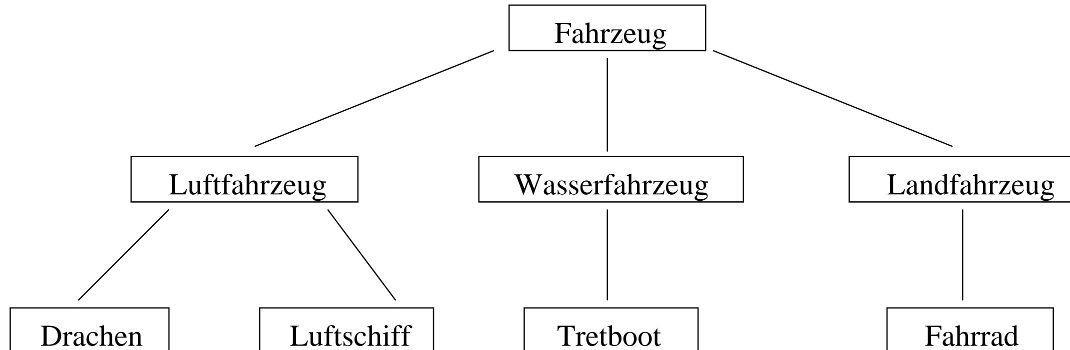
```
template <class T>  // print T objects
void print(T e)     // ordinary template function
{ cout << e;
}

template <class T>      // constructor
elem<T>::elem(T v, elem<T>* n)
: val(v), next(n)
{}

template <class T>   // destructor
elem<T>::~elem()
{ val = 0;
  next = 0;
}

template <class T>   // default constructor
tset<T>::tset() : objects(0)
{
}
```

```
template <class T>   // destructor
tset<T>::~tset()
{ elem<T> *optr;
  while (objects)
    { optr = objects;
      objects = objects->next;
      delete optr;
    }
}

template <class T>   // check if e is in set
int tset<T>::in(T e)
{ elem<T> *optr = objects;
  while (optr)
  { if (optr->val == e)
     return 1;
    optr=optr->next;
  }
  return 0;
}

template <class T>   // add element to set
void tset<T>::add(T e)
{ if (!in(e))
  {  elem<T> *eptr = new elem<T>(e, objects);
     objects=eptr;
  }
}
```

```
template <class T>   // print all set elements
void tset<T>::printset(void)
{ elem<T> *optr = objects;
  while (optr)
  { print(optr->val);
    cout << endl;
    optr=optr->next;
  }
}

int main()
{ tset<int> s;
  int i;
  while (cin >> i)   // read integers
    s.add(i);        // add them to set
  s.printset();      // print set
  cout << endl;
}
```

8 Klassen-Hierarchien und Vererbung

Konzepte der realen Welt können mit Klassen modelliert werden. Um Gemeinsamkeiten zwischen Konzepten auszudrücken, braucht man hierarchische Beziehungen zwischen Klassen:



Die Klasse "Wasserfahrzeug" ist von der **Basisklasse** "Fahrzeug" **abgeleitet**.
Basisklassen heißen auch **Elternklassen** oder **Superklassen**.
Abgeleitete Klassen heißen auch **Subklassen**.

Vorteile einer Klassenhierarchie

- Eine Klasse kann erweitert werden, ohne sie neu programmieren oder übersetzen zu müssen (hinzufügen von "Bus" ändert die Klasse "Landfahrzeug" nicht).
- Gemeinsamkeiten zwischen Klassen können formuliert und ausgenutzt werden (alle Fahrzeuge haben Preis, Gewicht und Höchstgeschwindigkeit).
- Verschiedene Klassen können eine gemeinsame Schnittstelle haben (z.B. "drucke_fahrzeugdaten").
- Datenrepräsentation kann besser strukturiert werden. Von allgemeinen Daten zu spezialisierten Eigenschaften.
- Unterstützung des modularen Programmierens (*Klassenbibliotheken*).
- Programmiereffizienz durch Wiederverwendung und Erweiterung vorhandener Klassen (*reusability, extensibility*).

8.1 Ein erstes Beispiel

Die Klasse `sigma` berechnet die laufende Summe der Zahlen, die übergeben werden:

```
class sigma
{ double sum;
public:
  sigma() : sum(0.0){}
  void input(double d) { sum += d;}
  double getsum()      { return sum;}
};

int main()
{ sigma x;
  x.input(1);
  x.input(12.5);
  cout << "The sum is "
       << x.getsum() << endl;
}
```

schreibt

```
The sum is 13.5
```

Nun soll jedoch der Durchschnitt der eingegebenen Zahlen ermittelt werden. Statt eine völlig neue Klasse zu definieren, kann `sigma` erweitert werden:

```
class mue : public sigma
{ int n;
public:
  mue() : n(0) {}
  void input(double d)
  { sigma::input(d); n++;
  }
  double getavg()
  { return n? getsum()/n : 0;}
  };
int main()
{ mue x;
  x.input(1);
  x.input(12.5);
  cout << "Sum " << x.getsum() << endl;
  cout << "Avg " << x.getavg() << endl;
}
```

schreibt

```
Sum 13.5
Avg 6.75
```

Beobachtungen:

- ```
  class mue : public sigma
  ```
  Spezifikation der Basisklasse: die `public` Elemente von `sigma` werden zu `public` Elementen von `mue`.

- ```
    mue() : n(0) {}
  ```
  Der Default-Konstruktor von `mue` ruft den Default-Konstruktor von `sigma` auf. Dies kann auch explizit gemacht werden:
  ```
  mue() : sigma(), n(0) {}
  ```

- ```
  void input(double d)
  { sigma::input(d); n++;
  }
  ```
  Die `input`-Funktion von `sigma` wird umdefiniert. Scope Resolution für den Zugriff auf das Original.

- ```
  double getavg()
  { return n? getsum()/n : 0;
  }
  ```
  `mue` erhält eine zusätzliche Elementfunktion `getavg()`. Die Funktion `getsum()` aus `sigma` wird übernommen.

8.2 Zugriff auf Basisklassen-Elemente

Klassenelemente sind `public` (default für `struct`), `protected` oder `private` (default für `class`):

```
class base
{   int privelem;    // privat
  protected:
    int protelem;    // protected
  public:
    int pubelem;     // public
  ...
};
```

Außerhalb der Klasse können die `public` Elemente benutzt werden:

```
void set(base *b)
{ b->pubelem=0;
}
```

Innerhalb der Klasse und von **Freunden** können alle Elemente benutzt werden:

```
int base::getpriv()
{ return privelem;
}
```

In abgeleiteten Klassen können die `protected` und die `public` Elemente benutzt werden (natürlich nicht die `private` Elemente):

```
class base
{private:   int privelem;   // privat
 protected: int protelem;   // protected
 public:    int pubelem;    // public
};
class derived : public base
{ private:  int derivedelem;
  public:   derived()
            { derivedelem=pubelem+protelem;}
};
```

Beispiel:

```
class name
{    char *vorname;
     char *nachname;
  protected:
     char *ganzername;
  public:
     name(char*, char*)   // Konstruktor
};
```

```
name::name(char* v, char* n)
{ vorname=new char[strlen(v)+1];
  nachname=new char[strlen(n)+1];
  strcpy(vorname,v); strcpy(nachname,n);
  int len = strlen(n) + strlen(v)+2;
  ganzername=new char[len];
  ostrstream os(ganzername,len);
  os << v << ' ' << n;
}

class address : public name
{    char *str;
     char *ort;
  public:
     address(char *v, char *n, char *s, char *o)
     : name(v,n)
     { str=new char[strlen(s)+1];
       ort=new char[strlen(o)+1];
       strcpy(str,s); strcpy(ort,o);
     }
     void printname()
     { cout << ganzername;
     }
};
```

8.3 Zugriff auf Basisklassen

Basisklassen werden `public` (default für `struct`), `protected` oder `private` (default für `class`) geerbt:

```
class deripub  : public base {...};
class deriprot : protected base {...};
class deripriv : private base {...};
```

Die Klasse `deripub` übernimmt die Zugriffsspezifikatoren von `base`, d.h. die `public` Elemente von `base` sind auch `public` von `deripub`, usw.

Nur innerhalb von `deriprot` und in davon abgeleiteten Klassen oder von Freunden kann auf die `public` und `protected` Elemente von `base` zugegriffen werden.

Nur innerhalb von `deripriv` oder von Freunden kann auf die `public` und `protected` Elemente von `base` zugegriffen werden.

8.4 Konstruktion und Destruktion

Wird ein Objekt einer abgeleiteten Klasse erzeugt, wird *implizit* auch (je) eines der Basisklasse(n) kreiert
$\Longrightarrow$ der Konstruktor der Basisklasse muß aufgerufen werden.

```
class uhr
{ int std;
  int min;
 public:
  uhr(int s, int m) : std(s), min(m) {};
  uhr() : std(0), min(0) {};
  void show() { cout << std << ':' << min;}
};

class wecker : public uhr
{ int alarm_std;
  int alarm_min;
 public:
  wecker() : alarm_std(0), alarm_min(0) {}
  wecker(int s, int m, int as, int am)
        : uhr(s,m),
          alarm_std(as),
          alarm_min(am) {}
  void show() { uhr::show();
                cout << " || "
                << alarm_std << ':'
                << alarm_min;
              }
};
```

Reihenfolge der Konstruktion

- Konstruktion des Basisklassenobjektes
- Konstruktion der Member
- Konstruktion des Objektes

Die Reihenfolge der Konstruktoraufrufe für Basisklassen bzw. Elemente bestimmt sich aus der **Reihenfolge der Deklaration** (nicht aus der Anordnung der Liste)!

Reihenfolge der Destruktion umgekehrt

Nicht erlaubt sind

- Konstruktoraufrufe für indirekte Basisklassen (außer bei virtuellen Basisklassen, siehe 8.7.1).
- Initialisierungen für geerbte Member.

Eine Klasse kann sowohl Basisklasse als auch Elementklasse sein:

```
// Besserer Wecker
class wecker : public uhr
{ uhr alarmzeit;
 public:
  wecker() : alarmzeit(0,0) {}
  wecker(int s, int m, int as, int am)
        : uhr(s,m),
          alarmzeit(as,am) {}
  void show() { uhr::show();
                cout << " || ";
                alarmzeit.show();
              }
};
```

8.4.1 Initialisierung durch Kopie

Wird ein Objekt einer abgeleiteten Klasse durch Kopie initialisiert

```
wecker w1(11,30,7,15);
wecker w2 = w1;
```

lassen sich vier Fälle unterscheiden:

1. Weder Basisklasse noch abgeleitete Klasse besitzt einen Copy-Konstruktor:

   – Basisklassenobjekt wird durch elementweise Kopie (der relevanten Daten) initialisiert.

   – Abgeleitetes Objekt wird durch elementweise Kopie initialisiert.

2. Die Basisklasse besitzt einen Copy-Konstruktor, die abgeleitete Klasse nicht:

   ```
   uhr(const uhr& ori) : std(ori.std),
                         min(ori.min) {}
   ```

   – Basisklassenobjekt wird durch den Copy-Konstruktor initialisiert.

   – Abgeleitetes Objekt durch elementweise Kopie.

3. Die abgeleitete Klasse besitzt einen Copy-Konstruktor, die Basisklasse nicht:

```
wecker(const wecker & ori)
        : uhr(ori),
          alarmzeit(ori.alarmzeit) {}
```

   – Basisklassenobjekt wird durch elementweise Kopie initialisiert. Allerdings **nicht automatisch**, sondern explizit im Copy-Konstruktor der abgeleiteten Klasse (Aufruf `uhr(ori)`). Fehlt der Aufruf, wird nur der Default-Konstruktor der Basisklasse benutzt.
   – Abgeleitetes Objekt wird durch den Copy-Konstruktor initialisiert.

4. Sowohl Basisklasse als auch abgeleitete Klasse besitzt einen Copy-Konstruktor:

   – Basisklassenobjekt wird durch Copy-Konstruktor initialisiert (explizit aufzurufen).
   – Abgeleitetes Objekt wird durch Copy-Konstruktor initialisiert.

8.5 Typanpassung

Ein Objekt einer abgeleiteten Klasse kann einem Objekt (einer) seiner Basisklasse(n) **ohne** explizite Typkonvertierung zugewiesen werden:

```
uhr tschibo;
wecker radio;
tschibo = radio;
// OK, jeder Wecker ist eine Uhr
```

Andersrum gilt dies nicht:

```
uhr kirchturm;
wecker hahn;
hahn = kirchturm;
// NEIN, eine Uhr ist i.a. kein Wecker.
```

Die gleichen Reglen gelten auch für Zeiger und Referenzen:

```
base b;
deriv d;
base *pb = &d;   // OK
base &rb = &d;   // OK
deriv *pd = &b;  // FEHLER
deriv &rd = &b;  // FEHLER
```

Will man auf Elemente der abgeleiteten Klasse über ein Objekt der Basisklasse oder einen Zeiger oder eine Referenz auf die Basisklasse zugreifen, braucht man eine explizite Konvertierung:

```
wecker w(11,30,7,15);  // ein Wecker.
uhr *uhrzeiger = &w;   // OK.

uhrzeiger->show();
          // das ist NICHT wecker::show !!
((wecker*)uhrzeiger)->show();
          // so ist's richtig!
```

Wie kann man im Allgemeinfall (z.B. eine Liste von `uhr`-Zeigern) sicherstellen, daß bei Pointerzugriff auf die richtigen Klassenelemente zugegriffen wird **???**

1. Man verweist nur auf einen Typ (`uhrzeiger` zeigt immer auf `uhr`). **Unbefriedigend!**
2. Man baut ein Typfeld in die Basisklasse ein. Konstruktoren setzen diese Feld. Funktionen lesen es. **Fehleranfällig!**
3. Man verwendet **virtuelle Funktionen**. **Ideal!**

8.5.1 Benutzer-definierte Typfelder

Ein Beispiel zum Lösungsvorschlag 2.:

```
class uhr
{ int std;
  int min;
public:
  enum uhrtyp { u, w };
  uhrtyp ut;
  uhr(int s, int m)
        : std(s), min(m), ut(u) {};
  uhr() : std(0), min(0), ut(u) {};
  void show();
};
class wecker : public uhr
{ public:
  uhr alarmzeit;
  wecker() : alarmzeit(0,0) {ut=w;}
  wecker(int s, int m, int as, int am)
           : uhr(s,m),
             alarmzeit(as,am){ ut=w;}
};
void uhr::show()
{ cout << std << ':' << min;
  if (ut == w)
  { wecker *w = (wecker*)this;
    cout << " || ";
    w->alarmzeit.show();
  }
}
```

Diese Methode ist besonders bei größeren Programmen fehleranfällig:

- Typfeldverwaltung liegt in den Händen des Programmierers.
- Daten der abgeleiteten Klasse müssen zum Zugriff aus der Basisklasse öffentlich sein (hier `alarmzeit`).
- Das wesentliche Prinzip der Modularität durch Vererbung ist verletzt, da die Basisklasse Eigenschaften der abgeleiteten Klassen kennen muß!

8.6 Virtuelle Funktionen

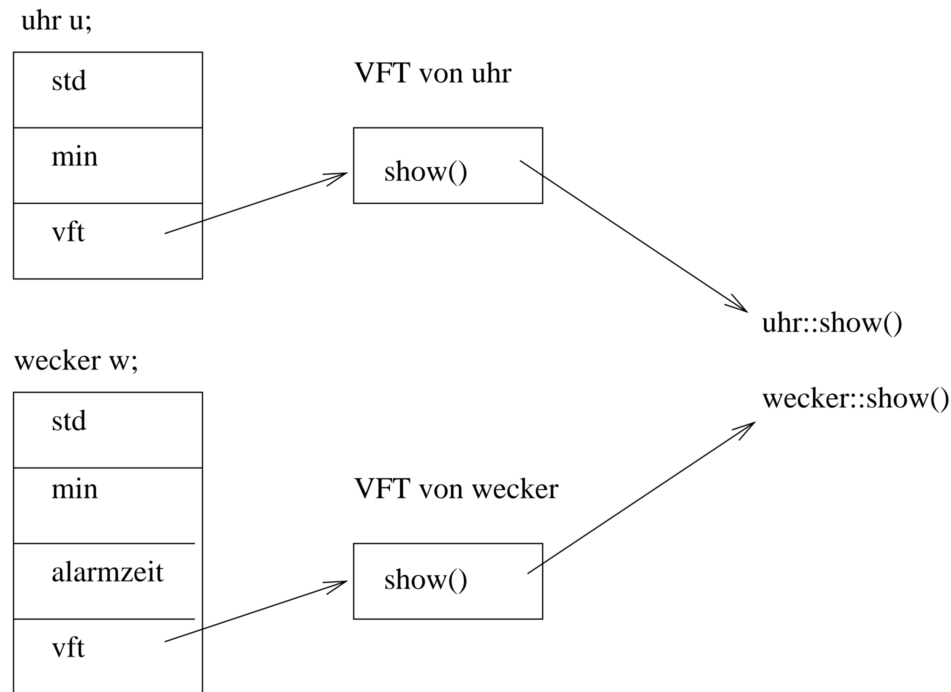
Virtuelle Funktionen lösen obige Probleme, indem der Compiler Code erzeugt, der zur Laufzeit die zum jeweiligen Objekt "passenden" Elementfunktionen bestimmt.

Dazu wird eine Funktion, die in einer Unterklasse (evtl.) redefiniert werden soll, in der Basisklasse als `virtual` gekenzeichnet.

```
class uhr
{ int std;int min;
public:
  uhr(int s, int m) : std(s), min(m) {};
  uhr() : std(0), min(0) {};
  uhr(const uhr& ori) : std(ori.std),
                        min(ori.min) {}
  virtual void show()
  { cout << std << ':' << min;}
};
class wecker : public uhr
{ uhr alarmzeit;
public:
  wecker() : alarmzeit(0,0) {}
  wecker(int s, int m, int as, int am)
           : uhr(s,m),
             alarmzeit(as,am) {}
  wecker(const wecker & ori)
           : uhr(ori),
             alarmzeit(ori.alarmzeit) {}
  void show() { uhr::show();
                cout << " || ";
                alarmzeit.show();
              }
};

int main()
{   uhr *up = new wecker(11,30,7,15);
    up->show();  // ruft wecker::show()
}
```

8.6.1 Implementierung mit Funktionstabellen

Jede Klasse, die virtuelle Funktionen benutzt, hat eine Tabelle VFT (*virtual function table*) mit Zeigern auf den Code der virtuellen Funktionen. Jedes Objekt einer solchen Klasse enthält einen Zeiger auf die VFT seiner Klasse:



Zur Laufzeit des Programms wird die aufzurufende Funktion durch Zugriff auf die virtuelle Funktionstabelle bestimmt (*Dynamic Binding*).
Dynamische Bindung ist eines der grundlegenden Merkmale objektorientierter Programmiersprachen.

8.6.2 Noch ein Beispiel

```
class sigma
{ double sum;
public:
  sigma()                 { sum = 0.0;}
  virtual void input(double d) { sum += d;}
  double getsum()         { return sum;}
};
class mue : public sigma
{ int number;
public:
  mue() : sigma()         { number=0;}
  void input(double d) { sigma::input(d);
                           number++;
                         }
  double getavg()         { return number?
                           getsum()/number:0;}
};

int main()
{ sigma s;
  mue m;
  sigma* ps1;
  for (int i=1; i <= 10; i++)
    { i%2 ? (ps1=&m) : (ps1=&s);
      ps1->input(i);
    }
  cout << s.getsum() << endl;
  cout << m.getsum() << endl;
}
```

8.6.3 Regeln für die Benutzung von virtuellen Funktionen

- Die Klasse, die die Funktion als `virtual` deklariert, muß sie auch definieren.
- Soll die virtuelle Funktion in einer abgeleiteten Klasse redefiniert werden, muß Ergebnistyp und Signatur exakt übereinstimmen.
- Die Kennzeichnung `virtual` kann in den abgeleiteten Klassen entfallen.
- Nicht jede abgeleitete Klasse muß eine virtuelle Funktion redefinieren.
- Die virtuellen Funktionen können in unterschiedlichen Klassen unterschiedliche Zugriffsrechte haben.

8.6.4 Virtuelle Destruktoren

```
uhr *uz = new wecker(16,0,17,30);
...
delete uz;   // FALSCH
```

benutzt den Destruktor von **uhr**.

Ausweg: Virtuelle Destruktoren

```
class uhr
{ int std; int min;
public:
  uhr(int s, int m) : std(s), min(m) {};
  virtual ~uhr(){cout << "~uhr ";}
};
class wecker : public uhr
{ uhr alarmzeit;
public:
  wecker(int s, int m, int as, int am)
           : uhr(s,m),
             alarmzeit(as,am) {}
  ~wecker() { cout << "~wecker ";}
};
int main()
{ uhr *uz = new wecker(16,0,17,30);
  delete uz;
}
```

schreibt

```
~wecker ~uhr ~uhr
```

8.6.5 Abstrakte Basisklassen

Virtuelle Funktionen in der obersten Basisklasse werden häufig nie benutzt, weil sie dort keinen Sinn ergeben. Man kann dies ausdrücken, indem man sie als **pure virtuelle Funktion** kennzeichnet:

```
class ding
{ public:
  // eine pure virtuelle Funktion:
  virtual int weile() = 0;
};
class gutding : public ding
{ public:
  int weile() { return 1000000;}
};
class schlechtding : public ding
{ public:
  int weile() { return 0;}
};
```

Eine Klasse, die mindestens eine pure virtuelle Funktion hat, heißt **abstrakte Klasse**.
Es ist natürlich **verboten**, Objekte von abstrakten Klassen zu definieren.
Eine abgeleitete Klasse **muß** die pure virtuelle Funktion definieren oder sie wiederum als pur kennzeichnen.

8.7 Mehrfache Vererbung

Eine Klasse kann von beliebig vielen Basisklassen abgeleitet werden:

```
class A { .... };
class B { .... };
class C { .... };
class D : public A, public B, public C
{ .... } ;
```

Eine Klasse darf nicht mehr als einmal als direkte Basisklasse auftreten:

```
class D : public A, public A // FALSCH!
{ .... };
```

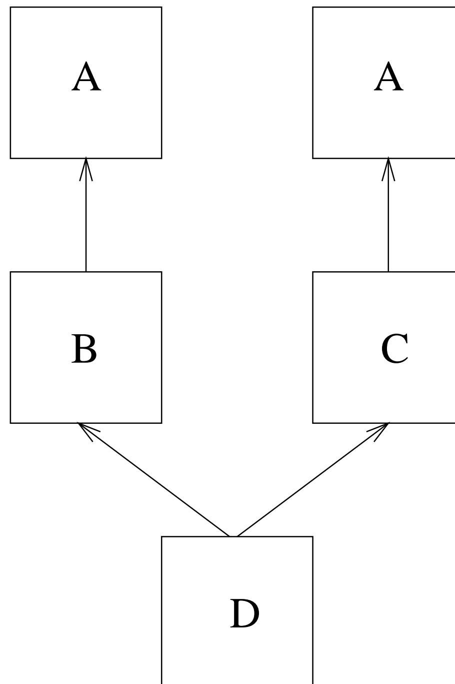
Grund: Jeder Zugriff auf die Elemente von `A` wäre mehrdeutig:

```
class A : { public: int val; };
class D : public A, public A // FALSCH!
{ .... };
void f (D* pd)
{ pd->val = 42;   // mehrdeutig !!!
}
```

Indirekte Basisklassen können jedoch mehrfach auftreten:

```
class A { .... };
class B : public A { .... };
class C : public A { .... };
class D : public B, public C   // OK
{ .... }
```

Das Diagramm der Klassenhierarchie sieht wie folgt aus:



Ein Objekt der Klasse `D` könnte folgendes Speicherlayout haben:

A-Teil (von B)
B-Teil
A-Teil (von C)
C-Teil
D-Teil

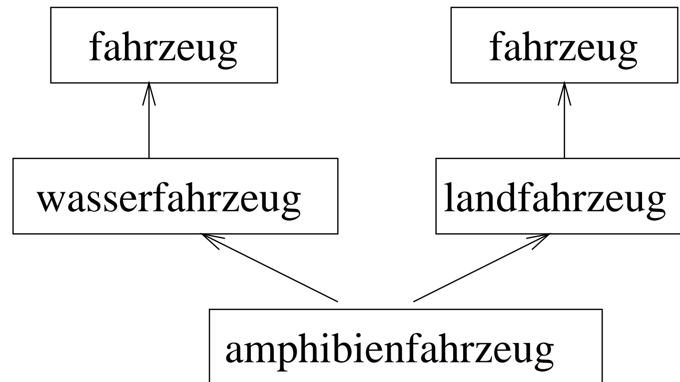
Mehrdeutigkeiten können durch Qualifizierung mit dem `::` Operator aufgelöst werden:

```
class A { public int val; };
class B : public A { .... };
class C : public A { .... };
class D : public B, public C
{ .... }
int D::f() { return C::val + B::val;}
```

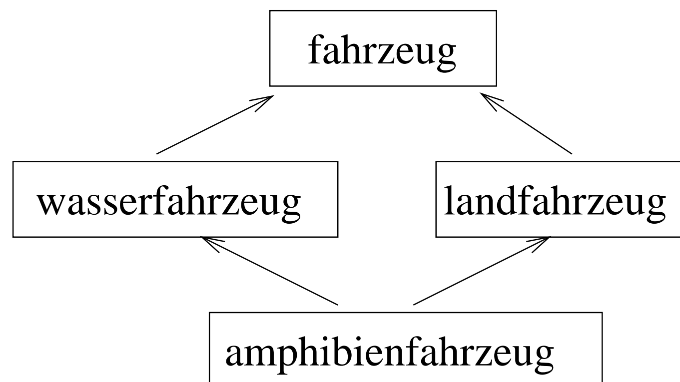
Mehrdeutige Namen führen nicht zu Fehlern, solange man sie nicht benutzt!

8.7.1 Virtuelle Basisklassen

In vielen Fällen ist die Mehrfachvererbung der selben Basisklasse nicht nötig und/oder erwünscht:



`amphibienfahrzeug` sollte nur eine Komponente der Basisklasse `fahrzeug` enthalten:



Man erreicht dies, in dem man die entsprechende Basisklasse *virtuell* vererbt:

```
class fahrzeug { .... };
class landfahrzeug
      : public virtual fahrzeug
{ ... };
class wasserfahrzeug
      : public virtual fahrzeug
{ ... };
class ampibienfahrzeug
      : public landfahrzeug,
        public wasserfahrzeug
{ .... };
```

Speicherlayout für ein Objekt der Klasse `amphibienfahrzeug`:

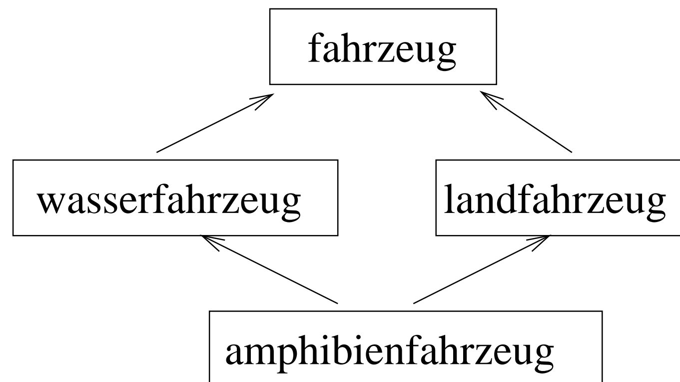
fahrzeug-Teil
landfahrzeug-Teil
wasserfahrzeug-Teil
amphibienfahrzeug-Teil

Mehrdeutigkeiten können so nicht auftreten.

Besonderheiten

- Virtuelle Basisklassen werden als erste konstruiert.
- Eine virtuelle Basisklasse muß einen Default-Konstruktor haben (falls sie überhaupt Konstruktoren hat).
- Virtuelle Basisklassen können von indirekt abgeleiteten Klassen konstruiert werden. Eine virtuelle Basisklasse wird von der am *weitesten abgeleiteten Klasse* konstruiert. Dazwischenliegende Konstruktoraufrufe werden nicht angewandt.

  In unserem Beispiel wird `fahrzeug` von `amphibienfahrzeug` konstruiert. Geschieht dies nicht explizit, wird der Default-Konstruktor aufgerufen (deshalb muß es ihn geben).



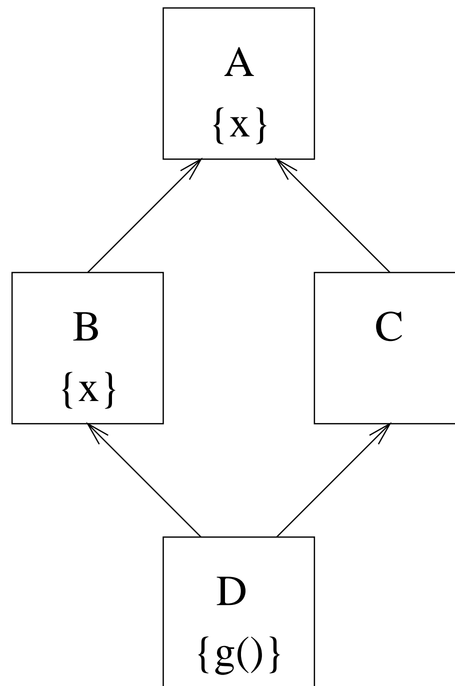
Beispiel:

```
class fahrzeug
{ public: fahrzeug()
  {cout << "\tdefault-fahrzeug\n"; }
};
class landfahrzeug: public virtual fahrzeug
{ public: landfahrzeug()
  {cout << "\tdefault-landfahrzeug\n"; }
};
class wasserfahrzeug: public virtual fahrzeug
{ public: wasserfahrzeug()
  {cout << "\tdefault-wasserfahrzeug\n"; }
};
class amphibienfahrzeug: public landfahrzeug,
                         public wasserfahrzeug
{ public:   amphibienfahrzeug()
  {cout << "\tdefault-amphibienfahrzeug\n"; }
};

int main()
{ cout << "Ein fahrzeug:\n";
  fahrzeug f;
  cout << "Ein landfahrzeug:\n";
  landfahrzeug lf;
  cout << "Ein amphibienfahrzeug:\n";
  amphibienfahrzeug af;
}
```

schreibt:

```
Ein fahrzeug:
        default-fahrzeug
Ein landfahrzeug:
        default-fahrzeug
        default-landfahrzeug
Ein amphibienfahrzeug:
        default-fahrzeug
        default-landfahrzeug
        default-wasserfahrzeug
        default-amphibienfahrzeug
```

Mehrdeutigkeiten bei Verwendung virtueller Klassen können durch die **Dominator-Regel** aufgelöst werden: Ein Name B::x dominiert einen Namen A::x wenn A eine Basisklasse von B ist.
Bei Mehrdeutigkeiten zwischen Namen, wird der genommen, der die anderen dominiert:




```
D::g()
{ x++; // das ist B::x
}
```

Mehrdeutigkeiten sind natürlich auch mit Dominator-Regel noch möglich:

```
class A
{ public : char x;
};
class B: public virtual A
{ public: char x;
          char y;
};
class C: public virtual A
{ public: char y;
};
class D: public B, public C
{ public: char z;
  D() { z=x+y;}
};
```

Compiler:

```
error: ambiguous B::y and C::y
Compilation failed
```

8.7.2 Mehrfache Vererbung und Typumwandlung

Bei Mehrfachvererbung kann Typumwandlung den Wert eines Pointers verändern:

```
class A {...};
class B {...};
class C: public A, public B
{...}
C* pc = new C;
B *pb;
```

Umwandlung von `pc` in einen `B*`

```
pb = pc;
```

oder

```
pb = (B*) pc;
```

setzt `pb` auf den `B`-Anteil von `pc`.

Experiment

```
#include <iostream.h>

class A { int a;};
class B { int b;};
class C: public A, public B
{int c;};

int main()
{ C* pc = new C;
  B *pb;
  pb = pc;
  cout << "pc: " << (long)pc << endl;
  cout << "pb: " << (long)pb << endl;
  if (pc == pb)
    cout << "pb und pc sind gleich\n";
}
```

erzeugt

```
pc: 142424
pb: 142428
pb und pc sind gleich
```

Der Vergleich von Zeigern auf Elemente von Klassenhierarchien berücksichtigt also die vorgenommenen Typumwandlungen.

9 50 Ways to Improve Your Programs

Dieses Kapitel folgt dem Buch

Effective C++
50 Ways to Improve your Programs and Designs
Scott Meyers
Addison-Wesley, 1992

9.1 `const` und `inline` statt ♯define

Den Compiler dem Präprozessor vorziehen:

```
const double RATIO = 1.472;
```

statt

```
#define RATIO 1.472
```

Vorteil:

- `RATIO` ist dem Compiler bekannt. Er kann sich in Fehlermeldungen darauf beziehen.
- Debugging.

Definitionen wie

```
#define MAX(a,b) ((a)>(b)?(a):(b))
```

müssen immer korrekt geklammert sein. Selbst dann kann es Ärger geben:

```
int a = 1;
int b = 0;
MAX(a++,b);   // a wird 2mal erhoeht
MAX(a++,b+2); // a wird 1mal erhoeht
MAX(a,"hi");  // Typen!!!
```

Die einfache und sichere Lösung ist:

```
inline int MAX(int a, int b)
{ return a > b ? a : b;
}
```

Der Präprozessor ist natürlich noch für bedingte Übersetzung wichtig (`#ifdef`, `#ifndef`, usw.).

9.2 iostream.h statt stdio.h

Die Funktionen aus stdio.h können auch in C++ Programmen benutzt werden, aber

- sie sind nicht typsicher
- sie sind nicht erweiterbar
- sie erfordern die Trennung von Formatierungsangabe und Daten (Fortran-Stil der 60er Jahre).

Die Stärken von << und >>:

- Erweiterbarkeit durch Überladen:

```
int i;
mycomplex c;
cin >> i >> c;
cout << "Complex: " << c;
```

- Ein- und Ausgabe hat die gleiche syntaktische Form:

```
int i;
char *pc;
scanf("%d %s",&i,pc);
...
cin >> i >> pc;
```

9.3 `new` und `delete` statt `malloc` und `free`

`malloc` und `free` wissen nichts über die Konstruktion und Destruktion von Objekten:

```
class String
{ char *value;
  public:
  String(const char* val=0);
  ~String();
};
String* array1=(String*)
        malloc(10*sizeof(String));
String* array2 = new String[10];
```

`array1` zeigt auf einen Speicherbereich für 10 Strings.
`array2` zeigt auf einen Speicherbereich mit 10 vollständig konstruierten Strings.

Ähnlich verläuft es bei der Speicherfreigabe:

```
free(array1);
delete [] array2;
```

`free` gibt den Speicher für die 10 Strings wieder frei. Die Stringobjekte, die evtl. selbst wieder Speicher allokiert haben, werden nicht zerstört.
`delete []` hat genau den gewünschten Effekt.

You're asking for trouble, if you use both `new/delete` and `malloc/free` in the same program.

9.4 C++ Kommentare benutzen

```
if (a>b)                 if (a>b)
{ int t=a;  //swap       { int t=a; /*swap*/
  a=b;                     a=b;
  b=t;                     b=t;
}                        }

 C++-Stil                C-Stil
```

Auskommentieren des Rumpfes führt zu folgender Situation:

```
if (a>b)                 if (a>b)
{// int t=a;  //swap     {/* int t=a; /*swap*/
 // a=b;                     a=b;
 // b=t;                     b=t;
}                         */}

 C++-Stil                C-Stil
```

Der C-Kommentar hört beim ersten `*/` auf.

9.5 Form der `new`- und `delete`-Aufrufe

Zusammengehörende Aufrufe von `new` und `delete` sollten die gleiche Form haben:

```
string *str = new string;
string *strarray = new string[100];
...
delete str;
delete [] strarray;
```

Der Effekt von

```
delete [] str;
delete strarray;
```

ist undefiniert $\Longrightarrow$ Nicht in der Form anwenden!

Außerdem folgt:
Wenn verschiedene Konstruktoren `new` verwenden, sollten sie das alle in der gleichen Form tun, da es nur einen Destruktor gibt.

9.6 `delete` für Pointer-Klassenelemente aufrufen

Ein Pointer-Member einer Klasse erfordert in der Regel:

- Initialisierung in jedem Konstruktor (mit 0, falls kein Speicher allokiert wird)
- Freigabe des Speichers und Neu-Allokation beim Wertzuweisungs-Operator
- Freigabe des Speichers im Destruktor

Ein Verstoß gegen die ersten beiden Regeln macht sich meist schnell bemerkbar.

Nichtausführung der Speicherfreigabe führt auf lange Sicht zur Unbrauchbarkeit des Programms.

`delete` auf den Null-Pointer ist ungefährlich.

9.7 Ergebniswert von `new` testen

Wenn **`new`** keinen Speicher allokieren kann, wird 0 zurückgegeben. Dies muß abgefangen werden, und zwar **IMMER!**
Präprozessor-Makros wie

```
#define NEW(PTR,TYPE)\
(PTR) = new TYPE\
assert ((PTR) != 0)
```

helfen dabei nicht, da new verschiedene Formen haben kann:

```
new T;
new T[size];
new T(arguments);
```

und durch Überladen neue Signaturen bekommen kann.

9.8 Überladen von new

Wenn man den Operator **new** überlädt, muß man bedenken, daß er an abgeleitete Klassen vererbt wird:

```
class base
{ ...
  public:
  void* operator new(size_t size);
  ...
};
class deriv: public base
{ ...
};
...
deriv *dp = new deriv; // base::new
```

`base::new` wird in der Regel **nicht** das Gewünschte tun.
Besser: Ein sicheres Basisklassen-`new`:

```
void* base::operator new(size_t size)
{ if (size != sizeof(base))
    // nimm globales new:
    return ::new char[size];
  ...
}
```

9.9 Das globale `new` nicht überdecken

Wenn eine Klasse den Operator **`new`** definiert (z.B. mit eigener Signatur) ist das Standard-**`new`** überdeckt:

```
typedef void (*fuptr())
class X
{ ...
  public:
  void* operator new(size_t size,
                     fuptr f);
};
void fu();
X *px1 = new(fu) X;
X *px2 = new X; // FEHLER!
```

Statt den Anwender zu zwingen, **`::new`** zu verwenden:

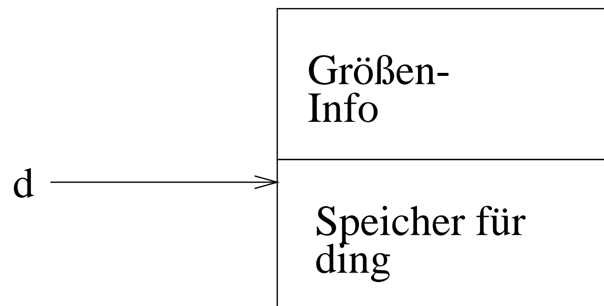
```
class X
{ ...
  public:
  void* operator new(size_t size,
                     fuptr f);
  void* operator new(size_t size)
  { return ::new char[size];
  }
};
```

9.10 Selbstdefinierte new/delete-Operatoren

Für kleine, häufig allokierte Objekte kann `new` zu teuer sein:

```
class ding
{ beschr *bs;
  public:
  ...
};
ding *d = new ding;
```

liefert:



Idee: Speicherverwaltung selbst schreiben:

```
class ding
{ beschr *bs;
  static ding* freiliste;
  public:
  void* operator new(size_t size);
  void operator delete(void * weg, size_t size);
}
ding* ding::freiliste = 0;
```

Dieses `new` allokiert Speicherplatz fuer mehrere Objekte des Typs `ding` und benutzt den Zeiger `bs`, um diese zu verketten:

```
void* ding::operator new(size_t size)
{ if (size != sizeof(ding))
     return ::new char[size];
  ding *p = freiliste;
  if (p)
  { freiliste = (ding*) p->bs;
    return p;
  }
  // neuer "roher" Speicher:
  ding *neu = (ding*)
              new char[256*sizeof(ding)];
  for (int i = 0 ; i < 255; i++)
      neu[i].bs = (beschr*)&neu[i+1];
  // Ende der Freiliste:
  neu[255].bs = 0;
  freiliste = &neu[1];
  return neu;
}
```

Natürlich braucht dieses `new` ein eigenes `delete`!

```
void ding::operator delete(void *weg,
                            size_t size)
{ // sende falsche Objekte
  // zum globalen delete:
  if (size != sizeof(ding))
  { ::delete[]((char*)weg);
    return;
  }
  ding *frei = (ding*) weg;
  frei->bs = (beschr*) freiliste;
  freiliste = frei;
}
```

9.11 Copy-Konstruktor und = Operator

Klassen mit dynamisch allokiertem Speicher brauchen Copy-Konstruktoren und Wertzuweisungsoperator.
Werden diese nicht definiert, greifen elementweise Zuweisung bzw. Kopie mit den bekannten (bösen) Folgen:

- Mehrfache Zeiger auf die gleichen Daten.
- Mehrfache `delete`-Aufrufe für die gleichen Daten.

Noch ein Beispiel:

```
class string
{ char *data;
  ...
};
void machnix(string local)
{}
...
string s("Tschuess schnoede Welt");
machnix(s);
```

9.12 Initialisierung durch Wertzuweisung vermeiden

```
class person
{ string name;
  date geburtstag;
  public:
  person(const string& n, const date& d);
};
```

Zwei Versionen des Konstruktors:

1.
```
person::person(const string& n,
               const date& d)
: name(n), geburtstag(d) {}
```

2.
```
person::person(const string& n,
               const date& d)
{ name = n;
  geburtstag = d
}
```

Die zweite Version funktioniert nicht für `const`- und Referenz-Member, da an diese nicht zugewiesen werden darf.

Ansonsten ist die zweite Version langsamer, da zusätzlich zu den Wertzuweisungen die beiden default-Konstruktoren ausgeführt werden müssen.

9.13 Reihenfolge der Initialisierung

```
class barray
{ int *data;
  unsigned size;
  int low, high;
public:
  barray(int l, int h);
};
barray::barray(int l, int h) // FALSCH!
: size(h-l+1),
  low(l),
  high(h),
  data(new int[size])
{// hier alle moeglichen Tests}
```

Das geht schief **!!!**
Bei der Initialisierung von `data` ist `size` noch nicht initialisiert, da die Initialisierung in der Reihenfolge der Deklaration geschieht.

Gute Idee:
Initialisierungslisten auch in dieser Reihenfolge schreiben.

9.14 Virtuelle Destruktoren

Basisklassen mit mindestens einer virtuellen Funktion sollten unbedingt einen virtuellen Destruktor definieren.

9.15 Rückgabewert des = Operators

Der Wertzuweisungsoperator `=` sollte eine Referenz auf `*this` zurückgeben.

Sonst funktionieren Folgen von Wertzuweisungen nicht:

```
string s1, s2;
s1 = s2 = "hallo";
```

9.16 Member-Zuweisung im = Operator

Wenn man den Zuweisungsoperator selbst definiert, muß man an alle Datenelemente der Klasse zuweisen.
Das ist klar, aber bei abgeleiteten Klassen gar nicht so einfach:

```
class base
{ int x;
  public:
  base(int i) : x(i) {}
};
class deriv: public base
{ int y;
  public deriv(int i): base(i), y(i)
  {}
  deriv& operator=(const deriv& rhs);
};
```

Falsch ist folgender Zuweisungsoperator für `deriv`:

```
deriv& deriv::operator=(const deriv& rhs)
{ if (this == &rhs)
     return *this;
  y=rhs.y;
  return *this;
}
```

da er die `x`-Komponente der Basisklasse nicht setzt.

So ist es richtig:

```
deriv& deriv::operator=(const deriv& rhs)
{ if (this == &rhs)
     return *this;
  ((base&) *this) = rhs;
  y=rhs.y;
  return *this;
}
```

Wichtig ist die Typumwandlung auf `base&`. Ohne `&` würde durch den Copy-Konstruktor ein temporäres Objekt vom Typ `base` kreiert.

Wenn `base` einen eigenen `=` Operator hat, ist es einfacher:

```
deriv& deriv::operator=(const deriv& rhs)
{ if (this == &rhs)
     return *this;
  base::operator=(rhs);
  y=rhs.y;
  return *this;
}
```

9.17 Selbstzuweisung abtesten

```
C& C::operator=(const C& rhs)
{ if (this == &rhs)
  return *this;
  ...
}
```

Gründe:

- Effizienz
- Korrektheit:
  Wenn vor der Wertzuweisung der alte Inhalt freigegeben wird, führt Selbstzuweisung zum Fehler.

9.18 Minimale Klassenschnittstellen

Die Schnittstelle der Klasse sind die Klassenelemente (meist Funktionen), die für den Benutzer der Klasse zugänglich sind.
Beim Entwurf gibt es zwei sich widersprechende Ziele:

- Die Klasse soll
  - leicht zu verstehen
  - leicht zu benutzen
  - leicht zu implementieren

  sein.
- Die Klasse soll
  - vollständig
  - mächtig
  - bequem zu benutzen

  sein.

Versuche ein Klasseninterface anzubieten, das *vollständig* und *minimal* ist.

Eine vollständige Schnittstelle erfüllt alle vernünftigen Wünsche an die Klasse.
Eine minimale Schnittstelle, bietet so wenig Funktionen wie möglich an (also keine Redundanzen und überlappenden Funktionalitäten).

9.19 Element-, Freund- oder globale Funktionen

Außer bei virtuellen Funktionen, die immer Elementfunktionen sein müssen, ist die Entscheidung oft nicht leicht.
Im Zweifelsfall: Benutze OO-Stil $\Longrightarrow$ Elementfunktionen.
Aber:

```
class bruch
{ int zaehler;
  int nenner;
  public:
  bruch (int z = 0, int n = 1);
  int zae() { return zaehler;}
  int nen() { return nenner;}
  bruch operator* (const bruch& b);
};
...
bruch viertel(1,4);
bruch achtel(1,8);
bruch result = viertel * achtel; // OK
result = result * 5; // OK
result = 2 * achtel; // FEHLER !
```

Der linke (implizite) Operand **muß** vom Typ `bruch` sein.

Also: Nicht-Element-Funktion:

```
bruch operator* ( const bruch& b1,
                  const bruch& b2)
{ return bruch(b1.zae() * b2.zae(),
               b1.nen() * b2.nen());
}
```

So funktioniert auch `result = 2 * achtel`.

Sollte `operator*` **Freund** der Klasse sein?
In unserem Fall: nein, da Zugriffsfunktionen (`inline`) auf die nicht-öffentlichen Datenelemente existieren.

Aber häufig ist es nötig:

```
class string
{ ...
  ostream& operator<<(ostream& o);
};
...
string s;
s << cout;
```

Das ist legal, entspricht aber nicht den Konventionen!
$\Longrightarrow$ Globale Freund-Funktion.

9.20 Vermeide öffenliche Datenelemente

Gründe:

- Konsistenz: Wenn alle öffentlichen Klassenelemente Funktionen sind, braucht der Anwender nicht zu überlegen, wie der Zugriff aussieht.
- Kontrolle über den Zugriff auf die Datenelemente:

```
class access
{ int readonly;
  int writeonly;
  int readwrite;
 public:
  int get_readonly();
  int set_writeonly(int);
  int get_readwrite();
  int set_readwrite(int);
};
```

- Funktionale Abstraktion: Man kann die Datenrepräsentation ändern, ohne die Anwendungsprogramme anpassen zu müssen.

9.21 const benutzen

Die `const`-Spezifikation erlaubt die Formulierung von Einschränkungen, deren Einhaltung vom Compiler überprüft wird.
Für Variablen:

```
char *p;
const char *p;       // konstante Daten
char *const p;       // konstanter Zeiger
const char *const p; // konstanter Zeiger
                     // + konstante Daten
```

In Funktionsdeklarationen:

- Konstantes Ergebnis

  ```
  class string
  { char *data;
   public:
    operator const char*() {return data;}
  };
  ```

  Verhindert, daß über das Funktionsergebnis Änderungen an privaten Elemente vorgenommen werden.

- Konstante Parameter dürfen im Rumpf nicht verändert werden.

- Konstante Elementfunktion

  dürfen auch auf konstante Objekte angewandt werden, da sie die Datenelemente nicht verändern dürfen.

  Die `const`-Markierung kann auch zum Überladen benutzt werden:

```
class string
{ char *data;
 public:
  char & operator[](int i)
  { return data[i]; }
  // die Version fuer const Objekte:
  const char & operator[](int i) const
  { return data[i]; }
};
...
const str cstring = "Hello";
string str = "World";
str[0] = 'x';   // OK
cstr[0] = 'x';  // Fehler!
```

9.22 Übergabe und Rückgabe per Referenz

Wenn nicht anders angegeben, wird Wertübergabe für Parameter und Funktionsergebnisse verwendet. Das bedeutet, daß Werte **kopiert** werden müssen (*Copy-Konstruktor*).

Beispiel:

```
class string {...};
class person
{ string name, vorname;
    ...
};
class student : public person
{ string uni, fach;
    ...
};
student machnix(student s)
{ return s;
}
```

Was kostet ein Aufruf von `machnix`???

1 Copy-Konstruktor `student` (Parameter)
  2 Copy-Konstruktor (`person`)
    3 Konstrukor `string` (`name`)
    4 Konstrukor `string` (`vorname`)
  5 Konstrukor `string` (`uni`)
  6 Konstrukor `string` (`fach`)
7 Copy-Konstruktor `student` (Ergebnis)
  8 Copy-Konstruktor (`person`)
    9 Konstrukor `string` (`name`)
    10 Konstrukor `string` (`vorname`)
  11 Konstrukor `string` (`uni`)
  12 Konstrukor `string` (`fach`)
13 Destruktor `student` (Parameter)
  14 Destrukor `string` (`fach`)
  15 Destruktor `string` (`uni`)
  16 Destruktor (`person`)
    17 Destrukor `string` (`vorname`)
    18 Destrukor `string` (`name`)
19 Destruktor `student` (Ergebnis)
  20 Destrukor `string` (`fach`)
  21 Destruktor `string` (`uni`)
  22 Destruktor (`person`)
    23 Destrukor `string` (`vorname`)
    24 Destrukor `string` (`name`)

Also: Übergabe und Ergebnisrückgabe per Referenz:

```
student & machnix( student &s)
{ return s;
}
```

Hier wird **kein** Konstruktor und **kein** Destruktor aufgerufen, da keine neuen Objekte kreiert werden.

Das geht natürlich nicht immer:

9.23 Keine Referenzrückgabe für neue Objekte

Es gibt zwei Möglichkeiten neue Objekte zu erzeigen:

1. Auf dem Keller:

```
bruch& operator* ( const bruch& b1,
                   const bruch& b2)
{ bruch result(b1.zae()*b2.zae(),
               b1.nen()*b2.nen());
  return result;  // FEHLER
}
```

2. Auf dem Heap:

```
bruch& operator* ( const bruch& b1,
                   const bruch& b2)
{ bruch* result = new bruch(
                   (b1.zae()*b2.zae(),
                    b1.nen()*b2.nen());
  return *result;  // GEFAHR
}
```

Die erste Version benutzt eine Referenz auf ein lokales Objekt ⟹ **funktioniert nicht!**

Bei dynamischer Allokation
besteht die Gefahr eines "Speicherlecks", wenn die zugehörige `delete`-Operation nicht ausgeführt wird.

Also: Wertrückgabe:

```
bruch operator* ( const bruch& b1,
                  const bruch& b2)
{ bruch result(b1.zae()*b2.zae(),
               b1.nen()*b2.nen());
  return result;  // OK
}
```

9.24 Überladen $\Rightarrow \Leftarrow$ Default-Parameter

```
void f();
void f(int i);
f();
f(10);
void g(int i=0);
g();
g(10);
```

Was ist besser **???**

Kommt darauf an,

- ob es einen vernünftigen Default-Wert gibt.
- ob der gleiche Algorithmus für die verschiedenen Versionen der Funktion geeignet ist.

Wenn beides zutrifft, nimm Default-Parameter.

9.25 Überladen von Pointer und numerischen Typen vermeiden

```
void f(int i);
void f(char *p);
...
f(0); Was ist das?
```

`0` ist zuallererst ein `int`, also wird `f(int)` aufgerufen.

Die zweite Version von `f` erhält man mit

```
f((char*) 0);
```

oder

```
const char* NULLCHAR = 0;
f(NULLCHAR);
```

9.26 Vorsicht bei potentieller Mehrdeutigkeit

In C++ ist potentielle Mehrdeutigkeit nicht verboten (siehe Mehrfachvererbung). Nur die Verwendung mehrdeutiger Konstrukte führt zu Fehlern.
Man braucht keine Klassen, um potentielle mehrdeutige Programme zu schreiben:

```
void h(char c);
void h(int i);
...
h(3.14);
```

Compiler:

```
error: ambiguous call: h ( double )
     choice of h()s:
        h(char );
        h(int );
Compilation failed
```

9.27 Compiler-generierte Elementfunktionen

Wenn die Benutzung Compiler-generierter Elementfunktionen (für Kopie und Wertzuweisung) nicht erwünscht ist, sollte sie explizit verboten werden:

```
class array
{ ...
 private:
  array& operator = (const array& rhs) {};
```

Das reicht noch nicht ganz: Freunde und Member dürfen Operator `=` immer noch benutzen (mit ungewolltem Ergebnis).

Trick: Nur deklarieren, nicht definieren (wirklich nicht):

```
class array
{ ...
 private:
  array& operator = (const array& rhs);
```

Benutzungen dieses Operators werden dann vom Linker beanstandet.

9.28 Partitionierung des globalen Namensraumes

In großen Anwendungen kann es leicht zu Namenskonflikten im globalen Namensraum kommen:

```
// xyz1.h
const float Version 3.4;
// xyz2.h
const int Version 7;
```

Dies kann man vermeiden, indem man seine globalen Objekte in einer Struktur ablegt:

```
 struct myglobals
 { static const float Version;
   static void f();
   enum fuzzy {wahr, falsch, weissnicht};
 };
 // Initialisierung der statischen Var.:
const float myglobals::Version = 3.4;
```

Benutzung wird dann mit `myglobals` qualifiziert.

9.29 Konstante Elementfunktionen

Konstante Elementfunktionen sollten keine "Handles" auf Datenelemente zurückgeben.

> Objekt A: "Liebling, bleib wie Du bist!"
>
> Objekt B: "Keine Sorge, Schatz, ich bin `const`."

Kann man B trauen???

Beispiel:

```
class string
{ char *data;
 public:
  operator char *() const;
};
string::operator char *() const
{ return data;
}
const string B("I'm const");
char *str = B; // ruft den operator char*
strcpy(str,"NoNoNo");
```

Dadurch wird das private `data` Element überschrieben.

Der Fehler liegt im Operator `char *()`, der ein "Handle" auf `data` liefert und andererseits als `const` markiert ist. Dadurch kann er auf konstante Objekte angewandt werden.
Bessere Lösung:

```
string::operator char *() const
{ char *copy = new char [strlen(data)+1];
  strcpy(copy,data);
  return copy;
}
```

9.30 Elementfunktionen, die Pointer auf weniger zugreifbare Elemente liefern

```
class person
{ private:
   char* name;
  protected:
   char *address;
  public:
   char *addr()
   {return address;}
};
```

Sowas sollte man vermeiden, da `address` jetzt nicht mehr `protected` ist.

9.31 Keine Referenzen auf lokale Objekte zurückgeben

Lokale Objekte existieren nicht mehr nach Rückkehr aus der Funktion.
Abhilfe durch dynamische allokierte Objekte birgt die Gefahr des Speicher-Lecks:

```
complex& operator +(const complex& c1,
                    const complex& c2)
{ complex* res = new complex
                     (c1.r + c2.r,
                      c1.i + c2.i);
  return *res;
}
```

Wer ruft das dazugehörige `delete` auf?.

```
complex& four = two + two;
...
delete &four;
```

Entweder der Programmierer vergißt es, oder er hat gar keine Chance:

```
complex eight = two + two + two + two;
```

Also: Wertrückgabe.

9.32 Ganzahlige Klassenkonstanten

Klassenlokale Konstanten sind leider nicht möglich. Auch statische Datenelemente helfen nicht:

```
class X
{ static const int BUFSIZ=100;  // FALSCH
  char buffer[BUFSIZ];
};
```

Auch so geht es nicht:

```
class X
{ static const int BUFSIZ;
  char buffer[BUFSIZ];        // FALSCH
};
const int X::BUFSIZ=100;
```

Was jedoch geht, sind klassenlokale Aufzählungstypen. Damit kann man wenigstens ganzahlige Konstanten implementieren:

```
class X
{ enum {BUFSIZ = 100};
  char buffer[BUFSIZ];        // OK
};
```

9.33 Benutze `inline` mit Bedacht

`inline` ist ein **Hinweis** an den Compiler, den Funktionsrumpf einzusetzen. Das kann, richtig angewandt, beträchtliche Laufzeitgewinne bringen. Preis: höherer Speicherbedarf.

Vorsicht: Wenn der Compiler beschließt, nicht zu "inlinen", kann es auch zusätzlichen Platz kosten:

```
// examp.h
inline void fun() { ... }

// source1.cc
#include "examp.h
...

// source2.cc
#include "examp.h
...
```

Sowohl `source1.o` also auch `source2.o` enthalten den Code von `fun()` (als `static` Funktionen wg. Linker).

Es kann sogar sein, daß eine Funktion "geinlined" wird und zusätzlich als aufrufbare Funktion im Code steht:

```
inline void fun() { ... }
void (*pf)() = fun;
int main()
{ fun();  // inline
  pf();   // non-inline
}
```

Tip:

Am Anfang der Programmentwicklung keine Funktionen "inlinen" (bzw. nur wirklich elementare). Dies hilft beim Debuggen.

Später zum Tuning die Funktionen in zeitkritischen Bereichen evtl. `inline` deklarieren.

9.34 Abhängigkeiten zwischen Programm-Moduln

In C++ sind Klassenimplementierung und Klassenschnittstelle nicht klar getrennt:

```
#include "string.h"
#include "date.h"
class person
{ string name;  // Implementierung
  date geb;     // Implementierung
 public:
  person(string, date); // Schnittstelle
  ...
};
```

⟹ auch wenn Änderungen sich auf die (private) Implementierung beschränken, müssen Anwender der Klasse neu übersetzt werden.

Abhilfe ist möglich über Zeiger auf die Implementierung:

```
// Forward Declarations
class string;
class date;
class personimpl;

class person
{ personimpl *impl;
 public:
  person(string, date); // Schnittstelle
  ...
};
```

9.35 Öffentliche Vererbung modelliert "istein"

Wenn eine Klasse B öffentlich von Klasse A abgeleitet wird, heißt das, daß jedes Objekt von Typ B ein Objekt von Typ A ist, aber nicht umgekehrt:

B *istein* A

- Ein Objekt vom Typ B kann einem Objekt von Typ A zugewiesen werden.
- Ein Zeiger auf B, kann einem Zeiger auf A zugewiesen werden.
- Eine Funktion, die ein A (Zeiger/Referenz auf A) erwartet, kann ein B (Zeiger/Referenz auf B) bekommen.

Bei *privater* Vererbung ist das nicht so. Sie modelliert eine andere Art von Klassenbeziehung (später).

Wofür *protected* Vererbung gut ist, weiß z.Zt. noch keiner.

9.36 Vererbung von Schnittstelle und/oder Implementierung

Wenn nur Funktionen vererbt werden (siehe 9.20), gibt es drei Möglichkeiten:

1. Erben einer puren virtuellen Funktion:

   Man erbt nur die Schnittstelle. Implementieren muß man selbst.

2. Erben einer virtuellen Funktion

   Man erbt die Schnittstelle und eine Default-Implementierung, die man überschreiben kann.

3. Erben einer nicht-virtuellen Funktion

   Man erbt die Schnittstelle und eine Implementierung, die man nicht überschreiben sollte (wg. statischer Bindung (siehe 8.6, 8.6.1, 9.37)).

Zwischen diesen Möglichkeiten ist schon beim Entwurf der Basisklasse sorgfältig auszuwählen.

9.37 Eine geerbte nicht-virtuelle Funktion nicht überschreiben

Mit nicht-virtuellen Funktionen erbt man auch deren Implementierung (siehe 9.36). Die sollte man nicht umdefinieren.

Darüberhinaus, kann dann *statische Bindung* zu Überraschungen führen:

```
class base
{ public:
  void memberfun()
  { ... }
};
class deriv : public base
{ public:
  void memberfun()
  { ... } // Besser nicht !!!
};
deriv d;
base *bp = &d;
bp->memberfun();  // ist base::memberfun
```

9.38 Geerbte Default-Parameterwerte nie ändern

```
#include <iostream.h>
enum preis { hoch=30000, niedrig=300};
class automobil
{ public:
  virtual void
    finanziere(int preis = hoch) = 0;
  };
class alte_ente : public automobil
{ public:
    void finanziere( int preis = niedrig)
    { cout  << preis << endl;
    }
};

int main()
{ automobil *ap = new alte_ente;
  ap->finanziere();
}
```

schreibt: `30000`.

<u>Der Grund:</u>

Die Funktion wird zwar **dynamisch** gebunden, der Default-Parameter jedoch **statisch**.

9.39 Vermeide “downcasts”

Explizite Typumwandlung von Basisklassen nach abgeleiteten Klassen heißen Downcasts. Sie sind erlaubt, aber **gefährlich**:

```
class mensch
{ char *name;
public:
  mensch();
  mensch(char *n);
  char *getname();
};

class student : public mensch
{ char *ort;
public:
  student();
  student(char* n, char *o);
  char *getort();
};

int main()
{ student s("Meier","Paderborn");
  mensch* mp = &s;
  // Ein Downcast:
  cout << ((student*)mp)->getort();
}
```

Kommt nun eine zweite abgeleitete Klasse hinzu, kann es gefährlich werden:

```
class wirt : public mensch
{   int umsatz;
    char *ort;
public:
  wirt();
  wirt(char* n, char *o);
  char *getort();
};

int main()
{ wirt w("Meier","Paderborn");
  mensch* mp = &w;
  cout << ((student*)mp)->getort();
}
```

liefert:

```
Segmentation fault (core dumped)
```

Eine Möglichkeit Downcasts sicherer zu machen:

```
class wirt;     // forward
class student;  // forward
class mensch
{ ....
  virtual student* cast2student()
  { return 0;}
  virtual wirt* cast2wirt()
  { return 0;}
};
class student : public mensch
{ ....
  student* cast2student()
  { return this;}
};
class wirt : public mensch
{ ....
  wirt* cast2wirt()
  { return this;}
};
int main()
{ wirt w("Meier","Paderborn");
  mensch* mp = &w;
  if (mp->cast2student())
    cout << mp->cast2student()->getort();
  else
    cout << mp->cast2wirt()->getort();
}
```

9.40 Modelliere "hatein" durch Elemente von Klassentyp

Durch Datenelemente aus anderen Klassen lassen sich neue Klasse auf alten "aufsetzen":

```
class address
{ ... };
class person
{ ...
  address addr;
  ...
};
```

Die Klassenbeziehung, die dadurch modelliert wird, heißt *hatein*:

person *hatein(e)* address

9.41 Private Vererbung

Objekte abgeleiteter Klassen können bei privater Vererbung nicht in solche der Basisklasse umgewandelt werden:

```
class a
{ int aa;
public:
  a(int i=0): aa(i) {}
};
class b : private a
{ int bb;
public:
  b(int i=0): bb(i) {}
};
int main()
{ b vb(20);
  a* pa = &vb;
}
```

Compiler:

```
error: no standard conversion of b* to a*
error: cast: b* -> base a*; private base class
```

Darüberhinaus wird nur die Implementierung der Elementfunktionen vererbt, nicht die Schnittstelle
$\Longrightarrow$ Private Vererbung modelliert nicht *istein* sondern *ist-implementiert-durch*.

9.42 Templates und Vererbung

Wann sollten neue Klassen mit Hilfe von Templates, wann durch Vererbung erzeugt werden

- Templates sollte man verwenden, wenn der Typ der Objekte das Verhalten der Klassenfunktionen nicht beeinflußt. Typische Anwendungsgebiete sind *Container-Klasse* (Listen, Keller, etc.).
- Vererbung zwischen Klassen sollte verwendet werden, wenn der Typ der Objekte das Verhalten der Klassenfunktionen beeinflußt.

9.43 Mehrfachvererbung

Mehrfachvererbung birgt die Gefahr der Mehrdeutigkeit:

```
class mensch
{ ...
  public:
  void augen();
  void nase();
  void mund();
  void backen();
};
class kuchen
{ ...
  public:
  void ruehren();
  void backen();
  void essen();
};
class baecker : public mensch,
                public kuchen
{ ... };
...
baecker vogt;
vogt.backen();  // FEHLER !
```

Außerdem gelten folgende Besonderheiten:

- Der Konstruktor einer virtuellen Basisklasse wird von der am weitesten abgeleiteten Klasse aufgerufen.
- Für virtuelle Basisklasse gilt die Dominator-Regel zur Identifikation von Elementnamen.
- Ein Downcast (siehe 9.39) von virtuellen Basisklassen ist verboten.

9.44 Verstehen was man tut

Zur Umsetzung eines Programmentwurfs in C++ (oder andere Sprachen) genügt es nicht, die Sprachkonstrukte zu beherrschen. Wichtig ist, wie die einzelnen Konstrukte die reale Welt modellieren.

Zusammenfassung

- Wenn zwei Klassen die gleiche Basisklasse haben, haben sie *gemeinsame Züge* (gleiche Datenelemente, Funktionen).
- `public` Vererbung bedeutet *istein*.
- `private` Vererbung bedeutet *ist-implementiert-durch*.
- Elemente von Klassetyp bedeutet *hatein*.
- Pure virtuelle Funktionen bedeuten, daß nur die Schnittstelle vererbt wird.
- Virtuelle Funktionen bedeuten, daß die Schnittstelle und eine Default-Implementierung vererbt wird.
- Nicht-virtuelle Funktionen bedeuten, daß die Schnittstelle und eine vorgeschriebene Implementierung vererbt wird.

9.45 Compiler-generierte Funktionen

Wenn man es nicht selbst tut, schreibt der Compiler für eine Benutzer-definierte Klasse folgende Funktionen:

- Ein Copy-Konstruktor
  Macht elementweise Kopien der Datenelemente
- Ein Wertzuweisungs-Operator
  Elementweise Zuweisung der Datenelemente
- Ein Adress-Operator `&`

  ```
  {return this;}
  ```

- Ein Adress-Operator `&` für konstante Objekte

  ```
  {return this;}
  ```

- Ein Default-Konstruktor (falls keine Konstruktoren definiert sind)

  ```
  { // tut nichts
  }
  ```

- Ein Destruktor (falls die Klasse von einer mit Destruktor abgeleitet ist)

  ```
  { // tut nichts
  }
  ```

9.46 Laufzeit-Tests

In C++ gibt es keine Laufzeittests (Division durch Null, Arraygrenzen, etc.). Grund: Platz- und Zeitersparnis.
In einem sauberen Programm müssen Laufzeitfehler explizit abgefangen werden. Man kann versuchen, die Anzahl der Tests zu reduzieren, indem Laufzeittests durch Übersetzungszeittests ersetzt werden (Bsp.: Eine Klasse `date`, bei der es genau 12 Möglichkeiten für die Monatskomponente gibt.)

9.47 Globale Objekte vor der Benutzung initialisieren

Es gibt drei Regeln:

- Alle globalen Objekte in einer Datei (*translation unit*) werden initialisiert, bevor eine Funktion oder ein Objekt aus dieser Datei benutzt werden.
- Die Initialisierung geschieht in der Reihenfolge der Deklaration.
- Wenn ein Objekt nicht explizit initialisiert wird, wird es vom Compiler mit 0 initialisiert. Dies geschieht vor den expliziten Initialisierungen.

Es gibt keine Aussage über die Initialisierungsreihenfolge bei mehreren Dateien:

```
#include <iostream.h>
int fun()
{ int inp;
  cin >> inp;
  return inp;
}

int i = fun();
int main()
{ cout << "i is " << i << endl;
}
```

Wer sorgt dafür, daß das Objekt `cin` initialisiert ist, bevor `i` initialisiert wird ???
Am Ende der Include-Datei `iostream.h` wird folgendes Objekt definiert:

```
static class Iostream_init {
   ...
public:
        Iostream_init() ;
        ~Iostream_init() ;
} iostream_init ;
```

Dessen Konstruktion sorgt dafür, daß `cin` initialisiert wird.

9.48 Compiler-Warnungen beachten

Compiler-Warnungen werden häufig nicht beachtet ("Wenn es was Ernstes wäre, wäre es eine Fehlermeldung").
Gerade in C++ sollte man jedoch Warnungen nicht mißachten:

```
class base
{ public:
  virtual void f() const;
};
class deriv : public base
{ public:
  virtual void f();
};
```

Compiler:

```
warning:  deriv::f() hides virtual base::f()
```

Durch das fehlende `const` wird nicht `f` der Basisklasse redefiniert, sondern ein eigenes `f` definiert, wodurch `base::f` versteckt wird.

9.49 Geplante Spracherweiterungen im Auge behalten

Mit einer Normung von C++ wird es zu Spracherweiterungen kommen.
Man sollte jetzt schon Programmstellen identifizieren, wo diese Erweiterungen gewinnbringend einsetzbar wären.

9.50 Das Referenz-Manual lesen

Margaret A. Ellis, B. Stroustrup
The Annotated C++ Reference Manual
Addison-Wesley, 1990

Neben der Sprachreferenz liefert das Buch viele Kommentare, die erläutern, warum die Dinge so sind, wie sie sind.

10 Java: Überblick und Vergleich

`Java vereint die elegante Einfachheit von C++`
`mit der rasenden Geschwindigkeit von Smalltalk.`

Unbekannter Autor

10.1 Ursprünge

Java ist entstanden aus der Sprache Oak (James Gosling) für "embedded consumer electronic applications".
1995 neu defininiert und in "Java" umbenannt. Autoren: James Goslign, Bill Joy, Guy Steele, Richard Tuck, Frank Yellin und Arthur van Hoff.
1997 erscheint Java 1.1 mit einigen neuen Spracheigenschaften.

10.2 Was ist interessant an Java?

Java ist eine Sprache wie viele andere und seine APIs (application programming interfaces) sind Klassenbibliotheken wie viele andere. Einige Spracheigenschaften von Java machen es jedoch sehr gut geeignet für die vernetzte, heterogene Rechnerlandschaft der späten 90er.

Sun hat Java mit der folgenden Schlagwort-Orgie beschrieben:

- Einfach
- Objekt-orientiert
- Dynamisch und verteilt
- Interpretiert
- Robust
- Sicher
- Architekturneutral und portabel
- High-performance
- Multi-threaded

Was heißt das im Einzelnen?

Einfach?
Falsch. Java ist nicht einfach!
Im Vergleich zu C++ gibt es jedoch Vereinfachungen:

- keine structs und unions (nur noch `class`)
- keine Zeiger
- automatische *garbage collection*.

Objekt-orientiert?

Anders als C++ wurde Java als rein objekt-orientierte Sprache entworfen:

- Fast alle Dinge in Java sind Objekte (Ausnahme: die Werte von Grundtypen)
- Klassen sind die Grundübersetzungseinheit

Dynamisch und verteilt?

- Java Klassen werden zur Laufzeit des Interpretierers geladen.
- Man kann dynamisch Informationen über Klassen erhalten (Java 1.1: *Reflection*)
- Laden von Programmen über das Internet
- Zugriff auf Ressourcen über Netze (java.net package)
- Java 1.1: RMI (remote method invocation).

Interpretiert?

Der Java Übersetzer erzeugt den sogenannten "byte code", der von der virtuellen Java Maschine (*JVM*) interpretiert wird.

Robust?
Natürlich kann man in Java fehlerhafte Programme schreiben. Einige Spracheigenschaften erleichtern jedoch die Konstruktion zuverlässiger Software:

- Statisch typisierte Sprache
- Fehlende Pointer verhindern all die damit zusammenhängenden Fehler
- Bereichsüberprüfung für Arrays
- Automatic Garbage Collection verhindert Speicherlecks
- Exceptions fest in die Sprache integriert

Sicher?
Wichtig wegen der Verteilung der Anwendungen über Netze.

- Sicherheit durch Robustheit (s.o.)
- Byte-Code-Verifikation durch den Interpretierer
- Sandkasten-Modell: Programme spielen im Sandkasten, abgeschottet von der wirklichen Welt (z.B. kein Dateizugriff)
- Java 1.1: digitale Unterschriften am Programmcode

Architekturneutral und portabel?
Der Byte-Code ist maschinenunabhängig und kann überall dort ausgeführt werden, wo ein Java-Interpretierer existiert (“Write Once, Run Anywhere”(Sun)).

High-performance?
Falsch. Java 1.0 Programme sind wegen der Interpretation etwa 20mal langsamer als C Programme. Der Java 1.1 Interpretierer ist doppelt so schnell, so daß der Faktor nun etwa 10 beträgt!

Für Netzwerk-basierte Programme, GUIs u.ä. OK.

Abhilfe: “Just in time compilation” von Byte-Code in Maschinen-Code zur Laufzeit vor. Außerdem ist es möglich, C- oder C`++` -Methoden zu Java-Anwendungen dazuzubinden (`native` Methoden-Kennzeichnung).

Multi-threaded?
Java unterstützt Multi-Threading (`Thread`-Klasse, Schlüsselwort `synchronized`). Um unabhängige Ausführungs-Stränge zu synchronisieren, benutzt Java *Monitore* als Mechanismus, geschützte Bereiche des Codes nur einem Prozess gleichzeitig zur Verfügung zu stellen.

10.3 Java Programme und Java Applets

"Hello World" als Java-Programm:

```
public class HelloWorld {
  public static void main (String args[]) {
    System.out.println("Hello World");
  }
}
```

Beobachtungen:

- Die `public` Kennzeichnung der Klasse erlaubt die Benutzung in anderen Übersetzungseinheiten
- Die Klassenmethode `main` ist der Einstiegspunkt für den Interpretierer.
- `out` ist der Standard-Ausgabe-Strom. Er ist ein Klassenattribut (*static field*) der Klasse `System` und stellt eine Methode `println` zur Verfügung.

Dieses Programm wird übersetzt mit (Sun JDK)

```
javac HelloWorld.java
```

wodurch der erzeugte Byte-Code in der Datei `HelloWord.class` abgelegt wird.

Ausführung durch:

```
java HelloWorld
```

“Hello World” als Java-Applet:

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;

public class HelloWorldApplet extends Applet {
    public void paint(Graphics g)
    {
        // display "Hello World",
        // with start of baseline at 10,15
        g.setColor(Color.red);
        g.drawString("Hello, World", 10, 15);
    }
}
```

Applets laufen in einem Applet-Viewer oder einem Web-Browser.
Um Applets in eine HTML-Web-Seite zu integrieren:

```
<applet code="HelloWorldApplet.class"
 width=120 height=30>
[Your browser doesn't handle Java]
</applet>
```

Für die Präsentation der Programmiersprache Java sollen uns Applets jedoch nicht weiter interessieren.

10.4 Unterschiede zwischen Java und C/C++

10.4.1 Programmstruktur

Ein Java-Programm besteht aus einer oder mehreren Klassendefinitionen, die in Byte-Code in `.class`-Dateien übersetzt worden sind.
Einer der `.class`-Dateien enthält die Klasse, die die `main`-Methode definiert:

```
public static void main(String args[])
```

10.4.2 Kommandozeilen-Argumente

```
String args[]
```

enthält die Strings die auf der Kommandozeile übergeben worden sind. Anzahl: `args.length()`.

10.4.3 Keine globalen Variablen und Funktionen

Es existieren nur Felder (Attribute) und Methoden von Klassen.
Jede Klasse ist Teil eines `package`.
Ein voll qualifizierter Java-Name besteht also aus

package_name class_name member_name

z.B.

```
java.lang.System.out.println("Hello World");
```

mit

- `java.lang` ist ein Paket
- `System` ist eine Klasse aus diesem Paket
- `out` ist ein static field in dieser Klasse
- `println` ist eine Methode für dieses field

Das `package` statement spezifiziert, daß Java-Code zu einem bestimmten Paket gehört.
Die `import`-Klausel erlaubt die abkürzende Schreibweise für zu qualifizierende Namen:

```
import java.util.Hashtable
```

erlaubt es, den Klassennamen `Hashtable` zu verwenden statt `java.util.Hashtable`.

10.4.4 Kein Präprozessor

statt dessen:

- Konstanten werden durch `static final`-Variablen beschrieben. Sie müssen initialisiert werden und dürfen nicht verändert werden

  ```
  ...
  public static final double PI = 3.14159;
  ...
  ```

- Makros werden ersetzt durch normale Methoden und fortschrittliche Compiler-Technologie, die ggf. inline-Code erzeugen kann.
- Datei-Inklusion: ersetzt durch die beschriebene Qualifizierung von Namen, bzw. `import`-Klauseln.
- Bedingte Übersetzung kann ersetzt werden durch Klammern von Anweisungsblöcken mit Bedingungen:

  ```
  if (false)
  { ...
  }
  ```

  Ein optimierender Übersetzer wird solchen toten Code nicht mitübersetzen.

10.4.5 Unicode-Buchstaben

Java characters, strings und Bezeichner bestehen aus 16bit Unicode-Buchstaben.

10.4.6 Grundtypen

Java führt zusätzliche Typen `byte` und `boolean` ein und legt für jeden primitiven Typ exakt seine Größe fest:

`boolean`	`true` or `false`	`false`	1 bit
`char`	Unicode character	`\u0000`	16 bits
`byte`	signed integer	0	8 bits
`short`	signed integer	0	16 bits
`int`	signed integer	0	32 bits
`long`	signed integer	0	64 bits
`float`	IEEE 754 float	0.0	32 bits
`double`	IEEE 754 float	0.0	64 bits

10.4.7 Referenz-Semantik

Nicht-primitive Werte in Java sind Objekte und werden über Referenzen adressiert. Werte von Grundtypen werden "by value" behandelt.
Zwei verschiedene Variablen können also auf das gleiche Objekt verweisen:

```
Button p,q;
p = new Button();
q = p;    // q zeigt auf den gleiche Button
p.setlabel("OK");  // verändert auch q
```

das ist anders für Grundtypen

```
int i = 3;
int j = i;
i = 2;      // das verändert j nicht
```

Sollen die Daten eines Objektes kopiert werden, muß dies explizit geschehen:

```
Vector b = new Vektor;
c = b.clone();
```

Analog dazu vergleicht der Operator "==" nicht, ob zwei Werte gleich sind, sondern nur ob zwei Referenzen sich auf das gleiche Objekt beziehen. Wertgleichheit muß mit eigenen Methoden geprüft werden (z.B. `equals()`).

10.4.8 Java hat keine Pointer

Vereinfacht die Sprache und verhindert viele Programmierfehler

10.4.9 `null`: Keine Referenz

Der Wert `null` steht für nicht vorhandene Referenz auf ein Objekt und kann an alle Variablen vom Klassen-, Interface- oder Array-Typ zugewiesen werden.

10.4.10 Objekt-Erzeugung

Mit `new` und Konstruktoren wie in C++ :

```
java.awt.Button b = new java.awt.Button();
ComplexNumber c = new ComplexNumber(1.0, 1.414);
```

Besonderheit: Strings, obwohl nicht eingebaut sondern Bibliotheksklasse haben eine Sondernotation:

```
String s = "This is a test";
```

10.4.11 Garbage Collection

Java benutzt automatische Garbage Collection zur Freigabe von Objekten, die nicht mehr gebraucht werden, weil keine Referenz mehr auf sie existiert.

10.4.12 Die `synchronized` Anweisung

Zum Schutz kritischer Abschnitte, wenn mehrere threads aktiv sind, die gemeinsame Daten benutzen:

```
synchronized (expression) statement
```

Das Schlüsselwort **`synchronized`** kann auch als Kennzeichnung einer Methode benutzt werden, was die gesamte Methode zum kritischen Abschnitt macht.

10.4.13 Ausnahmebehandlung (exception handling)

Begriffe:

- *Exception:* ein Objekt, das signalisiert, daß eine außergewöhnliche Bedingung eingetreten ist.
- *throw an exception:* eine außergewöhnliche Bedingung signalisieren.
- *catch an exception:* auf die Ausnahme reagieren (wie auch immer).

Wenn eine Ausnahme nicht im Block behandelt wird, wo sie erzeugt wird, propagiert sie in den umfassenden Block, dann in die aufrufenden Methode, usw. bis zur `main`-Methode. Wird sie auch dort nicht behandelt, erzeugt der Java-Interpretierer eine Fehlermeldung und beendet die Ausführung.

Beispiel:

```
// Definition einer Exception-Klasse
class IllegalAverage extends Exception{
public IllegalAverage() { super();}
public IllegalAverage(String s) { super(s);}
}

// Eine Klasse, die diese Exception erzeugt:
class MyClass {
  public double average(double [] vals, int i, int j)
     throws IllegalAverage
  { try {
           return (vals[i] + vals[j]) / 2;
        } catch (IndexOutOfBoundsException e)
          { throw new IllegalAverage();
          }
  }
```

Irgendwo höher in der Aufrufkette kann es nun ein

```
catch (IllegalAverage e)
```

geben, das angemessen auf die Ausnahme reagiert.

10.4.14 Keine Structs, Unions, Aufzählungen

Structs werden durch Klassen abgedeckt. Wichtige Eigenschaften von Vereinigungstypen können mit Klassenhierarchien simuliert werden. Das Fehlen von Aufzählungen in Java ist etwas erstaunlich.

10.4.15 Form der Klassendefinition

Beispiel:

```
 public class Circle
 {
   static int num_circles = 0;
   public double x,y,r;
   public Circle(double x, double y, double r)
   {
            this.x = x; this.y = y; this.r = r;
   }
   public Circle(double r)
   {
            this(0.0, 0.0, r);
   }
   public double area() {
            return 3.13159 * r * r;
   }
}
```

Beobachtungen:

- Klassenvariablen
- Überladene Konstruktoren (geht auch für Methoden)
- `this` zum Aufruf anderer Konstruktoren (nur erlaubt als erste Anweisung im Konstruktorrumpf!)
- Alle Methoden in der Klasse definiert.

10.4.16 Vererbung

Java Begriff: Erweiterung (*extension*).

```
public class GraphicCircle extends Circle
{
  Color outline, fill;
  public GraphicCircle(double x, double y, double r,
                       Color outline, Color fill)
  {
         super(x,y,r);
         this.outline = outline;
         this.fill = fill;
  }
  public void draw(DrawWindow dw)
  {
      dw.draw(x, y, r, outline, fill);
  }
}
```

Die Vererbung in Java ist Einfachvererbung. Jede Benutzer-Klasse hat genau eine Oberklasse. Wenn nicht spezifiziert, ist dies die vordefinierte Klasse `Object`.
Die Methodenbindung in Java ist dynamisch für in Unterklassen redefinierte Methoden.

10.4.17 Abstrakte Klassen

Wenn eine Klasse eine *abstrakte Methode* (in C++ *pur virtuelle*) enthält, wird sie zur abstrakten Klasse.

Beispiel:

```
public abstract class shape
{
  public abstract double area();
}
```

Regeln:

- Es darf keine Objekte von abstrakten Klassen geben
- Von einer Unterklasse einer abstrakten Klassen kann es Objekte geben, wenn diese Unterklasse alle geerbten abstrakten Methoden redefiniert.
- Redefiniert eine Unterklasse einer abstrakten Klasse nicht alle geerbten abstrakten Methoden, wird sie selbst zur abstrakten Klasse.

10.4.18 Interfaces

Abstrakte Klassen, die nur abstrakte Methoden und keine Attribute enthalten, lassen sich als Eigenschaftsklassen verwenden. Java führt hierfür das Sprachkonstrukt `interface` ein.

Beispiel:

```
public interface Drawable
{
   public void setColor(Color c);
   public void setPosition(double x, double y);
   public void draw(DrawWindow dw);
}
```

Konkrete Klassen *implementieren* Schnittstellenklassen, d.h. liefern Methodendefinitionen für die abstrakten Methoden der Schnittstellenklassen.

Beispiel:

```
public class DrawableRectangle
     extends Rectangle
     implements Drawable
     { private Color c;
       private double x, y;
       ...
       public void setColor(Color c)
       { this.c = c; }
       public void setPosition(double x, double y)
       { this.x = x; this.y = y; }
       ...
   }
```

Wichtig: Die `implements`-Klausel kann mehrere Interfaces aufführen.

10.4.19 C++ -Konstrukte, die wir in Java nicht finden

- Mehrfachvererbung. Die Möglichkeit, mehrere Interface-Klassen zu implementieren, wird oft als ausreichender Ersatz angesehen.
- Generizität. Es gibt keine Templates in Java. Stattdessen: Container vom allegemeinsten Typ `Object`.
- Überladen von Operatoren. Wird meist nicht als schmerzlich angesehen, da Benutzer-überladene Operatoren oft die Lesbarkeit von Programmen erschweren (eine Ausnahme, die in Java an overloading errinnert, ist der Operator `+` für Objekte der Klasse `String`.
- Typanpassung durch einstellige Konstruktoren. Dies ist nur eine Kurzschreibweise für die normale Objektkonstruktion.

10.5 Application Programming Interfaces

- package java.applet
- package java.awt
- package java.awt.datatransfer
- package java.awt.event
- package java.awt.image
- package java.beans
- package java.io
- package java.lang
- package java.lang.reflect
- package java.math
- package java.net
- package java.rmi
- package java.rmi.dgc
- package java.rmi.registry
- package java.rmi.server
- package java.security
- package java.security.acl
- package java.security.interfaces
- package java.sql
- package java.text
- package java.util
- package java.util.zip

10.6 Java-Information auf dem Netz

Java User Group Deutschland	http://www.java.de/
Java Seiten bei Sun	http://java.sun.com/
Official directory for Java	http://www.gamelan.com/
und viele, viele andere	

Index

&: 97
(): 154
*: 91
+: 144
--: 144
->: 152
->*: 134
.*: 134
...: 106
.cc file: 6
.h file: 6
::: 28, 122
<<: 59, 63, 72
=: 150, 222, 225
>>: 64, 69, 72
[]: 151

Abgeleitete Klasse: 172, 178
Abgeleitete Typen: 88
Abstrakte Basisklasse: 195
abstrakte Klasse: 286
ANSI C: 109
Anweisungen: 110
API: 268, 290
Applet: 273
Array: 93, 95, 124, 126
Assoziativität: 119
Attribut: 21
Aufzählung: 87, 246
Ausdruck: 111, 119
Ausgabe: 58, 210
Ausgabe in Strings: 73
Ausgabe v. Dateien: 69
Auswahlanweisung: 111

Basisklasse: 172, 180, 183, 196
Basisklassenelemente: 177
Bitfelder: 108
Block: 111, 118
Botschaft: 21
break: 116
byte code: 270

Call-by-Reference: 98, 102
call-by-value: 95
Call-by-Value: 102
case: 112
Cast: 79
catch: 281

`cerr`: 59
`char`-Array: 93
`cin`: 64
`class`: 21
Class Templates: 158, 165
clever: 139
`clog`: 59
Compiler-Aufruf: 3
Compound Statement: 111
`const`: 34, 86, 208, 223, 232
`continue`: 116
Copy-Konstruktor: 48, 49, 57, 98, 184, 222, 263
`cout`: 59

Data hiding: 26
Datei-I/O: 69
Datenabstraktion: 7
Default-Konstruktor: 17, 45, 46, 176, 263
Default-Parameter: 15, 105, 238, 253
Definition: 23, 101
Deklaration: 23, 88, 100
Deklarations-Anweisung: 118
Dekrement: 144
`delete`: 130, 156, 211, 214, 215, 219
Dereferenzierung: 91
Destruktor: 54, 57, 116, 181
`do`: 113, 114
Dominator-Regel: 204
downcast: 254
dynamische Allokation: 126
dynamische Bindung: 191, 252
dynamischer Speicher: 222, 237

Einfachvererbung: 285
Einfügeoperator: 59, 63
Eingabe: 58, 210
Eingabe v. Dateien: 69
Eingabe von Strings: 73
Eingabeoperation: 68
einstelliger Konstruktor: 80
Elementfunktion: 143
Elementfunktionen: 15, 28, 29, 31, 34, 229, 233, 241, 243
Elementklasse: 183
Elementweise Kopien: 49
Elementzugriff: 152
`else`: 111, 112
`enum`: 87
Ergebnisrückgabe: 234, 236
Exception handling: 281

Exceptions: 271
explizite Typumwandlung: 79
`export`: 160
`extends`: 285
Extraktionsoperator: 64, 69

Filecopy: 72
Fließkomma-Konstanten: 82, 84
Fließkommatypen: 75
`for`: 113, 115
`free`: 211
Freunde: 38, 180, 229
`friend`: 38
`fstream`: 69
Funktion: 100
Funktionsaufruf: 134, 154
Funktionsdefinition: 101
Funktionsdeklaration: 100, 232
Funktionsergebnis: 104, 117
Funktionsparameter: 100, 106
Funktionsschablonen: 158, 159
Funktionstabellen: 191

Ganzzahlige Typen: 75
garbage collection: 269
Garbage Collection: 280
generische Funktionen: 158
generische Klassen: 165
Generizität: 158
Geschichte: 4
`get`: 68, 72
globale Funktionen: 229
Gosling: 268
`goto`: 117
Grundtypen: 75
Gültigkeitsbereich: 110, 115, 116, 122

hatein: 257
Hex-Konstanten: 83

I/O-Bibliothek: 58
Identifikation bei Überladung: 138
`if`: 111, 112
`ifstream`: 69
Implementierung: 249, 251, 262
Implementierungsdatei: 6
`implements`: 288
implizite Typumwandlung: 77
Indizierung: 151
Initialisierung: 39, 41, 43, 48, 51–53, 93, 97, 128, 184, 223, 264

Inkrement: 144
inline: 101, 208, 247
Inline-Elementfunktionen: 29
Inline-Funktion: 101
Instanz: 21
Integer-Konstanten: 82, 83
Interface: 287
Internet: 270
iostream: 13, 58, 210
ist-implementiert-durch: 258
istein: 250
istrstream: 74
Iteration Statement: 113
Iterator: 155

Java: 268
Java Virtual Machine: 270
Jump Statements: 116
just in time: 272

Keller: 12
Klassendefinition: 23, 25
Klassendeklaration: 25
Klassenelemente: 257
Klassenhierarchie: 172
Klasseninstanz: 21
Klassenschablonen: 158, 165, 259
Klassenschnittstelle: 228, 249
Kommentar: 12, 213
Konstante: 82, 86, 232, 243, 246
konstante Elementfunktion: 34
konstanter Zeiger: 86
Konstruktor: 15, 39, 42, 52, 53, 80, 181

Label: 110, 112, 117
Laufzeit-Tests: 264
Lesen aus Strings: 74
Literal: 82

Makefile: 20
malloc: 211
Manipulator: 60, 66
Mehrdeutige Namen: 198
Mehrdeutigkeit: 200, 240, 260
Mehrfach-Inklusion: 15
Mehrfache Vererbung: 196, 206, 240, 260
Member Functions: 15
Methode: 21, 143
Meyers, Scott: 208

new: 126, 156, 211, 214, 216–219
null: 280

Oak: 268
Objekt: 21
Objektorientierte Programmierung: 9
ofstream: 69
Oktal-Konstanten: 83
OO: 9
Operand: 119
Operator: 119, 120
ostrstream: 73
Overloading: 15, 136, 238

Parameter: 100, 101
Parameterübergabe: 102, 234
Parametrische Polymorphie: 158
Pointer-Arithmetik: 92, 96
Pointer-Typen: 91
Polymorphie: 158
Postfix-Operator: 89
potentielle Mehrdeutigkeit: 240
Präfix-Operator: 89
Präprozessor: 209
Präzedenz: 119
private: 15, 21, 26, 177
private Vererbung: 250, 258, 262
Programmierung
  modular: 6
  objektorientiert: 9
  prozedural: 5
protected: 26, 177
protected Vererbung: 250
public: 15, 21, 26, 177, 250
pure virtuelle Funktion: 195, 262
put: 72
putback: 68

Reference Manual: 267
Referenz: 97, 98, 124, 223, 234, 245
Reflection: 270
Reihenfolge der Destruktion: 182
Reihenfolge der Initialisierung: 224
Reihenfolge der Konstruktion: 182
return: 117
RMI: 270

Sandkasten: 271
Schablone: 158
Schleifen: 113
Schnittstelle: 228, 249, 251, 262
Schnittstellendatei: 6
Schreiben in Strings: 73

Scope Resolution: 28, 122, 176
Selbstzuweisung: 227
Selection Statement: 111
Signatur: 107
`signed`: 77
`sizeof`: 76, 124
Speicherallokation: 126, 237
Speicherfreigabe: 130, 212, 215
Speicherleck: 237, 245
Speicherverwaltung: 156, 219
Spracherweiterungen: 267
Sprünge: 116
`static`: 36
statische Bindung: 252
Statische Klassenelemente: 36
Steele: 268
Stream-Ausgabe: 59
Stream-Eingabe: 64
String-I/O: 73
String-Konstanten: 82, 85
Stroustrup: 4, 267
`strstream`: 73
`struct`: 21
`switch`: 111, 112
`synchronized`: 281

`template`: 158
Template: 158, 259
Templates: 159, 165
Textzeichen-Konstanten: 82, 84
`this`: 33
`throw`: 281
Tuck: 268
Typanpassung: 186
`typedef`: 137
Typfelder: 188
Typumwandlung: 77, 80, 104, 206, 226, 254
Typumwandlungs-Operator: 81

Überladen: 15, 238
Überladen von Funktionen: 136
Überladen von Operatoren: 136, 140
Übersetzeraufruf: 3
Unicode: 278
`union`: 21
`unsigned`: 77

van Hoff: 268
Vererbung: 10, 172, 250–252, 262
VFT: 191
Virtuelle Basisklassen: 199
Virtuelle Destruktoren: 194

virtuelle Funktionen: 262
Virtuelle Funktionen: 189, 193, 229
Virtueller Destruktor: 224
`void`: 90
Vorausdeklaration: 25
Vorzeichen: 77, 83

Warnungen: 266
Wert-Parameter-Übergabe: 95
Wertrückgabe: 48, 104, 234, 236
Wertzuweisung: 43, 51, 150, 222, 223, 225
`while`: 113

Yellin: 268

Zeiger: 95
Zeiger auf Funktionen: 107
Zeiger auf Klassenelemente: 134
Zeiger-Arithmetik: 92, 96
Zeiger-Typen: 91
Zugriffskontrolle: 26, 231
Zugriffsspezifikation: 26

Inhaltsverzeichnis

1 Übersicht 2

2 C++ ist nicht C: Geschichte, Sprachkonzepte und ein Beispiel 3

2.1 Das erste (und wichtigste) C++ -Beispiel 3
2.2 Geschichte 4
2.3 Sprachkonzepte 5
2.3.1 Programmierstil Prozedurales Programmieren 5
2.3.2 Programmierstil Modulares Programmieren . 6
2.3.3 Programmierstil Datenabstraktion 7
2.3.4 Objektorientierte Programmierung 9
2.4 Das erste C++ -Projekt 11
2.4.1 `fstack`: Ein einfacher Keller 12
2.4.2 Die Klasse `fraction` 14
2.4.3 `fcalc.cc`: Das Hauptprogramm 18
2.4.4 Ein Makefile für den Bruch-Rechner 20

3 Klassen 21

3.1 Klassendefinition 23
3.2 Zugriffskontrolle 26

3.3 Elementfunktionen (“Member Functions“) 28
3.4 Inline-Elementfunktionen 29
3.5 Der `this`-Zeiger 33
3.6 Konstante Elementfunktionen 34
3.7 Statische Klassenelemente 36
3.8 Freunde 38
3.9 Konstruktoren 39
3.9.1 Primitive Datentypen 39
3.9.2 Konstruktoren für Klassen 42
3.9.3 Der Default-Konstruktor 45
3.9.4 Der Copy-Konstruktor 48
3.10 Initialisierung $\Longrightarrow \Longleftarrow$ Wertzuweisung 51
3.11 Destruktoren 54
3.12 Vordefinierte Klassen: Die C++ I/O-Bibliothek . . . 58
3.12.1 Stream-Ausgabe 59
3.12.2 Stream-Eingabe 64
3.12.3 Ein/Ausgabe von Dateien 69
3.12.4 Ein/Ausgabe von und in Strings 73

4 Grundtypen und abgeleitete Typen 75

4.1 Die Grundtypen von C++ 75

4.1.1 Implizite Typumwandlung 77

4.1.2 Explizite Typumwandlung 79

4.1.3 Benutzer-definierte Umwandlung 80

4.2 Konstanten 82

4.2.1 Integer-Konstanten 83

4.2.2 Fließkomma-Konstanten 84

4.2.3 Textzeichen-Konstanten 84

4.2.4 Strings 85

4.2.5 Benannte Konstanten 86

4.3 Abgeleitete Typen 88

4.3.1 Der Typ `void` 90

4.3.2 Zeiger-Typen 91

4.3.3 Arrays 93

4.3.4 Zeiger und Arrays 95

4.4 Referenzen 97

4.4.1 Funktionen 100

4.4.2 Zeiger auf Funktionen 107

4.5 Sonstiges 108

4.5.1 Bitfelder 108

4.6 Wichtige Unterschiede zu ANSI C 109

5 Anweisungen und Ausdrücke 110

5.1 Anweisungen 110

5.1.1 Benannte Anweisung (Labeled Statement) . 110

5.1.2 Ausdrucksanweisung (Expression Statement) 111

5.1.3 Blöcke (Compound Statements) 111

5.1.4 Auswahlanweisung (Selection Statement) . . 111

5.1.5 Schleifen (Iteration Statements) 113

5.1.6 Unbedingte Sprünge (Jump Statements) . . 116

5.1.7 Deklarations-Anweisung (Declaration Statement) 118

5.2 Ausdrücke 119

5.2.1 Übersicht über die Operatoren 120

5.2.2 Einige ausgewählte Operatoren 122

6 Überladen von Funktionen und Operatoren 136

6.1 Überladen von Funktionen 137

6.2 Überladen von Operatoren 140

6.2.1 Ein vollständiges Beispiel 146

6.2.2 Überladen der Wertzuweisung 150

6.2.3 Überladen der Indizierung 151

6.2.4 Überladen des Elementzugriffs 152

6.2.5 Überladen des Funktionsaufrufs 154
6.2.6 Überladen von new und delete 156
6.2.7 Warnung 157

7 Schablonen 158

7.1 Funktionsschablonen 159
7.2 Klassenschablonen 165

8 Klassen-Hierarchien und Vererbung 172

8.1 Ein erstes Beispiel 174
8.2 Zugriff auf Basisklassen-Elemente 177
8.3 Zugriff auf Basisklassen 180
8.4 Konstruktion und Destruktion 181
8.4.1 Initialisierung durch Kopie 184
8.5 Typanpassung 186
8.5.1 Benutzer-definierte Typfelder 188
8.6 Virtuelle Funktionen 189
8.6.1 Implementierung mit Funktionstabellen . . . 191
8.6.2 Noch ein Beispiel 192
8.6.3 Regeln für die Benutzung von virtuellen Funktionen 193

8.6.4 Virtuelle Destruktoren 194
8.6.5 Abstrakte Basisklassen 195
8.7 Mehrfache Vererbung 196
8.7.1 Virtuelle Basisklassen 199
8.7.2 Mehrfache Vererbung und Typumwandlung . 206

9 50 Ways to Improve Your Programs 208
9.1 `const` und `inline` statt ♯define 208
9.2 `iostream.h` statt `stdio.h` 210
9.3 `new` und `delete` statt `malloc` und `free` 211
9.4 C++ Kommentare benutzen 213
9.5 Form der `new`- und `delete`-Aufrufe 214
9.6 `delete` für Pointer-Klassenelemente aufrufen 215
9.7 Ergebniswert von `new` testen 216
9.8 Überladen von `new` 217
9.9 Das globale `new` nicht überdecken 218
9.10 Selbstdefinierte `new/delete`-Operatoren 219
9.11 Copy-Konstruktor und `=` Operator 222
9.12 Initialisierung durch Wertzuweisung vermeiden . . . 223
9.13 Reihenfolge der Initialisierung 224
9.14 Virtuelle Destruktoren 224

9.15 Rückgabewert des = Operators 225
9.16 Member-Zuweisung im = Operator 225
9.17 Selbstzuweisung abtesten 227
9.18 Minimale Klassenschnittstellen 228
9.19 Element-, Freund- oder globale Funktionen 229
9.20 Vermeide öffenliche Datenelemente 231
9.21 `const` benutzen 232
9.22 Übergabe und Rückgabe per Referenz 234
9.23 Keine Referenzrückgabe für neue Objekte 236
9.24 Überladen ⇒ ⇐ Default-Parameter 238
9.25 Überladen von Pointer und numerischen Typen vermeiden 239
9.26 Vorsicht bei potentieller Mehrdeutigkeit 240
9.27 Compiler-generierte Elementfunktionen 241
9.28 Partitionierung des globalen Namensraumes 242
9.29 Konstante Elementfunktionen 243
9.30 Elementfunktionen, die Pointer auf weniger zugreifbare Elemente liefern 244
9.31 Keine Referenzen auf lokale Objekte zurückgeben . 245
9.32 Ganzahlige Klassenkonstanten 246
9.33 Benutze `inline` mit Bedacht 247

9.34 Abhängigkeiten zwischen Programm-Moduln 249
9.35 Öffentliche Vererbung modelliert “istein” 250
9.36 Vererbung von Schnittstelle und/oder Implementierung 251
9.37 Eine geerbte nicht-virtuelle Funktion nicht überschreiben 252
9.38 Geerbte Default-Parameterwerte nie ändern 253
9.39 Vermeide “downcasts” 254
9.40 Modelliere “hatein” durch Elemente von Klassentyp 257
9.41 Private Vererbung 258
9.42 Templates und Vererbung 259
9.43 Mehrfachvererbung 260
9.44 Verstehen was man tut 262
9.45 Compiler-generierte Funktionen 263
9.46 Laufzeit-Tests 264
9.47 Globale Objekte vor der Benutzung initialisieren . . 264
9.48 Compiler-Warnungen beachten 266
9.49 Geplante Spracherweiterungen im Auge behalten . . 267
9.50 Das Referenz-Manual lesen 267

10 Java: Überblick und Vergleich 268

10.1 Ursprünge 268
10.2 Was ist interessant an Java? 268
10.3 Java Programme und Java Applets 273
10.4 Unterschiede zwischen Java und C/C++ 275
10.4.1 Programmstruktur 275
10.4.2 Kommandozeilen-Argumente 275
10.4.3 Keine globalen Variablen und Funktionen . . 276
10.4.4 Kein Präprozessor 277
10.4.5 Unicode-Buchstaben 278
10.4.6 Grundtypen 278
10.4.7 Referenz-Semantik 279
10.4.8 Java hat keine Pointer 280
10.4.9 `null`: Keine Referenz 280
10.4.10 Objekt-Erzeugung 280
10.4.11 Garbage Collection 280
10.4.12 Die `synchronized` Anweisung 281
10.4.13 Ausnahmebehandlung (exception handling) . 281
10.4.14 Keine Structs, Unions, Aufzählungen 283
10.4.15 Form der Klassendefinition 283
10.4.16 Vererbung 285

10.4.17 Abstrakte Klassen 286
10.4.18 Interfaces 287
10.4.19 C++ -Konstrukte, die wir in Java nicht finden 289
10.5 Application Programming Interfaces 290
10.6 Java-Information auf dem Netz 291