Studentische Mitschrift zu

Programmieren in C und C++

WS 1997/98, SS 1998

Autor: Verena Schneider

Inhaltsverzeichnis

1 Einführung

1.1 Objektorientierung - Grundbegriffe

Objekt:

lt. Mayers Lexikon: allgemeiner Gegenstand (auch nichtdinglich) (des Interesses), Gegenstand, der einer Beobachtung, Untersuchung, Messung unterworfen ist.

Webster´s: etwas, das man sehen oder fühlen kann, das man denken, über das man nachdenken kann (Studienobjekt); das emotionale Wirkung hervorruft.

Sprachlehre: (Gegensatzpaar) Subjekt - Objekt; auf das hin etwas gerichtet ist

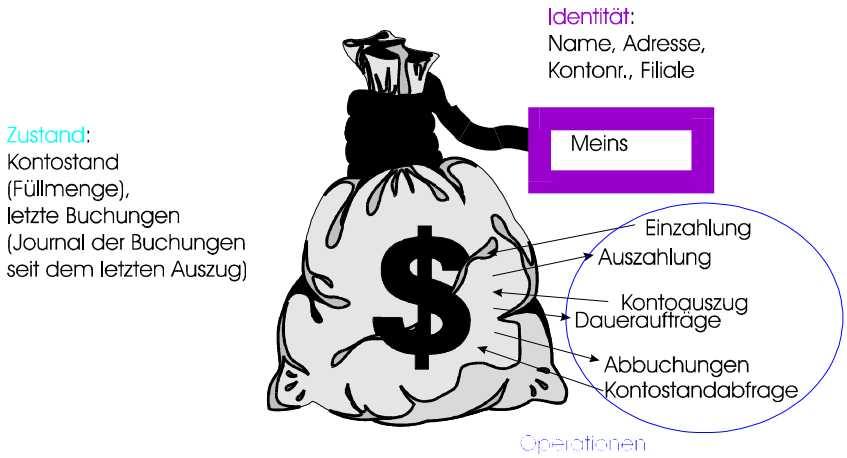
Objekt (im Sinne der OOP):

Erlenkötter: eine Variable (mit mehr Eigenschaften, als eine in C, Pascal, ...)
Horstmann: ein Objekt ist charakterisiert durch einen

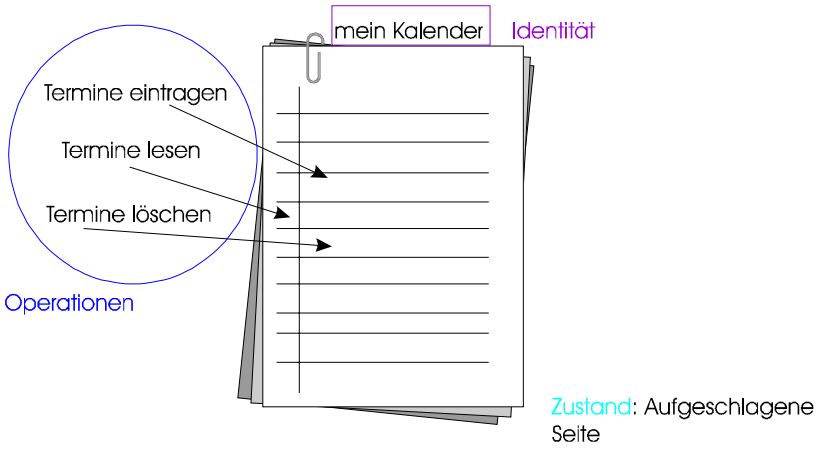
- Zustand
- Operationen
- Identität

Beispiele:

1. Bank-Konto



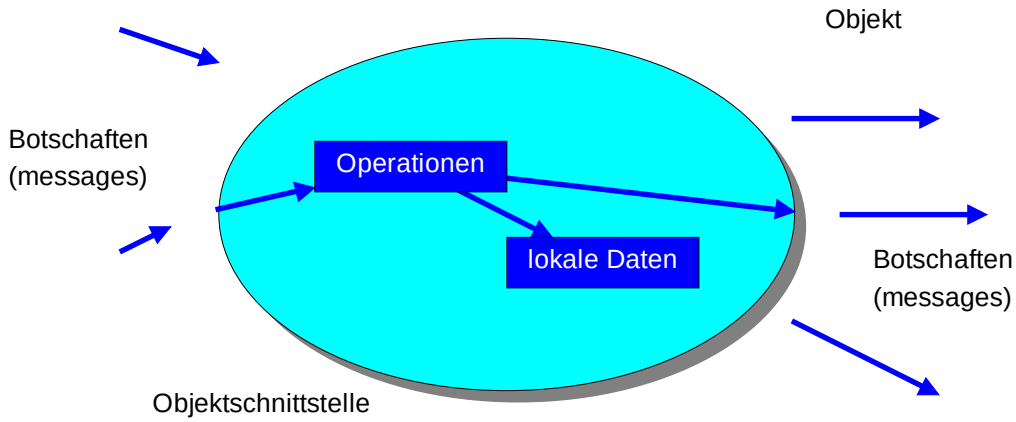
2. Terminkalender



3. Mailbox

Objekte sind: (Verbund-)Einheiten aus Daten und Operationen (Methoden) programmtechnische Realisierung abstrakter Datentypen.

Stichworte: Datenkapselung, Information-Hiding



Zustand ...
Operationen ...
Identität ...

Abstraktion und Modellierung:

- Objekte der realen Welt und ihre Beziehungen (Problembereich)
- Identifikation grundlegender Eigenschaften und Prinzipien
- Klassenbildung in der Modellwelt
- Individualisierung zu Objekten in der Modellwelt

Idee der OOP:
ein Programm besteht aus einer Menge (einfacher) miteinander kommunizierender Objekte (message passing)

1.2 Imperative Programmierung

(anweisungsorientierte Programmierung, prozedurale Programmiersprachen)

$a = b + c * c + 2 * (b - 1)$
einfuege (datei, zeile, spalte, zeichen)
$c = \mathrm{sqrt}(a * a + b * b)$
überwiegende Realisierung für Operationen / Methoden in Objekten

Programm:

```
/* der Anfang:
   Programm zur Ausgabe einer Zeile auf dem Bildschirm */
#include <iostream.h>
// eine Anweisung an den Präprozessor
// jetzt die Definition des Hauptprogramms, hier der
// main - Funktion in C
int main (void)   /* ANSI-C und Turbo-C++ */
{
 cout << "meine erste Zeile auf dem Bildschirm! "
         "\nund, weil es so schön ist gleich noch eine hinterher\n";
 return 0;
}
```

Erklärung:

/*...*/	Kommentar über mehrere Zeilen
//	Kommentar bis zum Zeilenende
main	Name der Funktion für das Hauptprogramm - mindestens einmal je Programm
int	Datentyp integer (ganze Zahl) für Rückgabewert der Funktion
void	(Parameter-)Typ, falls keine Übergabe- und Rückgabe-Parameter gewünscht
{...}	Block-Klammerung (Begin...End)
cout	Funktion Console Out (Ausgabe auf Console)
<< "..."	Einfüge-Operator für Zeichen "..."
\n	Pseudozeichenfolge (Escape-Sequenz) zur Steuerung der Ausgabe (neue Zeile)
return	Bestimmen des Rückgabewertes einer Funktion

`cout <<"..."`
Mit Hilfe dieser Funktion kann eine Zeichenfolge auf dem Monitor ausgegeben werden.

`cout << "Variable 1...."<< Variable1`
Ausgabe eines Variablenwertes auf dem Monitor.

`cin >> Variable1`
Der Benutzer wird mit diesem Befehl aufgefordert, einen Wert für die Variable 1 einzugeben.

`cin >> Variable1 >> Variable2`
In diesem Fall muß der Benutzer für mehrere Variablen einen Wert eingeben. Die Werte für die Variablen werden durch ein Leerzeichen getrennt.

`\n`
Mit diesem Befehl innerhalb einer Zeichenfolge wird der Cursor in die nächste Zeile bewegt.

```
cout << "\n Hallo"
→↵
   ≤Hallo
cout << "\nHallo"
→↵
   Hallo
```

\t

Tabulator

1.3 Aufgaben

1. Schreiben Sie ein einfaches Programm zur Ausgabe des Textes.

Bumerang
War einmal ein Bumerang;
War ein wenig zu lang.
Bumerang flog ein Stueck,
aber kam nicht mehr zurueck.
Publikum - noch stundenlang -
wartete auf Bumerang.
Ringelnatz

Programm:

```
#include <iostream.h>
int main (void)
{
 cout << "Bumerang";
 cout << "\nWar einmal ein Bumerang";
 cout << "\nWar ein weniges zu lang";
 cout << "\nBumerang flog ein Stueck";
 cout << "\naber kam nicht mehr zurueck";
 cout << "\nPublikum - noch stundenlang -";
 cout << "\nwartete auf Bumerang.";
 cout <<"\n                         Ringelnatz";
 return 0;
}
```

2. Wie könnte ein einfaches Programm aussehen, das die Berechnung der Hypothenuse aus den Katheten eines RWD durchführt? (Mathematik Funktionen in math.h)

```
// phytagoras
#include <iostream.h>
#include <math.h>
int main (void)
{
float a, b;
float c;
cout <<"\nBitte geben Sie einen Wert für a ein: "; cin >> a;
cout <<"\nBitte geben Sie einen Wert für b ein: "; cin >> b;
c=a*a+b*b;
c=sqrt(c);
cout<<"\n\nHier ist der berechnete Wert für c: "; cout<<c;
return 0;
```

2 Präprozessorbefehle

2.1 Präprozessor

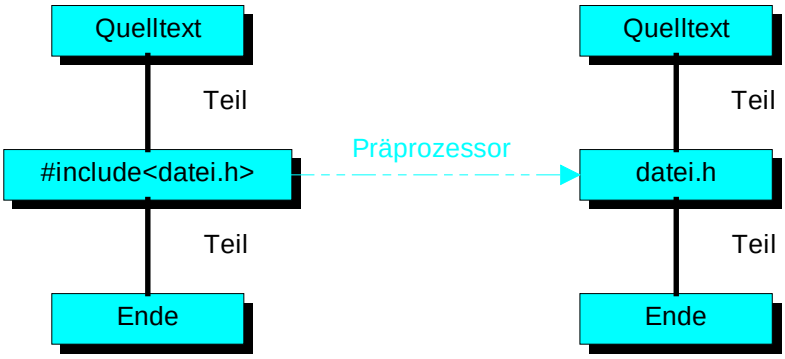
Prä- vor; Prozessor - Verarbeitungsphase (hier, sonst VE)

Bearbeitung des Quelltextes vor der eigentlichen Übersetzung zur Syntaxanalyse und Codeerzeugung

#include <headerdatei.h>

Mit diesem Befehl wird eine Headerdatei vom Präprozessor eingefügt. In einer Headerdatei sind bestimmte Funktionen und Konstanten schon deklariert. Um auf diese vorgefertigten Funktionen zurückgreifen zu können, muß man die jeweilige Headerdatei einbinden. Headerdateien haben die Erweiterung ".h". Der Präprozessor sucht die Headerdatei im include-Verzeichnis.

Der Präprozessor bearbeitet den Quelltext bevor er übersetzt wird. Er bindet die Headerdateien folgendermaßen in den Quelltext ein:



Die Headerdatei wird genau dort eingefügt, wo die include-Anweisung steht, also vor der "Mainfunktion". Das Programm verlängert sich um die Länge der Headerdatei.

#include "headerdatei.h"

Auch in diesem Fall wird eine Headerdatei vom Präprozessor in das Programm eingefügt. In diesem Fall sucht er jedoch zuerst im aktuellen Verzeichnis, erst danach im include-Verzeichnis.

Hinweis:

Im neuen Sprachstandard von C++ wurde festgelegt, daß Standard-Include-Dateien ohne die Endung .h verwendet werden. Bei einigen Compilern gibt es jedoch noch beide Versionen, also z.B. iostream und iostream.h. Hier unbedingt im Handbuch nachsehen und die gewählte Version bebehalten, also entweder alles ohne oder mit .h!

Beispiel:

```
iostream.h ⇒ iostream
```

#define Konstantenname Wert

Mit diesem Befehl kann eine Konstante, durch textuelle Ersetzung, definiert werden. Wenn im Programm die Konstante aufgerufen wird, so setzt der Computer immer den definierten Wert der Konstante ein.
(Siehe auch Kapitel 3.3)

Bedingte Compilierung mit #if, #elif, #else, #endif

Mit Hilfe der bedingten Compilierung kann man durch Veränderung eines Konstantenwertes gleich mehrere andere Konstanten verändern. Dies kann viel Arbeit ersparen.

Wenn man if und elif verwendet, dann wird die if-Unterscheidung abgebrochen, sobald eine Anweisung erfüllt ist. Es kann also nur eine Anweisung zutreffen.

Beispiel:

```
#define Test 12
#if Test==10
 #define Maxwert 99
#elif Test==11
 #define Maxwert 100
#elif Test==12
 #define Maxwert 101
#else
 #define Maxwert 50
#endif
```

Wenn man in diesem Beispiel die Konstante Test ändert, dann wird gleichzeitig auch die Konstante Maxwert geändert. Diese Anweisung wird effektiver, je mehr Konstanten durch die Veränderung von Test ebenfalls verändert werden müssen.

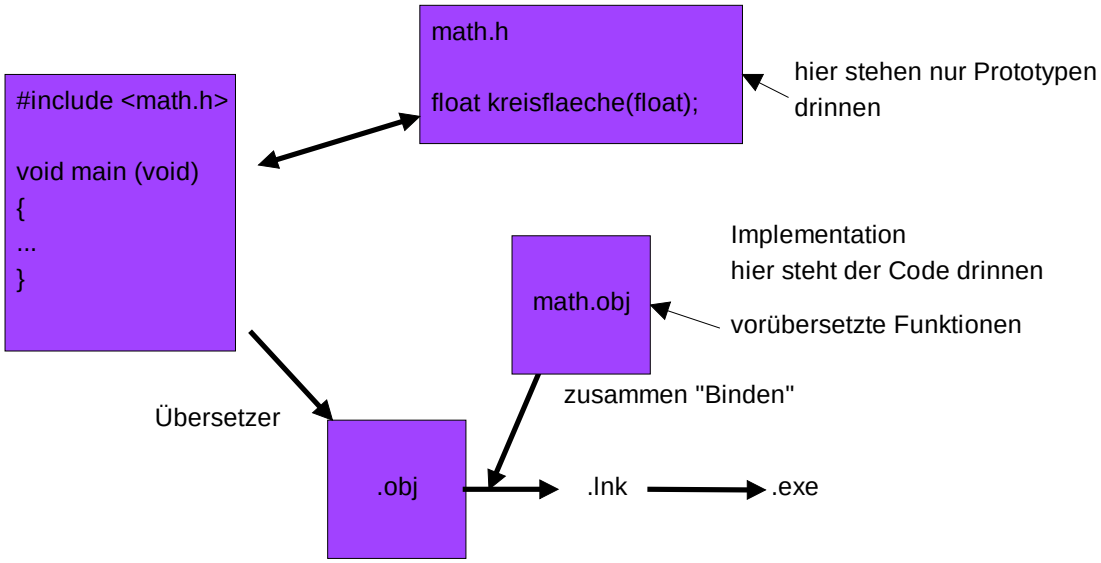
Definition überprüfen mit #ifdef, #ifndef

Hiermit wird überprüft, ob eine Konstante bereits definiert bzw. noch undefiniert ist.

Beispiel:

```
#define Demo 0

#ifdef Demo
 #define Version 0.9
#else
 #define Version 1.0
#endif
```

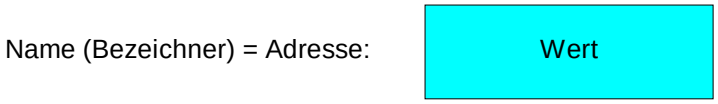
2.2 Die Verarbeitung eines Quellcode mit Hilfe des Präprozessors, Compilers und Binders



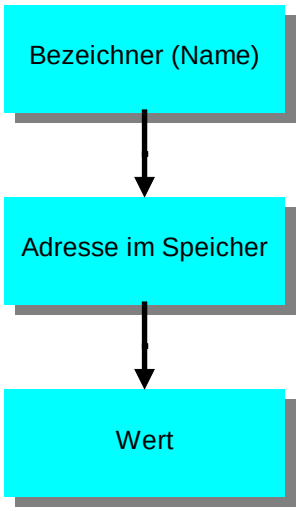
3 Variablen, Konstanten und Co.

3.1 Variablenmodell der imperativen Programmiersprachen

Variablen: Behältermodell



einfache skalare Variablen



Jede Variable besitzt einen Bezeichner (Name), diesem Bezeichner wird eine Adresse im Speicher zugewiesen. Der Speicheradresse wird dann der Wert der Variable zugewiesen.

In der Programmierung ist eine Variable ein Speicherplatz (Behälter), der Wert der Variablen kann sich mit der Zeit verändern. Der Zugriff auf den Wert ist über den Bezeichner und Adresse möglich.
In der Mathematik hingegen ist die Variable ein Platzhalter. Sie verallgemeinert oftmals Gleichungen.
Bsp.: $E = mc^2$

3.1.1 Deklaration

Bevor man Variablen im Programm verwenden kann, muß man sie deklarieren. Bei der Deklaration wird der Name der Variable eingeführt und jede Variable wird einem Datentyp zugewiesen. Dieser Datentyp muß dem Inhalt bzw. der Art der Variable entsprechen. Weiß man

schon, daß man verschiedene Variablen miteinander verknüpfen will, so müssen die entsprechenden Variablen, bezüglich dieser Operationen verträglich sein.

Beispiel:

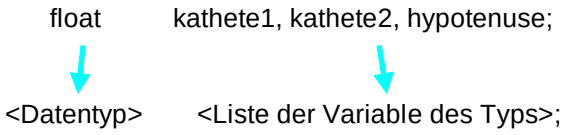
Es soll eine Variable ganz deklariert werden, die nur ganze Zahlen beinhalten soll:
```
int ganz;
```

Veränderungen des Wertes in der Zeit
Zugriff auf Wert über Bezeichner und Adresse möglich
Lebenszeit und -Dauer abhängig von Gültigkeitsbereich

Programmvariablen:

- müssen deklariert sein, d.h. ihre Namen müssen vor Benutzung eingeführt werden
- besitzen einen Datentyp
- die miteinander verknüpft werden, müssen verträglich Typen besitzen

Beispiel:



3.1.2 Warum unterschiedliche Datentypen?

Spezifische Operationen für unterschiedliche Datendarstellungen
(generische Operationen)

zum Beispiel:
interne Darstellung ganzer Zahlen ≠ interner Darstellung von Gleitkommazahlen
⇒ Addition ganzer Zahlen ≠ Addition von Gleitkommazahlen
...
bei Zahlentypen Verträglichkeit durch Konvertierbarkeit

Beispiel:

```
// Deklaration von Variablen für ganze Zahlen
int zaehler, nenner;
 // Deklaration von Gleitkommavariablen
float ergebnis, faktor;
 // Beispiel für Anweisung mit Typanpassung:
zaehler = 8;
nenner = 127;
faktor = 1.5;
ergebnis = zaehler / nenner * faktor;
// zaehler und nenner sind zwei integer-Variablen, es kommt also zuerst
// zur sog. Integerdivision, bei dem es keinen Rest gibt. In diesem Fall
// also 0. Multipliziert mit einem faktor ergibt das Ergebnis immer noch 0.
```

Wie man sieht unterscheidet der Compiler zum Beispiel bei einer Division die verschiedenen Datentypen eine Integerdivision wird anders ausgeführt als eine Floatdivision. Erst wenn man die Variablen "zaehler" und "nenner" durch ein sogenanntes Typcasting in eine Floatvariable umwandeln würde oder sie gleich als solche deklariert kommt ein Ergebnis ungleich 0 heraus.

Programm zur Berechnung der Hypotenuse aus den Katheten eines RWD:

- Eingabe von Daten (Parametern) in ein Programm
  cin >> var1 >> var2 >>...;
- Deklaration der Variablen: ...
- Bildschirmdialog zur Dateneingabe: ...

```
/* Programm Pythagoras --
berechnet die Hypotenuse eines rechtwinkligen Dreiecks aus den
Katheten */
#include <iostream.h>;
#include <math.h>; /* Headerfile für mathematische Funktionen */

int main (void)
{
 float kathete1, kathete2, hypotenuse;
 cout << "Bitte gib die Katheten ein: "<< endl;
 cin >> kathete1 >> kathete2;
 hypotenuse=kathete1*kathete1+kathete2*kathete2;
 hypotenuse=sqrt(hypotenuse);
 /* einfaches Ausgabeformat: */
 cout << "\nKathete 1 : \t" << kathete1
      << "\nKathete 2 : \t" << kathete2
      << "\nHypotenuse : \t" << hypotenuse << endl;
 return 0;
}
```

3.1.3 Wertzuweisung

- Zuweisungsoperator =
- rechte Seite zum Beispiel arithmetischer Ausdruck

3.1.4 Anweisung

- <ausdruck>;
- Semikolon erzeugt aus Ausdruck eine Anweisung
- x = 2 Zuweisung + Ausdruck
- x = 2; Anweisung
- a + 3; Anweisung

3.2 Datentypen

3.2.1 Übersicht

Man sollte immer den Datentyp verwenden, der für die jeweilige Variable am ehesten geeignet ist. Verwendet man Typen, die einen zu kleinen Wertebereich haben, so kann es leicht zu Überläufen und somit zu fehlerhaften Berechnungen kommen.
Anders als zum Beispiel in Pascal meldet sich C nicht bei einem Überlauf mit einer Fehlermeldung. Im Falle eines Überlaufes zum Beispiel nach oben zählt der Compiler einfach bei den negativen Zahlen weiter und umgekehrt. Man kann also einen Überlauf in C nur erkennen, wenn man feststellt, daß eine negative bzw. positive Zahl ausgegeben wird, wenn nach der richtigen Berechnung eigentlich eine positive bzw. negative Zahl ausgegeben werden sollte.
Aber auch die Verwendung von Variablentypen mit einem zu großen Wertebereich kann Probleme mit sich bringen, denn es wird mehr Speicherplatz benötigt und das Programm kann sich verlangsamen (Bsp.: Bei der Verwendung von long, anstelle von int, kann das Programm Einbußen an Rechengeschwindigkeit haben).
Deshalb sollte man stets darauf achten, daß man die richtigen Variablentypen verwendet.

Hier ist ein Auflistung der verschiedenen Typen:

Variablentyp	Mind. Anz. Bytes	Wertebereich	Bemerkung
signed char char	1	[-128, 127]	Ganzzahl mit Vz, Zeichen
unsigned char	1	[0, 255]	Zeichen vzlose Ganzzahl
short signed short signed short int short int	2	[-32768, 32767]	mind. so groß wie char; Unter MS-DOS sind short int identisch mit int
unsigned short unsigned short int	2	[0, 65535]	so groß wie short
signed int signed int	2	[-32768, 32767]	mind. so groß wie short int; Unter MS-DOS ist int identisch mit short int
unsigned unsigned int	2	[0, 65535]	so groß wie int
signed long int signed long long int long	4	$[-2^{31}, 2^{31}-1]$	mind. so groß wie int
unsigned long int unsigned long	4	$[0, 2^{32}-1]$	so groß wie long

Datentyp	Bytes (Turbo C)	Mantisse	Bemerkung

float	4	23 Bits + Vz	Mindestgen. 6 Stellen Turbo-C; 7 Stellen
double	8	52 Bits + Vz	Mindestgen. 10 Stellen mind. wie float Turbo:15 Stellen
long double	10	63 Bits + Vz	mind. wie double Turbo: 19 Stellen

Größe der Variablentypen:

$1 \equiv \text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$
$1 \leq \text{sizeof(bool)} \leq \text{sizeof(long)}$
$\text{sizeof(char)} \leq \text{sizeof(wchar\_t)} \leq \text{sizeof(long)}$
$\text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}$
$\text{sizeof(N)} \equiv \text{sizeof(signed N)} \equiv \text{sizeof(unsigned N)}$

3.2.2 Datentyp integer, short int und Co.

Es handelt sich hierbei um negative und positive ganze Zahlen.
Im Zehnersystem dürfen die Ziffern 0-9, im Oktalsytem 0-7 (es muß ganz vorne eine 0 stehen, um die nachfolgende Zahl als Oktalzahl zu definieren) verwendet werden und im Hexadezimalsystem sind die Ziffern 0-9 und die Buchstaben a-f (oder A-F) erlaubt (es muß vor der Zahl 0x oder 0X stehen).
Es dürfen außer dem Minuszeichen keinen weiteren Zeichen vorkommen.

Bei den Variablentypen ohne Vorzeichen (unsigned ...) ist das Minuszeichen nicht erlaubt.

Beispiele:

```
int i;       // Deklaration einer Integer-Variable
unsigned short int j;
i=5;         // richtige Deklaration ⇒ positive Zahl
i=-5;        // richtige Deklaration ⇒ negative Zahl
i=050;       // richtige Deklaration ⇒ 50 ist eine Oktalzahl
i=0x5a;      // richtige Deklaration ⇒ 50 ist eine Hexadezimalzahl
i=5.0;       // falsche Deklaration, weil ein Punkt nicht erlaubt ist
i=5,0;       // falsche Deklaration, ein Komma ist nicht erlaubt
j=-5;        // falsche Deklaration, da j negativ ist
```

Es ist meistens besser einen Integer zu verwenden, anstelle von vorzeichenlosen Zahlen.

3.2.3 Datentyp float, double und long double

Es handelt sich hierbei um Gleitkommavariablen.
Neben den Ziffern 0-9 ist auch das Minuszeichen und "e" oder "E" erlaubt. "e" oder "E" stehen für die Zehnerpotenz einer Zahl.
Die Nachkommastellen werden durch einen Punkt vom ganzzahligen Anteil abgetrennt.

Beispiele:

```
float i;           // Deklaration einer float-Variable
i=0.1;             // richtig
i=.1;              // richtig äquivalent zu 0.1
i=2e-8;            // richtig äquivalent zu 2E-8
i=0.222e-10;       // richtig
i=1;               // falsch, da weder eine Nachkommastelle noch ein
                   // Exponent vorhanden ist
i=2 E 10;          // falsch, Leerzeichen sind nicht erlaubt
i=1,000.0;         // falsch, Komma ist nicht erlaubt
i=2E+10.2;         // falsch, der Exponent muß ganzzahlig sein
```

Standardmäßig ist eine Gleitkommazahl ein double. Um dennoch eine Zahl vom Typ float abzuspeichern, benötigt man den Suffix f oder F.

Beispiel:

```
3.145f, 4.2F
```

Man sollte bevorzugt double verwenden, da die Genauigkeit dort höher liegt.

3.2.4 Datentyp char

```
// Programm zeibsp.cpp zur Verwendung von Zeichen
#include <iostream.h>
void main (void)
{
 int zahl;
 char x=65, y='B';

 zahl=234;
 cout << "\n" << x << " " << y << " ";
 cout << "\n" << (x+y);
 cout << "\n" << zahl;
}
```

→ Ausgabe:

```
A B
131  234
```

Zeichenbehandlung im Programm zeibsp.cpp:

- Deklaration von Zeichenvariablen: Typbezeichner char (character)
- Vorbesetzung der Variablen mit
    - Zahl-Code = Zahlkonstante=ASCII-Zahl
    - Zeichenkonstante
- Verwendung in arithmetischem Ausdruck ⇒ Zahl-Interpretation
- Verwendung unmittelbar in Ausgabeanweisung ⇒ Zeichen-Interpretation

3.2.5 Zeichenketten

Feld von Zeichen

Strings (Zeichenketten) werden in C / C++ durch ein Feld (siehe Kapitel 3.2.1) aus Zeichen hergestellt.

Man sollte bei der Initialisierung von Strings folgendes beachten:

- Jede Zeichenkette wird mit der binären Null "\0" abgeschlossen.
- Die Länge des Feldes muß um eins größer sein, als die maximale Anzahl einzulesender Zeichen. ⇒ Das letzte Feldelement wird benötigt um dort die binäre 0 hineinzuschreiben.

Beispiel:

```
char plz_falsch[5];    // Deklaration eines Strings mit 5 Elementen für eine fünfstellige PLZ
char plz_richtig[6];   // Deklaration eines für PLZ, aber mit 6 Elementen
...
cin >> plz_falsch;
cin >> plz_richtig;
```

- Wenn man von Anfang an weiß, das man nur ein einziges Zeichen eingeben möchte, so sollte man dieses nicht über einen String eingeben lassen, da man dort den doppelten Speicherplatz benötigt wie für die einfache Deklaration (eine Speicherzelle für Buchstaben und eine zweite für die binäre Null "\0"). Es besteht deshalb auch ein Unterschied zwischen der Eingabe ‘w’ und der Eingabe "w". Bei ‘w’ handelt es sich um ein einzelnes Charakter, bei "w" handelt es sich um einen String ("w" benötigt doppelt soviel Speicher wie ‘w’)

Die Zeichenkette "Hochschule" wird dem Charakterfeld "String" mit 11 Elementen zugwiesen.

```
char string[12];
strcpy (string, "Hochschule");
```

Es kommt nun zum folgenden "internen" Ergebnis:

Nummer des Buchstaben	**Feldnummer**	**Feldinhalt**
1	String[0]	H
2	String[1]	o
3	String[2]	c
4	String[3]	h
5	String[4]	s
6	String[5]	c
7	String[6]	h
8	String[7]	u
9	String[8]	l
10	String[9]	e
11	String[10]	/0

Lange Zeichenketten können durch Whitespaces unterbrochen sein, während man sie einer Variable zuweist.

Beispiel:

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

void main (void)
{
 char zeichenkette1[]="Bernd"
                      " Schneider";
 char zeichenkette2[]="Verena "   "Schneider";
 cout << zeichenkette1 << endl;
 cout << zeichenkette2 << endl;
```

```
}
```

Ausgabe:

Bernd Schneider
Verena Schneider

Unterschied Zeiger auf Charakter und Zeichenkette

...

Strings (Neu in C++)

...

3.3 Datentyp Bool

Der Datentyp Bool ist eine neue Variablenart in C++.
Mit ihr kann man Variablen die Werte true und false geben. Man verwendet diesen Datentyp zum Beispiel bei Vergleichen. Wenn man einer boolschen Variable einen Zahlenwert gibt (integer), dann werden Zahlenwerte, die ungleich Null sind, zu true (entspricht 1) und die Null zu false (entspricht der 0).

Beispiele:

```
void test (void)
{
 int a=5, b=-5;
 bool c;
 c= a==b; //falls a = b ist, dann erhält c den Wert true (entspricht 1),
          // ansonsten den Wert (false)
 c=a+b;   // c erhält den Wert false (a+b=0)
 c=a-b;   // c erhält den Wert true (a-b=10)
}
```

Beim Arbeiten mit Zeigern konvertiert ein NULL-Zeiger zu false und jeder andere zu true.

3.4 Oktale, Dezimale und Hexadezimale Zahlen

In C kann man nicht nur dezimale Zahlen darstellen, sondern auch Oktale und Hexadezimale Zahlen.
Oktale Zahlen besitzen als erstes Zeichen eine 0, hexadezimale Zahlen eine 0x.

dezimal:	0	2	63	83
oktal:	0	02	077	0123
hexadezimal:	0x0	0x2	0x3f	0x53

3.5 Namenskonventionen

- Der Name einer Variable, Konstante kann grundsätzlich beliebig lang sein. Es werden aber nur die ersten 32 Zeichen vom Compiler erkannt.

Beispiele für Variablennamen:

```
hallo_mir_geht_es_gut_und_dir_hoffentlich_auch   (46 Buchstaben)
hallo_mir_geht_es_gut_und_dir_hoffentlich_nicht  (47 Buchstaben)
```
Diese beiden Variablen sind für den Compiler gleich.

- Buchstaben, Ziffernstrich und Unterstrich "_" dürfen verwendet werden. Ziffern dürfen nicht am Anfang des Variablennamen stehen und der Unterstrich sollte - wenn möglich - auch nicht dort stehen. Den Unterstrich sollte man also, wenn überhaupt, nur in der Mitte des Namens verwenden, um zum Beispiel zwei Wörter logisch voneinander zu trennen.
- Die Groß- und Kleinschreibung ist relevant. ⇒ Aufpassen bei der Verwendung von Variablen!

Beispiel:
```
int Integer, integer;    // Dies sind zwei vollkommen unterschiedliche
                         // Variablen
```
- Man sollte nur aussagefähige Namen verwenden:

Beispiel:
```
int x;
besser
int zaehler;
```
- Man sollte bei den Variablennamen schwer unterscheidbare Buchstaben und Ziffern (als Unterschiede) vermeiden.

Beispiel:
```
int Oma1;   // endet mit Ziffer '1'
int Omal;   // endet mit Zeichen 'l'
```

- Man sollte den Konstanten- bzw. Variablennamen nicht zu kurz und nicht zu lang wählen. Ist der Name zu kurz, so kann ein zweiter Programmierer nur schwer die Aufgabe der Variable verstehen. Ist aber die Variable zu lang, so muß viel getippt werden und die Wahrscheinlichkeit, einen Fehler zu tippen steigt. Deshalb sollte man kurze aber ersichtliche, logische Variablennamen verwenden.

Beispiel:
```
int Saldo_des_Kontos_bei_der_Dresdner_Bank;   // zu lang
int Konto_Dreba;                              // sag alles, verständlich
int KD;                                       // zu kurz
```
- Schlüsselwörter dürfen nicht als Variablen- bzw. Konstantennamen verwendet werden.

Beispiele:
```
auto   break   case   char   const   continue   default   do   double
else   enum   extern   float   for   goto   if   int   long   register
return   short   signed   sizeof   static   struct   switch   typdef
union   unsigned   void   volatile   while
```

3.6 Typumwandlung

In vielen Fällen wird die Typumwandlung schon vom Compiler selbst vollzogen, so daß sie nicht manuell gemacht werden muß.

Beispiele:
```
int i, j;
long ix;
short s;
float x;
double dx;
char c;
```

```
i + c    ⇒ integer Variable
x + c   ⇒ float Variable
dx + x ⇒ double Variable
i + x    ⇒ float Variable
```

Es gibt aber ein paar Fälle, in denen man eine manuelle Typumwandlung machen muß. (sog. Typcasting)

Syntax:

```
(neuer Datentyp) (umzuwandelnder Ausdruck)
```

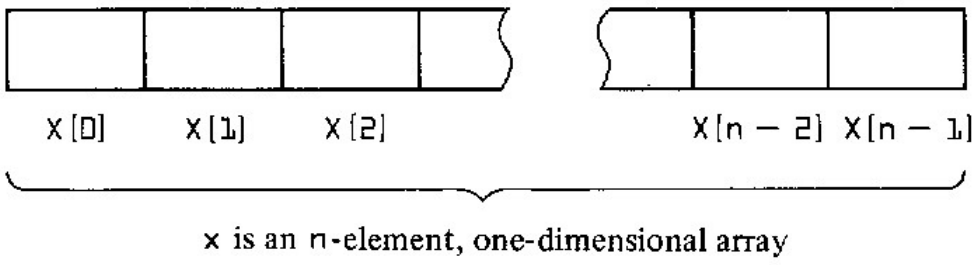
Beispiel:

```
int intvar = 10;
long longvar;
...
longvar = ((long)intvar)+5;
```

3.7 Zusammengesetzte Variablen

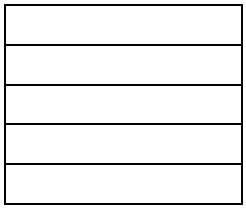
3.7.1 Felder (homogen)

eindimensionale Felder:



x is an n-element, one-dimensional array

Sie sind zusammengesetzt aus Variablen mit dem gleichen Datentyp.
Bei der Deklaration bekommt das Feld einen Bezeichner, es wird die Anzahl der Elemente angegeben und der Typ der einzelnen Elements (des Feldes).
In Turbo-Pascal entspricht dies der Deklaration eines Arrays.



Zu beachten ist, daß das erstes Element des Feldes den Index 0 hat. Es wird also nicht bei 1 angefangen zu zählen.

Anwendungsbereich: Matrizen, Felder, Tabellen

Beispiel:

Es wird ein Feld mit 5 Elementen deklariert. Es soll vom Typ integer sein und den Namen vektor haben.

```
int vektor [5];
```
⇒ Es gibt die Indizes 0 bis 4.
Vektor [0], vektor [1], vektor [2], vektor [3], vektor [4]

Man kann den einzelnen Elementen des Feldes gleich bei der Deklaration einen Wert zuweisen.

Beispiel:

```
int vektor [5] = {1,2,3,4,5};
```

→ vektor [0] = 1, vektor [1] = 2, vektor [2] = 3, vektor [3] = 4, vektor [4] = 5
{ ... } bezeichnet man als Initialisierung.

n<m:
Stehen in der Initialisierung n Elemente, hat das Feld jedoch m Elemente, so werden den ersten n Elementen des Feldes ein Wert zugewiesen. Die restlichen m-n Elemente des Feldes werden mit 0 initialisiert.

Beispiel:

```
int vektor [5] = {1,2}
```
→ vektor [0]=1, vektor[1]=2, vektor[2]=vektor[3]=vektor[4]=0

Beispiel:

Um in alle Elemente eines Feldes 0 hineinzuschreiben benötigt man keine Schleife. Man kann ein Feld auch anders initialisieren. Man muß jedoch dem ersten Element des Feldes eine 0 zuweisen, da der Compiler keine leere Menge annimmt.

```
int vektor[5]={0}
```
vektor[0]=vektor[1]=vektor[2]=vektor[3]=vektor[4] = 0;

mehrdimensionale Felder:

Mehrdimensionale Felder werden wie folgt deklariert:

```
dateityp dateiname [x1] [x2] [x3] ... [xn]
```

Zweidimensionale Felder (mit x_1 und x_2) kann man sich als Tabelle mit x_1 Zeilen und x_2 Spalten vorstellen. Dreidimensionale Felder kann man sich im Raum vorstellen, bei Feldern höherer Ordnung wird eine visuelle Vorstellung schwierig.

Beispiel:

Ein Feld mit 3 Zeilen und 2 Spalten. Da man in einem Feld von links nach rechts und von oben nach unten geht, wächst die zweite Variable, die für die Anzahl der Spalten steht schneller als die, die für die Anzahl der Zeilen steht.

```
int feld [3] [2];
feld[0][0] = 5; //allererstes Element (bildlich: Element links oben in der
                // Ecke)
feld[1][0] = 2; // erstes Element in der 2.Zeile
feld[2][1] = 3; // letztes Element in der letzten Zeile
```
Auch mehrdimensionalen Feldern kann man schon bei der Definition Werte zuweisen. Hierfür gibt es mehrere Möglichkeiten.

Beispiel:

```
int feld [2] [2]={1,2,3,4};
```

→feld[0] [0]=1 feld[0][1]=2
  feld[0] [1]=3 feld[1][1]=4

Beispiel:

```
int feld [2] [2]={1,2,3};
```

→feld[0] [0]=1 feld[0][1]=2
  feld[0] [1]=3 feld[1][1]=0

Beispiel:

```
int feld [2] [2]={ {1,2} , {3}};
```

→ feld [0][0]=1 feld[0][1]=2
  feld [1][0]=3 feld[1][1]=0

Beispiel:

Hier wird in das Feld Wort der String NORTH hineingeschrieben. Er wird mit einem \0 abgeschlossen.

```
char wort[ ]= "NORTH";
```

Beispiel:

In ein Feld werden die Charakters (Einzelbuchstaben) "N", "O", "R", "T", "H" hineingeschrieben. Das Feld enthält kein \0. Es hat also nur 5 Elemente.

```
char wort[ ]={´N´,´O´, ´R´,´T´, ´H´};
```

Beispiel:

```
char wort[6]={´N´,´O´, ´R´,´T´, ´H´};
```

→ In den Elementen 0 bis 4 stehen wieder die gleichen Buchstaben wie im obigen Beispiel, aber das Element 5 wird nun mit 0 initialisiert.

3.7.2 Strukturen und Unions (inhomogen)

Es handelt sich hier um einen Verbund von Variablen, die zu einem Paket gebündelt werden. Die einzelnen Variablen des Pakets haben unterschiedliche Typen.
Dies entspricht den *records* in Turbo-Pascal.

Anwendungsgebiet: Datenbankelemente, Personal-DB

Strukturen deklariert man mit Hilfe der struct-Anweisung.

Beispiel:

```
struct artikeltyp
{
 int    ArtikelNummer;
 char   ArtikelName[20];
```

```
 double Umsatz;
};
// Der Strichpunkt steht am Ende, weil man nach der Klammer noch
// Variablen vom Typ artikeltyp deklarieren kann. In diesem Fall ist keine
// deklariert, aber der Strichpunkt muß trotzdem dastehen.
...
// jetzt muß noch eine Variable deklariert werden
artikeltyp artikel;
```

Beispiel:

```
struct artikeltyp
{
 int    ArtikelNummer;
 char   ArtikelName[20];
 double Umsatz;
} artikel;
```

// In diesem Fall wird gleich die Variable artikel vom Typ artikeltyp deklariert.

In diesem Beispiel ist artikel der Strukturbezeichner. artikel besteht aus dem Verbund von drei Variablen. Einer Integer-Variable, einer Zeichenkette und einem double-Variable.

Eine Strukturdeklaration ähnelt einer Typendeklaration. Zum Zeitpunkt der Deklaration der Struktur sind noch keine Variablen definiert. Die passende Variablendeklaration wird im folgenden Beispiel erörtert.

Beispiel:

```
artikel Jacke, Hose, Schuhe;
```

→ Die Variablen Jacke, Hose und Schuhe sind vom Typ "artikel".
Der Zugriff auf die einzelnen Struktur-Komponenten erfolgt durch den Strukturoperator ".".

Beispiel:

```
Jacke.ArtikelNummer = 5555;
strcpy (Jacke.ArtikelName,"Jeans Jacke");
```

→ Es werden den einzelnen Elementen des Datenverbunds Werte eingegeben. Auf die gleiche Weise kann man auch wieder den Wert auslesen und weiter verarbeiten.

Zugriff auf die Struktur-Komponenten durch den Strukturoperator ".".
Wie bei "normalen" Variablen gibt es auch bei Strukturen die Möglichkeit sie gleich zu initialisieren.

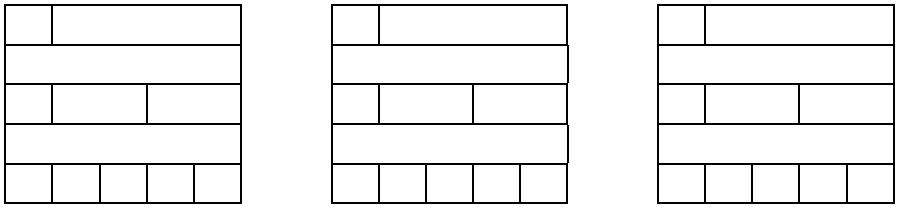
Beispiel:

```
struct adresse
{
 char *name;
 long int number;
 char *street;
};
adresse jd={"Jim Dandy", 61, "South St"}
```

3.7.3 Kombination von Felder und Strukturen

Es werden Felder gebildet, deren einzelne Elemente Strukturen sind. Die Elemente untereinander sind gleich, aber die Variablen, aus denen die einzelnen Elemente bestehen , können unterschiedliche Typen haben.

Anwendungsgebiet: Tabellen, Datenbanken



Eine Kombination von Feldern und Strukturen liegt vor, wenn der Typ eines Feldes mit struct deklariert wurde.

Beispiel:

Es soll ein Feld mit 5 Elementen (Index 0 bis 4) erstellt werden, das den Namen "umsatzliste" hat.

```
artikel umsatzliste [5];
```

3.8 Aufzählung

Eine Aufzählung ist wie eine Struktur oder eine Union ein Variablentyp.
Syntax: enum <enum_typname> {<Liste der Aufzählungskonstanten>}.

Beispiel:

```
enum tag {So, Mo, Di, Mi, Do, Fr, Sa};
```

Jeder Aufzählungskonstante erhält intern einen Integer-Wert.

Beispiel:

So=0, Mo=1, Di=2, ...

Eine Aufzählungskonstante kann initialisiert werden. Der Wertebereich den die Aufzählung dann umfaßt beträgt, falls die kleinste Zahl nicht kleiner als eins ist, die nächstgrößere Zweierpotenz minus eins.

Beispiel:

```
enum e1 {dunkel, hell}  // Bereich 0 : 1
enum e2 {a=3, b=9}    // Bereich 0 : 15
enum e3 {min=-10, max=1000000}  // Bereich –1048576 : 1048575
```

Konvertierung von Werten eines Integertyps in einen Aufzählungstyp.

Beispiel:

```
enum flag {x=1, y=2, z=4, e=8}   // Bereich 0 : 15
flag f1=5;          // Typfehler: 5 ist nicht vom Typ flag
flag f2=flag(5);   //OK: flag(5) ist vom Typ flag und im Bereich von flag
flag f3=flag(99);  // undefiniert: 999 ist nicht im Bereich von  flag
```

3.9 Konstanten

3.9.1 textuelle Konstanten

Sie werden mit #define definiert. Nach dem define steht der Name der Konstante und dann der Wert: #define max 1000. Im folgenden ersetzt der Präprozessor im Quelltext "max" immer durch 1000. Der Präprozessor arbeitet hier ähnlich wie ein Textverarbeitungsprogramm, wenn dieses ein Wort durch ein anderes automatisch ersetzt. Man kann auch die Konstante max 50+50 oder max sqrt(100) definieren. In diesen Fällen ersetzt der Präprozessor nur max durch den jeweiligen Wert. Erst der Compiler berechnet dann den Wert. Für die Berechnung des Wertes kann der Compiler dann auch auf die Funktionen der include-Dateien zurückgreifen.
Es wird aber nicht in Strings die Konstante max durch sqrt(100) ersetzt (cout, printf, usw.).
#define ist eine textuelle Ersetzung, es belegt im Unterschied zur const-Deklaration keinen Speicherplatz.
Hinter der define-Anweisung darf kein Strichpunkt stehen!

Beispiel

```
#define quadrat sqrt(2)
#define hallo "Das ist ein String"
#define zahl 2
```

Mit diesen textuellen Konstante ist es auch möglich Makros zu schreiben. Wenn diese Makros einfach sind und eine eigene Funktion ein zu großer Aufwand wäre, dann gebe ich gleich eine Alternative zu den textuellen Makros an. Diese werden nämlich nicht vom Compiler auf Fehler hin untersucht und so kann es sehr leicht sein, daß man durch einen falschen Makro mit Hilfe von #define sehr schnell sehr viele Fehler im Programm hat und nicht weiß woher sie kommen.
Zunächst zum Makro mit #define:

Beispiele:

Ich möchte gerne ein Makro zur Quadratberechnung einführen.

```
#define sq(x)  ((x)*(x)) // die äußere Klammer sollte man nicht vergessen,
// da es sonst nach der textuellen Ersetzung im Programm auf Grund von
//Klammerfehlern zu Fehlern kommen kann.

#define pl(x)   (x)+(x)
#define pl(x)   ((x)+(x))
a=pl(b)*c // wenn das die Abkürzung von a=2b+c ist, kommt es bei der
// Verwendung des ersten Makros zu unerwünschten und schwer findbare
// Fehlern
```

Nun gibt es als Alternative inline-Funktionen. Das sind ganz kleine, einfache "Funktiönchen", oftmals auch Einzeiler.

Beispiel:

```
inline int sq (int x)
{
  return x*x;
}
```

3.9.2 Nichtveränderliche Variable (Neu in C++)

Diese Variablen haben den Charakter von Konstanten. Sie werden mit const deklariert. Nichtveränderliche Variablen belegen nicht zwangsläufig Speicherplatz. Es kann vorkommen, daß der Compiler jede Verwendung der Konstanten direkt ersetzen kann und somit kein expliziter Verweis auf die Variable mehr bestehen muß – die Konstante wurde somit "wegoptimiert".

Const und #define sind sehr ähnliche Befehle. Früher verwendete man sehr häufig #define. Es stellte sich aber heraus, daß es zu Fehlern mit #define kommen kann, weil keine Typdeklaration vorgenommen wird. Const hat #define deshalb teilweise abgelöst.

Beispiel:

```
const float pi=3.14;
```

Jeder Versuch eine mit const deklarierte Konstante zu verändern verursacht eine Fehlermeldung.

Beispiel, wie man eine mit const deklarierte Konstante nicht verwenden darf:

```
const float pi=3.14;
float a=6, b=5;
...
pi=a+b;
```

3.9.3 Direkte Angabe von Zeichen, Zeichenfolgen und Zahlenwerten

Beispiel:

```
b = 455 + c;
```

In diesem Fall entspricht 455 einer direkten Angabe. Sie zählt zu den Konstanten.

3.10 Aufgaben

1. Schreiben Sie ein C++-Programm zur Berechnung der Volumina geometrischer Körper (mind. 3 verschiedene) - realisieren Sie einen komfortablen Bediendialog, Sie dürfen einen mündigen, d.h. fehlerfrei funktionierenden Bediener voraussetzen. (Unser bisheriger Kenntnisstand erlaubt nur lineare Programme!)

```
// Berechnung von 3 verschiedenen Volumina
#include <iostream.h>
#define pi 3.14159
int main (void)
{
 double a, b, c, G, h, r, V;
 cout << "\nVolumenberechnung eines Quaders";
 cout << "\nBitte geben Sie die Seitenlänge a ein: ";
 cin >> a;
 cout << "Bitte geben Sie die Seitenlänge b ein: ";
```

```
 cin >> b;
 cout << "Bitte geben Sie die Seitenlänge c ein: ";
 cin >> c;
 V=a*b*c;
 cout << "\nDas Volumen des Quaders beträgt: " << V;
 cout << "\n\nVolumenberechnung einer Pyramide";
 cout << "\nBitte geben Sie die Grundfläche G ein: ";
 cin >> G;
 cout << "Bitte geben Sie die Höhe h ein: ";
 cin >> h;
 V=1.0/3.0*G*h;
     // Der Compiler sieht die Division 1/3 als Ganzzahldivision.
     // Das Ergebnis dieser Division wäre 0. Man muß also 1.0/3.0
     // schreiben, damit der Compiler die Division richtig interpretiert.
 cout << "\nDas Volumen der Pyramide beträgt: " << V;
 cout << "\n\nVolumenberechnung einer Kugel";
 cout << "\nBitte geben Sie den Radius der Kugel ein: ";
 cin >> r;
 V=4.0/3.0*r*r*pi;
     // Wegen obigen Sachverhalts muß man auch hier 3.0/4.0 statt 3/4
     // schreiben.
 cout << "\nDas Volumen der Kugel beträgt: " << V;
 return 0;
}
```

Wo stecken die Fehler ?:

```
cout <<< max, moritz << ida, frieda; // max, moritz, ..variablen
cin << "bitte eine Zahl zwischen 0 und 10: " << zahl;

program Kreisflaeche;
int main (void);
{
float pi = 3.14159, g=9.81, flaeche=0, r;
flaeche=r*r*pi;
cout << "Kreisfläche = " << flaeche<<;
}.
```

Hier sind die Fehler markiert:

```
cout <<< max, moritz << ida, frieda; // max, moritz, ..variablen
cin << "bitte eine Zahl zwischen 0 und 10: " << zahl;

program Kreisflaeche;
int Main (void);
{
float pi = 3.14159, g=9.81, flaeche=0, r;
flaeche=r*r*pi;
cout << "Kreisfläche = " << flaeche<<;
}.
cout << "max, moritz" << ida << frieda;
  oder
cout << max << moritz << ida << frieda;
cout << "bitte eine Zahl zwischen 0 und 10: ";
cin >> zahl;

// program Kreisflaeche;
int main (void)
#include <iostream.h>
cout << "Kreisfläche = " << flaeche;
return 0;
```

4 Gültigkeitsbereiche von Variablen

Je nachdem, wo eine Variable in einem Programm definiert wird, ist ihr Verwendungsbereich unterschiedlich.

Gültigkeitsbereiche:

- Es gibt Variablen, die vom ganzen Programm und auch von zusätzlichen Programmen, die eingebunden werden, benutzt werden können. Solche Variablen nennt man *globale* Variablen.
- Variablen, die im ganzen Programm aber nicht von zusätzlich eingebunden Programmen benutzt werden können. (Schlüsselwort static bei der globalen Definition)
- Variablen die in Funktionen definiert werden können nur innerhalb einer Funktion benutzt werden, man nennt solche Variablen lokal. Ihr Wert wird beim Beenden der Funktion gelöscht. Ist im Hauptprogramm eine globale Variable definiert, die den gleichen Namen hat, wie eine lokale Variable hat, so geht der Wert der lokale Variable vor dem Wert der globalen. Lokale Variablen sind innerhalb der sie einschließenden geschweiften Klammern gültig (solange sie nicht von anderen (lokalen) Variablen überlagert werden).
- Mit Hilfe des Schlüsselwortes static bei Definitionen von Variablen in Funktionen, wird der Wert dieser Variable nach Beendigung der Funktion nicht gelöscht, sondern bleibt erhalten.

Beispiele:

```
int a, b=4; //  globale  Variable,  auch  von  externen  Programmdateien
                  aus veränderbar
static c;   //  globale  Variable,  die  nur  innerhalb  dieser  Programmdatei
                  bekannt ist

void test (void)
{
 int c=b;   // c hat den Wert 4, globales b
 int x;     // lokale Variable, die nur innerhalb der Funktion Test verwen–
            // det werden kann
 int  b;    // lokales b, verdeckt globales b
 static zaehler=0;  // statische Variable, die ihren Wert nach Beendigung
                    // beibehält, die Variable wird nur beim allerersten
                    // Funktionsaufruf mit 0 initialisiert
 b=22;         // lokales b erhält den Wert 22; 1. lokales b
    {
      int b;   //2.lokales b, verdeckt 1.lokales b und globales
      b=5;     // Wertzuweisung an 2.lokales b, Wert von 1.lokalen b bleibt
               // unverändert
    }          // hier endet der Gültigkeitsbereich des 2.lokalen b
 b=b+1;       // b=23, 1.lokale b wird verändert

 zaehler+=1;  // Zaehler zählt, wie oft die Funktion test aufgerufen wurde
}
```

- Man sollte in jedem Fall den Gültigkeitsbereich so klein wie möglich halten. Manche Programmierer sehen es aus diesem Grund als praktisch an, Variablen erst dann zu initialisieren, wenn die Variable auch wirklich benötigt wird (Dies geht jedoch nur in C++,in C müssen alle Variablen am Anfang eines Blocks definiert werden!).

Beispiel:

```
if (double d=prim(true))
{
 left/=d;
 break;
}
```

Möchte man einem globalen x ein Wert zuweisen, obwohl es von einem lokalen verdeckt wird, so kann man das globale x durch den Bereichsauflösungsoperator :: sichtbar machen.

Beispiel:

```
int a;

void test (void)
{
  int a=1; //Zuweisung an lokales a
  ::a=2;   //Zuweisung an globales a
}
```

Tips:

Man sollte, wenn möglich, keine Variablen überlagern. (Es gibt doch genügend Variablennamen!!!)

5 Ausdrücke

Ausdrücke sind Folgen aus Operatoren und Operanden

Ausdrücke:
- berechnen Werte
- bezeichnen Objekte
- legen Typen fest und / oder ändern sie
- erzeugen Seiteneffekte

5.1 Tabellarische Übersicht von Operatoren

Operator	Bedeutung	Klasse	Stelligkeit	Assoziativität
( )	Klammern in Ausdruck oder Funktion	1		l
[ ]	Klammerpaar für Indizes in Feldern	1		l
->	Pfeiloperator: dereferenziert + Strukturkomponente	1	binär	l
.	Punktoperator: Strukturkomponente	1	binär	l
!	Logische Negation	2	unär	r
-	Bitweise Negation	2	unär	r
++	Inkrement um Eins	2	unär	r
--	Dekrement um Eins	2	unär	r
-	Minusvorzeichen	2	unär	r
(typ)	Explizite Typumwandlung: casting	2	unär	r
*	Sternoperator (Inhaltsoperator): Derefernzierung	2	unär	r
&	Undoperator (Adressoperator)	2	unär	r
sizeof	Speichergröße in Bytes eines Objekts	2	unär	r
*	Multiplikation	3	binär	l
/	Division	3	binär	l
%	Modulo-Operator (Divisionsrest)	3	binär	l
+	Addition	4	binär	l
-	Subtraktion	4	binär	l
<<	Bitweiser Linksshift	5	binär	l
>>	Bitweiser Rechtsschift	5	binär	l
<	Vergleichsoperator	6	binär	l
<=	Vergleichsoperator	6	binär	l
>	Vergleichsoperator	6	binär	l
>=	Vergleichsoperator	6	binär	l
= =	Vergleichsoperator	7	binär	l
!=	Vergleichsoperator	7	binär	l
&	Bitweises AND	8	binär	l

^	Bitweises XOR	9	binär	l
|	Bitweises OR	10	binär	l
&&	Logisches AND	11	binär	l
||	Logisches OR	12	binär	l
? :	Bedingter Ausdruck	13	ternär	r
=	Zuweisung	14	binär	r
+=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
-=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
*=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
/=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
%=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
>>=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
<<=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
&=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
|=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
^=	zusammengesetzte Zuweisungsoperatoren	14	binär	r
,	Komma-Operator: Anweisungsverkettung	15	binär	l

Bindungstärke ·Klasse:
je kleiner die Klassenzahl ist, desto stärker ist die Bindung
⇒ Operator mit kleinerer Klassenzahl hat Vorrang vor Operatoren mit größerer Klassenzahl

Reihenfolge der Auswertung:
```
a=a+3*b;     // Multiplikation vor Addition
c=(a+b)+(c+d)=a+(b+c)+d
```

Man sollte bei verschiedenen Operatoren, die aus mehreren Zeichen bestehen, darauf achten, daß man zwischen diesen Zeichen kein Leerzeichen schreiben darf, da sonst andere Operatoren gemeint sind:

Beispiele:

```
+= bzw. + =
*= bzw. * =
```

5.2 Einfache Ausdrücke

5.2.1 Primärausdrücke

Ausdruck	**Wert**
Konstante	Wert der Konstante
Variablen	Wert der Variable
Feldvariablen	Adresse des Feldes
Funktionen	Adresse der Anweisungen der Funktion
konstante Zeichenfolge	Wert und Typ des Ausdrucks in Klammern

Ausdrücke für Feld- und Strukturkomponenten

Ausdruck	Wert und Wirkung
`x[Ausdruck]`	Auswahl einer Feldkomponente, x bezeichnet Feld (Name der Feldvariable oder Zeiger auf Feld) Ausdruck ist ein Integer-Ausdruck
`x.name`	Auswahl einer Strukturkomponente (x ist Name einer struct- oder union-Variable)
`x→name`	Auswahl der Komponente mit dem Bezeichner name aus der Struktur, auf die x verweist äquivalent: *(x.name)

Funktionsaufrufe

x (<Parameterliste>)
Aufruf der Funktion x mit der Parametern in <Paramterliste>

Adreßausdrücke

Ausdrücke	Wert / Wirkung
`&x`	Adresse der Variablen / Konstanten x
`*x`	Liefert den Wert des Objekts bzw. die Funktion, auf die x verweist

Explizite Typumwandlung (cast-Ausdrücke)

(typname) x

Anwendung der Vorzeichenoperanden: +, -
der unsren Operatoren: ++, --
der binären Operatoren: +, -, * /
Operanden mit Zahlentypen (Integer, Gleitkomma)

5.3 Unäre Ausdrücke

Unäre Operationen benötigen nur einen Operanden.

unäre Operationen: +, - Vorzeichenoperatoren
++ Inkrementieren
-- Dekrementieren

Beispiel:

```
b=a[i++];
oder
b=a[i];
i=i+1;


int i,j;
i=1;
```

```
j=++i+1;                // j=3, weil i=2 ist (++i wird vor der Addition
                           ausgeführt)

aber

i=1;
j=i++ +1;               // j=2, weil i erst nach Ausführung der Addition um
                        // eins erhöht wird

oder

i=1;
j=i+1;
i=i+1;
```

unabhängiges Beispiel:

Bei der for-Schleife ist es egal, ob man i++ bzw. i - - oder ++i bzw. - -i verwendet. In jedem Fall wird die Inkrementierung bzw. Dekrementierung am Ende der Schleife durchgeführt.

Es besteht also kein Unterschied zwischen:
`for (i=1; i<=100; i++);  = =  for (i=1; i<=100;++1);`

Ausdruck	Wert
++x	Wert von x um eins erhöhen und Ergebnis im Ausdruck verwenden
-- x	Wert von x um eins erniedrigen und Ergebnis im Ausdruck verwenden
x++	x im Ausdruck verwenden und anschließend um eins erhöhen
x--	x im Ausdruck verwenden und anschließen um eins erniedrigen

5.4 Arithmetische Ausdrücke

Ausdruck	Wirkung	Beispiel
+	Addition	5+1=6
-	Subtraktion	5-2=3
*	Multiplikation	5*5=35
/	Division; wenn beide Operanden Integer, dann ganzzahlige Division (in Pascal DIV)0	5/5=1 5/6=0 5.0 / 6.0 = 0.8333
%	Modulo-Bildung, nur Integer-Operanden a%b, Rest bei Division von a durch b. Falls b n	3%2 = 1

Zur Lesbarkeit von arithmetischen Ausdrücken:

Es kann vorkommen, daß man sehr komplexe mathematische Funktion berechnen muß. Es ist in diesem Fall sinnvoll, die Funktion, zwecks der besseren Übersicht, in kleine Teile zu zerlegen.

Beispiel:

```
w=2*((i%5)*(4+(j-3)/(k+2)));
```

```
oder einfach

u=i%5;
v=4+(j-3)/(k+2);
w=2*(u*v);
```

5.5 Vergleichausdrücke und logische Ausdrücke

Vergleichsoperatoren:

Wirkung	Ausdruck
gleich	= =
ungleich	!=
kleiner	<
größer	>
kleiner oder gleich	<=
größer oder gleich	>=

⇒ mit arithmetischen Operanden → gewohnte mathematische Interpretation
⇒ mit Zeigern:

- Vergleichbarkeit nur innerhalb derselben Struktur bzw. desselben Feldes
- Relation in Übereinstimmung mit Indexordnung und Abspeicherreihenfolge der Elemente

⇒ Vergleichsausdrücke erzeugen einen der Wahrheitswerte wahr (=1) oder falsch (=0), sie sind damit vom Typ int
⇒ Vergleichsausdrücke sind logische Ausdrücke
⇒ keine Kettenbildung entsprechend mathematischer Abkürzung: $a \leq 0 \leq b$
⇒ sondern Auflösen der Kette durch Benutzung des UND Operator a <= 0 UND b >= 0

Beispiel:

```
a=b>c;
```

Da Vergleichsausdrücke Wahrheitswerte erzeugen gehören sie auch zu den logischen Ausdrücken.
Verknüpfungen mit den logischen Operatoren erzeugt aus logischen Ausdrücken wieder logische Ausdrücke.

logische Operatoren:

- sind definiert über Ausdrücke, die zu Wahrheitswerten ausgewertet werden können
- verknüpfen Wahrheitswerte

- alle integer-Ausdrücke
- alle Vergleichsausdrücke

- alle durch Anwendung der Operatoren && (UND), || (ODER), ! (NICHT) auf logische Ausdrücke entstehenden Ausdrücke
- Zeiger (implizite Typumwandlung nach Integer, Überprüfung auf NULL-Zeiger)
- Auswertung von links - Abbruch, wenn Ergebnis vorliegt
  Bsp.: a && b, b wird nicht mehr ausgewertet, wenn a falsch ist
  Bsp.: a|| b, b wird nicht ausgewertet, wenn a wahr ist
- Vergleichsoperatoren binden stärker als logische Operatoren (siehe Tabelle in Kapitel 4.1)

Ergebnis der Anwendung eines logischen Operators: Wahrheitswert

Operatoren:

OPERATOR	SCHREIBWEISE	SCHEMA	WAHR, WENN	BEISPIEL
Nicht / Not	!	!a	a falsch ist	! (a<b)
Und / And	&&	a&&b	a und b wahr sind	(a = = b) && (b<c)
Oder / Or	||	a||b	wenn mind. einer wahr ist	(a<b) || (c= =b)

Verwendung:

- Bedingungsausdrücke (if-Anweisung, bedingte Schleife)
- Verknüpfung von Wahrheitswerten, Variablen und Vergleichsausdrücke
  Beispiel:
  ```
  If ( ( (a = = b) || (a = = c) ) && (b!=c) )
  ```
- in Wertzuweisungen für Wahrheitswerte:
  Beispiel:
  ```
  a = b = = c;
  c = b < c;
  ```

5.6 Logische Bit-Ausdrücke

- Boolesche Operationen auf der internen Bitrepräsentation von Operanden
- Operanden auf den Bits eines Operanden
- Operationen zwischen Bits zweier Operanden

AUSDRUCK	WERT / WIRKUNG
x << y	schiebe Inhalt von x um y Stellen nach links
x >> y	schiebe Inhalt von x um y Stellen nach rechts
x & y	bitweise UND Verknüpfung der Bits aus x und y
x | y	bitweise ODER Verknüpfung der Bits aus x und y
x^y	Bitweise "Exlusiv-Oder"[1] Verknüpfung der Bits aus x und y
~x	Bitweise Negation von x

[1] Dies bedeutet, daß entweder x oder y gleich 1 sein muß, damit der gesamte Ausdruck gleich 1 wird. Haben x und y den gleichen Wert, also beide gleich 0 oder gleich 1, so ergibt sich daraus das Ergebnis 0.

5.7 Bedingte Ausdrücke und Komma-Ausdrücke

- Bedingte Ausdrücke mit dem 3-stelligen Operator _?_:_
- Komma-Ausdrücke mit dem 2-stelligen Operator: _,_

Ausdrücke	Wert / Wirkung
`e0 ? e1 : e2`	e0 auswerten, • falls wahr → e1 auswerten • falls falsch → e2 auswerten e1, e2 sind alternativ
`e1, e2`	1. e1 auswerten 2. e2 auswerten 3. Typ und Wert des Gesamtausdrucks durch Typ und Wert von e2 bestimmt

Beispiel:

```
...
i=2;
...
x=f(y,(i++,a[i]);    // entspricht x=f(y,a[++i])
```

Anwendung des Komma-Ausdrucks:
Verwendung zweier Ausdrücke, wo nur ein Ausdruck erlaubt ist. (zum Beispiel Funktionsaufrufe, Parameter, Klammerung beachten)

5.8 Konstante Ausdrücke

Konstante Ausdrücke sind alle die Ausdrücke, die der Compiler zur Übersetzungszeit auswerten kann und muß (für konstante Ausdrücke wird kein Code zur Auswertung erzeugt):

- Argument der #if-Präprozessordirektive
- Dimensionierung von Feldern
- case-Ausdrücke in switch-Anweisungen
- explizite Definition von Aufzählungfswerten
- Inititialisierungswerte von static- und extern Variablen
- Konstante Ausdrücke können aus Integer- und Charkonstanten durch Anwendung der folgenden Operatoren gebildet werden:

  unär: + - ~ . !

  binär: + - * / % << >> = = != < <= > >= & ^ | && ||

  ternär: _?_:_

5.9 Zuweisungen

5.9.1 Abkürzungsoperatoren

ursprüngliche Zuweisungsoperator ist = zusätzliche Zuweisungsoperatoren zur Abkürzung der

- binärer arithmetischer und der
- binären Bitausdrücken

Schema:

Operand_1=Operand_1⊗ Operand_2 ⇒ Operand_1 ⊗= Operand_2
Operand_1=(Operand_1) ⊗ (Operand_2) ⇒ Operand_1 ⊗= (Operand_2)

int i=5, j=7;
float f=5.5, g=-3.25;

kurzer Ausdruck	langer Ausdruck	Endergebnis
`i+=5`	`i=i+5`	`10`
`f-=g`	`f=f-g`	`8.75`
`j*=(i-3)`	`j=j*(i-3)`	`14`
`f/=3`	`f=f/3`	`1.833333`
`i%=(j-2)`	`i=i%(j-2)`	`0`

5.9.2 Zuweisung als Ausdruck

Unterschied in C / C++ zu allen anderen geläufigen Programmiersprachen:
Zuweisungen sind Ausdrücke

Folgen:

Zuweisungen dürfen überall da verwendet werden, wo Ausdrücke des betreffenden Datentyps erlaubt sind!

Es ist auch erlaubt mehreren Variablen gleichzeitig einen Wert zuzuweisen.

Beispiel:

```
int i, j;
i=j=10;
```

Was passiert bei folgenden Beispielen?

int i,j,k;
float x=0.005, y;
char a, b, c, d;

z = k = x ⇒ k = 0, z = 0.0
k = z = x ⇒ z = 0.05, k = 0
k = c ⇒ k = 99

Nachtrag zu den Anweisungen:

Jeder Ausdruck ausdruck wird durch Anhängen von ; zu einer Anweisung

ausdruck;

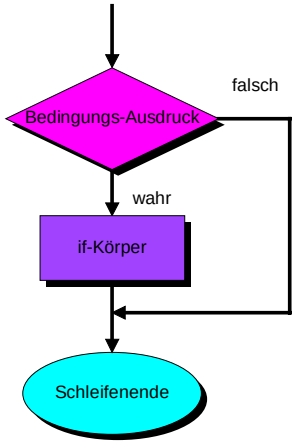
6 Einfache Verzweigungen

6.1 if-Verzweigung

Mit der if-Anweisung kann man erreichen, daß ein bestimmter Programmteil ausgeführt wird, wenn eine Bedingung zutrifft.
Es ist folgendes zu beachten: Sollen durch diese Verzweigung mehre Befehle ausgeführt werden, so müssen diese in Klammern stehen {...}. Es ist aber auch empfehlenswert eine einzige Anweisung in Klammern {...} zu schreiben, damit es bei Erweiterung der if-Bedingung keine Probleme gibt.

allgemeine Formen bedingter Anweisungen

```
if ( <Bedingungs-Ausdruck> )
 <Anweisung>;

if ( <Bedingungs-Ausdruck> )
{
 <Anweisungsfolge>;
}
```



Beispiel:

```
a = 5;
b = 4;
if (a > b)
 c = sqrt(a * a + b * b);
```

→ Falls a > b ist, dann ist $c = \sqrt{a^2 + b^2}$ (In diesem Fall braucht man um die Anweisung c = sqrt(a * a + b * b); keine Klammern machen.

```
a = 5;
b = 4;
if (a > b)
{
  c = a;
  a = b;
  b = c;
}
```

→ Die drei Anweisungen werden vom Computer bearbeitet, wenn a > b ist. In diesem Fall ist eine Klammer {...} notwendig. Würde man sie weglassen, so berechnet der Computer nur c = a; in der if-Schleife. Die Berechnungen a = b; b = c; würden bei jedem Programmablauf berechnet, unabhängig ob die Bedingung erfüllt ist oder nicht. Die Klammern geben dem Computer einen Hinweis auf einen zusammengehörigen Block, der im Zusammenhang mit der if-Anweisung abgearbeitet werden soll.

Bei bedingten Anweisungen gibt es eine abkürzende Schreibweise, wenn es darum geht, daß eine Schleife nur durchlaufen wird wenn ein Wert null bzw. ungleich null ist.

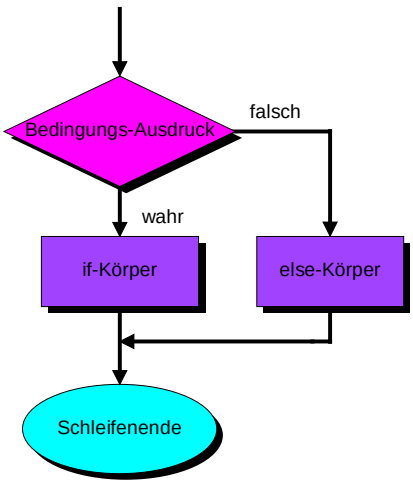
Beispiel:

```
if (x)   //wenn x ungleich 0
bzw.
if (x!=0)

if (!x)    //wenn x gleich 0
bzw.
if (x==0)
```

6.2 Bedingte Anweisungen mit Alternativen - if-else-Anweisung

Im Unterschied zur reinen if-Anweisung wird in der if...else-Anweisung eine Alternative angeboten. Das bedeutet, falls die Bedingung für die if-Anweisung nicht erfüllt wird, dann wird der Anweisungsblock der else-Anweisung abgearbeitet.

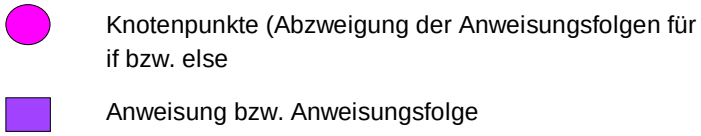
```
if ( <Bedingungs-Ausdruck> )
 <Anweisung>;
else
 <Anweisung>;


if ( <Bedingungs-Ausdruck> )
{
 <Anweisungsfolge>;
}
else
{
 <Anweisungsfolge>;
}
```



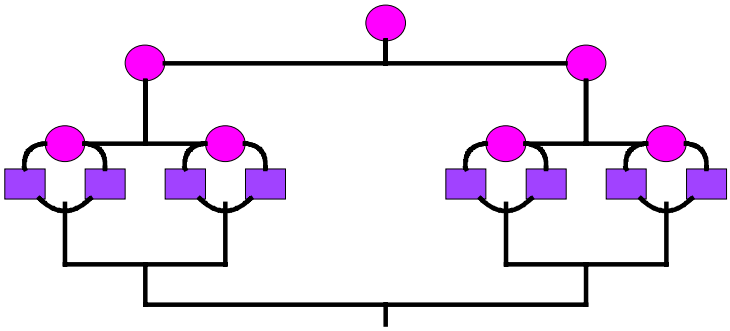
Beobachtungen:

- Bedingungsabfragen geschachtelt (bis zur Unübersichtlichkeit möglich)
- Einrücken im Programmtext ermöglicht Überblick
- Klären Komplexer Entscheidungssituation durch Entscheidungsbäume
- Achtung beim Test: im Extremfall Verdoppelung der Programmpfade mit jeder neuen Entscheidungsebene (Schachtelungsebene).

6.2.1 Anzahl der Programmpfade

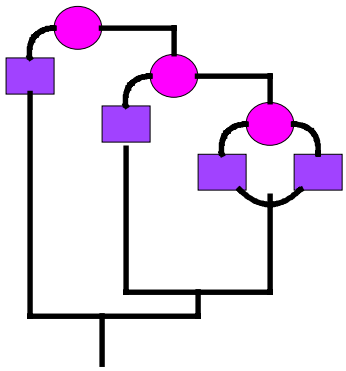


max 2 $^{\text{Schachtelungstiefe}}$ Pfade und Anweisungen / Blöcke



Ohne Schachtelung im else bzw. if-Zweig

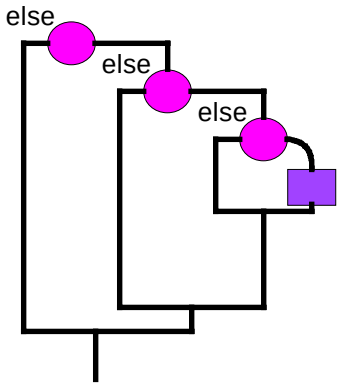
n+1 Pfade ... n+1 Anweisungen / Blöcke



geschachtelte bedingte Anweisung

Anzahl der Programmpfade im Intervall [1..$2^{\text{Schachtelungstiefe}}$] oder [n+1...$2^{\text{Schachtelungstiefe}}$]

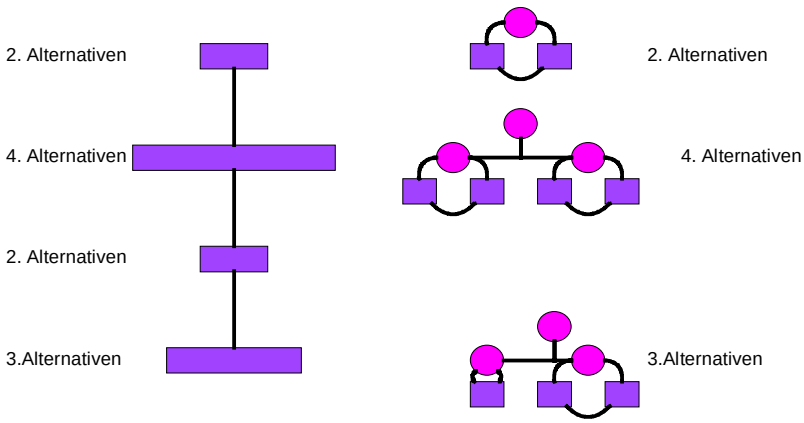
Das Diagramm bedeutet nichts anderes als, daß falls Bedingung 1 und Bedingung 2 und Bedingung 3 erfüllt ist, der Anweisungsblock ausgeführt wird.



<u>Hintereinanderausführung von Verzweigungen:</u>

Anzahl möglicher Pfade = Produkt der Alternativen der sequentiellen ausgeführten Verzweigungen.

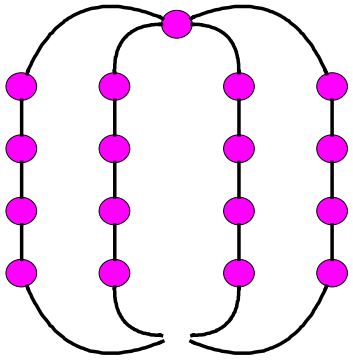
In diesem Fall:
$2 * 4 * 2 * 3 = 48$
In größeren Programmen können es leicht einige 1000 sein.



6.3 Fallunterscheidung, switch-Anweisung

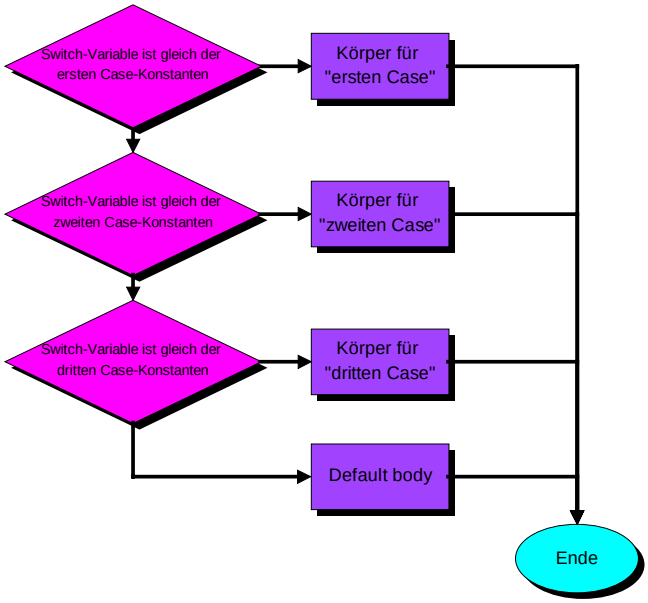
in C / C++ realisiert als Abfragekaskade

typische Anwendungen:
- Fallunterscheidung
- Auswahlentscheidungen - 1 aus n
- Anweisungskaskade mit berechneten Einstieg
- vollständige Mehrfachverzweigung



Syntax:

```
switch ( <Ganzzahl-Ausdruck> )
{
 case <Ganzzahl-Konstante>: <Anweisung>; // Fall 1
 ...
 case <Ganzzahl-Konstante>: <Anweisung>; // Fall n
 default: <Anweisung>;
}
```



- <Ganzzahl-Ausdruck> : Ausdruck, der zu einem ganzzahligen Wert ausgewertet wird
- <Ganzzahl-Konstante>: ganze Zahl oder Zeichen, keine Strings
- <Anweisung> : alle in C / C++ formulierbaren Anweisungen, besonders break (Verlassen der switch-Anweisung)
- <default-Anweisung>: sonst Fall, kann fehlen

Man muß bei einer Anweisungsfolge keine Blockklammerung benutzen.
Break sollte immer am Ende der Anweisungsfolge stehen, sonst werden die restlichen Anweisungen abgearbeitet.

Abarbeitung der switch-Anweisung:

1. Auswertung des <Ganzzahl-Ausdruck>
2. Vergleich mit <Ganzzahl-Konstante> der folgenden Fall-(case)-Anweisungen
3. Bei Gleichheit ausführen der <Anweisung> hinter:
4. <Anweisung> ohne / mit break
    - sequentielle Bearbeitung aller folgenden <Anweisung(-en)> bis zum ersten break
    - oder bis zum Ende der switch ...}-Anweisung
5. Keine Übereinstimmung mit Fall-Konstante, Ausführen der <default-Anweisung>
6. Fehlt <default-Anweisung> Fortsetzung hinter switch ... }

Außer der break-Anweisung gibt es noch die return-Anweisung, die einen Wert zurückliefert und gleichzeitig die switch-Anweisung verläßt.

6.4 Fragezeichenoperator

Der sogenannte Fragezeichenoperator ist eine abgekürzte Anweisung für einen if-else-Block.

```
Bedingung  ?  Anweisungen, falls Bedingung erfüllt : Anweisungen, falls Bedingung nicht erfüllt
```

Beispiel:

```
i>0 ? i=5 : i=2;
```

äquivalent zu:

```
if (i>0)
 i=5;
else
 i=2;
```

äquivalent zu:

```
i=i>0 ? 5 : 2; // hier wird gleich der Variable i das Ergebnis zugewiesen
```

In Verbindung mit dem Fragezeichenoperator kann man natürlich auch den "Komma-Operator" verwenden.
Mit Hilfe dieses Operators, kann man mehrere Befehle hintereinander ausführen lassen. Dies ist zum Beispiel auch bei den Schleifen hilfreich (Kapitel 6). Man muß jedoch darauf achten, daß die Abbruchbedingung (bei Schleifen vor allem wichtig) die letzte Anweisung ist.

Beispiel:

```
i>0 ? i=5, j=1,z=0 : i=2,j=-1,z=5;
```

äquivalent zu:

```
if (i>0)
{
 i=5;
 j=1;
 z=0;
}
else
{
 i=2;
 j=-1;
 z=5;
}
```

7 Die verschiedenen Schleifentypen

7.1 Zählschleife mit for

Wiederholung von Anweisungen abhängig von Zählvariable

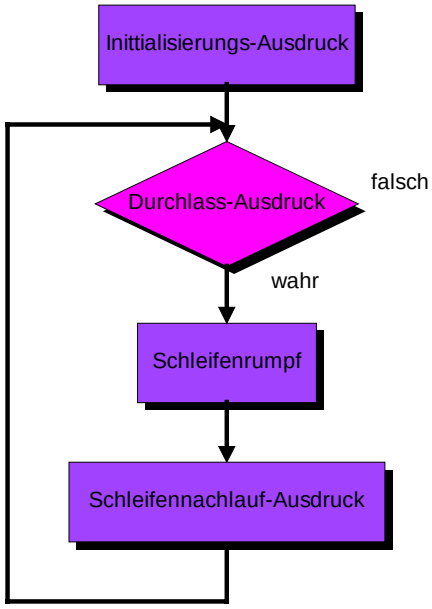
typische Anwendungen:
- Indexlauf, Indizierung von Feldelementen
- Matrizenmultiplikation

Syntax:

```
for ( <Initialisierungs-Ausdruck>;    // initialisiert Zählvariable
      <Durchlass-Ausdruck>;           // Bedingung für Durchlauf
      <Schleifennachlauf-Ausdruck>;)  // Schrittgröße
 <Schleifenrumpf-Anweisung>;          // Schleifenrumpf

oder


{
 <Schleifenrumpf-Anweisungsfolge>;    // Schleifenrumpf
}
```

BSP.:

```
for (i=1; i<=100; i++)
{
 <Anweisungsfolge>;
}

for (x=10; x<15; x++)
{
 cout << x << " ";
}

for (int x=10; x<15; x=x+1)
{
 if (feld [x-1]>max)
  max=feld[x-1];
}
```



Äquivalent: (in Form einer while-Schleife)

```
<Initialisierungs-Ausdruck>
while (<Durchlass-Ausdruck>)
{
  <Anweisung...>
  <Nachlauf-Ausdruck>
}
```

Beispiel (for-Schleife mit Komma-Operator):

```
for (i=0; j++,i>0;i++)
```

⇒ C-Gewohnheiten: kurze Anweisungen direkt als / im Nachlaufausdruck
⇒ fraglich: Übersichtlichkeit (der Mensch als multy pass compiler)

Vereinbarung: Nutzanweisung außerhalb des Nachlaufausdrucks

besondere Eigenschaften der for-Schleife (vgl. mit anderen Programmiersprachen):

- In C++ (nicht C!) ist die Deklaration der Zählvariable im Initialisierungsausdruck möglich: for (int i = 1; ...) Gültigkeitsbereich ist der Block der for-Schleife (übliche Definition des Gültigkeitsbereichs), aber keine weitere Deklarationsschachtel für gleichnamige Variable. (Nach Ausführung der for-Schleife ist der Variablennamen nicht mehr deklariert (Compiler kennt ihn nicht mehr). Es handelt sich hierbei um eine sogenannte lokale Variable. Möchte man in einer nachfolgenden Schleife (die nicht mit der 1.Deklarationsschleife verschachtelt ist) die Variable verwenden, so muß man sie wieder neu deklarieren.) (Sie hat dann aber nicht mehr den gleichen Inhalt wie vorher.)
    - Anwendung: gleichnamige, aber wertemäßige voneinander unabhängige Zählvariablen
    - Wert einer globalen Variablen (gültig für main); Zählvariable ist nach der for-Schleife deklariert

Beispiel:

```
for (int i=0; i<=10; i++) // Deklaration in der Schleife in C++ möglich
```

- Durchlaßausdruck: kann direkt mit der Zählvariable formuliert werden (Absicht der Zählschleife; häufigster Fall), muß aber nicht
- Die Berechnung innerhalb des Schleifenrumpfs muß der Abbruchbbedingung zustreben (Durchlaßausdruck = false)

Es gibt auch die Möglichkeit eine Schleife ohne explizite Abbruchbedingung (beabsichtigte Endlosschleife) zu programmieren. Man sollte aber in dieser Schleife entsprechende Abbruchbedingung einbauen, damit sie nicht zur totalen Endlosschleife wird.
Eine solche Schleife wird wie folgt deklariert:

```
for (;;)   //"forever"
```

7.2 while Schleife

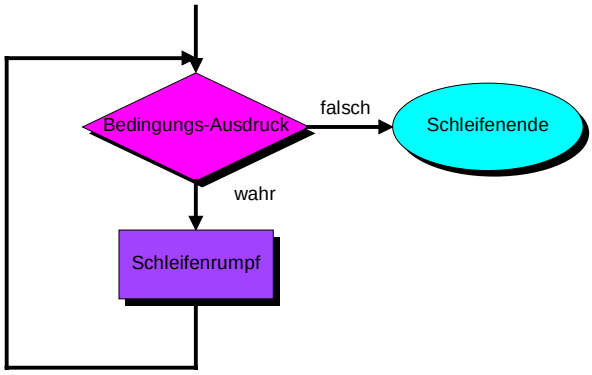
Anwendungen:

bedingte Wiederholung einer Anweisung (Anweisungsfolge)

Syntax der while-Schleife:

```
while (<Bedingungs-Ausdruck>)
 <Anweisung>
```

```
while (<Bedingungs-Ausdruck>)
{
 <Anweisungsfolge>
}
```



Semantik:

- Solange <Bedingungsausdruck> wahr, führe <Anweisung...> aus
- Prüfe vor erstem Durchlauf

Die while-Schleife in C++ entspricht der while-Schleife in Turbo Pascal.

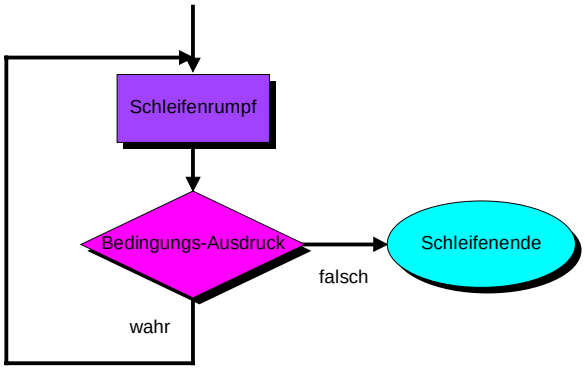
7.3 do...while Schleife

Syntax der do...while-Schleife:

```
do
 <Anweisung>
while (<Anweisungs-Ausdruck>);
```

oder

```
do
{
 <Anweisungen>
}
while (<Anweisungs-Ausdruck>);
```



Semantik:

- Führe die <Anweisung...> aus, solange der <Bedingungsausdruck> falsch wird
- Erster Durchlauf unbedingt

Die do...while-Schleife entspricht in etwa der Repeat-Until-Schleife in Turbo Pascal.

Für alle Schleifenanweisungen gilt:

1. Der <Bedingungsausdruck> muß durch die Operation im Schleifenrumpf den Wahrheitswert false zustreben.
   dann: Schleife terminiert
   sonst: Endlosschleife
2. Als <Bedingungsausdruck> zulässig:
   - alle arithmetischen Ausdrücke
   - alle Zeigerausdrücke
   - alle unzweideutig in solche konvertierbaren Ausdrücke

7.4 Aufgaben

1. Berechnen Sie mit Hilfe der drei in C möglichen Schleifenkonstrukte jeweils den Wert der Summe:

5+9+13+...+201

Wie können while-Schleifen in do...while-Schleifen umgewandelt werden?

```
// mit do...while-Schleife
#include <iostream.h>
int main (void)
{
 int zahl=5, summe=0;
 do
 {
  summe=zahl+summe;
  zahl=zahl+4;
 }
 while (zahl<=201);
 cout << "Die Summe beträgt: "<< summe;
 return 0;
}

// mit while-Schleife
#include <iostream.h>
int main (void)
{
 int zahl=5, summe=0;
 while (zahl<=201)
 {
  summe=zahl+summe;
  zahl=zahl+4;
 }
 cout << "Die Summe beträgt: "<< summe;
 return 0;
}

// mit for-Schleife
#include <iostream.h>
int main (void)
{
 int zahl=5, summe=0;
 for (zahl=5; zahl<=201; zahl=zahl+4)
  summe=summe+zahl;
 cout << "Die Summe beträgt: "<< summe;
 return 0;
}
```

2. Erzeugen Sie unter Verwendung einer for-Schleife eine Tabelle von sin(x) für $0 \leq x \leq 1.60$ mit folgenden Tabellenabständen:

delta = 0.1

delta = 0.2

```
#include <iostream.h>
#include <math.h>
int main (void)
{
 float sin_x;
for (float x=0; x<=160; x=x+0.1)   // für delta 0.2: x=x+0.2
 {
```

```
  sin_x=sin(x);
  cout << "\nsin ("<<x<<"):" << sin_x;
 }
 return 0;
}
```

2. Schreiben Sie ein Programm zur Berechnung von $u = s^2 + t^2$ mit $s = \sum_{m=1}^{100} \frac{m}{1+m}$ und $t = \sum_{m=1}^{100} \frac{2m+1}{3m^3 + 0{,}5m^2 - 0{,}25}$

```
#include <iostream.h>
int main (void)
{
 long u, s=0.0, t=0.0;
 for (long m=1.0; m<=100.0; m=m+1.0)
 {
  s=s+(m/(1+m));
  t=t+(2*m+1)/(3*m*m*m+0.5*m*m-0.25);
 }
 u=s*s+t*t
cout << "u beträgt: " << u;
 return 0;
}
```

3. Geben Sie einen eingegebenen Text wieder rückwärts aus.

```
// Text rückwerts ausgeben
#include <stdio.h>

void main (void)
{
 char text[80];
 int i, j;
 printf("\nBitte geben Sie einen Text ein, der weniger als 80");
 printf("\nZeichen hat.");
 printf("\nBitte schließen Sie die Eingabe mit \'Return\' ab:\n");
 scanf("%79[^\n]s",text);
 for (i=0; text[i]!='\0';i++);
 for (j=i-1; j>-1; j--)
  printf("%c",text[j]);
}
```

4. Berechnen sie mit Hilfe einer Schleife Fibonacci-Zahlen.
   Sie berechnen sich wie folgt: Die erste und zweite Fibonaccizahl ist 1. Die folgenden Zahlen werden gebildet durch die Addition der zwei letzten.
   $F_1=1$
   $F_2=1$
   $F_3=2=F_1+F_2$
   $F_4=3=F_2+F_3$
   ...

```
// Fibonacci-Zahlen
#include <stdio.h>

void main (void)
{
 int fib_liste[100]={1,1};
 int n, zaehler;
 do
```

```
 {
  printf("\nBitte geben Sie ein, wieviele Fibonaccizahlen berechnet");
  printf("\nwerden sollen: ");
  scanf("%d",&n);
 }
 while (n<3);
 for (zaehler=2; zaehler<n; zaehler++)
  fib_liste[zaehler]=fib_liste[zaehler-1]+fib_liste[zaehler-2];
 printf("\nHier ist die Liste mit den ersten %d Fibonaccizahlen",n);
 printf("\n  i     Fi");
 for (zaehler=0; zaehler<n; zaehler++)
  printf("\n%3d%6d",zaehler+1, fib_liste[zaehler]);
}
```

5. Ausgabe einer Raute:

```
     *
    ***
   *****
 *********
***********
 *********
  *****
   ***
    *
```

```
#include <stdio.h>
#include <math.h>


void sterne_drucken (int i, int k)
{
 int x;
 const int b=80;
 if (i+k >b)
  printf("%*s",b/2,"Flasche Parameter");
 else
 {
  printf("%*s",i,"");
  for (x=1; x<=k; x++)
   printf("*");
  printf("\n");
 }
}

void main (void)
{
 int i, j;
 for (i=-4; i<=4; i++)
  sterne_drucken(abs(i)+9,(1+2*(4-abs(i))) );
}
```

6. Schreiben Sie ein Programm, daß eine Zahl als römische Zahl ausgibt.

```
// Umrechnung in römische Zahlen
#include <stdio.h>
#include <string.h>

void main (void)
{
 int stelle_0, stelle_1, stelle_2, stelle_3;
 char rom_0[6], rom_1[6], rom_2[6], rom_3[6];
 int arab_zahl;
```

```
char rom_zahl[30];
do
{
  printf("\nBitte geben Sie eine positive Zahl bis 4000 ein, ");
  printf("\nwollen Sie das Programm beenden, so geben Sie eine 0 ein: ");
  scanf("%i", &arab_zahl);
} while ((arab_zahl<0)&& (arab_zahl>=4000));

while (arab_zahl!=0)
{
 // "Einserdezimalstelle"
 stelle_0=((arab_zahl%1000)%100)%10;
 switch (stelle_0)
 {
  case 1: strcpy (rom_0,"I"); break;
  case 2: strcpy (rom_0,"II"); break;
  case 3: strcpy (rom_0,"III"); break;
  case 4: strcpy (rom_0,"IV"); break;
  case 5: strcpy (rom_0,"V"); break;
  case 6: strcpy (rom_0,"VI"); break;
  case 7: strcpy (rom_0,"VII"); break;
  case 8: strcpy (rom_0,"VIII"); break;
  case 9: strcpy (rom_0,"IX"); break;
 }           // switch 0 Ende
 stelle_1=((arab_zahl%1000)%100)-stelle_0;
 switch (stelle_1)
 {
  case 10: strcpy (rom_1,"X"); break;
  case 20: strcpy (rom_1,"XX"); break;
  case 30: strcpy (rom_1,"XXX"); break;
  case 40: strcpy (rom_1,"XL"); break;
  case 50: strcpy (rom_1,"L"); break;
  case 60: strcpy (rom_1,"LX"); break;
  case 70: strcpy (rom_1,"LXX"); break;
  case 80: strcpy (rom_1,"LXXX"); break;
  case 90: strcpy (rom_1,"XC"); break;
 }            // switch 1 Ende
 stelle_2=(arab_zahl%1000)-stelle_1-stelle_0;
 switch (stelle_2)
 {
  case 100: strcpy (rom_2,"C"); break;
  case 200: strcpy (rom_2,"CC"); break;
  case 300: strcpy (rom_2,"CCC"); break;
  case 400: strcpy (rom_2,"CD"); break;
  case 500: strcpy (rom_2,"D"); break;
  case 600: strcpy (rom_2,"DC"); break;
  case 700: strcpy (rom_2,"DCC"); break;
  case 800: strcpy (rom_2,"DCCC"); break;
  case 900: strcpy (rom_2,"CM"); break;
 }             // switch 2 Ende
 stelle_3=arab_zahl/1000;
 switch (stelle_3)
 {
  case 1: strcpy (rom_3,"M"); break;
  case 2: strcpy (rom_3,"MM"); break;
  case 3: strcpy (rom_3,"MMM"); break;
 }             // switch 3 Ende
 if (stelle_3==0)
 {
  if (stelle_2==0)
  {
   if (stelle_1==0)
    strcpy(rom_zahl,rom_0);
   else
    {
```

```
      strcpy (rom_zahl,rom_1);
      if (stelle_0!=0)
       strcat (rom_zahl,rom_0);
     }
    }
    else
   {
    strcpy (rom_zahl,rom_2);
    if (stelle_1!=0)
     strcat (rom_zahl,rom_1);
    if (stelle_0!=0)
     strcat (rom_zahl,rom_0);
   }
  }
  else
  {
    strcpy (rom_zahl,rom_3);
    if (stelle_2!=0)
     strcat (rom_zahl,rom_2);
    if (stelle_2!=0)
     strcat (rom_zahl,rom_1);
    if (stelle_2!=0)
    strcat (rom_zahl,rom_0);
  }

 printf("\n\n%i als römische Zahl dargestellt: %s",arab_zahl,rom_zahl);

/* printf("%s",rom_2);
 printf("%s",rom_1);
 printf("%s",rom_0); */

 do
 {
  printf("\nBitte geben Sie eine positive Zahl bis 4000 ein, ");
  printf("\nwollen Sie das Programm beenden, so geben Sie eine 0 ein: ");
  scanf("%i", &arab_zahl);
 } while ((arab_zahl<0)&& (arab_zahl>=4000));
 }           // while Ende
}            // main Ende
```

7. Zahlenpyramide

Eine Zahlenpyramide sieht wiefolgt aus:

```
          1
         232
        34543
       4567654
      567898765
     67890109876
        usw.
```

```
// Zahlenpyramide
# include <stdio.h>
void main (void)
{
 int x=0, i, k, l, j;
 for (j=1; j<=10; j++)
 {
  printf("\n");
  x=j;
  for (i=1; i<=10-j; i++)
   printf(" ");
  for (k=1; k<= (2*j/2); k++)
```

```
   printf("%d",((x++%10)));
  x=x-1;
  for(; (k>=1) && (x>j);k--)
   printf("%d",((--x%10)));
   printf("       k=%d, j=%d",k,j);
 }
}
```

8 Struktogramme und Flußdiagramme

8.1 Struktogramme nach Nassi-Schneiderman

Hilfsmittel und Gegenstand bei der Methode der schrittweisen Verfeinerung zur Entwicklung strukturierter Programme

⇒ stepwise refinement

⇒ structured programming

Ziel:

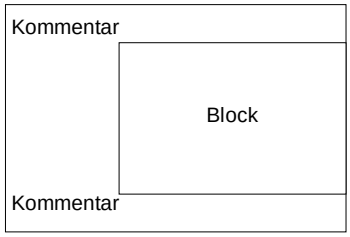
- übersichtliche, kompakte Darstellung der Logik, des Ablaufs eines Programms, eines Unterprogramms, eines Blocks auf einer Seite
- Auslagerung von Blöcken (Verfeinerung)
- getrennte, unabhängige Verfeinerung von Blöcken
- bis zu einem Verfeinerungsgrad, der möglichst direkte Umsetzung in Programmcode erlaubt.

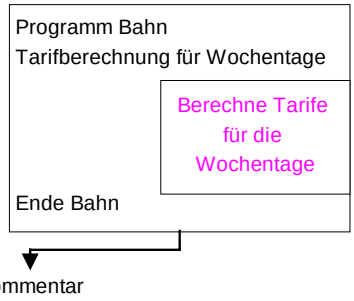
(neuerdings unterstützt durch rechnergestützte Werkzeuge: Compiler aided software engineering tools: case-tools)

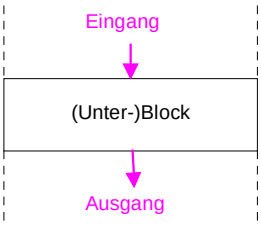
einige Schritte der Softwareentwicklung (in einer Firma)

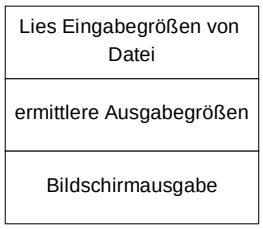
- Aufgabenstellung lesen und verstehen
- Der Softwareentwickler muß seinen ersten Termin mit dem Kunden ausmachen, um mit ihm zu prüfen, ob er die Aufgabenstellung richtig verstanden und interpretiert hat.
- Der Entwickler überlegt sich, wie er die Aufgabenstellung in die Tat umsetzten kann.
    - Problematik: der Programmierer arbeitet die Teile der Aufgabenstellung besonders aus, die sein Spezialgebiet sind. Andere Teile des Programms werden zeitweise nicht richtig ausgearbeitet.
- Programmierer muß sich wieder mit Kunden in Verbindung setzten und mit ihm die Umsetzung besprechen. Sonst kann es sein, daß der Programmierer verkehrte Schwerpunkte setzt oder sogar Programmteile falsch programmiert.

Das Problem, daß Softwareentwicklung zu teuer war und das Programmierer verkehrte Schwerpunkte setzten usw. waren mit Gründe für die sog. Softwarekrise.
(100% Programm, davon wurden 50% nicht oder nur selten benötigt, 30% waren falsch programmiert, nur die restlichen 20% konnte der Betrieb wirklich ausnutzen ⇒ Folge: Software war zu teuer (für 20%))



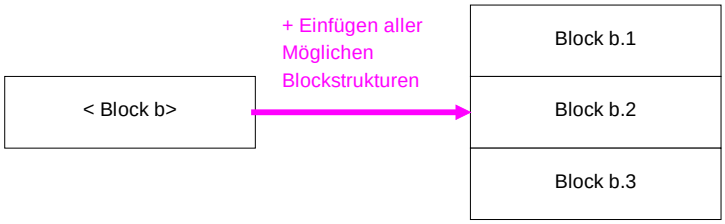


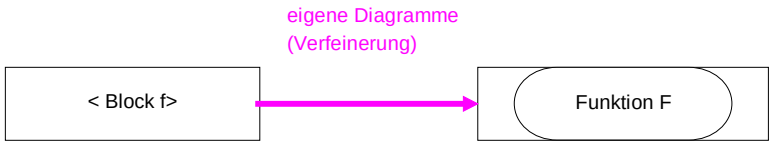


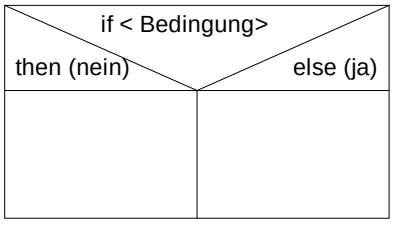


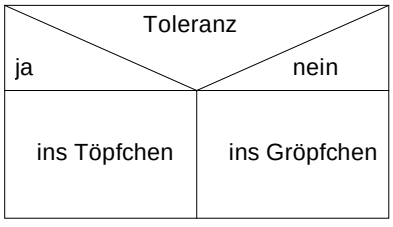
1. Verfeinerung

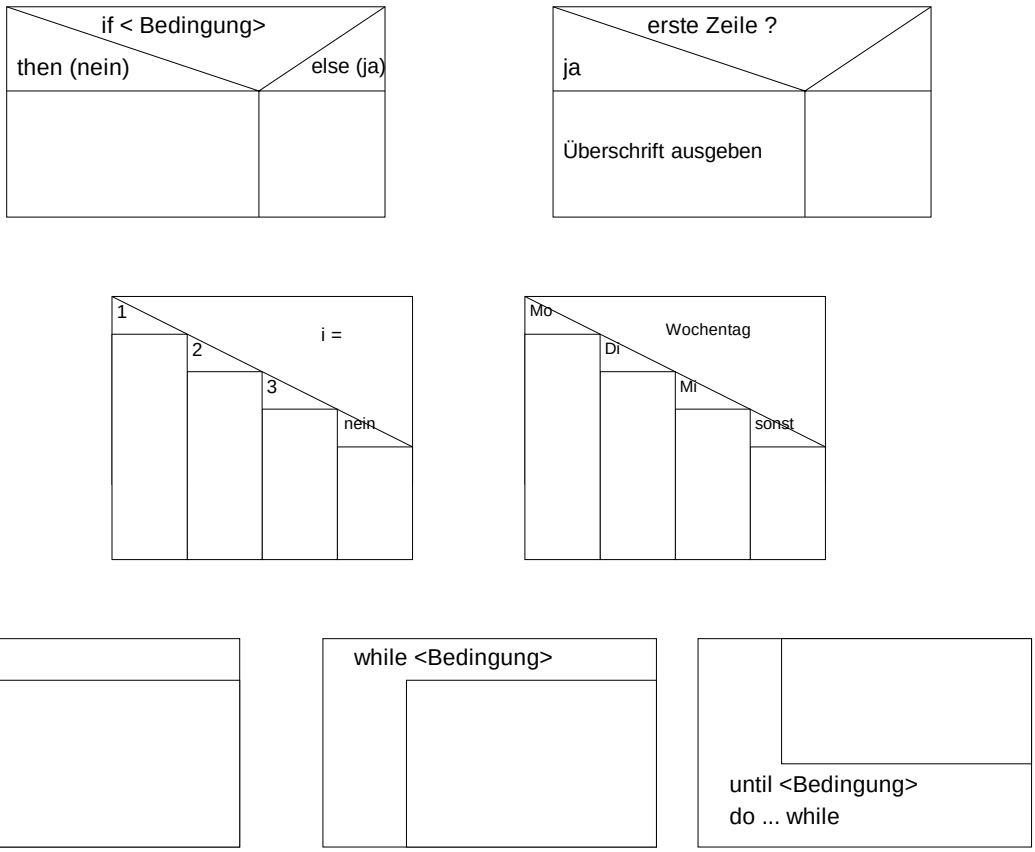
Allgemeine Verfeinerung:







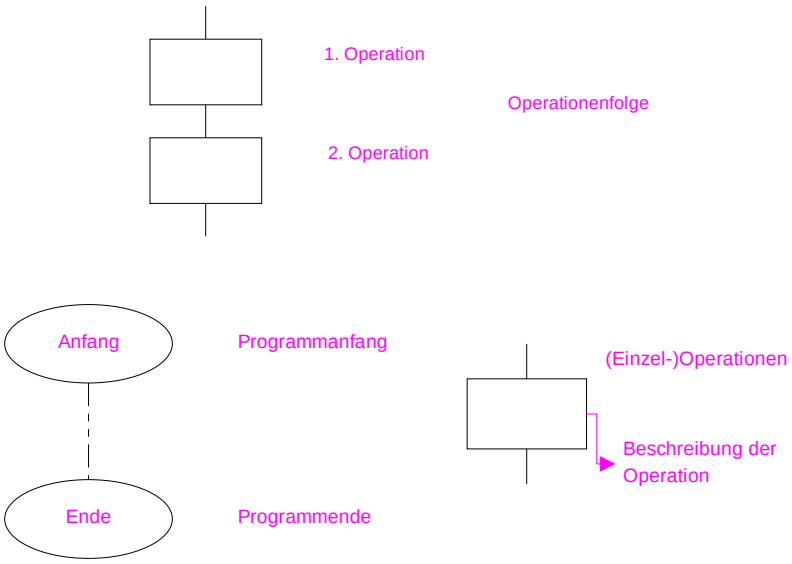


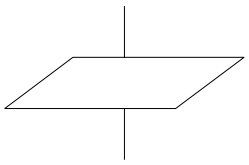


Vorteile der Nassi-Schneiderman:

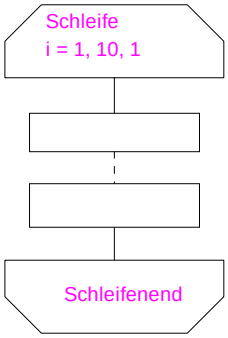
- Übersichtlich, weil mehr Text eingeführt werden kann
- Nassi-Schneiderman-Struktogramme sind / können übersichtlich auf einer DIN A4 Seite angegeben werden ⇒ durch Verfeinerung werden sie länger

8.2 Flußdiagramme



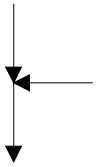


Einlesen / Ausgeben

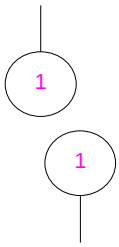


i wird mit 1 initialisiert, der Endwert ist 10 und die Schrittgröße ist 1

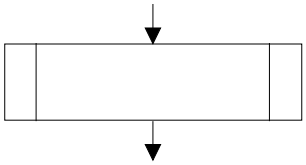
Anweisungen



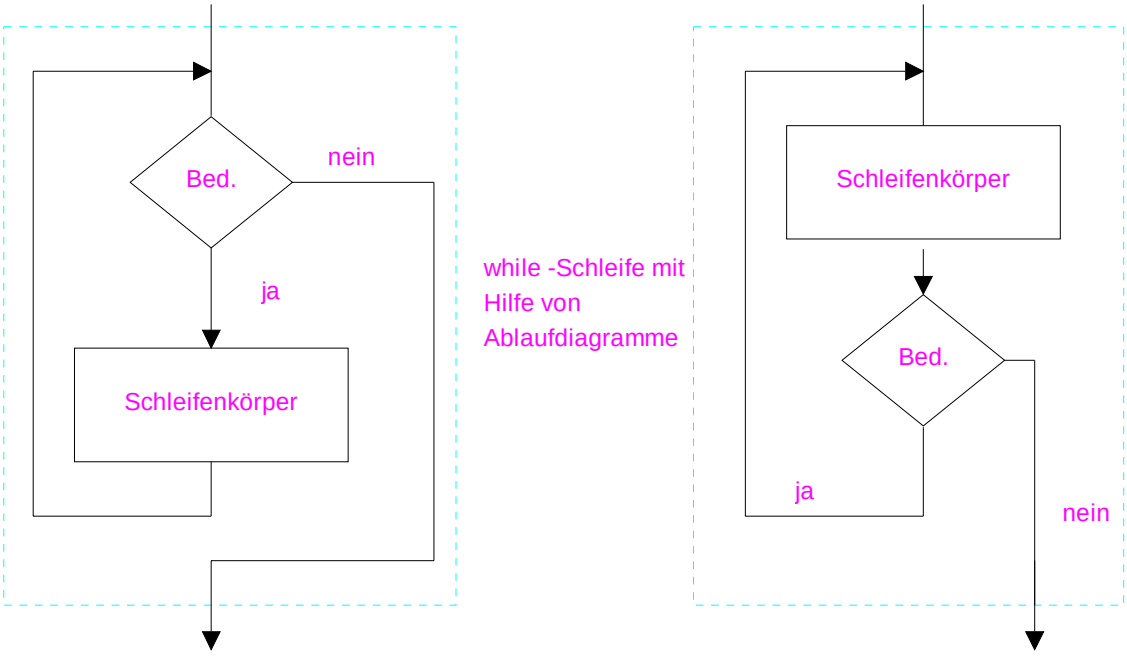
Zusammenführung



Übergangsstelle / Fortsetzungsstelle bei umfangreichen Diagrammen



Funktionsaufruf



Problematik von Flußdiagrammen:

- Nicht jedes Flußdiagramm kann man unmittelbar, ohne größere Probleme in einer modernen Programmiersprache umsetzten

Wohl strukturierte Flußdiagramme können in Nassi-Schneiderman - Struktogramme überführt werden.

8.3 Aufgaben

Schreiben Sie Algorithmen / Programme, die folgendes leisten:
1) In jedem Durchlauf Erstellen einer Liste der Potenzen zur Basis n (Eingabe) mit dem max. Potenzwert m (Eingabe)

gefällige Ausgabe;
Bediendialog für Fortsetzung / Abbruch
2) Ermitteln Sie für Aufgabe 2) den Index, der die Elemente in gleichgroße Summen teilt. (Mediumindex)

Bearbeitung der Aufgabe 1:
Zuerst Formulierung einer ungeordneten Liste von Anforderungen:

- Erstellen einer Liste von Potenzen
- Eingabe der jeweiligen Basis
- Eingabe des jeweiligen max. Potenzwertes
- gefällige Ausgabe
- wiederholter Durchlauf
- Bediendialog für wiederholten Durchlauf

- Berechne Liste (n-Elemente)

♦ Anfangsexponent auswählen
♦ Datentypen
♦ Basis ≥ 2 ?
♦ Speicherstruktur für Liste
♦ Schrittweite für Exponenten

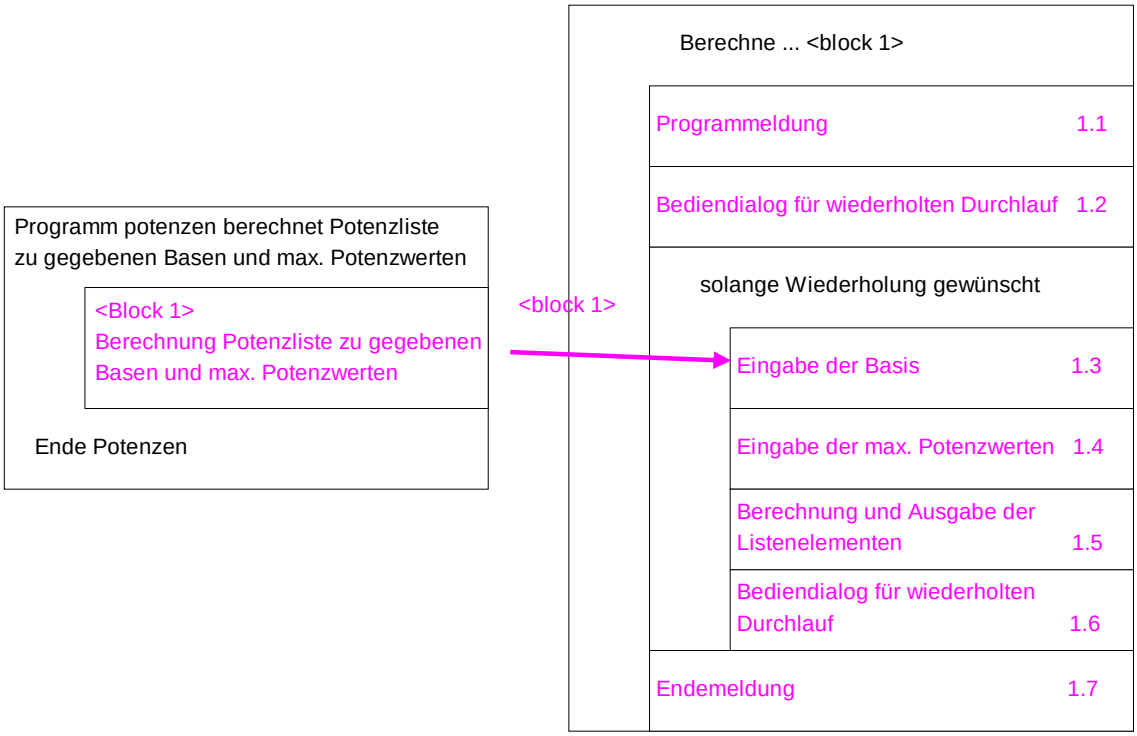
♦ steht für Regieanweisungen zur Überlegung des Programms ≠ zur Programmanforderung •

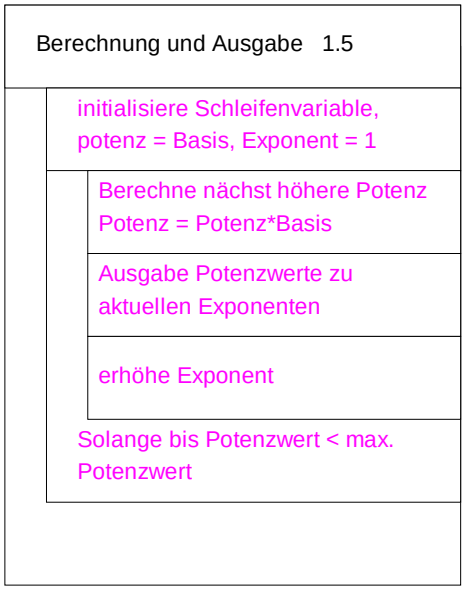
Aufgabensammlung (mit Reihenfolge):

1. Bediendialog für wiederholten Durchlauf
2. Wiederholter Durchlauf
3. Eingabe der Basis
4. Eingabe des max. Potenzwertes
5. Berechnung der Potenzwerte
6. gefällige Ausgabe der Potenzliste

Entscheidungen / Überlegungen (die nicht explizit in der Aufgabenstellung enthalten, aber bei der Programminitialisierung zu beachten sind):

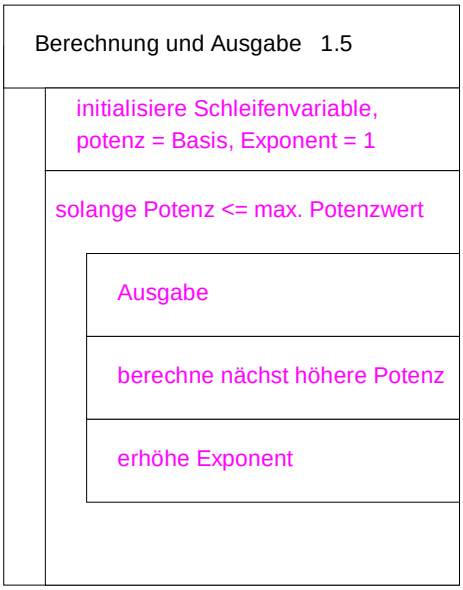
- Typ / Wertebereich der Basis
- Wertebereich des Exponenten
- Entscheidung für Wiederholungsstruktur
- Erzeugung / Speicherung / Ausgabe der Listenelemente





do...while-Schleife

oder



while-Schleife

```
// Aufgabe 1: Potenzberechnung
#include <iostream.h>
#include <math.h>
int main (void)
{
 int potenz, wert, basis;
 long ergebnis;
 cout << "\nDiese Programm berechnet die Potenzen einer angegebenen Zahl";
 cout << "\nBitte geben Sie die Basis ein: ";
 cin >> basis;
```

```
 cout << "Bitte geben Sie an wie viele Potenzen zur angegebenen Basis";
 cout << "\nberechnet werden sollen: ";
 cin >> potenz;
 ergebnis=1;
 cout << "0.Potenz: \t" << ergebnis;
 for (wert=1; wert<=potenz; wert++)
 {
  ergebnis=ergebnis*basis;
  cout << "\n" << wert << ".Potenz: \t" << ergebnis;
 }
 return 0;
}
```

9 Unbedingte Fortsetzungen mit goto, continue, break, return

9.1 goto marke

Herkunft:
Assembler, Fortran, Basic

Wirkung:
Unbedingte Fortsetzung des Programmlaufs bei der durch Marke angezeigten Stelle innerhalb derselben Funktion.

Anwendung:
Möglicher, aber nicht nötiger Anwendungsfall kann das Verlassen einer tiefen Schachtelung sein, wenn Fehler z.B. bei Funktionsaufrufen, eingetreten sind.

Vereinbarung:
Im Rahmen der Vorlesungen und Prüfungen Programmieren I und II verboten.

Beispiel:

```
for (...)
{...
 while (...)
 {
 if (schwerer_fehler) goto fehlerbehandlung;
 }
}
fehlerbehandlung: fehlerbehandlungsanweisung;
```

9.2 continue

Bedeutung:
Beendet aktuellen Schleifendurchlauf - setze Schleife mit neuen Durchlauf fort.

Wirkung:
Abbrechen des aktuellen Schleifendurchlaufs, nicht der ganzen Schleifenanweisung.

Anwendung:
Möglicherweise zur Zeitersparnis durch vorzeitigen Abbruch von Schleifendurchläufen.

Vereinbarung:
Anwendung im Rahmen der Vorlesungen und Prüfungen Programmieren I und II verboten.

Grund:

In der strukturierten Programmierung kein Beschreibungskonstrukt dafür definiert. Es wird Blockbildung verletzt - ein Block besitzt nur einen Eingang und einen Ausgang

Beispiel:

```
// Programm anz_ohne
// Beispiel für die Anwendung von continue
// Das Programm zählt die Zeichenvorkommen != Zeichen in e (Variable)
//

void                                main                                (void)
{
 const maxstr=20;
 char str[maxstr]="Maximilian";
 char c= 'i';                   // Vergleichszeichen
 int anzahl=0, anz_ohne_c=0;
 while (str[anzahl])     // solange stringende nicht erreicht
 {
  if  (str[anzahl]++]= =c) continue;
                                   // beende Schleifendurchlauf falls = = c
  anz_ohne_c=anz_ohne_c+1;           // sonst erhöhe Zähler
 }
}
```

Wie kann man auf dieses continue verzichten?
Wodurch kann man es ohne Verlust ersetzten?

Antwort:

```
...
 if (str[anzahl]!=c)                    // erhöhe Zähler wenn Zeichen != c
  anz_ohne_c=anz_ohne_c+1;
anzahl=anzahl+1;                        // erhöhe feldindex
...
```

9.3 break

Bedeutung / Wirkung:
Bricht die Bearbeitung des aktuellen umgebenden switch-Blocks oder der umgebenden Schleifen ab. Fortsetzung bei der Anweisung, die dem switch bzw. der Schleife folgt.

Anwendung:
Realisierung einer echten Mehrfachverzweigung in switch; Abbruch einer Schleife beim Eintreten von Ablaufbesonderheiten, evtl. von Fehlern.

Vereinbarung:
Nur bei der switch-Anweisung verwendbar.

9.4 return

Bedeutung / Wirkung / Anwendung:
Verlassen der aktuellen geöffneten Funktionsschachtel mit der Möglichkeit zur Rückgabe eines Wortes.

Zwingend vorgeschrieben für Wertübergabe an aufrufende Funktion, für void-Funktionen kann return-Angabe unterbleiben;
Evtl. sinnvoll zur Markierung des Funktionendes.

Syntax:

```
return ausdruck;
```

Anwendung:

- Typ von ausdruck muß gleich sein zu oder konvertierbar sein in Typ der Funktion.
- return ohne ausdruck nur erlaubt bei Funktion ohne Rückgabeparameter, d.h. Typ void.
- Wichtig bei Wertrückgabe ist tatsächlicher Durchlauf durch return-Anweisung.

Beispiel:

```
/* Funktion max_2 zur Berechnung des Maximums zweier integer Werte */
```

```
int max_2 (int a,int b)
{
  int max;
  if (a<b)
   max = a;
  else
   max = b;
  return max;
}
```

```
int max_2 (int a, int b)
{
 if (a<b)
   return a;
 else
   return b;
}
```

```
int max_2 (int a, int b)
{
 return (a<b) ? a:b;
}
```

Aus der letzten Spalte ist ersichtlich, daß man direkt den Wert eines Ausdrucks zurückliefern kann und diesen nicht vorher in Variablen abspeichern muß. Dies gilt auch für boolesche Ausdrücke, wie z.B. 'a<b'.

Beispiel:

```
bool test (int a, int b)
{
 bool test1;
 test1=a < b;
 return test 1;
}
```

oder

```
bool test (int a, int b)
{
  return a < b;
}
```

10 Funktionen

10.1.1 Unterprogramm

- Algorithmenschemata
- zusammengehörige Anweisungen
- Wiederholung als "Block"
- Eingabe, Parameter

10.1.2 Wirkungsmöglichkeiten von Unterprogrammen

- "feste Funktion", festgelegte Aktionenfolge ausführen
- mit denselben Objekten ⇒ Konstante Operation - Parametierung, Parameter
- Veränderung von Programmvariablen, Ein- / Ausgabeoperationen

Regeln:

- nur explizit benannte Objekte sollen verwendet werden
  Parameter
  lokale Variablen
- Unterprogramme sollen möglichst wenige versteckte globale Auswirkungen haben; keine Seiteneffekte (außer Ein- / Ausgabe)

10.1.3 Deklaration und Definition von Funktionen

```
typ funktionsname (parameterliste)
{
  Deklaration lokaler Variablen;
  Anweisung_mit return;
}
```

Typ:

- skalarer Datentyp
- Zeigertyp (beinhaltet auch Übergabe eines Feldes)
- strukturierte Datentypen möglich

Parameterliste:

- in C nur Wertparameter
- in C++ auch Referenzparameter
- es muß vor jedem Parameter der Typ geschrieben werden; die verschiedenen Parameter werden von Kommas abgetrennt

  Beispiel
  ```
  int funk (int var1, char var2, float var3);
  ```

Deklaration lokaler Variablen:

- alle Variablen, die hier deklariert werden, werden erst zur Laufzeit der Funktion erzeugt, d.h. Speicherplatz wird angelegt

- bei Beenden der Funktion wird der Speicherplatz freigeben / anderweitig verwendet

lokale Variablen verlieren ihre Gültigkeit

⇒ Gültigkeitsbereich für lokale Variablen und Parameter ist das Unterprogramm bzw. die Funktion

10.1.4 Anweisungen

- Anweisungsfolge, die die Operationen, die Aktionen der Funktion wiedergeben / beschreiben
- repräsentieren funktionale Abstraktion durch Funktionsname und Aufruf
- Ergebnisberechnung und Wertübergabe durch return-Anweisung

10.1.5 Ort der Deklaration

- Funktionsdeklaration auf einem Niveau
- keine geschachtelten Funktionsdeklarationen
- vor der Verwendung einer Funktion muß deren Deklaration erfolgt sein
    - vollständige Definition (Kopf inklusive Anweisungen)
    - Deklaration des Funktionsprototypen (z.B. auch in einer .h-Datei)
      in diesen Fall Nachholen der Definition nach Funktionsrumpf mit Aufruf (kann auch in einer Bibliotheksdatei .lib abgelegt sein)

10.1.6 Funktionsaufruf

Stellen für Funktionsaufruf:

- Funktionsaufruf an allen Stellen im Programm, an deren ein Operand des entsprechenden Datentyps erlaubt ist
- (mathematische Funktionen)
  ```
  a=funktions_name(aktual_parameterliste);
  ```
- Funktionsaufruf in Form eines Ausdrucks als eigenständige Anweisung
  ```
  funktions_name(aktuelle Parameter);
  ```

10.1.7 Aktualparameterliste

- beschreibt die Versorgung der Funktion mit aktuellen Parametern
- aktuelle Parameter:

  Ausdrücke des entsprechenden Parametertyps
    - Variablen
    - Konstanten
    - Funktionsaufrufe
    - komplexe Ausdrücke

10.1.8 Formalparameterliste

- Stelligkeit ist relevant

    - wie viele Parameter
    - von welchem Typ sind entsprechende Parameter
- formale Parameter sind Platzhalter, Stellvertreter für die im Aufruf verwendeten aktuellen Parameter
- alle Operationen, die in der Deklaration auf dem Formalparameter ausgeführt werden, werden zur Laufzeit der Funktion auf / mit dem Wert der aktuellen (aktual) Parameter ausgeführt.

Beispiel:

```
...
float kreisflaeche (float radius)
{
 const pi=3.14159;
 return radius*radius*pi;
}
...
void main (void)
{
 float r, h, zv, u, r1, r2, f;
 ...
   // Eingabe der Variablenwerte für r, h, r1, r2
 zv=kreisflaeche(r)*h;
 ...
}
```

10.1.9 Funktionsprototyp

Durch einen Funtionsprototypen wird eine Funktion deklariert. Sie wird dort noch nicht definiert.
Ein Funktionsprototyp sagt dem Compiler nur, daß später noch eine Funktion definiert wird, die die folgenden Parameter, den Rückgabewert und Funktionsnamen hat.
Ein Prototyp wird mit einem Strichpunkt abgeschlossen.

Beispiele:

```
void hallo (void);
int drucken (int, int);  //zwei Integerzahlen werden als Parameter erwartet
double weg (int a, double b);
```

10.1.10Parameterübergabe

- In C standardmäßig Wertparameter (parameter call by value)
- In C++ zusätzlich Adreßparameter (parameter call by reference)

Skalar als Aufruf-(Aktual)-Parameter:

beim Aufruf einer Funktion mit Skalar als Parameter wird der Wert übergeben:

```
int max_2 (int a, int b)
{
 if (a>=b)
  return a;
 else
  return b;
}
```

Aufruf:
`u=v+max_2(x,y)*w;`

```
int max (int x, int y)

{
 if (x>=y)
  return x;
 else
  return y;
}
```

Ausdruck als Aktualparameter:

Ausdruck auswerten - Wert bestimmen - Formalparameter

```
int max_2 (int a, int b)
{
 if (a>=b)
  return a;
 else
  return b;
}
```

Aufruf :
`u=v+max_2(x+y,x-y)*w;`

```
int max ( ,     )          // int max (x+y, x-y)
{
 if (x+y >= x-y)
  return x+y;
 else
  return x-y;
}
```

Jedes Vorkommen von a, b in der Definition bei wird bei Aufruf durch Wert von x+y, x-y ersetzt.

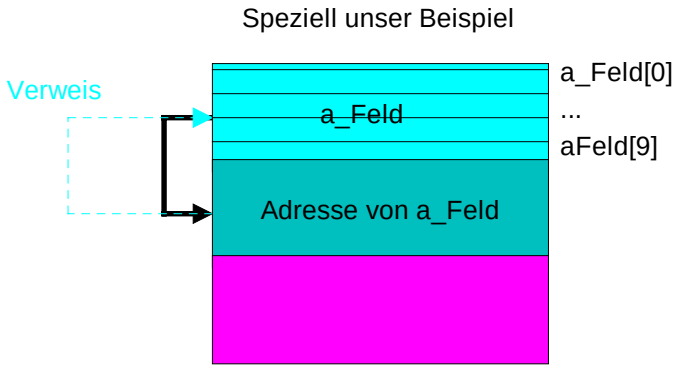
Felder als Aktualparameter:

Übergabe der Feldadresse - nicht Kopie des Feldes als Wertparameter

Deklaration:

```
int komp_sum (int feld[10])
{
   int sum = 0, i;
   for (i=0; i<10; i++)
     sum = sum + feld [i];
   return sum;
}
...
int a_feld [10]...;
...
u=v+komp_sum(a_feld)*w;
```

⇒ Implizite Behandlung als Referenz-Parameter:
Übergabe der Adresse, Dereferenzieren bei Zugriff auf Komponente



10.1.11Wertrückgabe über Parameter

Über Wertparameter keine Veränderung einer Variablen der aufrufenden Funktion möglich (außer bei Feldern), da nur Wert übergeben wird und Adresse verloren geht.

Wertrückgabe über Parameter ⇒ Notwendigkeit der Adreßübergabe
da nur Wertparameter (in C) ⇒ Adreßübergabe der Zielvariablen als Wertparameter vom Typ Zeiger auf...

Beispiel:

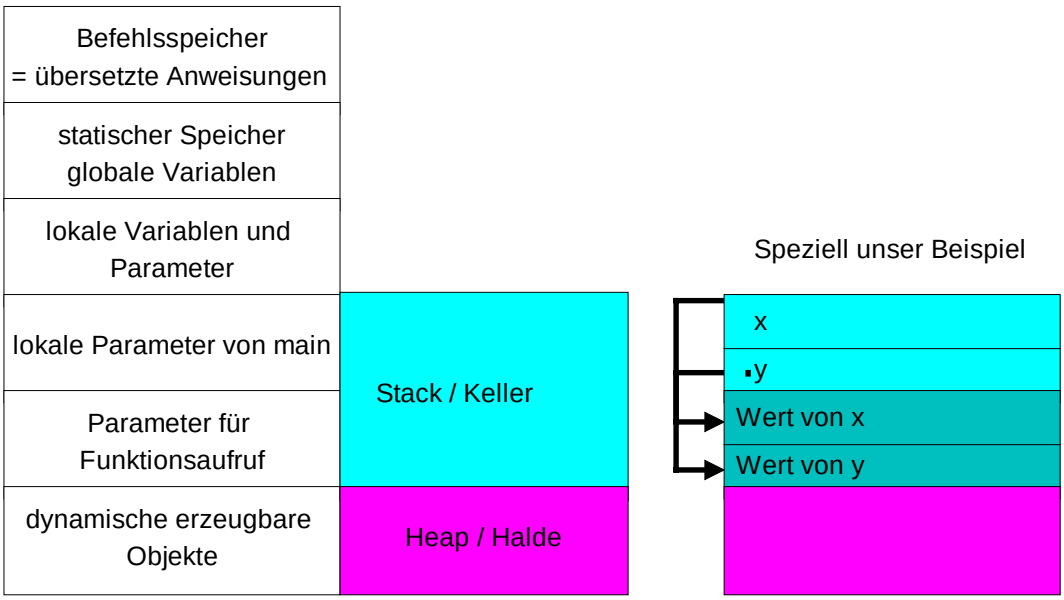
```
// Deklaration
void max_2 (int a, int b, int *ergeb) // Zeiger auf Ergebnisvariable
{
 if (a>b)
  *ergeb=a;                          // Dereferenzieren
 else                                // und
  *ergeb=b;                          // Speichern
 }

// Aufruf und Verwendung
{
 int a, b, c, u, v, w;
 ...
 a=...;
 b=...;
```

```
 ...
 max_2(a,b,&c);                    // Übergabe der Adresse
 ...                               // und
 u=v+c*w;                          // Benutzung des Wertes
 ...
}
```

10.2 Gültigkeitsbereiche von Variablen - lokale und globale Variablen

Zunächst sollte man sich mit dem Aufbau des Programmspeichers auseinandersetzen.

Programmspeicher:



Es gibt verschiedenen Arten von Variablen: lokale und globale Variablen.
Globale Variablen haben im ganzen Programm Gültigkeit, d.h. sie sind in allen Funktionen und in der "Main-Funktion" gültig.
Lokale Variablen hingegen sind nur in der Funktion gültig.

Beispiel:

```
Includedateien (#include<...>)
Deklaration von globalen Variablen

typ1 fkt_1 (Parameter von fkt.1)
{
  Deklaration von lokalen Variablen von fkt_1
  ...
}

typ fkt_2 (x)
{
  Deklaration von  lokalen Variablen von fkt_2
  ...
```

```
}

int main (void)
{
  lokale Variablen von main
  int x;
  fkt_2(x);
  ...
}
```

11 Eingabe- und Ausgabefunktionen

11.1.1 getchar( )

Prototyp:

```
int getchar(void);
```

- Rückgabe eines Zeichens vom Standardbediengerät (Tastatur-Eingabe)(

Anwendungsbeispiel:

```
c=getchar( );       // Zeichen einlesen, dann an c zuweisen
```

11.1.2 putchar( )

...

11.1.3 gets (str)

Prototyp:

```
char *gets(char *s);
```

- Einlesen eines mit <ret> abgeschlossenen Strings in die Feldvariable, deren Zeiger als Paramter übergeben wird: Stringende wird \0
- Rückgabe des Zeigers auf den eingelesenen String als Funktionswert
- ohne Fehler: Wertrückgabe des Zeigers, sonst Wertrückgabe des Zeiger NULL

Beispiel:

```
/* Zeichenfeld zeiger auf char / Zeichen */
char str[80], *s, *t;
t = str;
s = gets (t);
```

11.1.4 scanf ( )

Prototyp:

```
int scanf (char *formatierung, arg1, arg2,...)
```

- Einlesen einer Zeichenfolge von der Standardeingabe
- Umwandlung entsprechend der Angaben im formatstring
- Zuweisen der Eingabe an die Argumente arg1, arg2, ...
- arg1, arg2, ... sind Zeiger (da Rückgabeparameter)

einfaches Beispiel:

```
int i;
scanf ("%d", &i);
```

%d	Zeiger auf int	dezimale Eingabe

%D	Zeiger auf long	dezimale Eingabe
%u	Zeiger auf unsigned int	dezimale Eingabe
%U	Zeiger auf unsigned long	dezimale Eingabe
%h	Zeiger auf short integer	
%e	Zeiger auf float	Gleitkommazahl
%E	Zeiger auf float	Gleitkommazahl
%f	Zeiger auf float	Gleitkommazahl
%g	Zeiger auf float	Gleitkommazahl
%G	Zeiger auf float	Gleitkommazahl
%c	Zeiger auf char	einzelnes Zeichen
%s	Zeiger auf string	Zeichenkette
%i	Zeiger auf int	dezimale, oktale oder hexadezimale Eingabe
%I	Zeiger auf long	dezimale, oktale oder hexadezimale Eingabe
%x	Zeiger auf unsigned int	hexadezimale Eingabe
%X	Zeiger auf unsigned int	hexadezimale Eingabe
%o	Zeiger auf int	oktale Eingabe
%O	Zeiger auf long	oktale Eingabe
%n	Zeiger auf int	Anzahl der bisher eingelesenen Zeichen

Die oktale Eingabe beginnt mit 0 (Ziffer Null), die hexadezimale mit 0x beim Format %x bzw. mit 0X beim Format %X.

Beispiele:

```
int integer;
char string[5];
float gleit;

scanf("%i",&integer);
scanf("%s",string);
scanf("%f",&float);
```

Was ist bei der Eingabe von Zeichenketten mit scanf zu beachten?

Ohne Zusätze ist es mit scanf nicht möglich eine Zeichenkette mit Leerzeichen einzugeben. Sobald ein Leerzeichen eingegeben wurde, wird die Eingabe als beendet angesehen.

Beispiel:

```
...
char string[80];
scanf("%s",string);
printf("%s",string);
...
```

Eingabe:

```
Hallo Leute
```

Ausgabe:

```
Hallo
```

Man kann nun bei der scanf-Funktion angeben, welche Zeichen sie als zugehörig für diese Eingabe berücksichtigen soll. Sobald ein anderes Zeichen vorkommt wird die Eingabe beendet.

Beispiel:

```
...
char string[80];
scanf("%[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]",string);
printf("%s",string);
...
```

Eingabe:

ALLES GUTE zum Geburtstag

Ausgabe:

ALLES GUTE

"zum Geburtstag" wird nicht mehr berücksichtigt, da "z" ein Buchstabe ist, der nicht berücksichtigt wird.

Es wäre nun eine sehr umfangreiche Aufgabe jedesmal die erlaubten Zeichen anzugeben. Deshalb gibt es auch eine andere Möglichkeit.
Es soll nun eine Zeichenkette eingegeben werden. Sie darf alle Zeichen beinhalten und wird durch <Return> abgeschlossen. Dies kann man durch folgende Eingabe erreichen [^\n] (alles außer <Return>).

Beispiel:

```
...
char string[80];
scanf("%[^\n]",string);
printf("%s",string);
...
```

Eingabe:

ALLES GUTE zum Geburtstag

Ausgabe:

ALLES GUTE zum Geburtstag

Warum muß man vor einer Zeichenkette nicht den Adreßoperator schreiben?

Beispiel

```
char string[20];
...
scanf("%s",string);
```

gleichbedeutend:

```
scanf("%s",&string[0]);
```

"string" wurde oben als Feld [20] definiert. Hierbei ist die Variable "string" präziser eigentlich nur ein *Zeiger* auf das erste Element des Felds, hier das erste Zeichen. Der Adreßoperator *muß* hier entfallen, da ja "string" bereits ein Zeiger ist!

Will man dagegen nur ein Zeichen in die Zeichenkette hineinschreiben, so muß man hierbei das Feldelement angeben, den entsprechenden Formattyp '%c' und den Adreßoperator verwenden.

Beispiel:

```
char string[80];
scanf("%c",&string[5]);
printf("%c",string[5]);
```

Die Anweisung
`scanf("%s", &string[5]);`
würde hingegen einen String erwarten und diesen ab dem Element Nr. 5 des Strings abspeichern. Die ersten Elemente 0..4 würden unberührt bleiben!

Vergißt man bei der scanf-Anweisung den &-Operator, dann wird der eingegebene Wert nicht im Speicher an der Adresse der beabsichtigten Variable abgelegt, sondern an der Adresse, die dem Wert der Variablen gerade entspricht! Dies ist eine häufige Fehlerquelle mit sehr schwerwiegenden Auswirken, meistens Programmabsturz!

Verschiedene Eingabemöglichkeiten:

```
#include<stdio.h>
char string[20];
int    zahl;
float kosten;
void main (void)
{
  ...
 scanf("%s %d %f",string, &zahl, &kosten);
 ...
}
```

Mögliche Eingabeformen

hallo 122345 0.50	hallo 122345 0.50	hallo 122345 0.50	hallo 122345 0.50

11.1.5 printf( )

%d	Ausgabe von int	dezimale Ausgabe
%i	Ausgabe von int	dezimale Ausgabe
%u	Ausgabe von unsigned int	dezimale Ausgabe
%e	Ausgabe von float	Mantisse e Exponent
%E	Ausgabe von float	Mantisse E Exponent
%f	Ausgabe von float	Gleitkommazahl, ohne Exponent
%g	Ausgabe von float	Gleitkommazahl mit oder ohne Exponenten unter Streichung unnötiger abschließender Nullen

%G	Ausgabe von float	Gleitkommazahl mit oder ohne Exponenten unter Streichung unnötiger abschließender Nullen
%c	Ausgabe von char	einzelnes Zeichen
%s	Ausgabe von string	Zeichenkette

Beispiele:

```
char charakter;
char string[5];
int integer;
float gleit;

printf("%c",charakter);
printf("%s",string);
printf("%i",integer);
printf("%f",gleit);
```

11.2 Aufgaben

1. Schreiben Sie ein Programm, das folgendes leistet:

   main:

   - Eingabe eines 2-dimensionalen Feldes mit [5] [5]
   - Eingabe eines Vergleichwertes

   Funktion:

   - Vergleich der Feldelemente mit Vergleichswert
   - Ausgabe / Rückgabe der Indizes, des Feldelementes, das betragsmäßig den kleinsten Abstand zum Vergleichselement besitzt

   Hinweis: Verwenden sie die Funktion abs( ) aus der Include-Datei "math.h"

Meine Version:

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

// Funktion vergleich berechnet, welches Feldelement den betragsmäßig
// kleinsten Abstand zum Vergleichselement hat.
// Die Funktion gibt dann die Adresse des berechneten Feldelements zurück.
int *vergleich (int feld[3][3], int vergleichvar)
{
 int vergleich_variable, *kleinste_adresse, kleinster_wert;
 vergleich_variable=vergleichvar;

 for (int j=0; j<=2; j++)
 {
  for (int i=0; i<=2; i++)
  {
   if ((j==0) && (i==0))
   {

     // Vorbelegung der Variablen kleinste_adresse und kleinster_wert

     kleinste_adresse=&feld[j][i];
     kleinster_wert=abs(vergleich_variable-feld[0][0]);
   }
   else
   {
```

```
    if (abs(vergleich_variable-feld[j][i])<=kleinster_wert)
    {

      // Falls ein Element gefunden wurde, daß einen betragsmäßig kleineren
      // Abstand zum Vergleichselement als das vorherig berechnete,
      // wird der Variable kleinste_adresse die Adresse des Feldelements
      // Übergeben. Der Variable kleinster_wert wird der Wert des
      // Feldelements Übergeben.

      kleinste_adresse=&feld[j][i];
      kleinster_wert=abs(vergleich_variable-feld[j][i]);
    }
   }
  }
 }
 return kleinste_adresse;
}


// Hauptprogramm

int main (void)
{
 int feld[3][3], vergleichvar, *adresse, *adresse_vergl;

 // Eingabe der Feldelemente und des Vergleichswertes

 cout << "Bitte geben Sie eine 3 x 3 Matrix aus ganzen Zahlen ein: \n";
 for (int j=0; j <= 2; j++)
 {
  for (int i=0; i<=2; i++)
  {
   cout << "Bitte geben Sie die Zahl "<<j<<" "<<i<<" ein: ";
   cin >> feld[j][i];
  }
 }
 cout << "Bitte geben Sie nun eine Zahl, mit der alle Elemente der Matrix"
     "\nverglichen werden sollen: ";
 cin >> vergleichvar;

 // Aufruf der Funktion vergleich

 adresse=vergleich(feld,vergleichvar);

 // Da die Funktion vergleich nur die Adresse des gesuchten
 // Feldelements übergibt, muß nun gesucht werden, zu welchen
 // Indizes die von vergleich übergebene Adresse gehört.
 // Hierfür mache ich einen Adressenvergleich.

 for (int x=0; x <=2; x++)
 {
  for (int y=0; y<=2; y++)
  {
   adresse_vergl=&feld[x][y];
   if (adresse_vergl==adresse)
    cout << "\nDas Element mit folgenden Indizes hat den betragsmäßig"
            "\nkleinsten Abstand zum Vergleichselement : "<<x<<" "<<y;
  }
 }
 return 0;
} // Ende Hauptprogramm
```

Version von Dr. Eck:

```
// mindist-Programm zur Berechnung des Feldelements mit dem
// kleinsten Betragsabstand zu einem Vergleichswert
```

```
#include <iostream.h>
#define zeilen 3
#define spalten 3

float abs (float x)
{
 if (x<0)
  return (0.-x);
 else
  return (x);
}

// Funktion mindist zur Bestimmung des Feldelements mit der
// minimalen Distanz zu einem Vergleichselement

float mindist (float feld[zeilen][spalten], int *z_index,
          int *s_index, float v_wert)
{
 int z, s, i, j;
 float min, dist;
 min=abs(feld[0][0]-v_wert);       // Vorbestimmung des Distanzminimalwerts
 for (i=0; i<zeilen; i++)
 {
  for (j=0; j<spalten; j++)
  {
   dist=abs(feld[i][j]-v_wert);
   if (dist<min)
   {
    min=dist;                     // neues Distanzminimum
    z=i;
    s=j;                          //Indizes merken
   }
  }
 }
 *z_index=z;                      // Rückgabeparameter
 *s_index=s;
 return(min);                     // Wertrückgabe
}


// Hauptprogramm
void main (void)
{
 float feld[zeilen][spalten], vergl_wert;
 int i,j;
 char neufeld, neuwert, neudurch;

 do
 {
  // neue Feldeingabe gewünscht ?
  cout << "\n Neues Feld eingeben? (j/n) ";
  cin >> neufeld;
  if (neufeld != 'N' && neufeld != 'n')
  {
   // Eingabe der Feldwerte
   cout << "\n Eingabe der Feldelemente";
   for (i=0; i<zeilen; i++)
   {
    for (j=0; j<spalten; j++)
    {
     cout << "\nfloat-Element ["<<i<<"] ["<<j<<"]: ";
     cin >> feld [i][j];
    }
   }
```

```
  }

  // neue Vergleichswerteingabe gewünscht?

  cout << "\nNeuen Vergleichswert eingeben? (j/n) ";
  cin >> neuwert;
  if (neuwert !='N' && neuwert!='n')
  {
   cout << "\n Eingabe des Vergleichwertes: ";
   cout << "\n float-Vergleichswert: ";
   cin >> vergl_wert;
  }
  // Ausgabe der Ergebnisse
  cout << "\n\nDie minimale Distanz zwischen Vergleichswert und"
"Feldelement beträgt: "<<mindist(feld,&i,&j,vergl_wert);
  cout << "\nDas nächste Feldelement steht in Zeile "<<i<<" Spalte "<<j
       << "\nund hat den Wert " << feld[i][j];
  // neuer Durchlauf abfragen
  cout << "\n neuer Durchlauf gewünscht? (j/n) ";
  cin >> neudurch;
 }
 while (neudurch != 'N' && neudurch != 'n');
} // Programmende
```

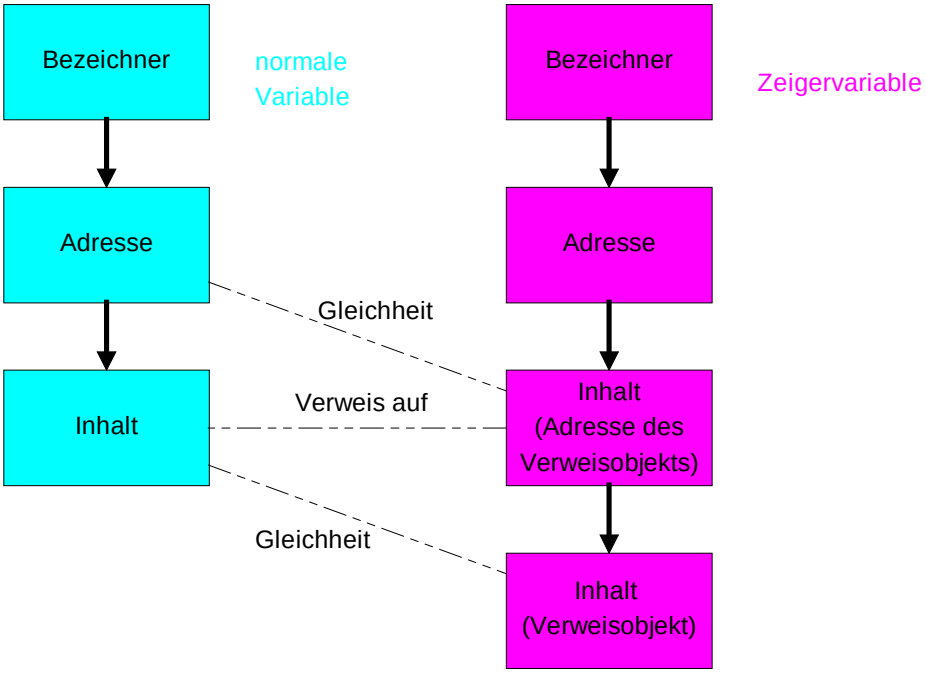
2. Schreiben Sie einen Sortieralgorithmus, der n eingegebene Zahlen der Reihe nach sortiert.

```
// Sortieralgorithmus

#include <stdio.h>

void main (void)
{
 int feld[100];
 int n,i,tausch,j;
 printf("Eingabe n: ");
 scanf("%d",&n);
 for (i=0;i<n; i++)
 {
  printf("%i.tes Element: ",i+1);
  scanf("%d",&feld[i]);
 }
 for (i=n-1; i>=0; i--)
  for (j=0; j<i; j++)
  {
   if(feld[j]>feld[j+1])
   {
    tausch=feld[j];
    feld[j]=feld[j+1];
    feld[j+1]=tausch;
   }
  }
 printf("\n");
 for (i=0; i<n; i++)
  printf(" %i ",feld[i]);
}
```

12 Zeiger

12.1 Deklaration und Arbeitsweise von Zeigern

Besonderheiten der Zeigervariablen:
Inhalt der Variablen (der Wert) ist die Adresse eines Objektes



Einsatzbereiche:

- Wertrückgabe bei Funktionen (außer Funktionswert) - Rückgabeparameter im Prozedurmodell
- Aufbau und Verwaltung von dynamischen Variablen und Datenstrukturen
- Hantieren von Funktionen und Objekten

Deklaration einer Zeigervariablen:

```
typangabe *bezeichner;
```

Sprechweise:

bezeichner verweist auf typangabe
bezeichner ist Zeiger auf typangabe

Beispiel:

```
int *intpointer                 // einzelner Zeiger; Zeiger intpointer zeigt
                                // auf Integertypen
float *floatzeiger;
```

```
int *zei_vec[10];              // Feld von Zeigern auf Integer
double *x, *y, u, v, w         // gemischte Deklaration
```

Man kann auch Zeiger, wie Variablen, gleichzeitig bei der Deklaration initialisieren. Es muß aber in diesem Fall, die Variable, auf die der Zeiger zeigt, schon vorher deklariert sein.

Beispiel:

```
int i;
int *j=&i;      // die Adresse der gerade erzeugten Variable i, wird dem
Zeiger j zugewiesen
```

12.2 Operatoren für Zeigervariablen

*-Operator:

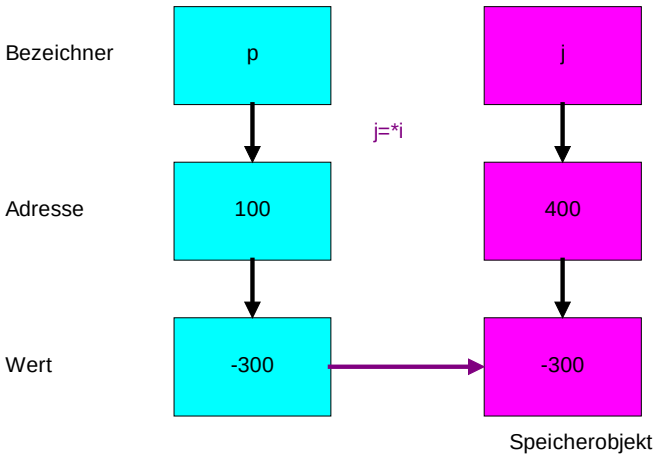
Dereferenzieren - Verfolge des Adreßverweises, Ermitteln des Wertes der Variable, auf die der Zeiger verweist.

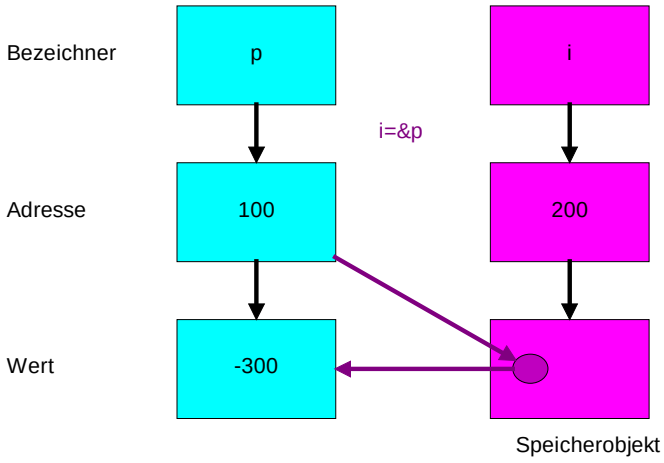
```
i=*p;
/* Der Inhalt der Zelle, auf die p verweist, wird i zugewiesen
   lies Inhalt von p, verwende den Inhalt als Adresse im Speicher, Suche
   die Zelle zu dieser Adresse auf, lies den Inhalt,... */
```

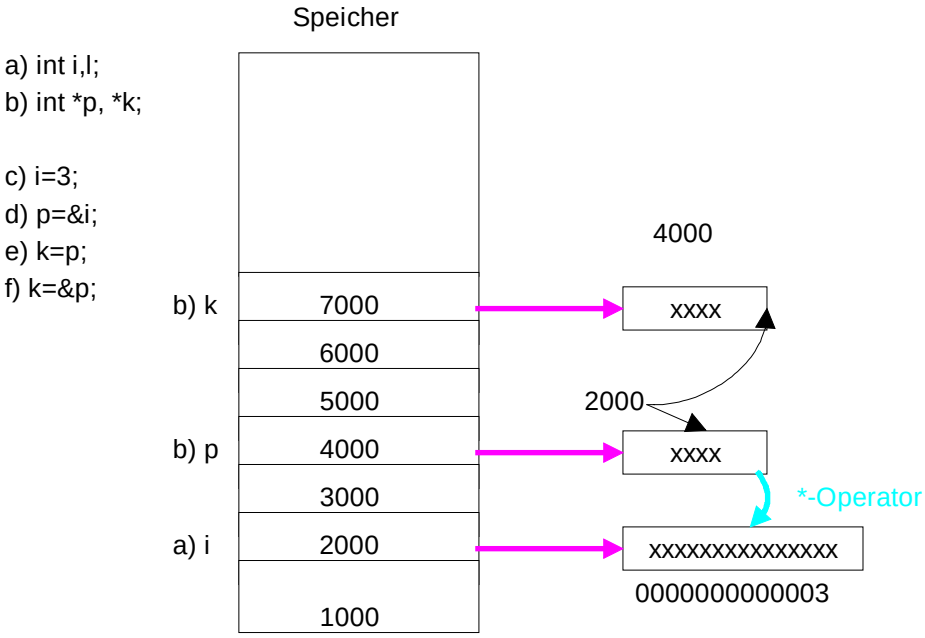
&-Operator:

Adreßoperator - ermittelt die Adresse der Variablen, des Objektes, auf das er angewendet wird.

```
i = &p;
/* Die Adresse der Variablen p wird ermittelt und der Variablen i
   zugewiesen - i ist jetzt ein Zeiger auf p */

int  * zeiger;
int *i, p;
int *l;
int j;
```







mit Zeigertypen verträglich:

- Variablenname eines Feldes
- Funktionsname

Bsp.:

```
int feld[el_anz];
int *zeiger;
...
zeiger=feld;          // zeigt auf Feldanfang
zeiger[i]=a*b+c;
zeiger = zeiger +1;   // zeigt auf 2.Element
                  // Adresse wird nicht um eins erhöht, sondern es verweist
                  // auf das nächste Objekt (Element des Feldes)
```

Bemerkung:

Der Befehl sizeof(typBezeichner) liefert, wieviel Speicherplatz eine Variable belegt.

Beispiel:

```
speicher_int=sizeof(int);
```

12.3 Zeiger und Funktionen

12.3.1 Übergabe von Arrays

Eine String besteht in C aus einem Array von Char-Variablen.

Beispiel:

```
char string[80];  // Zeichenkette für 79 Zeichen und der binären Null "/0"
```

Diesen String kann man mit folgenden Anweisungen einer Funktion übergeben.

Beispiel:

```
void scan_line (char kette[80]);
{
  ...
}

void main (void)
{
  char string[80];
  ...
  scan_line(string);
}
```

Beim Aufruf der Funktion "scan_line" wird lediglich der Zeiger auf den Anfang der Zeichenkette "string" der Funktion übergeben.

Hierdurch wird NICHT der Inhalt der Zeichenkette "string" nach "kette" kopiert, sondern nur ein zweiter Zeiger (innerhalb von scan_line mit dem Namen 'kette') ebenfalls auf den Anfang der Zeichenkette angelegt.

Die gemachten Angaben beschränken sich nicht für eine Zeichenkette, sondern sind für jedes (eindimensionale) Feld gültig.

Will man einer Funktion nicht die gesamte Zeichenkette sondern nur einen Restabschnitt übergeben, so wird die Adresse des betreffenden ersten Elements des Restabschnitts der Funktion übergeben.

Beispiel:

```
void prozess (float teilkette[40]);
    // Zugriff auf die Elemente von Kette[50..89] als teilkette[0..39]
{
  ...
}

void main (void)
{
  float kette[90];
  ...
```

```
  prozess (& kette[50]);     // Übergabe des Anfangs der Teilkette
}
```

12.3.2 Zeiger als Parameter

Es ist möglich bei Funktionen einen Wert durch return an die main-Funktion zurückzugeben. Es besteht nun die Möglichkeit, daß man in einer Funktion mehrere Funktionswerte zurückgeben möchte.
Um dieses Problem zu lösen, kann man einerseits mit globalen Variablen arbeiten oder zum Beispiel auch mit Zeigern.

Im zweiten Fall übergibt man der Funktion als Parameter die Adresse der verschiedenen, neuzubesetzenden Funktionen. In der Funktion wird durch Dereferenzierung der Variable ein neuer Wert zugewiesen werden. Dieser veränderte Wert bleibt nach Beendigung der Funktion erhalten.

Beispiel:

```
#include <stdio.h>
void main (void)
{
 int u=1;
 int v=3;
 void funkt1 (int u, int v);         // Funktionsprototyp
 void funkt2 (int *pu, int *pv);     // Funktionsprototyp

 printf("\n\nVor Ausführung von funkt1 : u=%d   v=%d",u,v);
 funkt1(u,v);
 printf("\nNach Ausführung von funkt1: u=%d   v=%d",u,v);

 printf("\n\nVor Ausführung von funkt2 : u=%d   v=%d",u,v);
 funkt2(&u,&v);
 printf("\nNach Ausführung von funkt2: u=%d   v=%d",u,v);
}

void funkt1 (int u, int v)
{
 u=0;
 v=0;
 printf("\nWährend funkt1            : u=%d   v=%d",u,v);
}

void funkt2 (int *pu, int *pv)
{
 *pu=0;
 *pv=0;
 printf("\nWährend funkt2            :*pu=%d  *pv=%d",*pu,*pv);
}
```

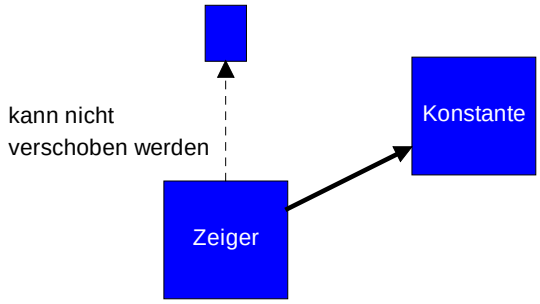
Ergebnis:

```
Vor Ausführung von funkt1  : u=1  v=3
Während funkt1             : u=0  v=0
Nach Ausführung von funkt1 : u=1  v=3

Vor Ausführung von funkt2  :  u=1      v=3
Während funkt1             :*pu=0    *pv=0
Nach Ausführung von funkt2 :  u=0      v=0
```

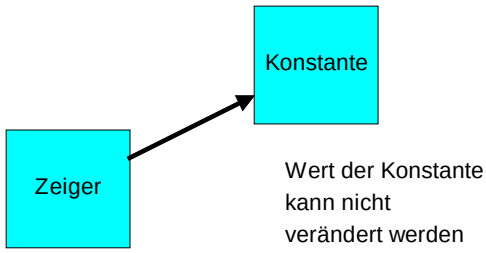
12.4 Zeiger und Konstanten

Auch in Verbindung mit Zeigern kann der Befehl const verwendet werden. Es gibt hierbei drei verschiedene Typen:

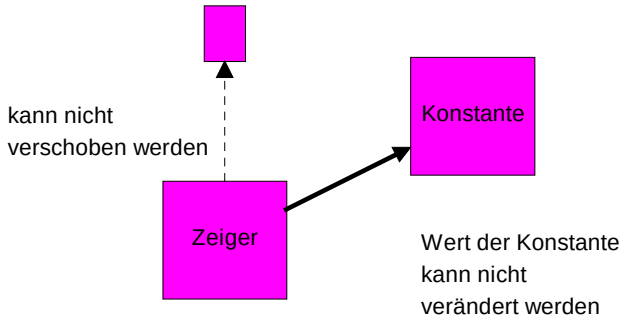
- Konstanter Zeiger:
  In diesem Fall darf die Zeigeradresse nicht verändert werden.
- Zeiger auf Konstante:
  Hier darf der Inhalt des Zeigers / Objektes nicht geändert werden.
- Konstanter Zeiger auf Konstante:
  In diesem Fall darf weder die Zeigeradresse noch der Inhalt des Zeigers / Objekts verändert werden.



konstanter Zeiger



Zeiger auf Konstante



konstanter Zeiger auf Konsante

Beispiele:

```
void test (char *p)
{
 char s[]="Gorm";
 const char *pc=s;    // Zeiger auf Konstante
 pc[3]='g';              // Fehler: pc zeigt auf Konstante
 pc=p;                   // OK

 char *const cp=s    // Konstanter Zeiger
 cp[3]='a';     //OK
 cp=p;    // Fehler: cp ist konstant

 const char *const cpc=s;    // Konstanter Zeiger auf Konstante
 cpc[3]='a';    //Fehler: cpc zeigt auf Konstante
 cpc=p;      //Fehler: cpc ist konstant
}
```

Man kann die Adresse einer Variable einem Zeiger auf eine Konstante zuweisen.

Beispiel:

```
void test1 (void)
{
 int a=1;
 const int c=2;
 const int* p1=&c;   // OK
 const int* p2=&a;   // OK
 int p3 = &c;        // Fehler: Initialisierung von int mit const int*
 *p3 = 7;            //Versuch, den Wert von c zu ändern
}
```

Anwendungsmöglichkeit:

Konstante Zeiger und Zeiger auf Konstanten können z.B. bei Funktionsaufrufen nützlich sein. Es gibt häufig das Problem, daß man eine Zeichenkette als Paramter übergeben will, diese jedoch in der Funktion nicht verändert werden soll. Man richtet nun einen Zeiger auf eine Konstante ein, um dieses Problem zu lösen.

void test (const char *text);

Falls der Zeiger in der Funktion nicht verändert werden darf, so verwendet man einen konstanten Zeiger.

void test (char const *text);

12.5 Zeiger und Felder

Zeiger und Felder sind nahe Verwandte.

Beispiel

```
int  Feldint[5];
int Feldchar[6];
int *Zeiger1=Feldint;    //nun zeigt Zeiger1 auf das erste Element
int *Zeiger2=&Feldint[0]; //auch hier zeigt Zeiger2 auf das erste Element
                           des Feldes
scanf("%s",Feldchar);    //da Feldchar der Zeiger auf das 1.Element des
                           Feldes ist, benötigt man hier keinen &-Operator
scanf("%s",&Feldchar[0]); //gleichbedeutend zu oben
```

```
scanf("%s",&Feldchar[0]); // dto.
```

Man kann nun die einzelnen Elemente eines Feldes mit Hilfe von Zeiger oder der eckigen Klammern ansprechen.

Beispiel:

```
for (int i=0; i<6; i++)
 feldchar[i]=0;
```

oder

```
for (int *zeiger=Feldchar; *zeiger!=0; zeiger++)
 *zeiger=0;
```

12.6 Vorsicht! Zeiger!

12.6.1 Komplexes Zeigerprogramm

Das nachfolgende Programm ist recht komplex. Es ist weniger als Anwendungsbeispiel, sondern mehr als Beispiel gedacht, was man in C alles machen kann.

```
#include <stdio.h>
#include <alloc.h>

int   a,b,erg,erg2;
int   *pi,*pb,*perg,*perg2;

void main (void)
{

a=3;
b=6;

/* Belegung von Speicherplatz */

pi=malloc(sizeof(int));
pb=malloc(sizeof(int));
perg=malloc(sizeof(int));
perg2=malloc(sizeof(int));
*pi=10;
*pb=20;
printf("\n-------------\n");
printf("a=%i,    &a=%i\n",a,&a);
printf("b=%i,    &b=%i\n",b,&b);
printf("*pi=%i,  pi=%i\n",*pi,pi);
printf("*pb=%i,  pb=%i\n",*pb,pb);
printf("sizeofint=%i\n",sizeof(int));

erg=a++ + ++b;                  // b wird sofort erhöht, dazu wird a ad-
                                   diert, a wird nach der Zuweisung erhöht
*perg=(*pi)++ + ++*pb;          // gleiche Funktion nur mit Zeigern
erg2=40;
*perg2=40;
printf("erg2=%i   &erg=%i\n",erg2,&erg2);
printf("*perg2=%i  perg2=%i\n",*perg2,perg2);
erg2 += a++ + ++b ;             // hier wird zusätzlich zur obigen Opera-
                                   tion noch erg2 hinzuaddiert
*perg2 +=a++ + ++b;             // gleiches Beispiel nur mit Zeigern
```

```
printf("ergebnis  erg=%i\n",erg);
printf("ergebnis perg=%i\n",*perg);
printf("ergebnis erg2 (nachher)=%i\n",erg2);
printf("ergebnis perg2 (nachher)=%i\n",*perg2);
printf("a=%i,    &a=%i\n",a,&a);
printf("b=%i,    &b=%i\n",b,&b);
printf("*pi=%i,  pi=%i\n",*pi,pi);
printf("*pb=%i,  pb=%i\n",*pb,pb);

/* Freigabe des belegten Speichers */

free(pi);
free(pb);
free(perg);
free(perg2);
}
```

Eingabe:

```
-------------
a=3,       &a=1020
b=6,       &b=1018
*pi=10    pi=2124
*pb=20   pb=2132
sizeofint=2
ergebnis erg=10
ergebnis perg=31
a=4,       &a=1020
b=7,       &b=1018
*pi=11,   pi=2124
*pb=21,  pb=2132
ERG2, PERG2
```

12.6.2 Subtraktion von Zeigern

Ich möchte hier der Vollständigkeit halber die Möglichkeit erwähnen, die Adressen von zwei Zeigern zu subtrahieren bzw. zu einem Zeiger eine Konstante zu addieren.

Die Subtraktion von zwei Zeigern liefert die Anzahl der Elemente, die zwischen diesen beiden Zeigern liegen.

Die Addition einer Konstante zu einem Zeiger bewirkt das 'Weiterschalten' des Zeigers um die angegebene Anzahl von **Elementen** und zwar in Abhängigkeit vom gewählten Datentyp des Zeigers! Dies wird auch als Zeigerarithmetik bezeichnet.

Beispiel:

```
void test (void)
{
 int v1[10];
 int v2[10];
 int i1=&v1[5]-&v1[3];  //i1=2 ~ Elemente
 int i2=&v1[5]-&v2[3]   //undefiniert, da zwei  verschiedene Felder verwen-
                        //det werden
 int *p1=v2+2           //p1=&v2[2]
 int *p2=v2-2           //undefiniert, weil man aus dem definierten
                        //Feldbereich herausläuft
}
```

12.7 Aufgaben

1. Das Plateauproblem (Prüfung Winter 88/89 1.Semester)
   Gegeben ist ein Fehld ganzer Zahlen a_i=1...n ≤ 100. Die Werte a_i sind in nichtfallender Folge geordnet: $a_i \le a_{i+1}$: i=1...n-1
   Gesucht ist:
   - Welcher Wert der a_i am häufigsten vorkommt: Plateauhöhe H. Da die Werte geordnet sind, kommt dieser Wert in lückenloser Folge
   - Wieviele Werte das Plateau umfaßt: Plateaulänge L
   - Bei welchen Index S das Plateau beginnt.

   Schreiben Sie eine Prozedur plateau, die diese Berechnungen durchführt.
   Für den Grenzfall $a_i < a_{i+1}$ für alle i gibt es kein Plateau. Geben Sie eine entsprechende Bemerkung aus, wie z.B. "Feld monoton steigend". (Dabei soll H=a(1), S=1 und L=1 sein.)

Hier ein Programmbeispiel:

```
// Winter 88/89 - Aufgabe 2 (Plateau-Problem)
#include <stdio.h>
void plateau (int *feld, int n);

void main (void)
{
 int feld[100];
 int n;
 printf("\nBitte geben Sie die Anzahl der einzugebenden Zahlen (n) ein: ");
 scanf("%d", &n);
 for (int i=0; i<=n-1; i++)
 {
  printf("Die %d.te Zahl: ",i);
  scanf("%d",&feld[i]);
 }
 plateau(feld,(n-1));
}
void plateau (int *feld,int n)
{
 int h,s,l;
 int max_1=1,max_2=1;
 int i, merker_1, merker_2, wert_1, wert_2;
 for (i=0; i<=n; i++)
 {
  if (feld[i]==feld[i+1])
  {
   max_1+=1;
   if (max_1==2)
   {
    merker_1=i;
    wert_1=feld[i];
   }
  }
  else
  {
   if (max_1>max_2)
   {
    max_2=max_1;
    merker_2=merker_1;
    wert_2=wert_1;
   }
   max_1=1;
  }
 }
```

```
 l=max_2;
 h=wert_2;
 s=merker_2;
 if (l!=1)
 {
  printf("\nDie Zahl %d kommt am h"ufigsten vor.", h);
  printf("\nDiese Zahl erscheint %d mal.",l);
  printf("\nDer Anfangsindex ist %d.",s);
 }
 else
  printf("\nEs kommt jede zahl nur einmal vor.");
}
```

13 Dateien

Manchmal kann es notwendig sein, daten dauerhaft abzuspeichern. In diesem Fall speichert man die eingegebenen Daten in einer Datei.
In Textdateien, kann man Daten hineinschreiben und wieder auslesen.

Es gibt zwei verschiedene Arten von Dateien: Textdateien und Binärdateien.

Textdateien:

- hier werden Zeichenfolgen abgespeichert
- sequentielle Dateien
- Zugriff: lesend, schreibend, lesend und schreibend
- Besonderheit: anhängen
- Funktionen:
  Datei öffnen: fopen ()
  Lesen: fscanf (), fgets ()
  Schreiben: fprintf (), fputs()
  Schließen: fclose ()

13.1 Die Schritte zur Dateibearbeitung:

- Ein Pointer auf den Datentyp Datei, diesem Pointer wird dann später die Adresse angegeben, wo die zu lesende / schreibende Datei vorhanden ist bzw. abzuspeichern ist.
- Man weist nun dem Pointer einen Wert zu. Dies funktioniert durch das Aufrufen der Funktion fopen mit den Parametern, Dateiname (und evtl. Verzeichnis) und der Angabe, ob aus der Datei gelesen werden soll, oder ob sie beschrieben werden soll. Für Lesen gibt man "r" oder "rt" und für Schreiben "w" oder "wt" an. Öffnet man eine bereits existierende Datei zum Schreiben, dann wird sie überschrieben. Mit "at" wird der Dateizeiger ans Ende der Datei gesetzt und es können Datensätze an die bestehende Datei angehängt werden. Durch diesen Schritt wird die Datei geöffnet.
- Nun muß man prüfen, ob die Datei ohne Fehler geöffnet werden konnte. Sind Fehler aufgetreten, so liefert die Funktion fopen einen NULL-Zeiger zurück.
- Möchte man nun den ganzen Dateiinhalt auslesen, so erfolgt dies mit der Funktion `fscanf (Adresspointer,"%Variablentyp Variable)` (ähnlich wie die scanf-Funktion). Die fscanf-Funktion kann nun in eine while-Schleife eingebunden werden, die sooft wiederholt wird, bis das Ende der Datei erreicht ist: `while(!feof (Adresspointer)`
- Datensätze kann man mit Hilfe der `fprintf(Adresspointer,"%Variablentyp Variable)`in der Datei abspeichern.

13.2 Aufgabe

1. Schreiben Sie ein Programm Dateikasten. In diesen Kasten sollen sie ihre Adressen abspeichern und auch wieder auslesen können.

```
// Dateikasten zur Speicherung von Adressen
#include <stdio.h>
#include <string.h>
struct adressen_typ
{
  char name[25],vorname[25];
};


void daten_lesen (void)
{
 FILE *fttp;
 char datei[13];
 adressen_typ adresse;
 printf("\nName der zu ”ffnenden Datei: ");
 scanf("%8s",datei);
 strcat(datei,".dat");
 fttp=fopen(datei,"rt");
 if (fttp==NULL)
  printf("Datei kann nicht zum lesen ge”ffnet werden.");
 else
 {
  if (feof(fttp))
   printf("Datei ist leer!");
  else
  {
   do
   {
    fscanf(fttp," %s",adresse.name);
    fscanf(fttp," %s",adresse.vorname);
   } while (!feof(fttp));
   fclose(fttp);
   printf("\nHier sind die Daten");
   printf("\nName        : %s",adresse.name);
   printf("\nVorname     : %s",adresse.vorname);
  }
 }
}

void daten_einlesen (adressen_typ *adresse, char *datei)
{
 printf("\nName der Datei (8 Buchstaben): ");
 scanf("%8s",datei);
 printf("\nA D R E S S E\n");
 printf("Name: ");
 scanf("%24s",adresse->name);
 printf("Vorname: ");
 scanf("%24s",adresse->vorname);
}

void daten_speichern (adressen_typ *adresse, char *datei)
{
 FILE *fttp;
 strcat (datei, ".dat");
 fttp=fopen (datei,"wt");
 if (fttp==NULL)
  printf("\nDatei kann nicht zum schreiben ge”ffnet werden.");
 else
 {
  fprintf(fttp,"\n%s",adresse->name);
  fprintf(fttp,"\n%s",adresse->vorname);
  fclose(fttp);
 }
}
```

```
void main (void)
{
 int wahl;
 adressen_typ adresse;
 char antwort[2];
 char datei[13];
 do
 {
  printf("\n%35s","Men□");
  printf("\n\n1.Daten einlesen und abspeichern");
  printf("\n2.Daten lesen");
  printf("\n3.Ende");
  printf("\nEingabe: ");
  scanf("%i",&wahl);
  switch (wahl)
  {
   case 1: do
           {
            daten_einlesen (&adresse, datei);
            daten_speichern(&adresse,datei);
            printf("Noch eine Adresse (j/n) :");
            scanf("%s",antwort);
           }
           while ((antwort[0]!='n') || (antwort[0]!='N'));
           break;
   case 2: daten_lesen();
           break;
  }
 }
 while (wahl !=3);
}
```

13.3 fprintf

Dieser Befehl wird im Prinzip genauso verwendet wie printf, allerdings erfolgt die Ausgabe in eine Datei. Als erster Parameter ist der Dateizeiger anzugeben, die weiteren Parametersind identisch mit dem printf-Befehl.

13.4 fscanf

...
Es ist auch möglich bestimmte Zeichen zu überlesen, d.h. sie "müssen" gelesen werden, werden aber nicht in der Variable abgespeichert.

Beispiele:

In diesem Beispiel soll ein Wort das in Anführungsstrichen steht gelesen werden, die Anführungsstriche selbst dagegen nicht in die Variable "Name" abgespeichert werden.
Zunächst erwartet der PC einen Anführungsstrich, dann liest er solange weiter und speichert auch in die Variable, bis ein weiterer kommt dieser wird zunächst nicht gelesen aber dafür gleich danach überlesen.

```
    fscanf (fttp, "\"%[^\"]\"", Name);
```

Hier wird zunächst das Wort "Kaffesorte" erwartet. Danach wird ein String in die Variable "ksorte" eingelesen.

```
fscanf (fttp, "Kaffesorte: %s", ksorte);
```

14 Übergang C nach C++

14.1 Was "gefällt" an C?

- Ermöglicht systemnahe Programmierung
- Effizienter Code
- Eigenschaften des Programmsystems können gut beeinflußt werden (i.e. Laufzeit, Garbage-Collection)
- Eigene Ansicht: Häufig Möglichkeit zur eleganten Formulierung eines Algorithmus

14.2 Was "gefällt" nicht an C?

- Flache Programmstruktur (keine explizite Kennzeichnung von Unterprogrammen)
- Wesentliche Datentypen (Bool, String, Liste, Queue und auch const) fehlen
- Modularisierung wird nur auf Dateiebene unterstützt (global, static)
- Keine benutzerdefinierten Datentypen, die sich ähnlich zu eingebauten Typen verhalten
- Initialisierungsfunktionen müssen explizit aufgerufen werden, d.h. man kann vergessen, sie aufzurufen

14.3 10 Ansätze in C++

- Engere Kopplung von Manipulationsfunktionen an Strukturen wäre wünschenswert
- Datenmanipulation an Strukturen jenseits der Manipulationsfunktionen ist bei C jederzeit möglich. Es wird kein Schutz geboten!
- Eine Kennzeichnung von Anwender- und Implementierungsschnittstelle auf einen Datentyp ist außer über Kommentare nicht möglich! Wiederum kein Schutz vor unberechtigter Datenmanipulation!
- Alle Funktionen brauchen einen eindeutigen Namen; gleichartige Funktionen auf unterschiedlichen Typen müssen dennoch unterschiedlich benannt werden (add_item_list, add_item_fifo, add_item_btree, ...)
- Manche kleine Funktionen, die aus Laufzeitgründen nicht als Funktionen aufgerufen werden sollen, müssen in C über Makros implementiert werden $\Rightarrow$ dort jedoch kein Schutz vor Typfehlern, Fehler im Makro sind sehr schwer zu finden, etc.
- Die Behandlung von Ausnahmen (Fehlerfälle, Überlauf, Unterlauf, etc.) wird konzeptuell nicht unterstützt
- Strukturierungs-/Modularisierungsmöglichkeiten sind insbesondere für große/sehr große Programmpakete nicht besonders gut

14.4 Ansätze in C++

Flache Programmstruktur (keine explizite Kennzeichnung von Unterprogrammen)

- Modularisierung wird nur auf Dateiebene unterstützt (global, static)

- Bleibt auch in C++

Jedoch: Möglichkeit der Zuordnung zu Namensbereichen, Klassen, Strukturen (auch mehrstufig)

Wesentliche Datentypen (Bool, String, Liste, Queue und auch const) fehlen

- Bool nun vorhanden
- typisierter const nun vorhanden
- String, Liste, Queue über Standardbibliothek

Keine benutzerdefinierten Datentypen, die sich ähnlich zu eingebauten Typen verhalten

- Nun besteht über Klassen und struct die Möglichkeit, eigene Datentypen zu definieren
- Durch Operator-Overloading können Operatoren wie +, =, [ ], etc. auf diesen eigenen Typen definiert werden

Initialisierungsfunktionen müssen explizit aufgerufen werden, d.h. man kann vergessen, sie aufzurufen

- Klassen und struct können (werden automatisch!) mit einer Initialisierungsfunktion versehen werden, die bei der Erzeugung eines dementsprechenden Objekts automatisch aufgerufen wird
- Versehentliches Vergessen des Aufrufs -- d.h. Arbeiten mit uninitialisierten Objekten -- ist nicht mehr möglich

Engere Kopplung von Manipulationsfunktionen an Strukturen wäre wünschenswert

- Klassen und struct können in C++ direkt die Manipulationsfunktionen enthalten
- Funktionsname besteht aus Klassenname::Funktionsname (Strukturierung und Hierarchisierung)

Datenmanipulation an Strukturen jenseits der Manipulationsfunktionen ist bei C jederzeit möglich. Es wird kein Schutz geboten!

- Für Klassen und struct können private und öffentliche Bereiche angegeben werden
- Daten und Funktionen in öffentlichen Bereichen können von außen aufgerufen bzw. manipuliert werden
- Daten in privaten Bereichen sind gegen Zugriff von außen geschützt
- private Funktionen können von außen nicht aufgerufen werden

Eine Kennzeichnung von Anwender- und Implementierungsschnittstelle auf einen Datentyp ist außer über Kommentare nicht möglich! Wiederum kein Schutz vor unberechtigter Datenmanipulation!

- bei Namensbereichen können mehrere Schnittstellen definiert werden, so daß ein Anwender, die für ihn wichtigen Funktionen leicht erkennen kann. Eine weitere Schnittstelle kann dann die Implementierung beschreiben
- Für Klassen/Struct s.o.

Alle Funktionen brauchen einen eindeutigen Namen; gleichartige Funktionen auf unterschiedlichen Typen müssen dennoch unterschiedlich benannt werden (add_item_list, add_item_fifo, add_item_btree, ...)

- Der Überlade-Mechanismus von C++ ermöglicht, bei einem Funktionsaufruf nicht nur den Funktionsnamen, sondern auch noch die übergebenen Parametertypen zu berücksichtigen. Dadurch könnte z.B. für alle oben genannten Datentypen die Übergabefunktion add_item lauten, wobei je nach nachfolgendem Parametertyp eine andere Funktion ausgewählt wird.

  bool add_item (list &Item) ⇒ ruft die Listenverarbeitungsfunktion add_item auf

  bool add_item (fifo &Item) ⇒ dto. Für Fifos

  bool add_item(btree &Item) ⇒ dto. Für BTrees

Manche kleine Funktionen, die aus Laufzeitgründen nicht als Funktionen aufgerufen werden sollen, müssen in C über Makros implementiert werden ⇒ dort jedoch kein Schutz vor Typfehlern, Fehler im Makro sind sehr schwer zu finden, etc.

- C++ bietet eine inline-Funktion, wodurch dem Compiler geraten wird, diese Funktion inline zu erzeugen
- Inline-Funktionen sehen ansonsten genauso aus wie "normale" Funktionen, bieten somit Typprüfung, formale Parameter etc.

Die Behandlung von Ausnahmen (Fehlfälle, Überlauf, Unterlauf etc.) wird in C konzeptuell nicht unterstützt

C++ bietet hier ein geschlossenes Konzept mit try-throw und catch. Hiermit können Fehlerzustände eindeutig identifiziert und vor allem von der richtigen Bearbeitungsinstanz aufgefangen werden.

15 Programmierparadigmen

Unter einem Paradigma versteht man einen Stil, eine Leitlinie, auf welche Weise ein Problem mit Hilfe einer Programmiersprache gelöst wird. "Eine Sprache unterstützt einen Programmierstil, falls sie Mittel bereitstellt, mit denen dieser Stil bequem (angemessen einfach, sicher und effizient) angewendet werden kann.
Eine Sprache unterstützt eine Technik nicht, falls solche Programme nur mit außergewöhnlichem Aufwand oder Wissen geschrieben werden können; in dem Fall ermöglicht sie nur die Technik. So kann man z.B. in FORTRAN 77 auch strukturiert und in C auch objektorientiert programmieren, dies ist aber unnötig schwierig, da diese Sprachen diese Techniken nicht direkt unterstützen.`` [B.S.]
Wichtig ist bei der nachfolgenden Betrachtung verschiedener Paradigmen, daß es das "beste Paradigma`` nicht gibt. Jedes Paradigma hat seine Berechtigung und Anwendungsgebiete, in denen es herausragende Eigenschaften (angemessen einfach, sicher und effizient) zeigt. Wesentlich ist mit allen Paradigmen vertraut zu sein und diese je nach Anforderung zielstrebig einsetzen zu können.

15.1 Prozedurales Programmieren

Entscheiden Sie, welche Prozeduren Sie benötigen; Verwenden Sie den besten Algorithmus, den Sie finden können

Der Schwerpunkt liegt hier auf Algorithmus. Beispiel Quadratwurzel:

```
double sqrt(double arg)
{
// Code zum Berechnen der Quadratwurzel
return Ergebnis;
}
```

Die zu lösende Programmieraufgabe wird durch eine Hierarchie von Funktionen bzw. Funktionsaufrufen beschrieben.
Dieser Stil eignet sich sehr gut für arithmetische Berechnungen sowie auch sonstige Berechnungen, in denen aufgrund von gegebenen Eingabeparametern entsprechende Rückgabewerte bestimmt werden sollen.
"Die sich mit diesem Paradigma befassende Literatur ist angefüllt mit Diskussionen über verschiedene Arten der Argumentübergabe, die Unterscheidungsmöglichkeiten verschiedener Argumentarten, unterschiedlichste Funktionsarten (z.B. Prozeduren, Routinen und Makros) usw.`` [B.S.]

15.2 Modulares Programmieren

Entscheiden Sie, welche Module Sie haben wollen. Unterteilen Sie das Programm so, daß die Daten in Modulen gekapselt sind.

"Im Laufe der Jahre verlagerte sicher der Schwerpunkt beim Entwurf von Programmen: Anstelle des Designs von Prozeduren rückte die Organisation der Daten in den Mittelpunkt. Unter anderem spiegelt sich darin ein Anwachsen der Programmgröße wider. Ein Satz verwand-

ter Prozeduren zusammen mit den von ihnen manipulierten Daten wird häufig als Modul bezeichnet. Dieses Paradigma ist auch als Datenkapselung bekannt.
Das bekannteste Beispiel eines Moduls ist die Definition eines Stacks. Die wesentlichen zu lösenden Probleme sind:

1. Bereitstellung einer Schnittstelle zur Benutzung des Stacks (z.B. Funktionen wie push() und pop())
2. Sicherstellen, daß auf die Repräsentation des Stacks (z.B. ein Feld aus Elementen) nur über die definierte Schnittstelle zugegriffen werden kann.
3. Sicherstellen, daß der Stack for der ersten Benutzung initialisiert wird

C++ unterstützt das modulare Programmieren i.w. durch das Konzept der Namensbereiche (namespace). Hierdurch können Daten, Strukturen und Funktionen in einem gemeinsamen Namensbereich zusammengefaßt werden.

```
namespace Stack // Anwenderschnittstelle
{
void push(int);
int pop();
}
```

Gleichermaßen wird es aber auch noch eine Implementierungsschnittstelle geben, in der für die Implementierung wichtige Details festgelegt sind, die jedoch den Anwender z.B. eines Stacks nicht interessieren müssen, wie z.B. die Repräsentation des Stacks.

```
namespace Stack // Implementierungsschnittstelle
{
const int max_size = 200;
int v[max_size];
int top = 0;
void push(int);
int pop();
}
```

Durch diese beiden Schnittstellen besteht somit eine anwendungs- und eine Implementierungstechnische Sicht auf die Funktionen und die Repräsentation des Stacks. Darüber hinaus wird auch die Initialisierung (hier: top = 0) in der Implementierungsschnittstelle realisiert.
Durch das "Verbergen`` der Implementierungsdetails wird ein erster - wenn auch noch nicht sehr strikter - Schutz vor einem direkten Zugriff auf die Repäsentationsdetails des Stacks erzeugt.
Die Implementierung der Funktionen push und pop kann an anderer Stelle außerhalb der namespace-Deklaration erfolgen, auch in einer anderen Datei z.B. stack.cpp
Der Stack kann in einem Anwendungsprogramm z.B. durch Einbinden der Stack-Anwendungsschnittstelle (hier: stack.h) und Zugriff auf die Funktionen push und pop über den Bereichsoperator :: erfolgen:

```
#include ``stack.h``
void f()
{
Stack::push(100);
cout << ''Datum auf dem Stack: '' << Stack::pop() << endl;
}
```

16 Dynamische Speicherverwaltung

16.1 new und delete

Bis jetzt haben wir Zeigern keinen eigenen Speicherplatz reservieren können, wir mußten die Zeiger auf schon initialisierte Variablen im Programm zeigen lassen.

Nun können wir aber auch für Zeiger einen eigenen Speicherplatz anfordern.

Der Sinn dabei ist, daß der Speicherplatz erst dann angefordert wird, wenn er gebraucht wird. Es wird also (fast) kein Speicherplatz am Anfang des Programmstartes (sinnlos) belegt.

Mit dem Befehl new, kann man für einen Zeiger einen Speicherplatz anfordern. Nach Beendigung des Programmes muß der Speicher dann wieder freigegeben werden. Dies wird mit dem C++ - Befehl delete gemacht.

Beispiel:

```
void main (void)
{
int *i;
i=new int;
...
delete i;
}
```

Auf den Inhalt / Adresse des Zeigers wird nach wie vor mit dem &- und *-Operator zugegriffen.

Man unterscheidet zwischen delete und delete[ ]. Es handelt sich hierbei um zwei getrennte Befehle!

delete löscht ein mit new erstelltes Objekt.

delete [ ] löscht ein Feld, das mit new [ ] erstellt wurde.

16.2 Verkettete Listen

16.2.1 Einfache Liste

Wenn man zum Beispiel eine Adressverwaltung mit dem Computer durchführen möchte, dann weiß man am Anfang noch gar nicht, wie viele Adressen eigentlich verwaltet werden sollen. Würde man mit Arrays arbeiten, so kann es leicht passieren, daß man zuviel oder zuwenig Speicherplatz beim Programmstart anfordert.

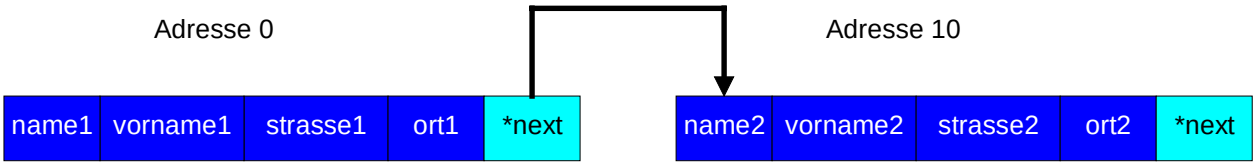
Aus diesem Grunde gibt es zum Beispiel die verketteten Listen. An sie kann man unbeschränkt neue Elemente anhängen und der Speicherplatz wird erst dann reserviert, wenn er auch benötigt wird. Eine solche Liste wird mit Zeiger realisiert.

In ein struct-Anweisung werden zunächst die einzelnen Elemente der Liste deklariert und dann wird auf diese Struktur selbst wieder ein Zeiger als Element der Struktur angelegt. Diese Rekursion ist der Schlüssel zum Erfolg.

Beispiel:

```
struct adressentyp
{
char name[25], vorname[25], strasse[25], ort[25];
adressentyp *next;
}
```

Zunächst werden in der struct-Anweisung die "datentragenden" Variablen deklariert, danach wird eine Variable next deklariert, die wiederum vom Typ adressentyp ist (Rekursion). Diese Variable next wird später die Adresse des nächsten Listenelements enthalten. Solange kein weiteres Element vorhanden ist, so hat *next den Wert NULL.



Der Inhalt von *next(1) = 10, der Inhalt von *next(2) dagegen ist NULL, weil es das letzte Element ist.

Vorgehensweise:

Um eine verkette Liste zu implementieren benötigt man folgendes:

- Eine struct-Anweisung, die einen Zeiger auf sich selbst hat (zum Beispiel: Adressentyp)
- Einen Zeiger, der vom Typ Adressentyp ist und der immer auf den Anfang der Liste zeigt. Er merkt sozusagen den Ausgangspunkt für alle späteren Operationen. (zum Beispiel: adr)
- Man benötigt einen Laufzeiger, der ebenfalls vom Typ Adressentyp ist und auf das aktuelle Listenelement zeigt. (zum Beispiel: laufz)
- Zuletzt benötigt man einen Zeiger, der vom Typ Adressentyp ist und immer für ein neues Listenelement den neuen Speicherplatz mit new anfordert. In ihn werden die Adressdaten hineingeschrieben. Dann wird der Zeiger next des letzten Listenelements auf den neuen Zeiger mit dem neuen Listeneintrag gebogen. Nun ist die Liste um das neue Element länger. (zum Beispiel: padr)
- Für den Laufzeiger laufz muß mit new kein Speicherplatz angefordert werden. Für den Zeiger adr dagegen muß man Speicherplatz anfordern, wenn die Liste ganz neu erstellt wird und noch keine Elemente enthält. Für jedes weiteres Element muß erst mit new padr ein neuer Speicherplatz angefordert werden.
- Die einzelnen Elemente einer Liste werden folgendermaßen angesprochen: Angenommen, es steht gerade laufz auf dem aktuellen Element, dann wird die Adresse wie folgt angesprochen: laufz->name, laufz->vorname, laufz->next usw. will man auf die übernächste

Adresse zugreifen, so schreibt man laufz->next->next, übernächster Name: laufz->next->next->name usw.

Beispielprogramm für eine verkettete Liste:

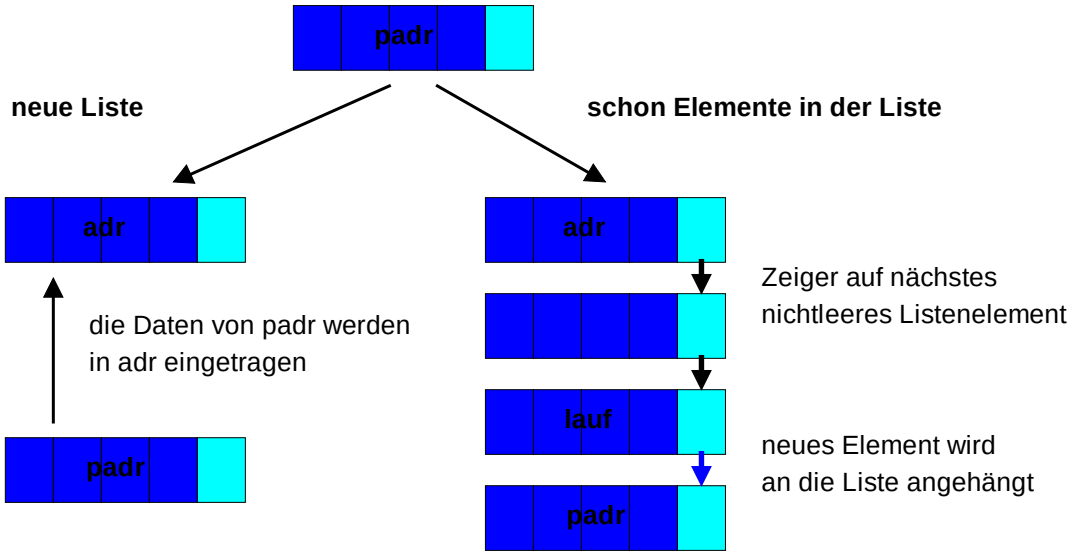
```
/*****************************************/
/*        einfach verkettete Liste       */
/*****************************************/

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

/*****************************************/
/*  Deklaration der struct-Anweisung für */
/*  die einzelnen Listenelemente         */
/*****************************************/

struct adresse_t
 {
char  name[25], vorname[25], strasse[30];
char  plz [6], ort[25];
adresse_t *next;
 };

/*****************************************/
/*             Hauptprogramm             */
/*****************************************/

void main (void)
{

adresse_t *adr, *padr,*laufz;
char antwort[2];

// Schleife wie viele Listenelement sollen eingefügt werden
adr=NULL;
printf("\nWollen Sie noch eine Adresse anhängen (j/n): ");
scanf ("%s",antwort);
while ((antwort[0]=='J') || (antwort[0]=='j'))
 {

// für neue Elemente wird in jedem Fall mit new padr
// neuer Speicherplatz angefordert
// existiert die Liste noch nicht, so wird dann später
// einfach ein Speicherplatz für adr erzeugt, ansonsten
// wird padr an die bestehende Liste angehängt
padr = new adresse_t;
printf("\n\nFamiliennamen: ");
scanf("%s",padr->name);
printf("Vorname     : ");
scanf("%s",padr->vorname);
printf("Strasse     : ");
scanf("%s[^\n]",padr->strasse);
printf("Postleitzahl: ");
scanf("%s",padr->plz);
printf("Wohnort     : ");
scanf("%s",padr->ort);
printf("\nWollen Sie noch eine Adresse anhängen (j/n): ");
scanf ("%s",antwort);

// falls noch kein Element in der Liste ist, muß zunächst
```

```
// adr initialisiert werden, adr erhält den Wert von padr
if(adr==NULL)
  {
adr=new adresse_t;
adr=padr;
adr->next=NULL;
laufz=adr;
  }

// das neue Element muß an das Ende der Liste angehängt werden
else
  {
laufz->next=padr;
laufz->next->next=NULL;
laufz=laufz->next;
  }
 }

printf("\n\nJetzt kommt die Ausgabe: ");
// der Laufzeiger wird auf den Anfang der Liste zurückgesetzt
laufz=adr;
// es wird solange die Schleife durchlaufen, solange das aktuelle
// Listenelement nicht auf NULL zeigt, d.h. leer ist
while (laufz!=NULL)
 {
  printf("\nFamilienname   : %s",laufz->name);
  printf("\nVorname        : %s",laufz->vorname);
  printf("\nStrasse        : %s",laufz->strasse);
  printf("\nPostleitzahl   : %s",laufz->plz);
  printf("\nWohnort        : %s",laufz->ort);

// der Laufzeiger wird nun auf das nächste Listenelement gerichtet
laufz=laufz->next;
 }

// es wird der Speicherplatz wieder freigegeben
delete adr;
delete padr;
delete laufz;
}
```

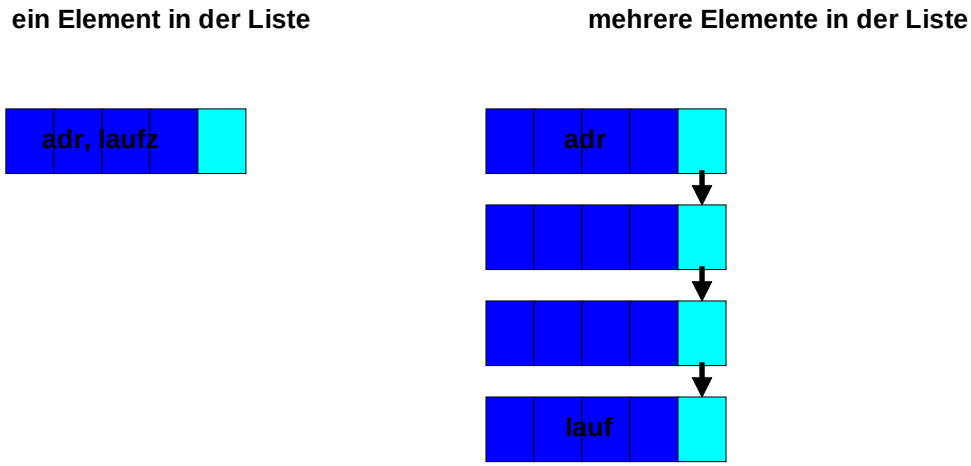
Bildliche Darstellung der Vorgehensweise des Programms:

Erster Schritt:

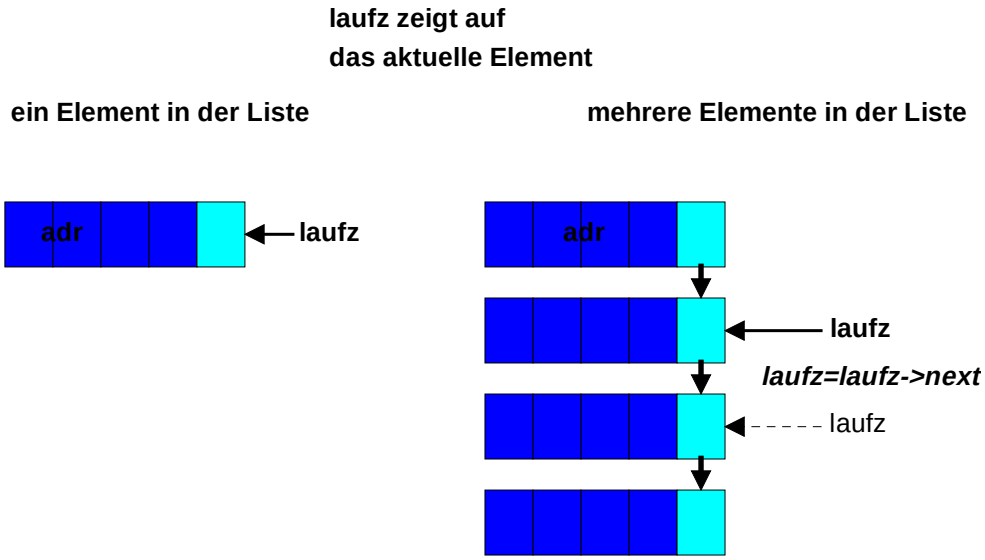
mit new wird ein neuer Speicherplatz für padr angefordert



Zweiter Schritt (Veränderungen nach dem Anhängen):



Dritter Schritt (herauslesen und drucken der Daten):



16.2.2 Sortierte, verkettete Liste

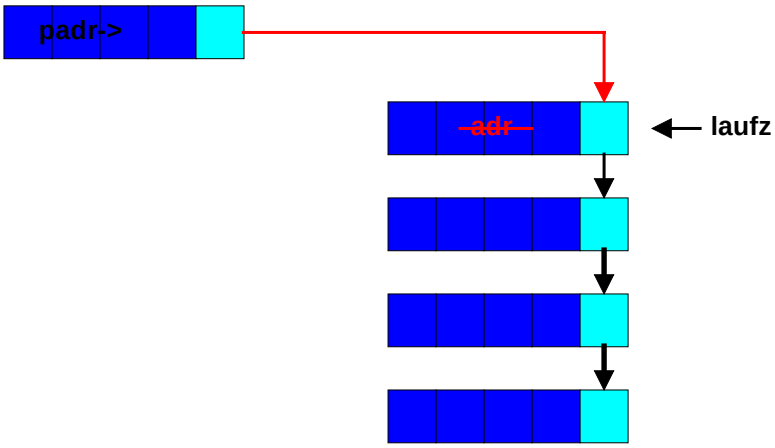
Wenn man eine Liste sortieren möchte, so muß man bei jedem Element schauen, an welche Stelle es in der Liste eingefügt werden soll.

Nehmen wir im folgenden an, daß wir unsere Liste nach dem Namen sortieren.

Es gibt 3 verschiedene Möglichkeiten:

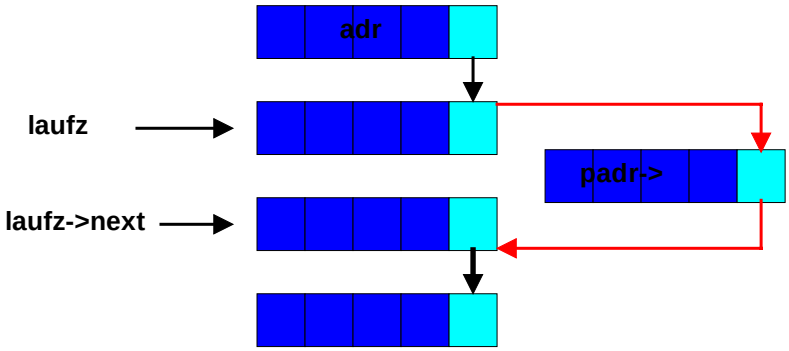
Das neue Element kommt an den Anfang der Liste:

In einer getrennten Abfrage muß geklärt werden, ob laufz auf adr steht und ob das neue Element vor adr eingefügt werden soll.
In diesen Fall muß das neue Listenelement der Anfang der Liste werden, d.h. es muß adr auf das neue Element zeigen. Adr-next wird der alte Anfang der Liste.



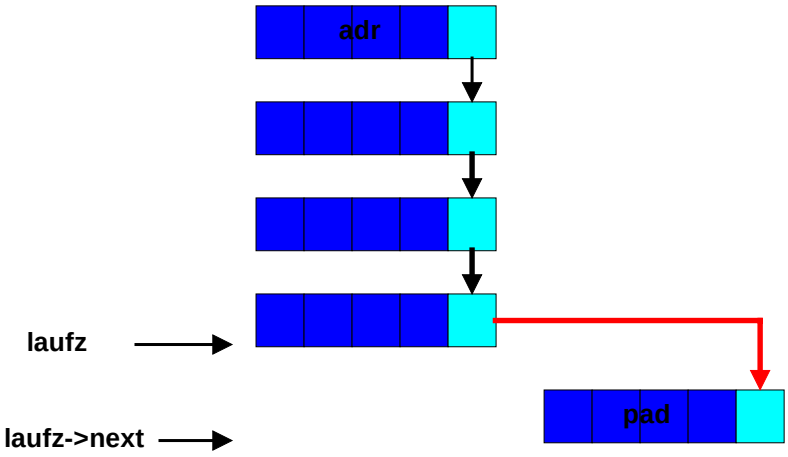
Das neue Element kommt zwischen zwei Listenelemente

In einer Schleife muß immer nachgefragt werden, ob zwischen laufz->name und laufz->next->name das Element hineinpaßt. Diese Abfrage ist in dieser Form notwendig, da man im Falle eines positiven Abfrageergebnisses die Adresse des Elements hat, nach dem das neue Element eingefügt werden soll.
Nun muß man folgendes machen: padr-> next muß auf laufz->next->next weisen, laufz->next darf nicht mehr auf das alte Element weisen, sondern muß auf das neue Element zeigen.



Das neue Element wird ans Ende der Liste angehängt:

In diesem Fall wird geprüft, ob das nächste Listenelement leer ist, ob man also beim Vergleichen am Ende der Liste angekommen ist.



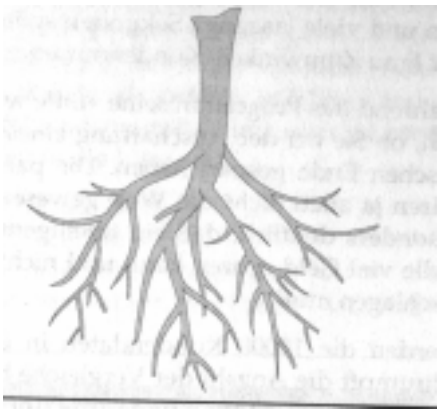
Beispielprogramm:

```
/*************************************************/
/*    einfach verkettete, sortierte Liste        */
/*************************************************/

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

/*************************************************/
/*    globale Definition von struct              */
/*************************************************/

struct adresse_t
 {
  char  name[25], vorname[25], strasse[30];
  char  plz [6], ort[25];
  adresse_t *next;
 };

void main (void)
{

 adresse_t *adr, *padr,*laufz;
 //int lauf=0;
 char antwort[2];
 char  name[25], vorname[25], strasse[30];
 char  plz [6], ort[25];
printf("\nWollen Sie noch eine Adresse anhängen (j/n): ");
 scanf ("%s",antwort);
 adr=NULL;
 while ((antwort[0]=='J') || (antwort[0]=='j'))
 {
  padr= new adresse_t;
  printf("\n\nFamiliennamen: ");
  scanf("%s",padr->name);
  printf("Vorname      : ");
  scanf("%s",padr->vorname);
  printf("Strasse      : ");
  scanf("%s",padr->strasse);
  printf("Postleitzahl : ");
  scanf("%s",padr->plz);
```

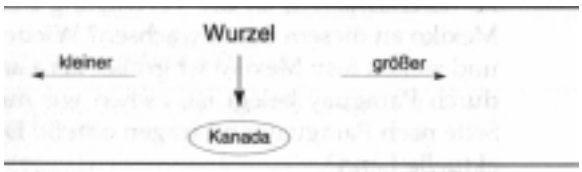
```
  printf("Wohnort      : ");
  scanf("%s",padr->ort);
printf("\n\nWollen Sie noch eine Adresse anhängen (j/n): ");
  scanf ("%s",antwort);

  // wenn noch kein Element in der Liste, dann wird adr gleich
  // padr gesetzt

  if(adr==NULL)
  {
   adr=padr;
   laufz=adr;
   laufz->next=NULL;
  }
  else
  {
    laufz=adr;
    while ((strcmp(laufz->name,padr->name)<0) &&
    (strcmp (laufz->next->name,padr->name)<0)
    && (laufz->next!=NULL)&&laufz!=NULL)
        laufz=laufz->next;
    if ((laufz->next==NULL)&&(strcmp(laufz->name,padr->name)<0))
{   // einfaches anhängen
     laufz->next=padr;
     laufz->next->next=NULL;
    }
    else
     if ((laufz==adr)&&(strcmp(laufz->name,padr->name)>=0))
     {    // vertauschen des 1.mit dem 2. Element
      adr=padr;
      adr->next=laufz;
      laufz=adr;
     }
     else
     {  // einschieben zwischen zwei
      padr->next=laufz->next;
      laufz->next=padr;
     }
    }

 }
 printf("\n\nJetzt kommt die Ausgabe: ");
 laufz=adr;
 while (laufz!=NULL)
 {
  printf("\n\nFamilienname   : %s",laufz->name);
  printf("\nVorname        : %s",laufz->vorname);
  printf("\nStrasse        : %s",laufz->strasse);
  printf("\nPostleitzahl   : %s",laufz->plz);
  printf("\nWohnort        : %s",laufz->ort);
  laufz=laufz->next;
 }
 delete adr;
 delete padr;
 delete laufz;
}
```

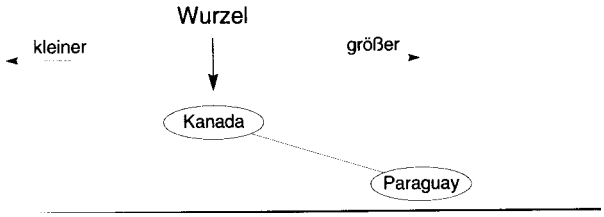
16.3 Binäre Bäume

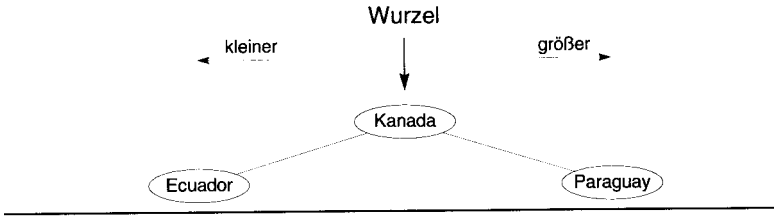
Ein binärer Baum ist wie eine Liste eine dynamische Speicherstruktur.

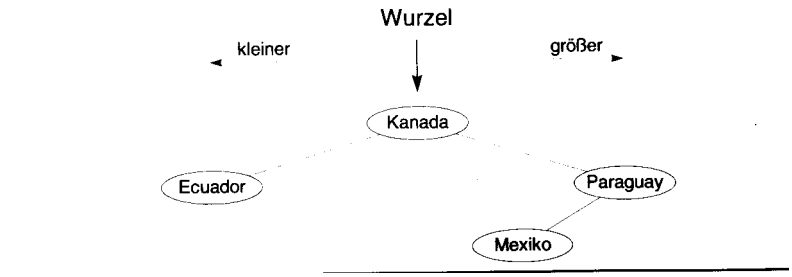


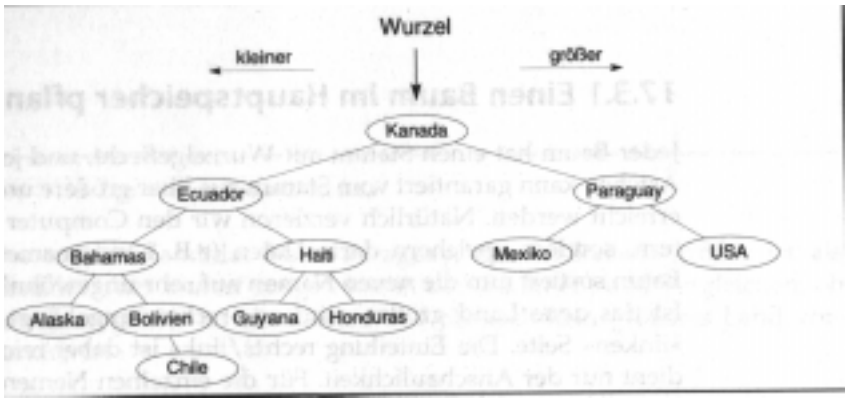
In einem binären Baum wird zunächst eine Wurzel angelegt. Die weiteren Elemente eines Baumes werden dann, wenn sie größer als die Wurzel bzw. einen Knoten sind, rechts eingeordnet, ansonsten links.











Ein binärer Baum besteht wie auch wie eine Liste aus einem Strukt, der einen Zeiger auf das rechte Element hat und einen Zeiger auf das linke Element hat.

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

class C_Baum
{
 /* private Variablen */
     struct Baum
      {
       char  name[80];
       int   i;
       Baum  *rechts;
       Baum  *links;
      };
 Baum *m_wurzel, *m_neu, *m_suchadresse, *m_vorher;
 int gefunden;
 /*  private Funktionen */
 void  m_einfuegen (Baum *m_aktuell);
 void  m_teil_ausgabe(Baum *m_aktuell);
 void m_freigabe (Baum *m_aktuell);
 void  m_anfsuchen (Baum *m_aktuell, int such_name);

 public:
       C_Baum (void);
       ~C_Baum (void);
 void  m_eingabe (void);
  void m_suchen (void);
 void  m_ausgabe (void);
 void  m_loeschen (void);
};

C_Baum::C_Baum (void)
{
 m_wurzel=NULL;
}

C_Baum::~C_Baum (void)
{
 m_vorher=m_wurzel;
 m_freigabe(m_wurzel);

}

void  C_Baum::m_freigabe (Baum *m_aktuell)
```

```
{
 if (m_aktuell->links!=NULL)
 {
  m_vorher=m_aktuell;
  m_freigabe(m_aktuell->links);
 }
 if (m_aktuell->rechts!=NULL)
 {
  m_vorher=m_aktuell;
  m_freigabe(m_aktuell->rechts);
 }
 if ((m_aktuell->rechts==NULL) && (m_aktuell->links==NULL))
   delete m_aktuell;

}



void C_Baum::m_einfuegen (Baum *m_aktuell)
{

 if (m_aktuell==NULL)
 {//komplett neues element
    m_wurzel=m_neu;
 }
 else
 {
   if (m_aktuell->i <= m_neu->i )
   {
     if (m_aktuell->rechts == NULL)
     {
        m_aktuell->rechts=m_neu;
     }
     else
        m_einfuegen(m_aktuell->rechts);
   }
   else
   {
        if (m_aktuell->links==NULL)
        {
           m_aktuell->links=m_neu;
        }
        else
           m_einfuegen(m_aktuell->links);
   }
  }
}


void C_Baum::m_eingabe (void)
{
 m_neu=new Baum;
 m_neu->name[0]='\0';
 m_neu->i=rand()%90;
 m_neu->rechts=NULL;
 m_neu->links=NULL;
 m_einfuegen(m_wurzel);
}

void C_Baum::m_suchen(void)
{
 int such_name;
 gefunden=0;
 cout << "Suchname: ";
```

```
 cin >> such_name;
 m_vorher=m_wurzel;
 m_anfsuchen(m_wurzel,such_name);
}

void C_Baum::m_anfsuchen(Baum *m_aktuelle, int suchen_name)
{
 int such_name=suchen_name;
 if ((such_name>m_aktuelle->i) && (m_aktuelle->rechts!=NULL))
 {
  m_vorher=m_aktuelle;
  m_anfsuchen(m_aktuelle->rechts, such_name);
 }
  if ((such_name<m_aktuelle->i) && (m_aktuelle->links!=NULL))
  {
     m_vorher=m_aktuelle;
     m_anfsuchen(m_aktuelle->links, such_name);
  }
 if (such_name==m_aktuelle->i)
    gefunden=1;
  if (gefunden==1)
   {
    cout << "String gefunden." <<endl;
    gefunden=2;
    m_suchadresse=m_aktuelle;
   }
  if (gefunden==0)
  {
   cout << "String nicht gefunden." <<endl;
   gefunden=2;
   m_suchadresse=NULL;
  }
}

void C_Baum::m_loeschen (void)
{
 Baum * loesch_r, *loesch_l;
 int wurzel=0;
 m_suchen();
 if(m_vorher->rechts==m_suchadresse)
  m_vorher->rechts=NULL;
 if (m_vorher->links==m_suchadresse)
  m_vorher->links=NULL;
 loesch_r=m_suchadresse->rechts;
 loesch_l=m_suchadresse->links;
 if (m_suchadresse==m_wurzel)
 {
  wurzel=1;
  if (loesch_r!=NULL)
  {
   m_wurzel=loesch_r;
   delete m_suchadresse;
   if (loesch_l!=NULL)
   {
    m_neu=loesch_l;
    m_einfuegen(m_wurzel);
   }
  }
  else
  {
   m_wurzel=loesch_l;
   delete m_suchadresse;
   if (loesch_r!=NULL)
   {
    m_neu=loesch_r;
```

```
    m_einfuegen(m_wurzel);
   }
  }
 }
 if (wurzel==0)
 {
  delete m_suchadresse;
  m_neu=loesch_r;
  if (loesch_r!=NULL)
     m_einfuegen(m_wurzel);
  m_neu=loesch_l;
  if (loesch_l!=NULL)
     m_einfuegen(m_wurzel);
 }
}

void C_Baum::m_teil_ausgabe(Baum *m_aktuell)
{
 if (m_aktuell->links!=NULL)
   m_teil_ausgabe (m_aktuell->links);

 cout << m_aktuell->i<<" ";

 if (m_aktuell->rechts!=NULL)
    m_teil_ausgabe(m_aktuell->rechts);
}

void C_Baum::m_ausgabe (void)
{
 char weiter;
 m_teil_ausgabe(m_wurzel);
 cout << "\n Weiter mit einer Taste" << endl;
 cin >> weiter;
}

void main (void)
{
 C_Baum Stadtbaum;
 int menue;
 cout << "New start\n\n";
 do
 {
  clrscr();
cout << "Menüauswahl"<<endl;
  cout << "Eingabe: 1" <<endl;
  cout << "Suchen : 2" <<endl;
  cout << "Löschen: 3" <<endl;
  cout << "Ausgabe: 4" <<endl;
  cout << "Beenden: 0" <<endl;
  cout << "Ihre Wahl: ";
  cin >> menue;
  switch (menue)
  {
   case 1: Stadtbaum.m_eingabe(); break;
   case 2: Stadtbaum.m_suchen (); break;
   case 3: Stadtbaum.m_loeschen(); break;
   case 4: Stadtbaum.m_ausgabe();break;
  }
 }
 while (menue!=0);
}
```

16.4 Fifo

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

class fifo_list
{
 struct list
 {
  list *before;
  int  content;
  list *next;
 } *liste;

 int max,number;
 public:
        fifo_list(int max);
        ~fifo_list (void);
  void  push (int data);
  void  pop (void);
  void  pop_stack(void);
};

fifo_list::fifo_list (int maxy=10)
{

 liste=new (list);
 number=0;
 liste->before=liste;
 liste->next=liste;
 liste->content=99;
 max=maxy;


}

fifo_list::~fifo_list (void)
{
 list *help;

 for (int i=0; i<=number; i++)
 {
  help=liste->next;
  delete liste;
  liste=help;
 }
}

void fifo_list::push (int data)
{
 list *neu;

 if (number<max)
 {
  neu=new (list);


  neu->next   = liste;          //ring
  neu->before = liste->before; //ok

  liste->before->next = neu;

  liste->before = neu;         //ok
```

```
  liste->before->content = data;

  number++;

 }
 else
  cout << "Es kann kein weiteres Element gespeichert werden" << endl;
}

void fifo_list::pop_stack (void)
{
 list *help;

 if (number != 0)
 {
  cout << liste->before->content << endl;
  cout << liste->next->content << endl;

  help=liste->before;

  liste->before->before->next=liste;
  liste->before=liste->before->before;
  delete help;
  number--;
 }
 else
  cout << "Es ist kein Element mehr im Speicher" << endl;
}



void fifo_list::pop (void)
{
 list *help;

 if (number != 0)
 {
  cout << liste->next->content << endl;

  help=liste->next;

  liste->next->before = liste;
  liste->next         = liste->next->next;

  delete help;

  number--;
 }
 else
  cout << "Es ist kein Element mehr im Speicher" << endl;
}

void main (void)
{
 clrscr();
 fifo_list first (4);
 first.push(1);
 first.push(2);
 first.push(3);
 first.push(4);
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();
```

```
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();



// cout << "First: "<< first.pop() << endl;
 /*for (int i=0; i<3 ; i++)
  first.push (i);
 //for (i=0; i<=10; i++)
 // second.push (i);
 for (i=0; i<3; i++)
  cout << "First: " <<first.pop() << endl;
 //for (i=0; i<=10; i++)
 // cout << "Second: " << second.pop() << endl;*/
}
```

Stack

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

class fifo_list
{
public:
 struct list
 {
  list *before;
  int  number;
  int  content;
  list *next;
 } *liste;

 int max;

        fifo_list(int max);
        ~fifo_list (void);
  void  push (int data);
  void  pop (void);
};

fifo_list::fifo_list (int maxy=10)
{

 liste=new (list);
 liste->number=0;
 liste->before=liste;
 liste->next=liste;
 max=maxy;


}

fifo_list::~fifo_list (void)
{
 list *help;
 int  number;

 number=liste->before->number;
 for (int i=0; i<=number; i++)
 {
  help=liste->next;
```

```
  delete liste;
  liste=help;
 }
}

void fifo_list::push (int data)
{
 list *neu;
 int  number;

 number=liste->before->number;
 if (number<max)
 {
  neu=new (list);


  neu->next   = liste;
  neu->before = liste->before;

  liste->before = neu;
  liste->before->next = neu;

  liste->before->content = data;
  liste->before->number = number+1;


//  neu->number = number+1;

 }
 else
  cout << "Es kann kein weiteres Element gespeichert werden" << endl;
}

void fifo_list::pop (void)
{
 list *help;

 if (liste->before->number != 0)
 {
  cout << liste->before->content << endl;

  help=liste->before;

  liste->before->before->next=liste;
  liste->before=liste->before->before;
  delete help;
 }
 else
  cout << "Es ist kein Element mehr im Speicher" << endl;
}

void main (void)
{
 clrscr();
 fifo_list first (4);
 first.push(1);
 first.push(2);
 first.push(3);
 first.push(4);
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();
```

```
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();



// cout << "First: "<< first.pop() << endl;
 /*for (int i=0; i<3 ; i++)
  first.push (i);
 //for (i=0; i<=10; i++)
 // second.push (i);
 for (i=0; i<3; i++)
  cout << "First: " <<first.pop() << endl;
 //for (i=0; i<=10; i++)
 // cout << "Second: " << second.pop() << endl;*/
}
```

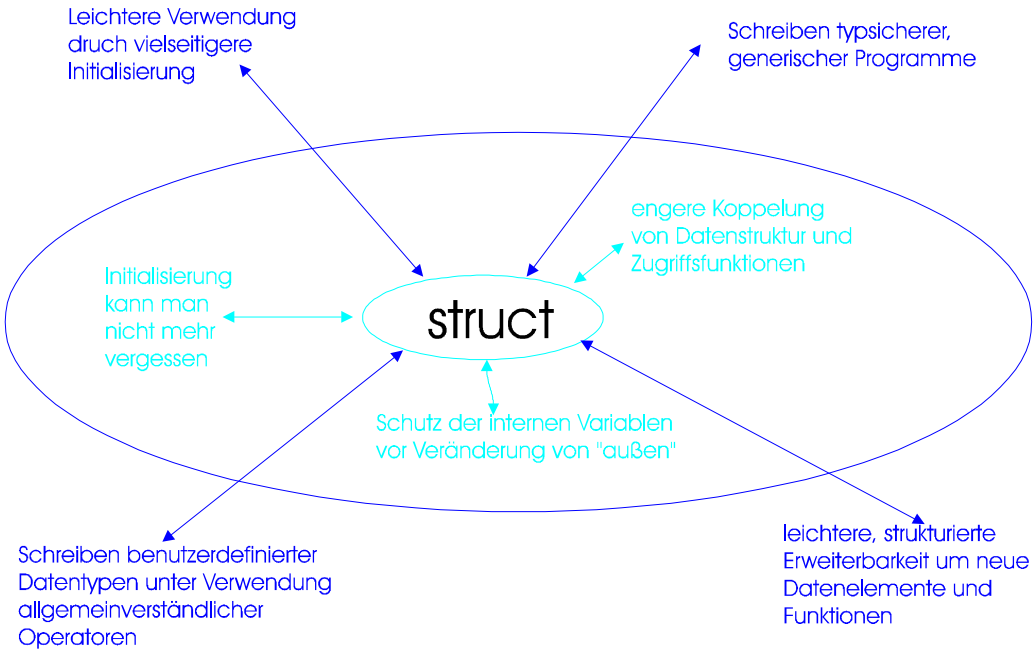
16.5 Stack

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

class fifo_list
{
public:
 struct list
 {
  list *before;
  int  number;
  int  content;
  list *next;
 } *liste;

 int max;

        fifo_list(int max);
        ~fifo_list (void);
  void  push (int data);
  void  pop (void);
};

fifo_list::fifo_list (int maxy=10)
{

 liste=new (list);
 liste->number=0;
 liste->before=liste;
 liste->next=liste;
 max=maxy;


}

fifo_list::~fifo_list (void)
{
 list *help;
 int  number;

 number=liste->before->number;
```

```
 for (int i=0; i<=number; i++)
 {
  help=liste->next;
  delete liste;
  liste=help;
 }
}

void fifo_list::push (int data)
{
 list *neu;
 int  number;

 number=liste->before->number;
 if (number<max)
 {
  neu=new (list);


  neu->next   = liste;
  neu->before = liste->before;

  liste->before = neu;
  liste->before->next = neu;

  liste->before->content = data;
  liste->before->number = number+1;


//  neu->number = number+1;

 }
 else
  cout << "Es kann kein weiteres Element gespeichert werden" << endl;
}

void fifo_list::pop (void)
{
 list *help;

 if (liste->before->number != 0)
 {
  cout << liste->before->content << endl;

  help=liste->before;

  liste->before->before->next=liste;
  liste->before=liste->before->before;
  delete help;
 }
 else
  cout << "Es ist kein Element mehr im Speicher" << endl;
}

void main (void)
{
 clrscr();
 fifo_list first (4);
 first.push(1);
 first.push(2);
 first.push(3);
 first.push(4);
 cout << "First: ";
  first.pop();
 cout << "First: ";
```

```
  first.pop();
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();


// cout << "First: "<< first.pop() << endl;
 /*for (int i=0; i<3 ; i++)
  first.push (i);
 //for (i=0; i<=10; i++)
 // second.push (i);
 for (i=0; i<3; i++)
  cout << "First: " <<first.pop() << endl;
 //for (i=0; i<=10; i++)
 // cout << "Second: " << second.pop() << endl;*/
}
```

17 Klassen



17.1 Unterschied zwischen der class und struct Anweisung

Die Unterschiede zwischen den beiden Anweisungen class und struct sind sehr gering. Zunächst besteht ein großer Unterschied bei der Voreinstellung der Zugriffsrechte die in einer mit class deklarierten Variablen und Funktion werden ohne Veränderung der Zugriffsrechte als private deklariert, die Variablen der struct Anweisung dagegen als public.

Bei der class Anweisung werden nicht nur die Variablen deklariert, sondern auch die Funktionen angegeben, die diese Variablen verändern dürfen. Die Variablen, die in einer class deklariert sind, sind einfach gesehen, global für die Funktionen der class deklariert (sie sind statics). Es kann also jede Funktion einer Klasse ohne zusätzliche Parameterübergabe auf den Inhalt der Variablen zugreifen und ihn verändern.

17.2 Aufbau einer Klasse

Eine Klasse wird zunächst mit dem Schlüsselwort class begonnen. Sie kann einen Namen tragen oder anonym bleiben.

In der Klasse werden normalerweise die Variablen als private deklariert, d.h. sie können nur intern von den in der Klasse definierten Funktionen benutzt werden. Auch Funktionen können als private deklariert werden. Da die Voreinstellung bei der Klasse private ist, muß nicht mehr ausdrücklich daraufhingewiesen werden, wenn Variablen und Funktionen private sind. Dage-

gen muß bei public Variablen und Funktionen ausdrücklich vorher darauf hingewiesen werden, daß sie diesen Status tragen. Generell muß mindestens eine Funktion einer Klasse als Public definiert sein, damit man auf die restlichen Variablen und Funktionen der Klasse überhaupt zugreifen kann.

Beispiel:

```
class neu                                   class neu
{                                           {
 // Variablen die nur intern von den          private:
Funktion                                     int i, h;
 // schreiben und drucken verwendet          char string[10];
werden                                       long *x;
 // dürfen
  int i, h;                                  public:
  char string[10];                            void schreiben (void);
  long *x;                                    int drucken (void);
// Die Funktion kann nur von schrei-        }
// ben oder drucken aus aufgerufen
// werden
  void hallo (void);

  public:
// öffentliche Funktionen der Klasse
   void schreiben (void);
   int drucken (void);
}
```

Beide Deklarationsmöglichkeiten sind möglich.

In der Klassendefinition selber sind die Funktionen nur als Prototypen definiert. Sie müssen deshalb außerhalb der Klasse noch einmal genau implementiert werden.

Normale Funktionen (keine Konstruktoren bzw. Destruktoren), werden im Programm wie folgt deklariert, dabei muß man im Programm definieren, zu welcher Klasse die implementierte Funktion gehört:

Datentyp_der_Funktion Klassennamen :: Funktionsname (Parameter)

Beispiel:

Im obigen Beispiel wurden nur die Prototypen schreiben und drucken der Klasse neu definiert. Nun wird die Funktion schreiben implementiert.

```
void neu::schreiben (void)
{
printf("%i",i);
}
```

17.3 Konstruktoren und Destruktoren

Es kann sehr leicht passieren, daß man zum Beispiel am Ende des Programms vergißt, den Speicherplatz, den man mit new angefordert hat, wieder mit delete freizugeben.

Aus diesem Grund gibt es bei den Klassen die Möglichkeit, die Initialisierung und die Speicheranforderung automatisch durchführen zu lassen.

Dies geschieht mit den sogenannten Konstruktoren und Destruktoren. Der Konstruktor wird selbständig aufgerufen, so bald das "Leben eines Objektes beginnt", der Dekonstruktor, so bald das "Leben eines Objekts endet", z.B. bei den schließenden Klammern eines Blockes.

Es ist nicht notwendig, Konstruktor und Destruktor in einer Klasse zu definieren. In diesem Fall werden trotzdem beide während des Programmablaufes aufgerufen. Sie sind jedoch leer und führen deshalb keine Befehle aus.

Werden sie dagegen in der Klasse definiert, so werden die "leeren" Konstruktoren bzw. Destruktoren "einfach überschrieben". An den richtigen Stellen im Programm werden dann die neuen Konstruktoren bzw. Destruktoren ausgeführt.

17.3.1 Deklaration von Konstruktoren und Destruktoren

Beispiel (es wird kein Konstruktor bzw. Destruktor definiert):

```
class neu
{
private:
int i, h;
char string[10];
long *x;

public:
void init(); //Initialisierung
void schreiben (void);
int drucken (void);
}
```

Konstruktoren werden wie folgt in der Klasse definiert:

Klassenname (Parameterübergabe);

Es ist möglich dem Konstruktor Parameter zu übergeben. Es muß kein void vorliegen.

Destruktoren werden wie Konstruktoren definiert, haben jedoch zusätzlich noch am Anfang eine Tilde "~":

~Klassenname (Parameterübergabe);

Beispiel (Deklaration eines Konstruktors):

```
class neu
{
private:
int i, h;
char string[10];
long *x;

public:
neu(); // Initialisierung, anstelle von init
void schreiben (void);
int drucken (void);
}
```

Beispiel:

```
class A {
public:
A(); // Konstruktor
~A(); // Destruktor
private:
int nummer;
};
```

Konstruktoren und Destruktoren können (wie jede andere Elementfunktion der Klasse auch) innerhalb der Klasse oder auch extern definiert sein.

Beispiele:

```
A:: A()
{
  ...
}


A::~A()
{
...
}
```

17.3.2 Überlagern von Konstruktoren

In einer Klasse kann es notwendig sein verschiedene Konstruktoren zur Verfügung zu stellen. (siehe Kapitel 3)
Mit Hilfe der Überlagerung von Konstruktoren können zum Beispiel unterschiedliche Konstruktoren aufgerufen werden.

Beispiel:

```
class Datum
{
 int t, m, j;
 public:
 Datum (int, int, int);    // Tag, Monat, Jahr
 Datum (int, int);        // Tag, Monat und aktuelles Jahr
 Datum (int);             // Tag, aktueller Monat und aktuelles Jahr
 Datum ( );               //heutiges Datum
}
```

In manchen Fällen ist es auch möglich einen einzigen Konstruktor zu verwenden. Man kann dies mit Hilfe eines Default-Wertes realisieren.
Bei einem Default-Wert, werden die (einige) Variablen in der Parameterliste mit einem Wert initialisiert. Wird beim Aufruf der Funktion / Konstruktors kein Wert übergeben, dann wird die Variable, mit dem in der Parameterliste stehenden Wert initialisiert.

Beispiel:

```
Datum (int t=0, int m=0, int j=0);
```

```
datum a(1,12,33);   // t=1, m=12, j=33
datum b(2,5);       // t=2, m=5, j=0
datum c;            // t=0, m=0, j=0
```

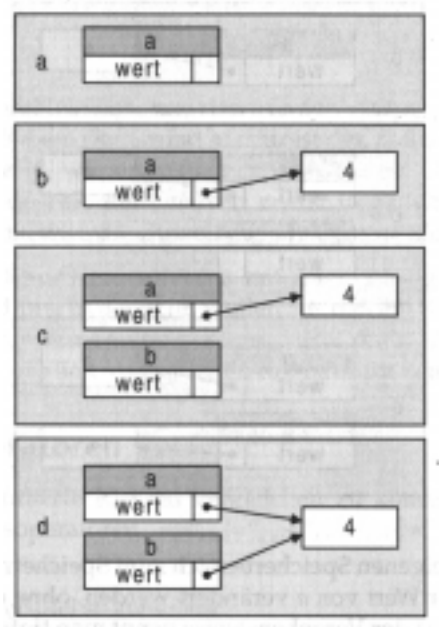
Außerdem gibt es die Möglichkeit gleich die übergebenen Werte der Membervariablen zuzuweisen:

```
class Datum
{
 int t, m, j;
 public:
 Datum (int tag, int monat, int jahr): t(tag), m(monat), j(jahr) {};
};
```

Der Strichpunkt am Ende der geschweiften Klammern ist optional. Diese Zeile entspricht:
```
t=Tag;
m=monat;
j=jahr;
```

17.4 Copy-Konstruktor

Es gibt nun die Möglichkeit, daß man einem neues Objekt mit dem Wert eines bestehenden Objekts initialisiert. Man benötigt hierfür einen geeigneten Konstruktor, da sonst C++ einen Standard-Copy-Konstruktor verwendet. Die Anwendung dieses Standard Konstruktors führt dazu, daß alle Elemente des zu kopierenden Objekts einfach kopiert werden. Wenn diese "Vorlage" einen Zeiger auf Daten, oä. enthalten sollte, so wird dieser Zeigerwert ebenfalls nur kopiert, d.h. nun verweisen mehrere Objekte auf den gleichen Speicher. Wird das erste Objekt verändert, so verändert sich auch der Wert des zweiten Objekts. Dies ist normalerweise nicht richtig.

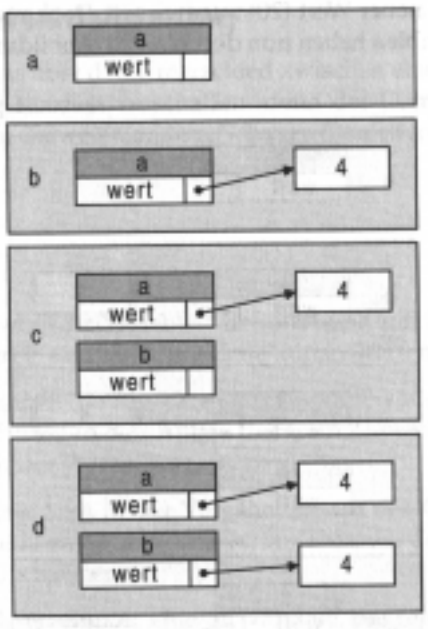


Um dieses Problem zu beseitigen schreibt man einen eigenen Copy-Konstruktor. Man überladet die anderen vorhandenen Konstruktoren. Als Übergabeparameter verwendet man in die-

sem Fall eine konstante Referenz auf den bestimmten Typen oder auch Zeiger (Referenz siehe nächstes Kapitel).

Beispiel:

```
Class Ownint
{
  private:
    int *wert;

  public:
   Ownint(int w)
   {
      wert=new int;
      *wert=w;
   }
   Ownint (const Ownint &k);
   ~Ownint() {if (wert) delete (wert);}
   void Print (void) {cout << *wert << endl; }
   void Set(int w) {*wert=w;}
}

Ownint::Ownint (const Ownint &k)
{
   wert=new int;
   wert=k.wert;
}
```



17.5 Statische Klassenelemente

...

17.6 Übungsaufgaben

1. Schreiben Sie ein Programm, daß einen Fifo mit Hilfe von Klassen implementiert. Das Fifo soll eine, bei der Initialisierung, angebbare Menge an Daten speichern können.

Hier sind 2 Beispielprogramme für Aufgabe 1:

```
/* FIFO mit Klassen, FIFO wird in Feldern gespeichert */

#include <iostream.h>

class Fifo
{
 int *buf;
 int top;  //letzer freier platz!!!!!
 int max_size;

 public:
  Fifo (int s=10);
  ~Fifo();
  void push (int data);
  int pop();
}

Fifo::Fifo (int s)
{
 max_size=s-1;
 top=max_size;
 buf=new int[max_size];
}

Fifo::~Fifo ()
{
 delete [] buf;
}

void Fifo::push (int data)
{
 if (top>=0)
  buf[top--]=data;
 else
  cout << "Fifo full\n";

}

int Fifo::pop ()
{
 int buffer=-1;
 int i;

if (top<max_size)  //elemente noch da sonst top==max_size
 { // top innerhalb vom fifo== elemente drinnen
  buffer=buf[max_size]; //letzte temp. sichern

  // weiterschieben der elemente

i=max_size;  //laufzeiger
  while (i!=top+1)  //laufen bis zum letzten element
  {
   buf[i]=buf[i-1];
   i--;
  }
  //top erhoehen
  top++;

  return buffer; // element zurueckliefern
 }
 else  // top==max_size ==> keine elemente
```

```
 {
  cout << "Fifo empty";
  return -1;
 }

}

main ()
{
 Fifo var1;
 Fifo *var2 = new Fifo(20);
 cout << "Neuer Start"<< endl;
 for (int zaehler=0; zaehler <=10; zaehler++)
  var1.push(zaehler);

 for (int i=0; i<=10; i++)
  cout << "Pop ("<<i<<") = "<<var1.pop() << endl;

 cout << "Test auf Fifounderflow: ";

 return 0;
}


/* FIFO verwirklicht mit einem Listenring und Klassen */
#include <iostream.h>
#include <conio.h>

class fifo_list
{
 struct list
 {
  list *before;
  int  content;
  list *next;
 } *liste;

 int max,number;
 public:
        fifo_list(int max);
        ~fifo_list (void);
  void  push (int data);
  void  pop (void);
  void  pop_stack(void);
};

fifo_list::fifo_list (int maxy=10)
{
 liste=new (list);
 number=0;
 liste->before=liste;
 liste->next=liste;
 liste->content=99;
 max=maxy;
}

fifo_list::~fifo_list (void)
{
 list *help;

 for (int i=0; i<=number; i++)
 {
  help=liste->next;
  delete liste;
  liste=help;
```

```
 }
}

void fifo_list::push (int data)
{
 list *neu;

 if (number<max)
 {
  neu=new (list);
  neu->next   = liste;
  neu->before = liste->before;
  liste->before->next = neu;
  liste->before = neu;
  liste->before->content = data;
  number++;
 }
 else
  cout << "Es kann kein weiteres Element gespeichert werden" << endl;
}

void fifo_list::pop_stack (void)
{
 list *help;

 if (number != 0)
 {
  cout << liste->before->content << endl;
  cout << liste->next->content << endl;
  help=liste->before;
  liste->before->before->next=liste;
  liste->before=liste->before->before;
  delete help;
  number--;
 }
 else
  cout << "Es ist kein Element mehr im Speicher" << endl;
}



void fifo_list::pop (void)
{
 list *help;

 if (number != 0)
 {
  cout << liste->next->content << endl;
  help=liste->next;
  liste->next->before = liste;
  liste->next         = liste->next->next;
  delete help;
  number--;
 }
 else
  cout << "Es ist kein Element mehr im Speicher" << endl;
}

void main (void)
{
 clrscr();
 fifo_list first (4);
 first.push(1);
 first.push(2);
 first.push(3);
```

```
 first.push(4);
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();
 cout << "First: ";
  first.pop();
}
```

2. Aufzug-Programm:

Aufgabenstellung:

Konzipieren (1) und implementieren (2) Sie einen Simulator für eine Aufzugssteuerung, etwa wie man sie im FH-Gebäude A (für einen Aufzug) findet. Der Simulator soll alle wesentlichen Vorgänge eines Aufzugs simulieren können und eine Bewertung der Aufzugssteuerung erlauben.

Vorgänge des Aufzugs:

-- Anforderungsknopf in einer (oder mehreren) Ebene(n) gedrückt
-- Tür auf/zu
-- Lichtschranke frei
-- Personen treten ein und drücken den Knopf für das gewünschte Fahrziel
-- Fahrstuhl setzt sich gemäß bekannter Strategie in Bewegung
-- Personen verlassen den Aufzug
-- Gewichtsüberwachung, d.h. Alarmglocke, wenn zu viele Personen eintreten. Keine Fahrt antreten, Türen nicht schließen, bis Gewicht wieder reduziert wurde Bewertungskriterien für eine Aufzugssteuerung
  (Für diese Kriterien soll der Simulator entsprechende Daten liefern)
-- Durchsatz (jede Aktion (Tür auf/zu, Fahrt des Aufzugs zum nächsten Stockwerk, Licht schranke frei, auf frei warten, etc.) kostet Zeit. Gesucht ist die durchschnittliche Zeit, vom Eintreffen von Personen vor dem Aufzug bis sie dann wieder den Fahrstuhl auf der gewünschten Etage verlassen.
-- Akzeptanz; Menschen tendieren dazu, nach einer bestimmten Zeit des Wartens vor dem Aufzug, ohne daß dieser eingetroffen ist, das Warten abzubrechen und die Treppe zu nehmen. Gesucht ist die Abbrechquote (Abbruch zur Gesamtzahl der Personen)

Hinweise

-- Überlegen Sie zuerst, welche Aktionen Sie in Klassen zusammenfassen wollen (z.B. eigene Klassen für Personen, Aufzug, Simulation, ...)
-- Überlegen Sie, wie Sie den Simulationsvorgang gestalten wollen, z.B. zufälliges Bestimmen einer Etage, auf der eine neue Person den Aufzug anfordern soll. Diese fordert den Aufzug an, Fahrstuhlsimulation geht einen Schritt weiter, falls Fahrstuhl ankommt, dann tritt die Person ein, falls Überlast, etc.)
-- Listen Sie die Beziehungen/Verknüpfungen zwischen den Klassen auf.
-- Implementieren Sie den Simulator derart, daß der eigentliche Aufzug mit seinen Sensoren, Aktoren und insbesondere der Fahrstrategie gut gekapselt ist (eigene Klasse). Es soll leicht möglich sein, mit alternativen Fahrstrategien zu experimentieren.
-- Bei der Simulation können Sie so vorgehen, daß Sie die kleinste Zeiteinheit bestimmen, die im (Aufzug-) System vorkommen kann. Diese können Sie als Simulationstakt wählen und alle anderen Aktionen relativ dazu wählen bzw. abtesten.

-- Implementieren Sie die einzelnen Teilaufgaben des Simulators in eigenen Dateien!

Gedanken vor dem Programmieren:
Fahrstrategie:

- Wenn keine Person im Aufzug ist, oder draußen wartet, so soll die Türe geöffnet sein und der Aufzug im aktuellen Stockwerk halten und warten.
- Sobald die 1.Person gedrückt hat, fährt der Aufzug in diese Richtung. Da die Person schon von "außen" auch ihr Ziel eingibt, wird nach der Zielrichtung die Vorzugsrichtung festgelegt. Wenn weitere Personen einsteigen, fährt der Aufzug zunächst in dieser Vorzsrichtung, oder bis der oberste bzw. unterste Stock erreicht ist. Dann wird die Vorzugsrichtung geändert.

Hier ist ein Teil des Aufzugprogramms (es sind noch Verbesserungen notwendig, um die Aufabe vollständig zu erfüllen):

```
#include <iostream.h>
#include <time.h>
#include <stdlib.h>
#include <conio.h>
#include "auf_pers.cpp"
#define max 6

class aufzug
{
int     tuer;               //Tür auf oder zu  0=zu 1=offen
 int     max_gewicht;        //maximales Gesamtpersonen der Insassen
 int     mom_gewicht;        //momentanes Gewicht
 int     zeit;               //Uhr
 int     fahren_erlaubt;     //darf der Aufzug fahren? 0=nein 1=ja
int     knopf[max];         //Feld für gedrückte Knöpfe
 int     stock;              //momentanes Stockwerk
 int     richtung;           //0=runter 1=rauf  2=stehen
 int     anzahl;
 int     ausgestiegen;
 int     zu_spaet;

 public:
 personen pers_liste;
 aufzug (void);
 void strategie         (void);
 void tuere             (void);
 int  rueck_ausgestiegen(void);
 void neue_person       (void);
 void fahren            (void);
};


aufzug::aufzug (void)
{
 tuer=1;
 max_gewicht=8*75;
 mom_gewicht=0;
 zeit=0;
 anzahl=0;
 richtung=2;
 stock=1;
 randomize();
 ausgestiegen=0;
 for (int i=0; i<max; i++)
  knopf[i]=0;
}
```

```
void aufzug::strategie (void)
{
 int gedrueckt=0;
 int akt_stock=stock;
 int min_stock=max-1;
 int max_stock=0;
 for (int i=0; i<max; i++)
  if (knopf[i]!=0)
  {
   gedrueckt++;
   if (min_stock>i)
    min_stock=i;
   if (max_stock<i)
    max_stock=i;
  }
 if ((gedrueckt==0)  || (akt_stock==min_stock) || (akt_stock==max_stock))
// Aufzug bleibt stehen
  richtung=2;
 else
 {
  if (gedrueckt==1)    //Richtung wird bestimmt
  {
   if (min_stock<akt_stock)
    richtung=0;
   if (min_stock>akt_stock)
    richtung=1;
  }
  else
  {
   if ( (akt_stock==min_stock) || (akt_stock==max_stock))
   {
    if (richtung==0)
     richtung=1;
    else
     richtung=0;
   }
   else
      if (akt_stock>max_stock)
       richtung=0;
      else
       richtung=1;

    }
  }

 }



void aufzug::tuere (void)
{
 int jemand_da;
 fahren_erlaubt=0;
 if (knopf[stock]!=0)
 {
  if (tuer!=1)
  {
    tuer=1;
    zeit+=3;
  }
  ausgestiegen=pers_liste.suchen_ziel(stock,&zeit, &mom_gewicht);
  knopf[stock]=0;
  zu_spaet=pers_liste.suchen_start(stock, zeit, knopf, mom_gewicht,
```

```
                                    max_gewicht);
  ausgestiegen=ausgestiegen+zu_spaet;
  zeit+=3;
  tuer=0;
 }
}

void aufzug::neue_person (void)
{
 int x;
 x=random(2);
 if (x==1)
 {
  pers_liste.eingabe(knopf, zeit);
  pers_liste.einsortieren();
  anzahl++;
 }

}

int aufzug::rueck_ausgestiegen (void)
{
 return ausgestiegen;
}

void aufzug::fahren (void)
{
 if (anzahl<3)
  neue_person();

 if (richtung==2)
 {
  stock+=0;
  tuer=1;
  zeit+=3;
 }
 if ( (richtung==0) && (stock>0))
 {
  stock-=1;
  zeit+=6;
  tuer=0;
 }
 if ( (richtung==1) && (stock<5))
 {
  stock+=1;
  zeit+=6;
  tuer=0;
 }
 //tuere();
 strategie();
 tuere();

 if (anzahl<3)
  neue_person();
}



 void main (void)
{

 int anzahl=0;
 int richt;
 int abbruch;
 int aus=0;
```

```
 aufzug aufzug_1;
 clrscr();
 do
 {
  aufzug_1.fahren();
  aus=aufzug_1.rueck_ausgestiegen();
  //  aufzug_1.pers_liste.ausgabe();
 }
 while (  (aus!=3) );
}

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



class personen
{
 private:
  struct list
  {
   list     *forward;
   float    gewicht;   //Personengewicht
   int      start;     //von außen gedrücktes Stockwerk
   int      ziel;      //von innen gedrücktes Zielstockwerk
   int      akzeptanz; //wie lange wartet Person auf Aufzug?
   int      startzeit; //Beginn der Fahrtzeit
   int      endezeit;  //Ende der Fahrzeit
   int      fahrtzeit; //Gesamtfahrzeit
   int      wartstart; //Beginn der Wartezeit
   int      eingestiegen;
   list     *next;
  };

   list *liste; //speicherplatz fuer neues element
    list *adr;   //erstes element
    list *laufz;
  public:
    personen();
    ~personen();

//    void init (void);


    void eingabe (int *knopf, int zeit);
    void einsortieren (void);
    void ausgabe (void);
    int suchen_ziel (int stock, int *zeit, int *mom_gewicht);
    int suchen_start (int stock, int &zeit, int *knopf, int &mom_gewicht,
    int &max_gewicht);
    void loeschen (void);
};


personen::personen()
{

 adr=NULL;
 randomize();
}


personen::~personen (void)
{
```

```
 laufz=adr;
 while (laufz!=NULL)
 {
  adr=laufz;
  laufz=laufz->next;
  delete adr;
 }
}



void personen::eingabe (int *knopf, int zeit)
{

 liste= new list;
 liste->next=NULL;
 liste->forward=NULL;


 liste->gewicht=random(60)+50;
 liste->start=random(6);
 do
 {
  liste->ziel=random(6);
 }
 while (liste->ziel==liste->start);
 liste->akzeptanz=random(30);
 liste->eingestiegen=0;
 knopf[liste->start]=1;
 liste->wartstart=zeit;
 cout << "\nGewicht:" << liste->gewicht << " Start " << liste->start << "
Ziel " << liste->ziel;

}




void personen::einsortieren (void)
{



 if (adr==NULL)
 {
  adr=liste;
  adr->next=NULL;
  adr->forward = NULL;
 }
 else
 {
  laufz=adr;
  liste->next=adr;
  liste->forward=NULL;
  laufz->forward=liste;
  adr=liste;
 }
}



int personen::suchen_ziel (int stock, int *zeit, int *mom_gewicht)
{
```

```
 int zahl=0;
 laufz=adr;
 while (laufz!=NULL)
 {
  if ((laufz->ziel==stock) && (laufz->eingestiegen!=0))
  {
   laufz->fahrtzeit=laufz->endezeit-laufz->startzeit;
   *zeit+=3;
   *mom_gewicht=*mom_gewicht-laufz->gewicht;
   cout << "\nStock " << stock << "Zeit " << *zeit<< "  Person steigt aus:
   Start " << laufz->start << "Ziel " << laufz->ziel << endl;
   loeschen();
   zahl++;
  }
  laufz=laufz->next;

 }
 return zahl;
}


int personen::suchen_start (int stock, int &zeit, int *knopf, int
&mom_gewicht, int &max_gewicht)
{

 laufz=adr;
 int i=0;

 while (laufz!=NULL)
 {
  if ( (laufz->start==stock) && (laufz->eingestiegen!=1))
  {

   if (zeit - laufz->wartstart <= laufz->akzeptanz)
   {
    mom_gewicht=mom_gewicht+laufz->gewicht;
    if (mom_gewicht<max_gewicht)
    {
     laufz->startzeit=zeit;
     laufz->eingestiegen=1;
     zeit+=3;
     knopf[laufz->ziel]=1;
     cout << "\nStock " << stock << "Zeit " << zeit<<"  Person steigt ein:
Start " << laufz->start << "Ziel " << laufz->ziel << endl;
    }
     else
      knopf[laufz->start]=1;
   }
   else
   {
    cout << "Zulange gewartet: " <<zeit-laufz->wartstart << "< " <<
    laufz->akzeptanz;
    loeschen();
    i++;
   }
  }
  laufz=laufz->next;
 }
 ausgabe();
 return i;
}



void personen::loeschen (void)
```

```
{
 if ((laufz==adr) && (laufz->next==NULL))
 {
  delete laufz;
  adr=NULL;
 }
 else
  if ((laufz==adr) && (laufz->next!=NULL))
  {
    laufz->next->forward=NULL;
    adr=laufz->next;
    delete laufz;
  }
  else
  {
   laufz->forward->next=laufz->next;
   laufz->next->forward=laufz->forward;
   delete laufz;
  }
}

void personen::ausgabe (void)
{
 laufz=adr;
 while (laufz!=NULL)
 {
  cout << "\nStart " << laufz->start << "Ziel " <<laufz->ziel;
  laufz=laufz->next;
 }
 if (adr==NULL)
cout << "Ätsch";
}
```

18 Überlagern von Funktionen

18.1 Überlagern von Funktionen

In C müssen Funktionen, deren Aufgabe gleich ist, der aber unterschiedliche Parameter übergeben werden unterschiedliche Namen erhalten. Man muß dann auch beim Funktionsaufruf darauf achten, daß man den richtigen Funktionsnamen aufruft.
In C++ ist es nun möglich Funktion zu überladen. D.h. Funktionen die den gleichen Inhalt haben aber unterschiedliche Parameterlisten haben können den gleichen Namen bekommen.

Beispiele:

```
double addiere (double a, double b)
{
 return a+b;
}

int addiere (int a,int b)
{
 return a+b;
}

void main (void)
{
 int  x,y,z;
 double m,n,o;
 float a,b,c;
 x=addiere (y,z); // es wird die Addition von zwei Integers durchgeführt
 m=addiere(n,o); // es wird die Addition von zwei Double durchgeführt
  // Was passiert nun?
  a=addiere (b,c); // die Float Zahl wird in ein Double umgewandelt und
                   // dann wird die double Funktion ausgeführt
}
```

Im letzten Fall wird eine interne Typkonvertierung durchgeführt. Eine solche Invertierung wird nach folgenden Rangfolgen durchgeführt:

- Genaue Übereinstimmung: ohne oder nur mit trivialen Konvertierungen (zum Beispiel Feldname nach Zeiger, Funktionsname nach Funktionszeiger und T nach const T)
- Übereinstimmung mit Promotionen; das heißt integrale Promotionen (bool nach int, char nach int, short nach int und ihre unsigned Gegenstücke) float nach double und double nach long double
- Übereinstimmung mit Standardkonvertierungen (zum Beispiel: int nach double, double nach int, Abgeleitet * nach Basis *, T* nach void*, int nach unsigned int)
- Übereinstimmung mit benutzerdefinierten Konvertierungen
- Übereinstimmung mit "..." in einer Funktionsdeklaration

- Mehrere Übereinstimmung auf der gleichen Stufe: Aufruf mehrdeutig
- Rückgabewerte werden bei der Typkonvertierung nicht berücksichtigt: Auflösung für einen einzelnen Operator oder Funktionsaufruf soll kontextunabhängig sein

18.2 Überladen von Operatoren

Eine andere wichtige Anwendung beim Überladen ist, daß man zum Beispiel Operatoren wie +, - selber definieren kann. Dadurch kann man zum Beispiel die Addition zweier komplexen Zahlen ganz normal mit a+b addiert werden.

Folgende Operatoren können überladen werden:

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

- Die Operatoren :: (Bereichsauflösung), *. (Elementauswahl durch einen Funktionszeiger) und . (Elementauswahl) sind nicht durch den Benutzer definierbar.
- Man kann neben den oben aufgeführten Operatoren keine weiteren Operatoren dazudefinieren. (z.B.: operator <> (Pascalnotation, entspricht != in C,C++))

- Desweiteren unterscheidet man ein-, zweistellige Operatoren bzw. Präfix- und Postfixoperatoren.
- Ein einstelliger Operator benötigt nur eine Variable mit der er etwas machen soll. Eine Funktion, die einen einstelligen Operator überlädt, benötigt als Elementfunktion einer Klasse keine Parameter, als nicht Elementfunktion einen Parameter.
- Präfix-Operator
  aa.operator@() oder als operator@(aa)
- Postfix-Operator @
  aa.operator@(int) oder als operator@(aa,int)
- Ein Operator kann nur entsprechen der für ihn in der Grammatik definierten Syntax deklariert werden. ⇒ kein einstelliges % oder dreistelliges +

Beispiele:

- (Präfixminus; negatives Vorzeichen)
+ (Präfixplus; positives Vorzeichen)
& (einstelliger Präfixoperator &; Adresse)

```
class X
{
 X operator- (void);   // keine Parameter
};

X operator- (X);    // ein Parameter
```

Zweistellige Operatoren benötigen zwei Variablen mit denen sie arbeiten können. Beim Überladen eines solchen Operators werden in einer Elementfunktion ein Parameter angegeben und außerhalb zwei Parameter.

Beispiele:

+ (zweistelliges Plus; Addition zweier Zahlen)
- (zweistelliges Minus; Subtraktion zweier Zahlen)

```
class Y
{
 Y operator- (Y);      // ein Parameter
};
Y operator- (Y,Y);  // zwei Parameter
```

Funktionsaufrufe:

```
Y aa,bb,cc;
cc=aa.operator-(bb);    //möglicher  Aufruf  für  eine  Memberfunktion;  ent-
spricht cc=aa+bb
cc=operator-(aa,bb);     //Aufruf für eine nicht Memberfunktion
```

18.3 Konstruktoren und Konvertierungen

Wenn man zum Beispiel komplexe Zahlen addieren möchte, so kann es folgende Möglichkeiten für die Addition geben:

- komplexe Zahl + komplexe Zahl
- Reale Zahl + Reale Zahl
- komplexe Zahl + Reale Zahl
- Reale Zahl + komplexe Zahl

Wenn man nun für Addition, Subtraktion, Division und Multiplikation jeweils 4 verschiedene Funktionen schreiben. Dies bewirkt eine hohe Fehleranfälligkeit.
Eine Alternative wäre, wenn man einen Konstruktor bereitstellt, der eine Realzahl in eine komplexe Zahl umwandelt.
Die Folge ist, daß nun nur noch eine Version des Operators benötigt wird.

Mit Hilfe von friend Funktionen kann eine Funktion, die außerhalb der Klasse deklariert wird aber ein "guter Kumpel" der angegebenen Klasse ist. Die friend Funktion kann auf die private und protected Membervariablen und Memberfunktionen zugreifen.
Einer solchen friend Funktion übergibt man bei zweistelligen Operatoren zwei Parameter. Bei der Übergabe der Parameter müssen die angegebenen Variablen automatisch, auf Grund der Anwendung der entsprechenden Konstruktoren, in den entsprechenden Typ umgewandelt werden.

Beispiel:

```
int
{
  private:
    int *wert;

  public:
   Ownint(int w);
   Ownint (const Ownint &k);
   ~Ownint(void);
   void Print (void);
   void Set(int w);
   friend operator+ (const Ownint &k1, const Ownint &k2);
}

Ownint::Ownint (const Ownint &k)
{
   wert=new int;
   wert=k.wert;
}
```

```
Ownint operator+  (const Ownint &k1, const Ownint &k2)
{
  Ownint k(*k1.wert+*k2.wert);
  Return k;
}
```

Durch diese Deklaration wird folgendes möglich:

a+b

x+2

3+y

Verwendet man jedoch eine Klassenfunktion, dann würde folgendes nicht funktionieren:

4+b

Bei einer Klassenfunktion wird nur der zweite Parameter (hier das b) übergeben und der entsprechende Konstruktor aufgerufen. Dies führt dazu, daß die 4 nicht in das entsprechende Format umgewandelt wird

18.4 Übungsaufgaben

1. Schreiben Sie ein Programm, das die Operatoren + und – für komplexe Zahlen überlädt.

Verwirklichung von Aufgabe 1:

```
#include <iostream.h>

class complex
{

 private:
   float r;
   float i;

 public:
   complex(float, float);  // Konstruktor
   ~complex();             // Destruktor
   float real();           // Rückgabe des Realteils; r ist privat
   float imag();          // Rückgabe des Imaginärteils

   complex operator+ (complex);   // Überlagerung von +
   complex operator- (complex);   // Überlagerung von -
};

complex::complex(float a=0, float b=0)
{
 r=a;
 i=b;
}

complex::~complex()  {}

float complex::real()
{
 return r;
}

float complex::imag()
{
 return i;
}
```

```
complex complex::operator+(complex b)
{
 complex c;
 c.i = i+b.i; // a.operator+(b); i= zahl auf der operator+ ausgefuehrt wird
 c.r = r+b.r;
 return (c);
}

complex complex::operator-(complex b)
{
 complex c;
 c.i = i-b.i; // a.operator-(b); i= zahl auf der operator+ ausgefuehrt wird
 c.r = r-b.r;
 return (c);
}

ostream& operator<<(ostream& s,complex zahl) // Überlagerung von <<
{
 return s<<zahl.real()<< (zahl.imag()>=0?"+":"") <<zahl.imag()<<"i";
}

istream& operator>>(istream& s,complex& zahl) // Überlagerung von >>
{
 float re,im;
 char kl;
 cin >> re;
 cin >> kl;
 cin >> im;
 cin >> kl;
 zahl=complex(re,im);
 return s;
}
void main(void)
{

  complex  a,b,c;   // Konstruktor wird aufgerufen

  cout << "Zahl a: ";
  cin >> a;         // Einlesen einer komplexen Zahl, zum Beispiel 5+6i
  cout << "Zahl b: ";
  cin >> b;
  c = a + b;         //Komplexaddition
  cout << a <<" + "<< b << " = " << c <<endl;
  c = a - b;
  cout << a <<" - "<< b << " = " << c <<endl;
}
```

2. Hier kommt ein aufwendigeres Programm. Schreiben Sie eine Klasse String.
   - Sie soll einen privaten Zeiger auf einen dynamischen Speicher besitzen.
   - Es sollen Variablen für Stringlänge und Stringpuffergröße angelegt werden. Der Stringpuffer soll bei der Konstruktion 15 Zeichen größer sein als die Stringlänge selbst.
   - Sie soll drei Konstruktoren besitzen: String (void), String(const char*), String (const char) diese besetzen den Speicherplatz entsprechend
   - Schreiben Sie drei Zuweisungsoperatoren operator=(String), operator=(const char*) und operator=(const char) sowie einen copy-Konstruktor String(const String&).
   - Überladen Sie die Operatoren << und >> für Strings.
   - Überladen Sie die Operatoren + (Hintereinanderschreiben zweier Strings) und += (String1+=String2 bedeutet, daß hinter String1 noch String2 angehängt werden soll).

- Fügen Sie eine Einfügefunktion hinzu, die an der Stelle x im Stringspeicher die ersten n Buchstaben eines zweiten Strings einfügt.
- Überladen Sie den []-Operator, um auf die einzelnen Zeichen eines String-Objekts genau zugreifen zu können
- Überladen Sie für die Klasse String die Vergleichsoperatoren.
- Überladen Sie den ()-Operator so, daß Sie ihn mit (pos,len) aufrufen können und er dann einen Teilstring erzeugt, der an der Position pos beginnt und len Zeichen lang ist.
- Überladen Sie den -= Operator. String1-=String2: hier sollen alle Vorkommnisse von String2 aus String1 entfernt werden.

```
#include <iostream.h>
#include <string.h>
#include <assert.h>

class String
{
   private:
     char *string_puffer;
     int  string_laenge;
     int  puffer_groesse;

   public:
                String    (void);
                String    (const char text);
                String    (const char *text);
                String    (const String *text);
                ~String   (void);
     String     &operator= (const String &text);
     String     &operator= (const char   *text);
     String     &operator= (const char    text);
     void       einfuegen  (int position, int laenge, const char *string);
     char       &operator[](int position);
     String     operator() (int position, int laenge);
     String     &operator-= (const char *text);
     friend     String operator+ (const String &text1, const String &text2);
     friend     String operator+= (String &text1, const String &text2);
     friend     int    operator== (String const &text1, const String
                &text2);
     friend     int    operator<  (String const &text1, const String
                &text2);
     friend     int    operator>  (String const &text1, const String
                &text2);
     friend     int    operator>= (String const &text1, const String
                &text2);
     friend     int    operator<= (String const &text1, const String
                &text2);
     friend     ostream& operator<<(ostream& s,String text)

};
int laengenbestimmung (const char *text);

String::String (void)
{
  char text[81];

  cout <<"Bitte geben Sie einen Text ein, dieser darf jedoch nicht"<<endl;
  cout <<"l"nger als 80 Buchstaben sein (wenn Sie einen l"ngeren" << endl;
  cout <<"Text eingeben wollen, dann m□ssen sie die Variable gleich" <<
endl;
  cout <<"mit ihm initialisieren."<<endl;
  cout << "Eingabe: ";
  cin >>text;
```

```
  string_laenge=laengenbestimmung(text);
  puffer_groesse=string_laenge+15;
  string_puffer=new char[puffer_groesse];
  strcpy (string_puffer,text);
}

String::String (const char text)
{
  char text_neu[2];
  string_laenge=1;
  puffer_groesse=16;
  text_neu[0]=text;
  text_neu[1]='\0';
  string_puffer=new char[puffer_groesse];
  strcpy (string_puffer,text_neu);
}

String::String (const char  *text)
{

  string_laenge=laengenbestimmung(text);
  puffer_groesse=string_laenge+15;
  string_puffer=new char [puffer_groesse];
  strcpy (string_puffer,text);
}
String::String (const String *text)
{
  string_laenge=text->string_laenge;
  puffer_groesse=string_laenge+15;
  string_puffer=new char[puffer_groesse];
  strcpy (string_puffer,text->string_puffer);
}

String::~String (void)
{
  delete [] string_puffer;
}

String &String::operator=(const String &text)
{
  if (string_laenge<text.string_laenge)
  {
   delete [] string_puffer;
   string_puffer=new char[text.puffer_groesse];
  }
  strcpy (string_puffer,text.string_puffer);
  return *this;
}

String &String::operator= (const char *text)
{
  int text_laenge;
  text_laenge=laengenbestimmung(text);
  if (string_laenge<text_laenge)
  {
    delete [] string_puffer;
    string_puffer=new char [text_laenge+15];
  }
  strcpy (string_puffer, text);
  return *this;
}

String &String::operator= (const char text)
{
  char neu_text[2];
```

```
  neu_text[0]=text;
  neu_text[1]='\0';
  strcpy (string_puffer,neu_text);
  return *this;
}

void String::einfuegen (int position, int laenge, const char *string)
{
  char *neu_string;
  if (string_laenge+laenge>puffer_groesse)
  {
    puffer_groesse=string_laenge+laenge+15;
    neu_string=new char [puffer_groesse];
  }
  else
  {
    neu_string=new char [puffer_groesse];
  }
  for (int i=0; i<position-1; i++)
    neu_string[i]=string_puffer[i];
  for (int j=0; j<laenge-1; j++)
    neu_string[j+position-1]=string[j];
  delete [] string_puffer;
  string_puffer=new char[puffer_groesse];
  strcpy (string_puffer, neu_string);
  delete [] neu_string;
}

char &String::operator[] (int position)
{
 assert (position<string_laenge);  //pr□ft ob man noch im Index ist
 return (string_puffer[position]);
}

int laengenbestimmung (const char *text)
{
  int zaehler=0;
  int string_laenge=0;
  while (text[zaehler])
  {
    string_laenge++;
    zaehler++;
  }
  return string_laenge++;
}

String operator+ (const String &text1, const String &text2)
{
  int l1, l2;
  l1=text1.string_laenge;
  l2=text2.string_laenge;
  char *neu_text;
  neu_text=new char[l1+l2+14];
  strcpy (neu_text, text1.string_puffer);
  strcat (neu_text, text2.string_puffer);
  String neu (neu_text);
  return neu;
}

String operator += (String &text1, const String &text2)
{
  if (text1.string_laenge+text2.string_laenge-1>text1.puffer_groesse)
  {
   delete [] text1.string_puffer;
   text1.string_laenge=text1.string_laenge+text2.string_laenge;
```

```
   text1.puffer_groesse=text1.string_laenge+15;
   text1.string_puffer=new char [text1.puffer_groesse];
  }
  strcat (text1.string_puffer,text2.string_puffer);
  return text1;
}

int operator== (const String &text1, const String &text2)
{
  if (strcmp(text1.string_puffer,text2.string_puffer)==0)
    return 1;
  else
    return 0;
}

int operator < (const String &text1, const String &text2)
{
  if (strcmp(text1.string_puffer,text2.string_puffer)<0)
   return 1;
  else
   return 0;
}

int operator > (const String &text1, const String &text2)
{
  if (strcmp(text1.string_puffer,text2.string_puffer)>0)
   return 1;
  else
   return 0;
}

int operator <= (const String &text1, const String &text2)
{
  if (strcmp(text1.string_puffer,text2.string_puffer)<=0)
   return 1;
  else
   return 0;
}

int operator >= (const String &text1, const String &text2)
{
  if (strcmp(text1.string_puffer,text2.string_puffer)>=0)
   return 1;
  else
   return 0;
}

String String::operator () (int position, int laenge)
{
   char *neu_String;
   neu_String=new char[laenge+15];
   for (int i=position; i<=position+laenge-1; i++)
     neu_String[i-position]=string_puffer[position];
   String neu (neu_String);
   return neu;
}

String &String::operator-= (const char *text)
{
  int text_laenge;
  int i, j,l,k;
  char *hilfe=NULL;
  i=l=j=0;
  text_laenge=laengenbestimmung(text);
  do
```

```
  {
   if (text[l]==string_puffer[i])
   do
   {
     j++;
     l++;
   }
   while (( l<text_laenge) || (text[l]==string_puffer[j]));
   if ((l==text_laenge) && (text[text_laenge]==string_puffer[j]))
   {
     hilfe = new char[puffer_groesse];
     for (int zaehler=0; zaehler<=i;zaehler++)
       hilfe[zaehler]=string_puffer[zaehler];
     for (zaehler=j+1, k=i+1; zaehler<string_laenge; zaehler++,k++)
       hilfe[i]=string_puffer[zaehler];
     strcpy (string_puffer,hilfe);
    }
   i++;
   l=0;
   j=i;
  }
  while (i<string_laenge);
  if (hilfe!=NULL)
  {
    strcpy (string_puffer,hilfe);
    delete [] string_puffer;
    string_laenge=laengenbestimmung(hilfe);
    string_puffer=new char[string_laenge+15];
    delete [] hilfe;
  }
  return *this;
}


ostream& operator<<(ostream& s,String text)
{
 return s<<text.string_puffer;
}

void main (void)
{
  String s1('a');
  String s2("Hallo");
  String s3(s2);
  int gleichheit;
  cout << "S1: " <<s1<<endl;
  cout << "S2: "<<s2<<endl;
  cout << "S3: "<<s3<<endl;
  s3=s3+s3;
  cout << "S3 neu: "<<s3<<endl;
  s2+=s3;
  cout << "S2: "<<s2<<endl;
  String s4;
  s4-="otto";
  cout<<"S4: "<<s4<<endl;
  gleichheit=s2==s3;
  cout << "Gleicheit: "<<gleichheit<<endl;
}
```

19 Referenzen

Eine Referenz ist einfach ein anderer Name für eine Variable. Eine Referenz greift auf die gleiche Adresse wie die Orginalvariable zurück. Wird der Wert der Referenz verändert, dann wird auch die Orginalvariable verändert.
Referenzen müssen initialisiert werden, d.h. es muß gleich angegeben werden, für welche Variable die Referenz steht.

Beispiele:

```
void test (void)
{
 int a = 11;
 int &c;       // Fehler c muß initialisiert werden
 int &b=a;     // b und a beziehen sich auf denselben int
 int x=b;      // x=11
 b = 22;       // a=22
 b++;          // a=23
}
```

Man kann Referenzen anstelle von Zeigern bei Funktionsaufrufen in Parameterlisten verwenden. Wie bei Zeigern bleibt die Änderung innerhalb einer anderen Funktion auch im Hauptprogramm usw. bestehen.

Beispiele:

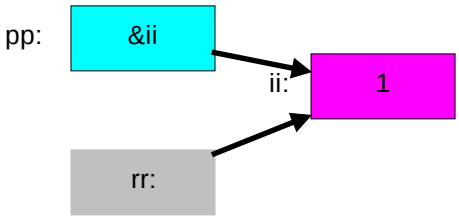
```
void test2 (int &b)
{
 b++;
}

void main (void)
{
 int x=11;
 test2 (x);    //x=12;
}
```

Unterschied zwischen Zeigern und Referenzen:

Bei Zeigern und Referenzen muß man unterscheiden, daß man eine Referenz nicht so manipulieren kann wie Zeiger.

Bildliche Darstellung:

Zeiger pp zeigt auf die Adresse von ii, ebenfalls die Referenz rr



Im Programm:

```
void test (void)
{
 int ii=1;
 int &rr=i;
 int *pp=&rr;
}
```

Die Unterschiede zwischen Zeigern und Referenzen liegt darin, daß man während des Programmablaufes einem Pointer jeder Zeit auf ein anderes Objekt zeigen lassen kann. Eine Referenz kann man jedoch nicht ändern. Diese Tatsache, kann man damit begründen, daß eine Referenz nur ein anderer Name für ein Objekt ist. Warum sollte man nun diesen anderen Namen auf ein neues Objekt "umlenken"??

Beispiel:

```
#include<iostream.h>

void main (void)
{
 int ii=1;
 int &rr=ii;
 int *pp=&rr;
 cout <<"Referenz: "<< rr << endl;
 cout <<"Pointer: "<<pp<<endl;
 int kk=5;
 *pp=kk;
 &rr=kk;   //Fehler: cannot convert 'int' to 'int *'
 cout << "Referenz: " << rr << endl;
 cout << "Pointer: "<<pp<<endl;
}
```

Wie kann man die Änderung innerhalb einer anderen Funktion beibehalten?

- Man kann die Änderung mit return zurückgeben (eingeschränkt auf einen veränderbaren Rückgabewert)
- Man kann Zeiger verwenden
- Man kann Referenzen verwenden

Beispiel:

```
#include <iostream.h>

void incr(int *p)          //Verwendung eines Zeigers
{
 (*p)++;
}

int next (int p)           //Verwendung einer return-Anweisung
```

```
{
 return p+1;
}

void increment (int &aa)   //Verwendung einer Referenz
{
 aa++;
}
void main (void)
{
  int x=1;
  increment(x);
  cout << "x: " << x<<endl;
  x=next(x);
  cout << "x: "<<x<<endl;
  incr(&x);
  cout << "x: "<<x<<endl;
}
```

Ausgabe:

x: 1
x: 2
x: 3

Auch Funktionen können Referenzen zurückliefern. In diesem Fall wird nicht der gefundene Wert einer, sondern es wird eine Referenz auf diese spezielle Variable zurückgeliefert. Dies führt dazu, daß nun ein Funktionsaufruf auf der linken Seite wie auch auf der rechten Seite in einer Zuweisung stehen kann.

Beispiel:

```
#include <iostream.h>

 int x=5, y=6;
int &test (void)
{
 if (x>y)
  return x;
 else
  return y;
}
void main (void)
{

 cout << "Adresse von y                              : " << &y<<endl;
 cout << "Zurückgeliefertes Objekt                   : " <<&test()<<endl;
 cout << "y vor Aufruf der Funktion test             : " << y<<endl;
 test()=7;
 cout << "y nach Aufruf der Funktion test            : " << y << endl;
}
```

Ausgabe:

```
Adresse von y                              : 0x7e9900ac
Zurückgeliefertes Objekt                   : 0x7e9900ac
y vor Aufruf der Funktion test             : 6
y nach Aufruf der Funktion test            : 7
```

19.1 Problem Referenz - Zeiger

Referenzen sind zwar im Umgang mit Funktionen handlicher, haben aber das Problem, daß man beim Funktionsaufruf nicht direkt mitbekommt, daß eine Wertänderung möglich ist.

Beispiel:

```
Void inc (int &i)
{...}
inc (a);    //hier wird nicht deutlich, daß es sich um eine Referenz
// handelt, die auch den Wert von a verändern kann

void inc (int *a)
{...}
inc (&a);  // hier handelt es sich um einen Zeiger; es muß damit gerechnet
//  werden, daß der Wert von a durch die Funktion geändert wird
```

Konstante Referenz

Es ist eventuell notwendig eine Parametervariable als Referenz zu übergeben. Diese soll jedoch nicht in ihrem Wert verändert werden. Es besteht also das gleiche Problem, als wenn man einen Zeiger übergibt. Für dieses Problem gibt es deshalb auch die gleiche Lösung: man deklariert die übergebene Referenz als konstant.

Beispiel:

```
void test (const char &text);
```

20 Selbstreferenz – this

```
class Datum
{
  int t, m, j;

  public:
  Datum (int tt=0, int , mm=0, int jj=0);
  void addiere_Tag           (int plus_t);
  void addiere_Jahr          (int plus_j);
  void addiere_Monat    (int plus_m);
};

void Datum::addiere_Jahr (int plus_j)
{
  if  (t==29 && m==2 && !=schaltjahr(j+plus_j))
  {
    t=1, m=3;
  }
  j+=plus_j;
}


Datum j1(1,1,1998)   // das Datum j1 wird mit dem Datum 1.1.1998
                     // initialisiert
```

Wenn man nun das Datum von j1 jeweils um einen Tag, Monat und Jahr erhöhen möchte, so daß das Datum 2.2.1999 herauskommt, so müßte man folgende Funktionen anwenden:

```
j1.addiere_Tag(1);
j1.addiere_Jahr(1);
j1.addiere_Monat(1);
```

Diese Schreibweise ist sehr umständlich. Zusätzlich beziehen sich alle Memberfunktionen der Klasse Datum auf das Objekt j1. Außerdem liefern die verwendeten Funktionen keine Werte zurück.
Es wäre nun schön, wenn man folgendes schreiben könnte:

```
j1.addiere_Tag(1).addiere_Jahr(1).addiere_Monat(2);
```

Dies kann man mit der sogenannten Selbstreferenz this verwirklichen. Dies bedeutet, daß die Funktion addiere_Jahr zurückliefert, auf welches Objekt die nächste Funktion angewendet werden soll. In diesem Fall liefert die Funktion den Wert j1.
Man kann sich die Abarbeitung so vorstellen:

```
J1.addiere_Tag(1).addiere_Jahr(1).addiere_Monat(1);
                                          //addiere_Tag wird ausgeführt
```

J1.addiere_Tag(1) liefert den Wert j1 zurück, deshalb kann man folgendes schreiben:

```
J1.addiere_Jahr(1).addiere_Monat(1);     //addiere_Jahr wird ausgeführt
```

J1.addiere_Jahr liefert ebenfalls den Wert j1:

```
J1.addiere_Monat(1); // zum Schluß wird nun addiere_Monat ausgeführt
```

Damit man auch folgende Schreibweise anwenden kann, müssen die einzelnen Funktion eine Referenz auf das aktuelle Element zurückliefern. Dies wird im Programm folgenderweise realisiert:

```
Class Datum
{
  //...
 Datum &addiere_Jahr (int plus_j);
  //...
}

Datum &Datum::addiere_Jahr (int plus_j)
{
  if  (t==29 && m==2 && !=schaltjahr(j+plus_j))
  {
    t=1, m=3;
  }
  j+=plus_j;
  return *this;
}
```

Eigenschaften von this:

This zeigt immer auf das aktuelle Objekt.
Man könnte die letzte Funktion auch anders schreiben:

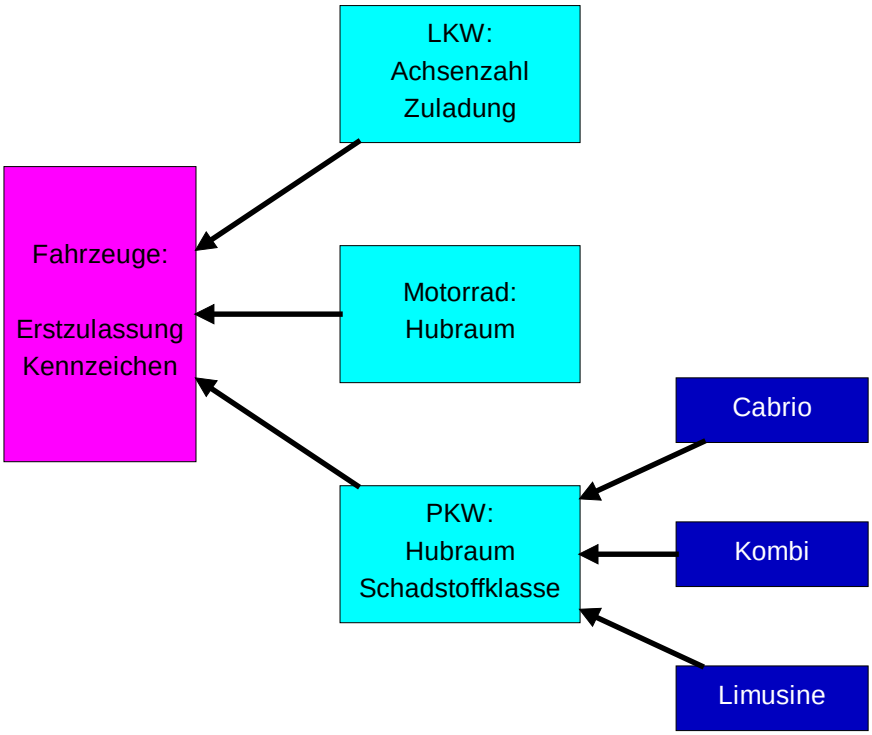
```
Datum &Datum::addiere_Jahr (int plus_j)
{
  if  (this->t==29 && this->m==2 && !=schaltjahr(this->j+plus_j))
  {
    this->t=1;
    this->m=3;
  }
  this->j+=plus_j;
  return *this;
}
```

21 Ableiten von Klassen

21.1 Gedanke

Warum Ableiten von Klassen?
Ganz einfach, es gibt verschiedene Objekte, die Gemeinsamkeiten haben. Um diese Gemeinsamkeit auszudrücken, schreibt man diese in eine sog. Basisklasse und die spezifischen Eigenschaften in sog. abgeleitete Klassen.

Beispiel:



Jedes Fahrzeug besitzt eine Erstzulassung und ein Kennzeichen. Diese müssen in Variablen abgespeichert werden. Zusätzlich hat jeder Fahrzeugtyp noch andere wichtige Daten, die auch abgespeichert werden müssen.

21.2 Public, Protected, Private

21.2.1 Zugriffsarten

Es gibt nun 3 verschiedene Zugriffsspezifizierer:

Private:

Abgeleitete Klassen können auf die Membervariablen bzw. –funktionen der Hauptklasse nicht zugreifen.
Wird eine Hauptklasse mit private abgeleitet, so bewirkt dies, daß alle Memberfunktionen und –variablen in der abgeleiteten Klasse privat sind, auch wenn die Funktionen und Variablen in der Hauptklasse protetected bzw. public waren.

Protected:

Abgeleitete Klassen können auf die Funktionen und Variablen der Klasse, die als protected deklariert wurden zugreifen, jedoch ist kein Zugriff von Außen möglich.
Bei einer protected Ableitung werden alle public Variablen und Funktionen protected, der restliche Zugriffsschutz bleibt gleich.

Public:

In diesem Fall kann jeder auf die Funktionen und Variablen der Klasse zugreifen.
Wird eine Klasse public abgeleitet, so verändert sich nichts am Zugriffsschutz.

Zugriffstyp	Eigene Klasse	Abgeleitete Klasse	Sonstige
Private	Ja	Nein	Nein
protected	Ja	Ja	Nein
public	Ja	Ja	Ja

21.2.2 Vererbungstyp

Basisklasse	Public-Element wird	Protected-Element wird	Private-Element wird
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Private	Private	Private

Wird bei der Vererbung kein Vererbungstyp angeben, so wird private vererbt.

Beispiele:

```
Class X
{
    public int a;

    //...
};

class y1: public X {};
class y2:protetected X { };
class y3:private X{};

void test (y1 &r1, y2 &r2, y3 &r3)
{
  r1.a=7;   //OK
  r2.a=7;   //Fehler, X wurde protected vererbt ⇒ a wurde protected
  r3.a=7    // Fehler, X wurde private vererbt ⇒ a wurde private
}
```

21.3 Konstruktion

Beispiel:

```
class Hund
{
  // Elementfunktion, Membervariablen;
};

Class Dackel:public Hund
{
  // Elementfunktionen, Membervariablen
};
```

Es ist nun Dackel von Hund abgeleitet.

21.3.1 Konstruktor

Konstruktor nicht innerhalb der Klasse definiert :

Der Konstruktor der Basisklasse wird ganz normal definiert.
Bei der Definition des Konstruktor der abgeleiteten Klasse muß man jedoch aufpassen: Der Konstruktor der Basisklasse wird selbständig aufgerufen, bevor die restlichen Anweisungen für den Konstruktor der abgeleiteten Klasse bearbeitet werden. Wenn man selber noch den Konstruktor der Basisklasse aufruft, so kommt es zu Fehlern.

Beispiel:

```
class C_Autos
{
  private:

  int     m_erstzulassung;
  char    m_kennzeichen[10];

  public:
                  C_Autos  (void);
};


class C_PKW:public C_Autos
{
  private:
  int     m_hubraum;
  int     m_leistung;
  int     m_schadstoff;

  public:
          C_PKW     (void);
};

C_Autos::C_Autos (void)
{
  cout << "Kennzeichen: ";
  cin >> m_kennzeichen;
  cout << "Erstzulassung: ";
  cin >> m_erstzulassung;
}

C_PKW::C_PKW(void)
```

```
{
   // Es wäre hier falsch, noch einmal den Basiskonstruktor aufzurufen
  cout << "Hubraum   : ";
  cin >> m_hubraum;
  cout << "Leistung  : ";
  cin >> m_leistung;
  cout << "Schadstoffklasse: [0..2]: ";
  cin >> m_schadstoff;
}
```

Konstruktor innerhalb der Klasse definiert (mit Parametern):

```
Class PKW
{
  PKW (char *kfz, int j, int h, int l, short s=0):
  Fahrzeug(kfz,j), m_hubraum(h), m_leistung(l), m_schadstofftyp(s) {};
};
```

21.3.2 Virtuelle Funktionen

Es kann nun notwendig sein, daß man z.B. für die Basisklasse wie auch für die abgeleitete Klasse eine print-Funktion benötigt. Um diese Funktion überschreiben zu können und der Compiler dann später beim Aufruf auch wieder die richtige findet, schreibt man vor dem Funktionsname virtual.
Möchte man, daß nur in den abgeleiteten Klassen eine Funktion print stehen muß, dann schreibt man hinter der Deklaration =0. Dies bedeutet einerseits, daß es sich um eine abstrakte Basisklasse handelt, d.h. es kann keine Instanz von Ihnen geben. Zum anderen bedeutet dies, daß jede abgeleitete Klasse eine solche Funktion (z.B. print) haben muß.

Beispiel:

```
Class Fahrzeug
{
   virtual void print (void);
   virtual void steuer(void) = 0;
//für jedes Fahrzeug muß man Steuern zahlen, deren Höhe jedoch vom
//Fahrzeugtyp abhängt
};

Class PKW
{
  void print (void);
  void steuer(void);
};

class LKW
{
  void steuer (void);   //OK, Funktion print muß nicht vorhanden sein
};

class Motorrad
{
  void print (void);    //Fehler es fehlt die Funktion steuer;
};
```

```
void main (void)
{
  Fahrzeug Auto;  //Falsch, abstrakte Klasse
  PKW meins;      // OK
  LKW seins;      // OK
}
```

21.4 Destruktor

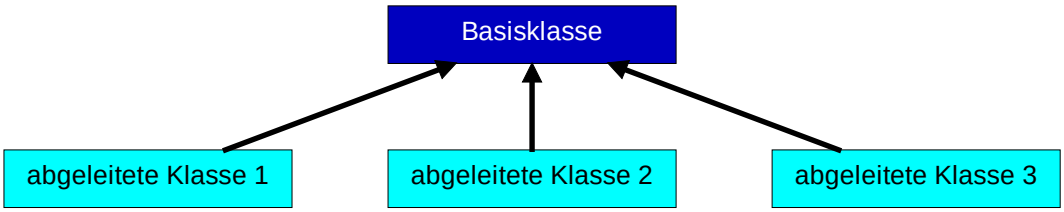
Destruktoren sollten als virtual definiert sein, d.h. sie werden in den abgeleiteten Klassen überdefiniert. Im Gegensatz zur Konstruktion, bei der erst der Konstruktor der Basisklasse und dann der Konstruktor der abgeleiteten Klasse aufgerufen wird, sollte bei der Destruktion zuerst der Konstruktor der abgeleiteten Klasse und dann der Destruktor der Basisklasse aufgerufen werden.

Was passiert wenn man virtual vergißt?

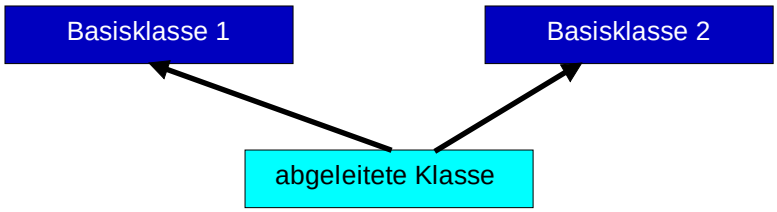
Es wird nur die Basisklasse zerstört.
Mit virtual wird zuerst die abgeleitete Klasse zerstört und dann die Basisklasse.

21.5 Vererbungshierarchie

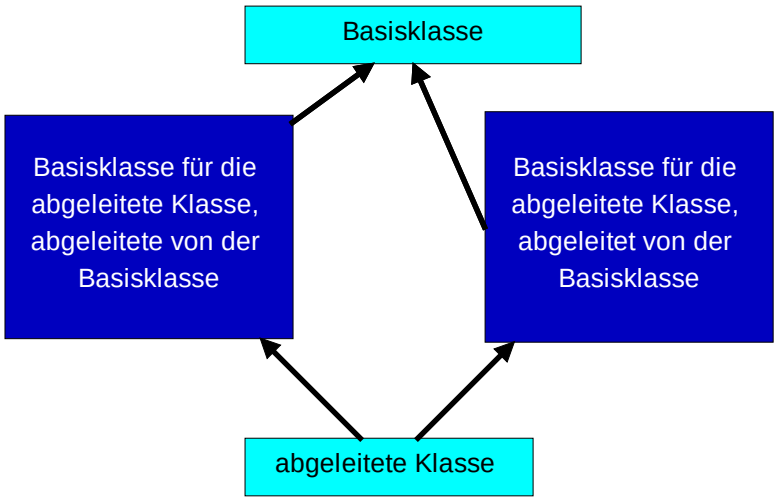
Um darzustellen, welche Klasse von wem abgeleitet ist, gibt es eine Vererbungshierarchie.



Einfachvererbung



Mehrfachvererbung



21.6 Aufgabe

Hier kommt noch einmal eine Aufgabe. Diese wurde zwar in den Beispielen schon teilweise behandelt, hier kommt sie ganz:

Organisation von Fahrzeugdaten
Gesucht ist für eine Stelle wie z.B. das Kraftfahrtbundesamt eine neue, übersichtliche und leicht erweiterbare Organisation der verschiedenen Fahrzeugdaten.
Für die unterschiedlichen Fahrzeugarten (PKWs, Motorrädern und LKWs) sind jeweils unterschiedliche Daten zu erfassen. Auch die Weiterbearbeitung der Daten (z.B. Steuerberechnung) erfolgt nach unterschiedlichen Kriterien. Zu erfassende und zu speichernde Daten:

-- Für alle Fahrzeugtypen Kennzeichen, Jahr der Erstzulassung
-- PKWs, zusätzlich: Hubraum, Leistung, Schadstoffklasse [0-schadstoffarm, 1-normal, 2-Diesel]
-- Motorräder, zusätzlich: Hubraum
-- LKWs, zusätzlich: Anzahl der Achsen, Zuladung in t

Entgegen dem alten System möchte man sich bei der reinen Verwaltung der Daten (z.B. Datenablage, Suche nach Fahrzeugen gemäß Kennzeichen, etc. Steuerberechnung und TÜV-Anmahnung) nicht mehr mit unterschiedlichen Fahrzeugtypen "herumschlagen".
Implementieren Sie oben gefordertes System mit Hilfe von abgeleiteten C++-Klassen. Bieten Sie für die Auswertung/Weiterbearbeitung der Daten folgende Funktionen an:

-- Suchen nach einem Fahrzeug anhand dem eingegebenen Kennzeichen und Ausgeben aller betreffenden Fahrzeugdaten
-- Ausgeben aller Daten von allen gespeicherten Fahrzeugdaten
-- Berechnung der Steuerschuld für 1) ein angegebenes Kennzeichen und 2) für alle gespeicherten Fahrzeuge gemäß folgenden Formeln:
-- PKW: (Hubraum+99) / 100 * 10 DM * (Schadstoffklasse+1)
-- Motorrad: (Hubraum+99) / 100 * 20 DM
-- LKWs: Zuladung*100 DM/t
-- Auflisten der TÜV-Fälligkeit für alle Fahrzeuge in einem anzugebenden Jahr.

(Sie können hier davon ausgehen, daß die erste TÜV-Untersuchung nach drei Jahren zu erfolgen hat und anschließend alle zwei Jahre eine Untersuchung fällig ist)

Hier kommt mein Programm. Es ist diesmal in Headerdatei und Programm aufgeiteilt:

KFZ.h

```
#ifndef _KFZ
#define _KFZ

class C_Autos
{
  private:

  int     m_erstzulassung;
  char    m_kennzeichen[10];

  protected:
  float   m_steuer;


//  enum    m_tuev  {nicht_ok,ok};
  int       m_tuev;
  public:
                  C_Autos  (void);
//  virtual        ~C_Autos (void);
  virtual void    ausgabe  (void);
  virtual void    steuer   (void)=0;
          void    tuev     (int jahr);
          void    ausgabe_2(void);
          int     vergleich(C_Autos *obj);
          int     vergleich(char *such_name);
};


class C_PKW:public C_Autos
{
```

```
  private:
  int     m_hubraum;
  int     m_leistung;
  int     m_schadstoff;

  public:
          C_PKW     (void);
          ~C_PKW    (void);
  void    ausgabe   (void);
  void    steuer    (void);
};

class C_Motorrad:public C_Autos
{
   private:
   int    m_hubraum;

   public:
          C_Motorrad  (void);
          ~C_Motorrad (void);
   void   ausgabe     (void);
   void   steuer      (void);
};

class C_LKW:public C_Autos
{
   private:
   int    m_achsen;
   int    m_zuladung;

   public:
          C_LKW     (void);
          ~C_LKW    (void);
   void   ausgabe   (void);
   void   steuer    (void);
};


#endif
```

KFZ.cpp

```
#include <iostream.h>
#include <string.h>
#include "kfz.h"

C_Autos::C_Autos (void)
{
  cout << "Kennzeichen: ";
  cin >> m_kennzeichen;
  cout << "Erstzulassung: ";
  cin >> m_erstzulassung;
}

void C_Autos::ausgabe (void)
{
  cout << "Kennzeichen  : " << m_kennzeichen << endl;
  cout << "Erstzulassung: " << m_erstzulassung << endl;
}

void C_Autos::ausgabe_2 (void)
{
  cout << "T□v          : " ;
```

```
  if (m_tuev==0)
   cout << "nicht in Ordnung" << endl;
  if (m_tuev==1)
   cout << "in Ordnung" << endl;
  cout << "Steuer       : " << m_steuer << " DM" <<endl;
}


void C_Autos::tuev (int jahr)
{
 int differenz;
 differenz=jahr-m_erstzulassung;
 if ((differenz%2==0) || (differenz==1))
    m_tuev=1;
 else
    m_tuev=0;
}

int C_Autos::vergleich(C_Autos *obj)
{
  char kenn_o[20],kenn_v[20];

  strcpy(kenn_o,m_kennzeichen);

  strcpy(kenn_v,obj->m_kennzeichen);

  if ( strcmp(kenn_o,kenn_v) == 0)
    return 0;
  else

    if (strcmp (kenn_o,kenn_v) >0)
      return 1;
    else
      return -1;

}

C_Autos::vergleich(char *such_name)
{
  char kenn_o[20],kenn_v[20];

  strcpy(kenn_o,m_kennzeichen);

  strcpy(kenn_v,such_name);

  if ( strcmp(kenn_o,kenn_v) == 0)
    return 0;
  else

    if (strcmp (kenn_o,kenn_v) >0)
      return 1;
    else
      return -1;

}

C_PKW::C_PKW(void)
{
  cout << "Hubraum   : ";
  cin >> m_hubraum;
  cout << "Leistung  : ";
  cin >> m_leistung;
  cout << "Schadstoffklasse: [0..2]: ";
  cin >> m_schadstoff;
  steuer();
```

```
  m_tuev=1;
}

void C_PKW::ausgabe (void)
{
  static char *klassen[3]={"Sauber", "Dreckschleuder", "Diesel"};
  C_Autos::ausgabe();
  cout << "Fahrzeugtyp  : PKW" << endl;
  cout << "Hubraum      : "<< m_hubraum << endl;
  cout << "Leistung     : "<< m_leistung << endl;
  cout << "Schadstofftyp: "<< klassen[m_schadstoff] << endl;
  ausgabe_2();
}

void C_PKW::steuer (void)
{
 m_steuer=(m_hubraum+99) / 100 * 10 * (m_schadstoff+1);
}

C_LKW::C_LKW (void)
{
  cout << "Achsenzahl: ";
  cin >> m_achsen;
  cout << "Zuladung  : ";
  cin >> m_zuladung;
  steuer();
  m_tuev=1;
}


void C_LKW::ausgabe(void)
{
  C_Autos::ausgabe();
  cout << "Fahrzeugtyp   : LKW" << endl;
  cout << "Achsenzahl    : " << m_achsen << endl;
  cout << "Zuladung      : " << m_zuladung << endl;
  ausgabe_2();
}

void C_LKW::steuer(void)
{
 if (m_zuladung>1000)
  m_steuer=m_zuladung*0.1;
 else
  m_steuer=m_zuladung*100;
}

C_Motorrad::C_Motorrad (void)
{
  cout << "Hubraum : " ;
  cin >>m_hubraum;
  steuer();
  m_tuev=1;
}

void C_Motorrad::ausgabe(void)
{
  C_Autos::ausgabe();
  cout << "Fahrzeugtyp   : Motorrad" << endl;
  cout << "Hubraum       : " << m_hubraum << endl;
  ausgabe_2();
}

void C_Motorrad::steuer(void)
{
```

```
  m_steuer=(m_hubraum+99)/100*20;
}
```

Baum.h

```
#ifndef _Baum
#define _Baum

#include "kfz.h"
class C_Baum
{
 /* private Variablen */
     struct Baum
      {
       C_Autos  *daten;
       Baum  *rechts;
       Baum  *links;
      };
 Baum *m_wurzel, *m_neu, *m_suchadresse, *m_vorher;
 int gefunden;
 /*  private Funktionen */
 void  m_einfuegen (Baum *m_aktuell);
 void  m_teil_ausgabe(Baum *m_aktuell);
// void  m_freigabe (Baum *m_aktuell);
 void  m_anfsuchen (Baum *m_aktuell, char *such_name);
 void  m_tuev_erneuern (Baum *m_aktuell, int jahr);

 public:
       C_Baum (void);
//       ~C_Baum (void);
 void  m_eingabe(C_Autos *obj);
 void  m_suchen (void);
 void  m_ausgabe (void);
 void  m_tuev (void);
// void  m_loeschen (void);
};

#endif
```

Baum.cpp:

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

#include "kfz.h"
#include "baum.h"

C_Baum::C_Baum (void)
{
 m_wurzel=NULL;
 m_wurzel->daten=NULL;
 m_wurzel->links=NULL;
 m_wurzel->rechts=NULL;
}
```

```
/*
C_Baum::~C_Baum (void)
{
 m_vorher=m_wurzel;
 m_freigabe(m_wurzel);

}

void  C_Baum::m_freigabe (Baum *m_aktuell)
{
 if (m_aktuell->links!=NULL)
 {
  m_vorher=m_aktuell;
  m_freigabe(m_aktuell->links);
 }
 if (m_aktuell->rechts!=NULL)
 {
  m_vorher=m_aktuell;
  m_freigabe(m_aktuell->rechts);
 }
 if ((m_aktuell->rechts==NULL) && (m_aktuell->links==NULL))
   delete m_aktuell;

}
*/



void C_Baum::m_einfuegen (Baum *m_aktuell)
{

 if (m_aktuell==NULL)
 {//komplett neues element
    m_wurzel=m_neu;
 }
 else
 {
   if ( m_aktuell->daten->vergleich(m_neu->daten)  >0 )
   {
     if (m_aktuell->rechts == NULL)
     {
        m_aktuell->rechts=m_neu;
     }
     else
        m_einfuegen(m_aktuell->rechts);
   }
   if (m_aktuell->daten->vergleich(m_neu->daten)<0)
   {
        if (m_aktuell->links==NULL)
        {
           m_aktuell->links=m_neu;
        }
        else
           m_einfuegen(m_aktuell->links);
   }
   if (m_aktuell->daten->vergleich(m_neu->daten)==0)
    cout << "Dieses Kennzeichen wurde schon einmal vergeben!!!" << endl;
  }
}

void C_Baum::m_eingabe(C_Autos *obj)
{
 m_neu=new Baum;
```

```
 m_neu->daten=obj; //!!!!!!!!!!

 m_neu->rechts=NULL;
 m_neu->links=NULL;
 m_einfuegen(m_wurzel);
}


void C_Baum::m_suchen(void)
{
 char such_name[10];
 gefunden=0;
 cout << "Zu suchendes Kennzeichen: ";
 cin >> such_name;
 m_vorher=m_wurzel;
 m_anfsuchen(m_wurzel,such_name);
}
void C_Baum::m_tuev(void)
{
  int jahr;
  cout << "Aktuelles Jahr: ";
  cin >> jahr;
  m_tuev_erneuern(m_wurzel, jahr);
}

void C_Baum::m_tuev_erneuern(Baum *m_aktuell, int jahr)
{
  if (m_aktuell->links!=NULL)
   m_teil_ausgabe (m_aktuell->links);
  m_aktuell->daten->tuev(jahr);
  m_aktuell->daten->ausgabe();

 if (m_aktuell->rechts!=NULL)
    m_teil_ausgabe(m_aktuell->rechts);
}
void C_Baum::m_anfsuchen(Baum *m_aktuelle, char *such_name)
{
 if ( (m_aktuelle->daten->vergleich(such_name) >0) && (m_aktuelle-
>rechts!=NULL))
 {
  m_vorher=m_aktuelle;
  m_anfsuchen(m_aktuelle->rechts, such_name);
 }
  if ((m_aktuelle->daten->vergleich(such_name)  < 0) && (m_aktuelle-
>links!=NULL))
  {
     m_vorher=m_aktuelle;
     m_anfsuchen(m_aktuelle->links, such_name);
  }
 if (m_aktuelle->daten->vergleich(such_name)  ==0 )
    gefunden=1;
  if (gefunden==1)
   {
    m_aktuelle->daten->ausgabe();
    gefunden=2;
    m_suchadresse=m_aktuelle;
   }
  if (gefunden==0)
  {
   cout << "String nicht gefunden." <<endl;
   gefunden=2;
   m_suchadresse=NULL;
  }
}
```

```
/*
void C_Baum::m_loeschen (void)
{
 Baum * loesch_r, *loesch_l;
 int wurzel=0;
 m_suchen();
 if(m_vorher->rechts==m_suchadresse)
  m_vorher->rechts=NULL;
 if (m_vorher->links==m_suchadresse)
  m_vorher->links=NULL;
 loesch_r=m_suchadresse->rechts;
 loesch_l=m_suchadresse->links;
 if (m_suchadresse==m_wurzel)
 {
  wurzel=1;
  if (loesch_r!=NULL)
  {
   m_wurzel=loesch_r;
   delete m_suchadresse;
   if (loesch_l!=NULL)
   {
    m_neu=loesch_l;
    m_einfuegen(m_wurzel);
   }
  }
  else
  {
   m_wurzel=loesch_l;
   delete m_suchadresse;
   if (loesch_r!=NULL)
   {
    m_neu=loesch_r;
    m_einfuegen(m_wurzel);
   }
  }
 }
 if (wurzel==0)
 {
  delete m_suchadresse;
  m_neu=loesch_r;
  if (loesch_r!=NULL)
     m_einfuegen(m_wurzel);
  m_neu=loesch_l;
  if (loesch_l!=NULL)
     m_einfuegen(m_wurzel);
 }
}
*/

void C_Baum::m_teil_ausgabe(Baum *m_aktuell)
{
 if (m_aktuell->links!=NULL)
   m_teil_ausgabe (m_aktuell->links);

  m_aktuell->daten->ausgabe();

 if (m_aktuell->rechts!=NULL)
    m_teil_ausgabe(m_aktuell->rechts);
}

void C_Baum::m_ausgabe (void)
{
 char weiter;
 m_teil_ausgabe(m_wurzel);
 cout << "\n Weiter mit einer Taste" << endl;
```

```
 cin >> weiter;
}
```

22 Ausnahmebehandlungen

In jedem Programm können Fehler auftreten. Damit man das Programm nicht neu starten muß und der Benutzer möglichst selten mit Fehler umgehen muß, gibt es in C++ das Konzept, daß man Störfälle, die in einem Programmablauf auftreten, abfängt und bearbeitet und dadurch eventuell behebt.

22.1.1 Aspekte

- Aufrufende Routine kann den Fehler noch nicht vorhersehen (z.B. fopen)
- Aufgerufene Routine kann den Fehler selbst nicht behandeln
- "Begleitende" Fehlerbehandlung macht den Code unhandlich, unelegant, schwer verständlich; man möchte i.w. eigentlich den Hauptfall betrachten
- Eine "passende" Stelle im Programm soll sich um Ausnahmen kümmern

22.1.2 Idee

Für die Ausnahmebehandlung benötigt man im Prinzip folgende Befehle:

- try: Man versucht Befehle auszuführen
- throw: Aufgerufene Routine erzeugt einen Fehlerfall
- catch: falls ein Fehler aufgetreten ist wird er aufgefangen und bearbeitet

22.1.3 Vorteile

- "Saubere" Programmierung des Gut/Schlecht-Falls in der aufgerufenen Routine
- Keine Überlastung der Rückgabewerte durch die Kodierng eines Schlecht-Zustands
- Fehlerbehandlung muß nicht in der direkt übergeordneten Funktion durchgeführt wären, sondern kann an der passendsten Stelle geschehen

22.2 Vorgehen

22.2.1 try:

Mit Hilfe von try wird das Auffangen von Fehler ermöglicht. Mit diesem Befehl wird dem Compiler mitgeteilt, daß in diesem Programmteil eine Ausnahme auftreten kann. Dieser "kritische" Bereich wird mit einem try begonnen und steht in geschweiften Klammern.

22.2.2 catch

Catch fängt den Fehler wieder auf.

22.2.3 throw

- bricht die Abarbeitung der Routine an dieser Stelle ab
- der passende Ausnahme-Handler wird gesucht
- evtl. "dazwischenliegende" Routinen werden übersprungen
- die Ausführung wird direkt im Handler fortgesetzt

Achtung: throw ist kein einfacher Rücksprung:

- Alle zwischen throw und Ausnahme-Handler liegenden Daten auf dem Aufruf-Stack müssen entfernt werden
- Returnwerte werden nicht erzeugt
- Strukturierte Programmierung ????

Die Aunsahmetypen werden mit in der Klasse als Elemente definiert.

Beispiel:

```
Main ()
{ ...
   try
   {
     switch (Cmd)
     {
       case ‘i‘:  cin >> Name;
                     fifo.enqueue(Name);
                  break;
       case ‘o‘: cout << "Eintrag: "<<fifo.dequeue()<<endl;
                  break;
     }
   }
  catch (CFifo::Overflow)
  {
     cerr << "Hey, kein Platz mehr verfuegbar !" << endl;
  }
  catch (CFifo::Underflow)
  {
     cerr << "Woher soll ich bitte die Daten nehmen??? " << endl;
  }
}
```

23 Friends

Normalerweise können nur Elementfunktionen auf private Variablen und Funktionen zugreifen. Es kann nun jedoch erforderlich sein (einfacher), daß auch eine andere Funktion auf die privaten Variablen und Funktionen ausnahmsweise zugreifen darf, ohne daß sie zu dieser Klasse gehört. Eine solche Funktion wird als Friend-Funktion bezeichnet.

Anwendungsbeispiel:

Eine Funktion die auf private Variablen zwei verschiedener Klassen zugreifen muß.

24 Templates

Templates dienen zur generischen Programmierung. Zum Beispiel möchte man nicht für jeden Typen einen eigenen Stack oder Fifo implementieren, sondern es wäre praktisch, wenn man eine Klasse schreibt und dann den Datentyp (int, float,...) angibt, der dort eingespeichert werden soll.
Man nennt dann so etwas ein Template.

Beispiel:

```
template <class T> class C_Ring
{
  private:
  T         *puffer;
  int       rein;
  int       raus;

  public:
            C_Ring    (int s=10);
            ~C_Ring   (void);
  void      eingabe   (T daten);
  T         ausgabe   (void);
  T         anzeige   (void);
  void      operator= (const char *neu);
};

template <class T> C_Ring<T>::C_Ring (int s=10)
{
  puffer=new T[s];
  raus=0;
  rein=s-1;
}

template <class T> C_Ring<T>::~C_Ring (void)
{
  delete puffer [];
}

template <class T> void C_Ring<T>::eingabe (T daten)
{
  if (raus-rein == 1)
//   overflow ;
  else
  {
    puffer[rein]=daten;
    if (rein<s-1)
     rein++;
    else
     rein=0;
  }
}
```

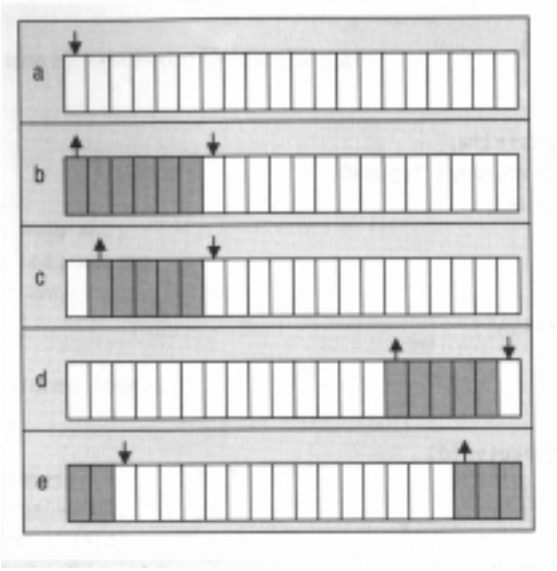
...
C_Ring<int> integer_Ring; //Ring für integer Werte

Alle Stellen, die farbig (rosa) hervorgehoben sind, werden durch die Angabe int ersetzt (oder andere Variablentypen).

Template gilt nur innerhalb der jeweiligen geschweiften Klammern. Außerhalb dieser Klammern muß man dem Computer wieder neu sagen, daß es sich um ein Template handelt. Deshalb muß auch bei der Funktionsdeklaration immer vorher Template stehen.
Der Klassenname besteht bei zusätzlich aus den hier T in spitzen Klammern.

24.1 Aufgabe

Programmieren Sie mit Hilfe von Templates einen Ring. Diese Datenstruktur ist ein Mischung von Stack und Fifo.
Diese Bild soll Ihnen bei der Implementierung helfen:



Mein Programm:

```
#include <iostream.h>
#include <math.h>
template <class T> class C_Ring
{
  T        *puffer;
  int      rein; //zeigt auf das GERADE eingefuegte element
  int      raus; //zeigt auf das zu lesende element
  int      max;
  int      flag;
  int      Anzahl;
  public:
            C_Ring   (int s=10);
           ~C_Ring   (void);
  void     eingabe   (T daten);
  void     ausgabe   (void);
  void     print     ();
};


template <class T> C_Ring<T>::C_Ring (int s=10)
{
  puffer=new T[s];
  raus=0;
  rein=0;
  flag=0;
  max=s-1;
```

```
  Anzahl=0;
  for (int i=0; i<=max; i++)
   puffer[i]=0;
}



template <class T> C_Ring<T>::~C_Ring (void)
{
  delete  [] puffer;
}



template <class T> void C_Ring<T>::eingabe (T daten)
{//push
  int hier=0;
  if (Anzahl==0)
  {
   puffer[0]=daten;
   rein=1;
   flag=0;
   Anzahl++;
  }
  else
  {
   if (Anzahl==max+1)
    cout << "Voll. Nix geht mehr" <<endl;
   else
   {
    if ((rein==max+1) && (raus!=0))
    {
     puffer[0]=daten;
     rein=1;
     flag=1;
     hier=1;
     Anzahl++;
    }
    if ((rein<=max) && (hier==0))
    {
     cout << "Rein: "<<rein<<endl;
     puffer[rein]=daten;
     rein++;
     Anzahl++;
     cout << "Rein: "<<rein<<endl;
    }
   }
 }
 cout << "Rein: " << rein << "Flag"<<flag;
 cout << "Raus: "<< raus << endl;

}

template <class T>void C_Ring<T>::print()
{
 int i;
 for (i=0; i<=max; i++)
   cout <<"    "<<i;
 cout<<endl;
 for (i=0; i<=max; i++)
   cout <<"    "<<puffer[i];
 cout<<endl<<"Rein:"<<rein<<"  Raus:"<<raus<<endl;
}
```

```
template <class T> void C_Ring<T>::ausgabe (void)
{//pop
  if (Anzahl==0)
     cout<<"Bin leer"<<endl;
  else
  {
   cout << "Element Nr." << raus<<" Daten:"<<puffer[raus]<<endl;
   puffer[raus]=0;
   if (raus<max)
    raus++;
   else
   {
    if (rein!=max+1)
     raus=0;
    else
    {
     raus=0;
     rein=0;
    }
   }
    cout << "Anzahl"<<Anzahl<<endl;
    cout << "Raus: "<<raus<<"Rein: "<<rein<<endl;
    Anzahl--;
    cout << "Anzahl"<<Anzahl<<endl;

  }
}

template <class T> void C_Ring<T>::print()
{
 int i;
 for (i=0; i<=max; i++)
  cout << "    "<<i;
 cout << endl;
 for (i=0; i<=max; i++)
  cout << "    "<<puffer[i];
 cout <<endl<<"Rein: "<<rein<<" Raus: "<<raus<<endl;
}

void main (void)
{
 C_Ring<int>    var1(5);
 int i;
 float f;

 do
 {
   cout<<"Cmd:"; cin>>i;
   switch (i)
   {
   case 1:cout<<"Zahl:"; cin>>f;
          var1.eingabe(f);
          break;
   case 2:
          var1.ausgabe();
          break;
   case 3:
          var1.print();
          break;
   }
 }while (i!=0);
}
```

25 Versionskontrollsysteme

25.1 Aspekte bei der Arbeit von mehreren Programmierern an einem Projekt

- Möglichst unabhängige Arbeit (1), jedoch auf dem gleichen Datenbestand (2)
  1 – lokale Projektkopie für jeden
  2 – zentrale Verwaltung der Daten
- Änderungen an einer Datei sollen erst dann an alle weitergehen, wenn die Routinen "laufen"
  Schwierig bei zentraler Verwaltung
- Es dürfen nicht zwei Personen gleichzeitig eine Datei (schreibend) editieren
  Zugriffsregelung
- Änderungen sollen (personell) erkennbar und auch (beliebig mehrstufig) wieder rückgängig gemacht werden können.

⇒ Versionsverwaltungssysteme

25.2 Idee

- Gemeinsame Datenbasis (zentral)
- Jeder Entwickler kann aus der Datenbasis eine Kopie (des gesamten Codebestands) erhalten
- Jeder arbeitet auf seiner Kopie (nur lesbar!)
- Sind Änderungen an einer Datei erforderlich, so wird diese ausgeliehen ("ausgecheckt") – diese Datei ist dann beschreibbar
- Hat die Datei nach den Programmtests einen befriedigen Stand erreicht, so kann sie wieder "eingecheckt" werden
- Konsequenz: Alle Entwickler erhalten beim nächsten Compilieren automatisch eine aktuelle (NUR-LESE-)Kopie der geänderten Datei (Realisierung über MAKE)
- Somit sind jetzt auch wieder bei allen Entwicklern die Dateien aktuell

25.3 Realisierung

- Änderungen an Dateien werden in der zentralen Datenbasis als Deltas zur vorangegangenen Version vermerkt

⇒ jede beliebige alte Version der Datei kann wiederhergestellt werden

⇒ jede einzelne Änderung kann nachvollzogen werden

26 Zum Schluß: Häufige Konventionen und optischer Aufbau

Man sollte den Quellcode auch optisch gut aufbereiten, da es dadurch einfacher wird ihn zu lesen. Für den Programmierer selbst mag das Lesen seines Quellcodes keine Probleme bereiten, aber wenn ein zweiter sich das Programm anschaut, dann kann es leicht zu welchen kommen.
In großen Projekten wird ein bestimmter optischer Aufbau eines Quellcodes zwischen den Mitgliedern des Programmierteams abgesprochen und sollten auch eingehalten werden.

26.1 Kommentare

Man soll Kommentare dort verwenden, wo
- sie sinnvoll sind
- sie das Programm beschreiben

Programme sollten auch aktuell sein.
Sobald ein Sachverhalt einfach und klar durch die Programmiersprache dargestellt werden kann, sollte man einen Kommentar weglassen.

Beispiele:

```
a=b+c; // a wird b+c zugewiesen
zaehler++;  //Inkrementiere zaehler
```

Man sollte zu jeder im Programm vorkommenden Funktion einen kleinen Kommentar schreiben, der beinhaltet:
- Aufgabe der Funktion
- letzte Änderung

Beispiel:

```
/*********************************************************************/
/* Funktion Berechnung: Sie berechnet die Zinsen für ein Kalenderjahr */
/*letzte Änderung: 11.3.98                                           */
/*********************************************************************/
```

26.2 Klammersetzung

Es ist empfehlenswert die Klammern in eine eigene Zeile zuschreiben. Der Code nach den Klammern sollte man einrücken. Code der in einer Schachtelungstiefe ist, sollte die gleiche Einrückung besitzen. Die öffnende und die schließende Klammer sollten den gleichen Einrückabstand besitzen. Bei vielen Verschachtelungen ist es hilfreich die einzelnen Klammern zu beschriften.

Beispiel – so nicht:

```
{ cout << "blablabla...";
       cout << "Ätsch";
      cout << "So nicht!!";
              }
```

Beispiel – so schon eher:

```
{
    cout << "blablabla...";
    cout << "Ätsch";
    cout << "So nicht!!";
}
```

26.3 Klassen

Um Programme übersichtlicher zu gestalten, gibt es ein paar Konventionen, die häufig verwendet werden:

- Klassen beginnen häufig mit einem "C": CBoot, C_Baum,...
- Membervariablen haben als Präfix "m_": m_kasse, m_haus