Vorlesung Programmieren in C und C++

Edwin Schicker, FH Regensburg

Inhaltsverzeichnis

1 Einleitung ... 2

2 Einführung in C ... 4

2.1 Das erste Programm ... 4
2.2 Allgemeine Informationen zu C ... 4
2.3 C und Pascal im Vergleich ... 5
2.4 Einfache Datentypen ... 6
2.5 Ausgabe mit printf ... 8
2.6 Ein- und Ausgabe mit den Ausgabeströmen in C++ ... 9
2.7 Funktionen ... 12
2.8 Eingabefunktionen in C und C++ ... 14
2.9 Weitere Kontrollstrukturen ... 17
2.10 Variablen, Typen und Operatoren ... 19
2.11 Weitere Möglichkeiten in C++ ... 25

3 Zeiger und Felder ... 26

3.1 Felder ... 26
3.2 Zeichenketten ... 30
3.3 Einschub: Konstanten und selbstdefinierte Datentypen ... 35
3.4 Mehrdimensionale Felder (Matrizen) ... 36
3.5 Dynamische Speicherverwaltung in C ... 39
3.6 Dynamische Speicherverwaltung in C++ ... 40
3.7 Zeigerfelder ... 41

4 Dateibearbeitung ... 43

4.1 Dateibearbeitung in C ... 43
4.2 Dateibearbeitung in C++ ... 46

5 Modulare Programmentwicklung in C ... 47

5.1 Gültigkeitsbereich von Bezeichnern in einer Datei ... 47
5.2 Speicherklassen ... 48
5.3 Getrennte Übersetzung ... 49
5.4 Headerdateien ... 52
5.5 Getrennte Übersetzung an einem Beispiel ... 53
5.6 Makefile ... 55

6 Datenstrukturen (struct und union) ... 57

6.1 Die Datenstruktur struct ... 57
6.2 Unions ... 59

7 Rekursion ... 60

8 Dynamische Datenstrukturen ... 61

8.1 Lineare Listen ... 61
8.2 Bäume ... 65

9 Sonstige Möglichkeiten mit C ... 68

9.1 Arbeiten mit Bitoperatoren (Wiederholung) ... 68
9.2 Präprozessoranweisungen ... 69
9.3 Makros ... 70
9.4 Aufzählungen ... 72
9.5 Zeiger auf Funktionen ... 72

10 Anwendungen und Vertiefung des Stoffs ... 74

10.1 Einige nützliche Funktionen in C ... 74
10.2 Curses ... 75

10.3 Hash-Verfahren 76
10.4 Fallstricke 79
10.5 Eine komplexe Funktionsdefinition 79

11 Objektorientierte Programmierung und Klassen 81
11.1 Objektorientierte Programmierung 81
11.2 Objektorientierte Implementierung eines Stacks 82
11.3 Friend- und Memberfunktionen 86
11.4 Das Schlüsselwort „this“ 87

12 Überladen von Operatoren und Funktionen 88
12.1 Überladen von Funktionen 88
12.2 Überladen von Operatoren 89
12.3 Standardfunktionen und –operatoren einer Klasse 90
12.4 Klassen und dynamisch reservierter Speicher 91
12.5 Klassen und statische Variable 93
12.6 Ein umfassendes Beispiel: Die Klasse der rationalen Zahlen 94
12.7 Überladen spezieller Operatoren 98

13 Weitere Möglichkeiten in C++ 99
13.1 Ausnahmebehandlung 99
13.2 Templates 101
13.3 Klassenhierarchie und Vererbung 103
13.4 Virtuelle Funktionen 105
13.5 Abstrakte Basisklassen 106
13.6 Konstanten innerhalb von Klassen 109

1 Einleitung

Prüfung: Keine Zulassungsvoraussetzungen, Prüfungen PGC und PGP zählen zusammen als eine Note. Zugelassen in der Prüfung sind dieses Skript, eigene Mitschriften zur Vorlesung, eigene Formelsammlungen, nicht jedoch Programmcode, der nicht in der Vorlesung besprochen wurde.

Übungen: Sie finden im Raum U511 statt. Geübt wird auf SUN-Workstations mit dem Betriebssystem SOLARIS (Unix Derivat). Insgesamt sind im Raum U511 12 Übungs-Workstations, 1 Server (+ weitere Workstations), 1 Nadeldrucker und 1 Postscript-Laserdrucker. Es stehen je nach Geschmack mehrere grafische Oberflächen zur Verfügung. Zum Schreiben von Programmen stehen mehrere Editoren zur Verfügung (vom einfachen vi bis zu grafischen Editoren).

Kennung: Zum Arbeiten mit den SUNs werden die NDS-Benutzerkennungen benötigt.

Übungsbetrieb: In den Vorlesungen werden Aufgaben gestellt, die in den Übungen besprochen werden. In den Übungsstunden stehe ich mit Rat und Tat zur Seite. Musterlösungen werden angeboten.

Vorlesungsinhalt: Dieses Skript.

Vorlesungsvoraussetzungen: Es wird das erste Semester in Programmieren (Pascal) vorausgesetzt, also im wesentlichen mein Pascal-Skript. Viele Teile (Rekursion, Zeiger) werden aber nochmals ausführlich wiederholt.

Literatur: Kernighan/Ritchie: Programmieren in C, Hanser Verlag (ca. 60 DM), zu empfehlen
Koenig: Der C Experte, Addison-Wesley (ca. 40 DM)
Stroustroup: Die C++ Programmiersprache, Addison-Wesley (ca. 70 DM)
Lippmann, C++ Primer, Addison-Wesley
Ladd: Applying C++, Prentice Hall
Ellis/Stroustroup: The Annotated C++ Reference Manual, Addison-Wesley (ca. 70 DM)
Heck: Standard Betriebssystem UNIX, eine Einführung, Rororo (ca. 17 DM)
im UNI-Rechenzentrum erhältlich: Skripte zu C, C++, UNIX (je ca. 6 DM)

UNIX: UNIX ist ein modernes Betriebssystem mit einer Vielzahl von Möglichkeiten. Es besitzt eine 32 bis 64 Bit Architektur (Adressen). Es ist ein Multi-User und Multi-Tasking System mit Paging und Zeitscheiben. Die Verdrängung ist preemptive.

UNIX entstand Anfang der 70er Jahre und ist damit ca 10 Jahre älter als MS-DOS. Glücklicherweise hat MS-DOS das für damalige Verhältnisse sehr moderne Konzept des Dateibaums von UNIX übernommen. Aus diesem Grund dürfte der Umstieg von MS-DOS auf UNIX nicht allzu schwer fallen.

In UNIX dient allerdings der Slash (‘/’) statt des Backslash (‘\’) als Trennzeichen zwischen Unterverzeichnissen. In UNIX dürfen Dateinamen beliebig lang sein. Auch besitzt der Punkt keine Sonderstellung wie in MS-DOS. Es gibt hier keine Extension, mehrere Punkte im Dateinamen sind daher zulässig. UNIX unterscheidet zwischen Groß- und Kleinschreibung. Auch kennt UNIX keine Geräte. Der Dateibaum beginnt immer bei der Root (/).

Einige UNIX-Kommandos:

```
ls    ≡ dir          cat       ≡ type
cd    ≡ cd           more      ≡ type (seitenweise)
mkdir ≡ md           pwd       ≡ akt. Verzeichnis ausgeben
rmdir ≡ rd           who       ≡ wer ist im System
rm    ≡ del          who am I  ≡ wer bin ich
cp    ≡ copy         lp        ≡ print
mv    ≡ ren          man       ≡ Manualseiten anzeigen
exit  ≡ Verlassen    yppasswd  ≡ Paßwort ändern
```

Insgesamt existieren mehr als 200 UNIX-Kommandos, etwa: chmod, chown, ps, kill, stty, lpstat, … Die meisten dieser Kommandos besitzen noch zahlreiche Parameter, die in UNIX in der Regel mit einem Minuszeichen (‘-’) eingeleitet werden. Weiter sind der Stern (‘*’) und das Fragezeichen (‘?’) Wildcardsymbole für Dateinamenersetzungen. Beispiel:

`ls -l *.c`	listet alle Dateien im ausführlichen Modus (l=long) auf, die mit ‘.c’ enden
`rm *`	löscht alle Dateien im aktuellen Verzeichnis (nicht *.* verwenden!)

Wie in DOS bedeuten der alleinstehende Punkt ‘.’ das aktuelle Verzeichnis und der doppelte Punkt ‘..’ das übergeordnete Verzeichnis.

Erstellen von C-Programmen:

- Die C-Programme werden mit einem beliebigen Editor erstellt und dann abgespeichert. Der C-Compiler erkennt nur Dateien, die mit ‘.c’ enden.
- Übersetzen mit: `cc dateiname.c` oder `make dateiname`
- Ablauf mit: `a.out` bzw. `dateiname`

Hier wird als Dateiname der Name des Programms ohne die Endung ‘.c’ verstanden. Ein Programmabsturz macht sich durch die Ausgabe der Meldung ‘core dumped’ bemerkbar und eventuellen weiteren Informationen. Die dabei erzeugte Datei ‘core’ dient zum Feststellen der Absturzursache. Solange wir deren Verwendung nicht kennen, sollte diese eventuell sehr große Datei gelöscht werden (*rm core*).

2 Einführung in C

2.1 Das erste Programm

Betrachten wir das erste C-Programm.

```
/* Das erste Programm */

#include <stdio.h>

int main ()
{
   printf ("Hello world\n");
   return 0;
}
```

Programm: 2-einf/prog1.c

Folgerungen:

- C erlaubt einen praktisch genauso freien Aufbau wie Pascal. Einrücken des Programmtextes oder Zeilenvorschübe sind überall erlaubt (natürlich nicht innerhalb von Bezeichnern und Namen!).
- Kommentare werden in /* … */ eingeschlossen.
- Einen Programmkopf wie Pascal besitzt C nicht. Statt dessen beginnt ein C-Programm immer mit der ersten Zeile der Funktion 'main'.
- Mit *#include* werden fremde Programmteile hinzugefügt. In diesem Fall ist es die Headerdatei der Ein-/Ausgabebibliothek *stdio.h* . Sie enthält alle Funktionsköpfe der Ein-/Ausgabe-Funktionen.
- In C schreibt man '{' und '}' statt 'Begin' und 'End'.
- Zeichenketten werden in C in " … " gesetzt, nicht in ' …' !
- *printf* ist die Standardausgabefunktion. 'Hello world' wird ausgegeben. Das Sonderzeichen '\n' steht für eine neue Zeile und darf an beliebiger Stelle in der Zeichenkette vorkommen, auch mehrfach.
- Ein C-Programm endet automatisch am Ende der Funktion *main* oder mit einem *return*. Im letzten Fall wird die angegebene Zahl an das aufrufende Betriebssystem zurückgeliefert. Dieses wertet diesen Rückgabewert aus und kann gegebenenfalls reagieren. Wird kein *Return* verwendet, so ist der Rückgabewert nicht definiert, ein schlechter Programmierstil liegt vor, wenn nicht sogar ein Fehler.
- Das Semikolon (';') besitzt in C eine geringfügig andere Bedeutung als in Pascal. In Pascal trennt das Semikolon zwei Anweisungen. In C beendet ein Semikolon eine Anweisung. Auf diesen feinen Unterschied kommen wir noch zu sprechen. Hinter *#include* steht kein Semikolon!
- C unterscheidet zwischen Groß- und Kleinschreibung! Wird *Main* statt *main* geschrieben, so liegt ein Syntaxfehler vor.

2.2 Allgemeine Informationen zu C

C entstand wie Pascal um 1970, und zwar als Systemprogrammiersprache für UNIX. Pascal war hingegen als eine einfache strukturierte Sprache geplant in Konkurrenz zu PL/1, zu dem wegen seiner Komplexität kaum Compiler zur Verfügung standen.

C wurde zunächst nur in Zusammenhang mit UNIX verwendet, fand ab etwa 1980 aber auch außerhalb von UNIX Verbreitung. C wurde 1988 normiert (ANSI-C). Im Gegensatz zu der sehr bescheidenen Norm von Pascal ist die ANSI-C Norm so umfangreich, daß C auch fast allen praktischen Anforderungen genügt. Der Vorteil ist, daß ein ANSI-C Programm auf alle Plattformen portiert werden kann. Jedes fehlerfreie ANSI-C Programm kann daher jederzeit von SUN auf MS-DOS portiert werden.

C wurde von Kernighan und Rhitchie in den Bell Laboratories entwickelt. C Kenntnisse sind heute stark gefragt.

```
Analyse der gefragten DV-Kenntnisse in Stellenanzeigen
Quelle: CDI Stellenmarktanalyse 1993


Betriebssysteme:                Programmiersprachen:
  UNIX          30,4%             C,C++           37,7%
  MS-DOS        18,5%             Cobol           27,8%
  IBM OS/400    16,2%             RPG              9,3%
  usw.                            usw.
```

Info: Gesuchte DV-Kenntnisse

Während Pascal eine kleine Sprache mit einer hervorragenden Struktur ist, ist der Sprachumfang von C etwas umfangreicher. C läßt sich ähnlich strukturiert programmieren, allerdings erlaubt C auch extrem viele „unsaubere" Programmiermöglichkeiten. Die Folge ist, daß in C syntaktisch extrem viel erlaubt ist. Während demnach Pascal viele Fehler schon beim Übersetzen findet, müssen diese in C meist sehr zeitraubend während der Laufzeit aufgespürt werden. Beispiele: in C ist fast jeder Variablentyp mit jedem anderen verträglich, in C existiert keine Bereichsüberprüfung.

2.3 C und Pascal im Vergleich

Vorab einige Meinungen zu C:

- C ist eine „Hacker"-Programmiersprache, die fast alle „Wünsche" erfüllt!
- Mit ANSI-C kann fast alles programmiert werden!
- C besitzt viele Fallen und Tücken (ich werde darauf gezielt aufmerksam machen)!
- C ist eine Zeigersprache (dies ist eine Tatsache)!
- Strukturierte Programmierung mit C macht Spaß!
- Nichtstrukturierte Programmierung mit C kann die Hölle auf Erden sein! (C erzieht?)

```
 /* Was tut dieses Programm ? */
int v,i,j,k,l,s,a[99];
main ()
{   for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,
    j+=(v=j<s&&(!k&&!!printf(2+"\n\n%c"-(!l<<!j),
    " #Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-j))
    &&!(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i])
        ;
}

/*  Antwort: Loesungen des n-Damenproblems in der Form:
    Q# # # #
    # # #Q#
     # # # Q
    # Q # #
     # # #Q#
    # #Q# #
     Q # # #
    # # Q #           */
```

Programm: 2-einf/dame.c

Grober Vergleich zwischen C und Pascal:

Thema	Vergleich (Informationen)
Schleifen	sehr ähnlich, C besitzt mächtigere For-Schleife
Verzweigungen	ähnlich, C besitzt gewöhnungsbedürftige Mehrfachverzweigung
Zuweisungen	C besitzt keine strenge Typüberprüfung!
Unterprogramme	ähnlich, C unterscheidet nicht zwischen Prozeduren und Funktionen
Einfache Datentypen	ähnlich, C besitzt mehr Typen, etwa vorzeichenlose Variablen
Operatoren	C besitzt wesentlich mehr Operatoren, etwa zur Bitmanipulation
Komplexe Typen	C kennt keine Mengen und Teilbereichstypen
Felder	in C beginnen alle Felder mit dem Index 0. C erlaubt den Zugang zu Feldern mittels Zeiger
Zeiger	in C werden Zeiger nicht nur bei dynamischen Strukturen verwendet, sondern sehr intensiv auch bei Feldern und Zeichenketten
Strukt. Typen	sehr ähnlich
Ein-/Ausgabe	in C nicht im Sprachumfang enthalten. Stattdessen gibt es eine sehr umfangreiche Standardbibliothek (in ANSI-C enthalten!)
Getrennte Übersetzbarkeit	in Standard-Pascal nicht vorgesehen, in Turbo-Pascal mittels Units. In C gibt es kaum Einschränkungen

Vorgehen in der Vorlesung:

– Schleifen, Verzweigungen und Zuweisungen werden im Eiltempo behandelt,
– Funktionen, Rekursion, Datentypen, Operatoren und die Ein-/Ausgabe erfolgen ausführlicher,
– Ausführlich werden Felder, Zeichenketten, Zeiger und getrennte Übersetzbarkeit behandelt.

2.4 Einfache Datentypen

Beispiele:

```
int i, j;           /* entspricht Integer in Pascal */
short k;            /* 16 Bit Integer */
long l;             /* >= 32 Bit Integer */
                    /* short <= int <= long */
float x, y;         /* entspricht real in Pascal */
double z;           /* lange Gleitpunktzahl (8 Byte) */
long double w;      /* sehr lange Gleitpunktzahl */
char ch;            /* Zeichen: 1 Byte Zahl */
```

Alle diese Typen (außer Gleitpunktzahlen) können noch mit dem Attribut ‘unsigned’ versehen werden. Dann wird das 1. Bit nicht als Vorzeichen interpretiert, z.B.:

```
char ch;                    Bereich zwischen -128 und +127
unsigned char ch;           Bereich zwischen 0 und +255
```

Bereits bei der Deklaration darf eine Zuweisung erfolgen, so daß Variablen initiiert werden können:

```
float x = -1.1, y = 2.4;      /* Deklarationen mit Wertzuweisung */
```

Wir erkennen, daß in C der Zuweisungsoperator das Zeichen ‚=‘ ist und nicht ‚:=‘. Dies gilt für alle Zuweisungen. Wir schreiben daher:

Variable = Ausdruck;

Mehrfachzuweisungen sind möglich, etwa ‘a=b=10;’ . Hier erhält zunächst b den Wert 10 und dann a den Wert von b (Abarbeitung von rechts nach links).

Betrachten wir jetzt unser zweites Programm:

```
  /* Ausgabe der Wurzeln der Zahlen von 1 bis 20  */
#include <stdio.h>
#include <math.h>      /* Bibliothek fuer math. Funkt. */
int main ()
{  int i;            /* Ganzzahl */
   float x;          /* Gleitpunktzahl */
   /* Ausgabe der ersten 10 Werte mit while */
   i = 1;      /* nicht := !!! */
   while ( i <= 10 )
   {  x = sqrt (i);
      printf ("Wurzel aus %d ist %f\n", i, x);
      i++;     /* entspricht i = i+1   */
   }
   /* Jetzt die Werte 11 bis 20 mit for  */
   for (i=11; i<=20; i++)
      printf ("Wurzel aus %5d ist %10.6f\n", i, sqrt(i) );
   return 0;
 }
```

Programm: 2-einf/prog2.c

Folgerungen:

- Die mathematischen Funktionen stehen in der mathematischen Bibliothek *libm.a* . Die dazugehörige Headerdatei heißt *math.h* . Sie muß dem Compiler bekannt gemacht werden. Damit der Linker die benötigten Module aus der mathematischen Bibliothek findet, muß zusätzlich beim Compilieren die Option '-lm' mit angegeben werden. Der Compileraufruf lautet daher:

  ```
  cc prog2.c -lm
  ```

- In C existieren eigene Inkrement- und Dekrementoperatoren. Der Ausdruck ‘i++’ entspricht ‘i=i+1’, entsprechendes gilt für ‘i--’, also ‘i=i-1’.
- Die While-Schleife besitzt in C die Syntax

while (Ausdruck)
  Anweisung
	analog zu Pascal:	while Ausdruck do
		  Anweisung

Das Verhalten der While-Schleife ist analog zur While-Schleife in Pascal.

- Die For-Schleife besitzt in C die Syntax

```
for ( Ausdruck1 ; Ausdruck2 ; Ausdruck3 )
    Anweisung;
```

Diese For-Schleife ist in C äquivalent zu:

```
Ausdruck1;                /* Initialisierung */
while ( Ausdruck2 )     /* Bedingung */
{   Anweisung;
    Ausdruck3;            /* Fortschalten, z.B. Inkrement */
}
```

Wir hätten damit die beiden Schleifenkonstrukte *while* und *for* vollständig behandelt. Während *while* identisch zu Pascal verwendet wird, ist in C die For-Schleife nur eine Abkürzung für eine komplexere While-Schleife. Im Gegensatz zu Standard-Pascal besitzt demnach der Schleifenzähler in der For-Schleife in C am Ende immer einen definierten Wert, dieser Schleifenzähler darf ebenso wie die untere und obere Grenze beliebig manipuliert werden. Wir können sogar For-Schleifen ohne Schleifenzähler definieren!

Als eine Art „Vergewaltigung“ der For-Schleife sei das bereits gezeigte Beispiel des Achtdamenproblems auf Seite 5 genannt. Es besteht praktisch nur aus einem For-Schleifen-Kopf.

Namen in C und Pascal stimmen nahezu überein. In beiden Sprachen enthalten Namen beliebig viele Zeichen, die aus Buchstaben, Zahlen oder dem Unterstrichzeichen (‘_’) bestehen. Als erstes Zeichen ist in Pascal nur ein Buchstabe erlaubt, in C auch das Unterstrichzeichen.

2.5 Ausgabe mit printf

Der Funktionskopf von *printf* besitzt die Form (aus stdio.h):

int printf (char * format, arg1, arg2, …)

Dies bedeutet, daß die Funktion *printf* beliebig viele Parameter besitzen kann und ein int-Wert zurückgeliefert wird. Dieser int-Wert enthält die Anzahl der geschriebenen Zeichen und ist im Fehlerfall negativ. Meist wird er in Programmen aber nicht ausgewertet. Der erste Parameter ist die C-Darstellung für die Übergabe einer Zeichenkette. Wir kommen darauf zurück. Hier steht also in der Praxis meist

" … <text> … "

Wenn neben Text auch der Inhalt von Variablen ausgegeben werden soll, so wurde in Pascal der Text einfach abgebrochen, der Name der Variablen angegeben und anschließend der Text wieder fortgesetzt. Die Vorgehensweise von C ist völlig anders und erinnert an den Formatsatz in Fortran. Zur Ausgabe von Variableninhalten dienen in C Formatangaben, die in der Formatzeichenkette zusammen mit dem auszugebenden Text stehen. Diese Formatangaben beginnen immer mit dem Prozentzeichen ('%'). Diesem Zeichen folgen detaillierte Angaben zur Ausgabeform der Variable. Die Variablennamen selbst stehen als Argumente, durch Komma getrennt, hinter dem Formatstring.

Die Syntax der Formatangabe lautet:

% [Flag] [Breite] [. Präzision] [Präfix] Typ

Die in eckigen Klammern stehenden Ausdrücke sind wahlfrei. Sie haben im einzelnen folgende Bedeutung (keine vollständige Aufzählung):

Flag:	-	linksbündige Ausgabe (standardmäßig: rechtsbündig)
	+	Pluszeichen vor positiven Zahlen wird ausgegeben
	0	Führende Zeichen werden mit Nullen aufgefüllt
	#	vor oktalen und hexadezimalen Zahlen wird '0' bzw. '0x' geschrieben Gleitpunktzahlen besitzen immer einen Dezimalpunkt
Breite:	Zahl	Die angegebene Zahl gibt die Mindestbreite der Ausgabe an. (entspricht in Pascal der Zahl nach dem 1. Doppelpunkt in write)
	*	die nächste Zahl der Argumentliste wird als Breite interpretiert (int-Zahl)
Präzision:	Zahl	Anzahl der Nachkommastellen bei Gleitpunktzahlen (entspricht in Pascal der Zahl nach dem 2. Doppelpunkt in write) Anzahl der maximal auszugebenden Zeichen bei Zeichenketten
	*	die nächste Zahl der Argumentliste wird als Präzision interpretiert (int)
Präfix:	l	Variable wird als Typ *(unsigned) long* interpretiert und ausgegeben
	L	Variable wird als Typ *long double* interpretiert und ausgegeben
	h	Variable wird als Typ *short* interpretiert und ausgegeben
Typ:	d, i	Dezimalzahl (dezimal, integer)
	o	Dezimalzahl oktal ausgeben
	x, X	Dezimalzahl hex ausgeben mit Klein-/Großschreibung der Buchstaben
	u	Dezimalzahl ohne Vorzeichen (unsigned)
	c	einzelnes Zeichen (char)
	f	Gleitpunktzahl im double-Format (float)
	e, E	Gleitpunktzahl in Exponentialdarstellung (Klein-/Großschreibung)
	g, G	Gleitpunktzahl, bei kleinem Exponenten f-Darstellung, sonst e bzw. E
	s	Zeichenkette (String)
	p	Zeigerausgabe (Pointer)
	%	Ausgabe des Prozentzeichens

Außer der Formatangabe in der Formatzeichenkette dürfen auch spezielle Zeichen verwendet werden. Diese Spezialzeichen beginnen alle mit dem Backslash ('\') und sind in allen Zeichenketten erlaubt:

\n	Neue Zeile	\t	Tabulator
\a	Glocke (alert)	\b	Backspace
\\	Backslash	\?, \', \"	entwertet die Zeichen ?, ', "
\0	Stringendezeichen	\xhh	Hexzahl des ASCII-Codes

Beispiel zur Ausgabefunktion *printf*:

```
#include <stdio.h>
#include <math.h>          /* enthaelt math. Funktionen */

int main ()
{  int         i;
   long double lpi;
   double      dpi;
   float       fpi;
   char      * p = "123456789";

   for (i=1;i<=5;i++)
      printf (">%*d<\n",i+4,i*i);
   printf ("Text: %20s\n",p);
   printf ("Text: %-20.5s\n",p);
   lpi = 4 * atan(1);        /* arctan */
   dpi = 4 * atan(1);        /* arctan */
   fpi = 4 * atan(1);        /* arctan */
   printf ("pi = <%+-30.20Lf<\n", lpi);
   printf ("pi = <%+30f<\n", dpi);
   printf ("pi = <%30.20f<\n", dpi);
   printf ("pi = <%030.20f<\n", fpi);
   return 0;
 }
```

Programm: 2-einf/printf.c

Dieses Programm erzeugt als Ausgabe:

```
>    1<
>     4<
>      9<
>      16<
>        25<
Text:                123456789
Text: 12345
pi = <+3.14159265358979324000        <
pi = <                      +3.141593<
pi = <        3.14159265358979312000<
pi = <000000003.14159274101257324000<
```

Ausgabe des Programms: 2-einf/printf.c

2.6 Ein- und Ausgabe mit den Ausgabeströmen in C++

Das Arbeiten mit Ein-/Ausgaben wurde in C++ völlig neu gestaltet. Dies war notwendig, um in Klassen die Ein-/Ausgabe optimal einbinden zu können. Natürlich steht in C++ die Ausgabefunktion *printf* weiterhin zur Verfügung. Man muß nur weiterhin die Headerdatei *stdio.h* mit Include hinzufügen. Im weiteren wird auf diese Möglichkeit aber meist verzichtet. Zum einen wollen wir uns an die C++ Möglichkeiten gewöhnen, zum anderen sind die Möglichkeiten der Ausgabe in C++ noch vielfältiger als in C!

In C++ werden die Ein- und Ausgaben als Ströme (streams) bezeichnet. Die für uns wichtigen Ströme sind:

- ostream: Ausgabestrom
- istream: Eingabestrom

Realisiert sind diese Ströme als Klassen, eine Erweiterung von Strukturen analog zu Delphi (Record wurde dort zu Object bzw. Class). Die obigen Ströme sind demnach Datentypen, die in der Headerdatei *iostream.h* definiert sind. Standardmäßig sind folgende Klassenvariablen, auch als Klassenobjekte bezeichnet, vordefiniert:

- cout als Objekt der Klasse ostream, schreibt auf die Standardausgabe
- cerr als Objekt der Klasse ostream, schreibt auf die Standardfehlerausgabe
- cin als Objekt der Klasse istream, liest von der Standardeingabe

Das Arbeiten mit *cout* und *cin* funktioniert fast so wie in Pascal mit *write* und *read*:

```
printf ("Wurzel aus %d ist %f\n", i, x);              /* C-Code */
cout << "Wurzel aus " << i << " ist " << x << "\n";   // C++ -Code
```

Wir sehen, daß der Datentyp einer Variable automatisch erkannt und die Variable entsprechend ausgegeben wird. Dies reduziert die Möglichkeiten von Fehlerquellen enorm. In C++ gibt es darüberhinaus noch eine weitere Kommentarklammer, nämlich ‚//'. Folgende Zeichen bis zum Ende der Zeile gelten als Kommentar.

Auch das Einlesen von Variablen ist mit Strömen einfach. Sollen die Gleitpunktzahl x, die Ganzzahl a und das Zeichen ch nacheinander eingelesen werden, so erledigt dies:

```
cin >> x >> a >> ch;
```

Zu beachten ist, daß hier immer führende White-Spaces überlesen werden!

Wir wollen jetzt das Beispiel des letzten Abschnitts auch mit den Ausgabeströmen lösen. Dazu müssen wir auch hier Formatanweisungen kennenlernen, die meist als Funktionen aufgerufen werden:

```
cout.width (n)          // Die Int-Zahl n gibt die Breite an, gültig nur für die
                        // nächste Ausgabe!!!!
cout.precision (n)      // Die int-Zahl n ist die Anzahl der Nachkommastellen,
                        // von Gleitpunktzahlen, gültig bis zur Änderung!!!
cout.fill (ch)          // Das Zeichen ch ist ab sofort das Füllzeichen
cout.setf ( … )         // Funktion zum Setzen von Flags
cout.setf ( … , … )     // Funktion zum Setzen und Rücksetzen von Flags
cout.unsetf ( … )       // Funktion zum Rücksetzen von Flags
```

Diese Funktionen sind natürlich auch auf die Standardfehlerausgabe *cerr* anwendbar. Die Flags der Ein-/Ausgabeströme sind intern als Bitleiste implementiert und sind in der Klasse *ios* als Konstanten definiert. Sie heißen:

ios::left	Linksbündiges Ausrichten der Ausgabe
ios::right	Rechtsbündiges Ausrichten der Ausgabe
ios::internal	Die Füllzeichen stehen zwischen Vorzeichen und Zahl
ios::dec	Dezimaldarstellung von Ganzzahlen
ios::oct	Oktaldarstellung von Ganzzahlen
ios::hex	Hexadezimaldarstellung von Ganzzahlen
ios::fixed	Verwendung der Dezimaldarstellung von Gleitpunktzahlen
ios::scientific	Exponentialdarstellung von Gleitpunktzahlen
ios::showpos	Positives Vorzeichen wird mit ausgegeben
ios::uppercase	Die Hexbuchstaben A bis F werden als Großbuchstaben ausgegeben
ios::showpoint	Gleitpunktzahlen besitzen immer einen Dezimalpunkt
ios::showbase	Die Basis (oktal, hex) wird mit angezeigt

Alle diese Flags können jetzt mit der Funktion *setf* gesetzt werden. Beispiele:

```
cout.setf (ios::right);                    // Ab jetzt rechtsbündige Ausgabe
cout.setf (ios::fixed);                    // Ab jetzt Dezimaldarstellung von Gleitpunktzahlen
cout.setf (ios::showpos | ios::uppercase);        // setzt 2 Flags gleichzeitig
```

Die Flags sind als Bitleiste implementiert. Es wird mit *setf* folglich das entsprechende Bit gesetzt. Will man mehr als ein Bit gleichzeitig setzen, so sind die einzelnen Bits mit ODER zu verknüpfen. Ein bitweises Oder existiert in C, der Oder-Operator lautet ‚|' (siehe auch später in diesem Kapitel).

Nun ist es nicht sinnvoll, gleichzeitig linksbündige und rechtsbündige Ausrichtung einzustellen. Aus diesem Grund ist es ratsam, vorher die nicht mehr benötigten Bits zurückzusetzen. Soll etwa wieder linksbündiges Ausrichten eingestellt werden, so ist folgendes Vorgehen zu empfehlen:

```
cout.unsetf (ios::right);          // Setzt Flag zurück
cout.setf (ios::left);             // Setzt linksbündiges Flag
```

Kürzer läßt sich dies mit nur einem Funktionsaufruf erledigen:

```
cout.setf (ios::left, ios::right);
```

Dieser erweiterte setf-Befehl setzt zunächst das im zweiten Parameter angegebene Flag zurück und

setzt erst dann das im ersten Parameter angegebene Flag. Hat man jedoch vergessen, welche Flags vorher schon gesetzt wurden, so wäre folgender Befehl angebracht:

```
cout.setf (ios::left, ios::left | ios::right | ios::internal);
```

Hier wurden sicherheitshalber alle Flags des Ausrichtungstyps zurückgesetzt. Das Gleiche ist auch für die Darstellung von Gleitpunktzahlen und für die Basisausgabe von Ganzzahlen angezeigt. Aus diesem Grund sind in der Klasse ios noch folgende Konstanten definiert:

```
ios::adjustfield    // entspricht:  ios::left | ios::right | ios::internal
ios::basefield      // entspricht:  ios::dec | ios::oct | ios::hex
ios::floatfield     // entspricht:  ios::fixed | ios::scientific
```

Der obige Funktionsaufruf wird demnach vereinfacht zu:

```
cout.setf (ios::left, ios::adjustfield);
```

Meist ist es lästig, zwischen obigen Funktionsaufrufen und der Ausgabe mit cout zu wechseln. Aus diesem Grund können viele Formatangaben auch direkt in den Ausgabestrom geschrieben werden. Solche Formatangaben heißen in C++ Manipulatoren. Die wichtigsten sind:

hex	schaltet auf Hexadezimalausgabe von Ganzzahlen um
oct	schaltet auf Oktalausgabe von Ganzzahlen um
dec	schaltet auf Dezimalausgabe von Ganzzahlen um
flush	schreibt den Ausgabepuffer sofort auf die Standardausgabe
endl	gibt ein Zeilenende ('\n') aus, gefolgt von einem "flush"
ws	überliest Whitespaces (' ', '\n', '\t'), einziger Eingabemanipulator
setw (n)	wie die Funktion cout.width (n)
setfill (ch)	wie die Funktion cout.fill (ch)
setprecision (n)	wie die Funktion cout.precision (n)

Die Manipulatoren *setw*, *setfill* und *setprecision* benötigen die Headerdatei <iomanip.h>. Konvertieren wir jetzt das Programm printf.c nach C++:

```
#include <iostream.h>
#include <iomanip.h>  // wegen Manipulatoren
#include <math.h>     // math. Bibliothek

int main ()
{  int    i;                 long double lpi;
   double dpi;               float       fpi;
   char * p = "123456789";   int         prec;

   for (i=1;i<=5;i++)
      cout << '>' << setw(i+4) << i*i << "<\n";
   cout << "Text: " << setw(20) << p << '\n';

   lpi = 4 * atan(1);       dpi = 4 * atan(1);
   fpi = 4 * atan(1);                        // arctan

   cout.setf(ios::left | ios::showpos);
   prec = cout.precision(20);
   cout << "pi = >" << setw(30) << lpi << "<\n";
   cout.unsetf(ios::left);     cout.precision(prec);
   cout << "pi = >" << setw(30) << dpi << "<\n";
   cout.unsetf(ios::showpos);  cout.precision(20);
   cout << "pi = >" << setw(30) << dpi << "<\n";
   cout << "pi = >" << setfill('0') << setw(30) << fpi;
   return 0;
}
```

Programm: 2-einf/printf.cpp

Wir sehen noch eine Anwendungsmöglichkeit einiger Ein-/Ausgabefunktionen: Als Funktionswert wird die bisherige Einstellung zurückgeliefert. Es fehlt eine Zeile, in der in C die Präzision bei Strings ausgenutzt wurde. Diese Möglichkeit gibt es nicht bei den Strömen.

Dieses Programm müssen wir jetzt mit einem C++ -Compiler übersetzen. Ersetzen Sie auf den Unix-Rechnern einfach ‚cc' durch ‚g++'. Alle anderen Parameter, wie etwa ‚-lm' sind vollkommen gleich.

2.7 Funktionen

Vom Funktionskopf her sind Pascal und C sehr ähnlich. Die Schwierigkeit von C liegt allerdings darin, daß C keine Var-Parameter wie Pascal kennt. Die Übergabe von Referenzen mittels call-by-reference muß daher direkt programmiert werden, womit wir schon bei Zeigern angelangt sind. Glücklicherweise hilft uns C++ wieder aus der Patsche, da hier der Referenzoperator & dem Var-Parameter in Pascal entspricht.

Doch beginnen wir am Anfang. Unsere Aufgabe lautet, zwei Ganzzahlen zwischen 0 und 9 zyklisch zu addieren. Entsprechende Unterprogramme in Pascal lauten:

a)
```
function summe1 (a, b : integer) : integer;
begin
   summe1 := (a+b) mod 10
end;
```
b)
```
procedure summe2 (a, b : integer; var c : integer);
begin
   c := (a+b) mod 10
end;
```

Die zweite Prozedur *summe2* liefert das Ergebnis im dritten Parameter zurück. Schreiben wir diese beiden Funktionen in C, und die zweite auch in C++:

```
#include <iostream.h>
int summe1 ( int a, int b )         /* erste Funktion */
{  return ( a + b ) % 10 ;          /* % = Modulo */
}
void summe2 ( int a, int b, int *c )
/* '*c' heisst: c ist ein Zeiger auf eine Zahl */
{  *c = ( a + b ) % 10 ;
}
void summe2pp (int a, int b, int &c)  // nur in C++
{  c = ( a + b ) % 10 ;               // &c entspricht var c
}
int main ()
{  int i, j;
   cout << "3 und 5 mod 10 ist " << summe1(3,5) << endl;
   summe2 (5, 7, &i);              /* jetzt mit summe2: */
/* '&i' heisst: Adresse von i, in i steht das Ergebnis */
   summe3 (5, 7, j);               // wie in Pascal
   cout << "5 und 7 mod 10 ist << i << " bzw. " << j <<endl;
   return 0;
}
```
Programm: 2-einf/summe.cpp

Dieses Programm liefert folgende Ausgabe:

```
(3 und 5) mod 10 ist 8
(5 und 7) mod 10 ist 2 bzw. 2
```

Aus dem Programm können wir mehrere Ergebnisse ableiten:

- der Modulo-Operator heißt in C: '%'
- in C gibt es nur Funktionen, die allerdings immer auch wie Prozeduren aufgerufen werden können. Betrachten wir nur *printf*. Diese Funktion liefert eine Ganzzahl zurück, trotzdem wird sie meist wie eine Prozedur aufgerufen werden. In diesem Fall wird der Rückgabewert einfach ignoriert.
- einer Funktion, die definitiv keinen Rückgabewert besitzt, wird *void* vorangestellt. Fehlt eine Datentypangabe ganz, so nimmt C standardmäßig an, daß ein int-Wert zurückgegeben wird!!
- eine Funktion besitzt immer die Form:

```
[ <Datentyp> ]  <Funktionsname>  (  <Argumentliste>  )   { Anweisung ; … }
```

- die Argumente in der Argumentliste werden <u>einzeln</u> aufgelistet und sind durch Kommata getrennt. Die Argumentliste darf auch leer sein. Dies bedeutet in C aber leider nicht, daß dann auch keine Argumente existieren, sondern daß die Anzahl der Argumente beliebig ist! Wir geben zur Vermeidung von Fehlern immer alle Argumente an. Wie schon erwähnt, wird bei fehlendem Datentyp implizit *int* angenommen.
- mit *return* wird der Funktionswert zurückgegeben und die Funktion beendet. Funktionen, die keinen Rückgabewert besitzen (Datentyp *void*), brauchen kein *return* enthalten. Sie werden automatisch mit der letzten Anweisung beendet. Wieder ist es C völlig egal, ob eine Funktion (auch main!) ein *return* enthält oder nicht. In diesem Fall ist der Rückgabewert undefiniert und zufällig. Glücklicherweise geben viele Compiler zumindest eine Warnung bei fehlendem *return* aus, alle C++ -Compiler sowieso. Übrigens ist *return* keine Funktion, ein Klammern des Rückgabeausdrucks ist nicht erforderlich.

Ältere C-Programme, die noch nicht nach der ANSI-Norm geschrieben sind, unterscheiden sich hauptsächlich durch einen anderen Funktionskopf von der ANSI-Norm. Die ältere Kernighan-Ritchie-C-Variante benutzte nämlich folgenden Kopf:

```
int summe1 (a, b)
int a, int b;
{  …  }
```

Wir werden diese Version, die aus Kompatibilitätsgründen von allen C-Compilern unterstützt wird, nicht mehr verwenden. In C++ wird dieser Funktionskopf grundsätzlich abgewiesen.

Wohl oder übel müssen wir uns noch mit der Übergabe call-by-reference beschäftigen. In C muß das, was uns Pascal und C++ abnimmt, selbst programmiert werden. Das Problem der Werteübergabe (call-by-value) war, daß im Unterprogramm eine Kopie des Originals angelegt wird. Jede Änderung dieser Kopie bleibt für das Original demnach ohne Belang. Soll eine Änderung eines Parameters auch zur Änderung des Originals führen, so muß dieser Parameter mit dem Original identifiziert werden. Übergeben wir statt einer Variable also die Adresse dieser Variable, so kann das Unterprogramm auf diese ihm jetzt bekannte Adresse zugreifen und dort Änderungen vornehmen. Diese Übergabe der Adresse oder Referenz heißt call-by-reference.

Wir übergeben demnach statt einer Variablen eine Adresse. Diese Adresse wird beim Aufruf des Unterprogramms in den lokalen Parameter kopiert. Adressen sind Zeiger, wir übergeben also einen Zeiger.

In C kann die Adresse zu jeder Speichervariablen ermittelt werden. Ist *i* eine Variable, so ist *&i* die Adresse dieser Variablen (bitte nicht mit einer Referenz in C++ verwechseln). Umgekehrt: ist *p* ein Zeiger, so ist **p* der Inhalt, auf den dieser Zeiger verweist.

Definiert wird beispielsweise ein Zeiger auf eine Ganzzahl in C durch: `int * p;`

Als Eselsbrücke dienen geschickt gesetzte Klammern: Mit int (*p) ist demnach **p* eine int-Variable. Folglich muß *p* selbst ein Zeiger auf eine int-Variable sein. Mit dieser Hilfestellung können wir jetzt unser call-by-reference Problem in C lösen:

Wir übergeben zum einen statt der Variablen *i* die Adresse *&i* an das Unterprogramm. Das Unterprogramm muß daher einen Zeiger auf *int* statt einer int-Variablen erwarten. Wir schreiben demnach int *c als Parameter. Im Unterprogramm wollen wir in die Speicherstelle von *i* schreiben. Im Parameter *c* steht die Adresse dieser Variablen, somit ist **c* gleich dem Inhalt an dieser Adresse, also identisch mit *i*.

Wir wollen diese Arbeitsweise mit Zeigern noch an einem Beispiel festigen. Dazu betrachten wir eine int-Variable und einen Zeiger auf *int*. Wir setzen den Wert der int-Variable auf 17, einmal direkt und einmal indirekt:

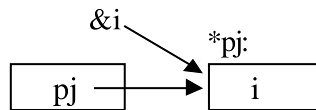
```
int i;      /* int Variable */
int *pj;    /* Zeiger auf einen int-Wert */
```

Lösung a:

```
i = 17;     /* direkt */
```

Lösung b:

```
pj = &i;    /* pj zeigt jetzt auf die Variable i */
*pj = 17;   /* der Inhalt, auf den pj zeigt, wird auf 17 gesetzt */
```



Beide Lösungen liefern letztendlich das Gleiche: die Speichervariable *i* enthält den Wert 17! Dies kann mittels *printf* oder *cout* leicht überprüft werden.

Fazit: Gut, daß es C++ und den Referenzoperator & gibt!

Beachten wir noch einige Hinweise:

– Eine Funktion kann zunächst, etwa am Anfang des Programms nur deklariert werden. Hierzu gibt man den Kopf der Funktion an und beendet diese mit einem Semikolon. Die Funktion ist dann ab dieser Zeile bekannt. Die eigentliche Definition erfolgt erst an einer beliebigen Stelle im Programm.

– Eine Prozedur ist eine Funktion ohne Rückgabewert. Diese wird in C und C++ als Funktion mit dem Rückgabewert *void* definiert. Jede Funktion darf aber auch als reine Anweisung aufgerufen werden. In diesem Fall verfällt der Rückgabewert einfach.

– Eine Funktion ohne Parameter hat in C und C++ unterschiedliche Bedeutung. In C++ ist damit eine parameterlose Funktion gemeint; in C heißt dies, daß diese Funktion eine unbestimmte Anzahl von Parametern besitzt. Um Mißverständnisse zu vermeiden, sollte bei der Definition einer parameterlosen Funktion immer der Parameter *void* angegeben werden. Dieser Parameter drückt in C und C++ genau die Parameterlosigkeit aus! Beispiele:

```
int func1 ( ) ;      // Deklaration einer Funktion, in C++: keine Parameter,
                                          in C: unbestimmte Anzahl von Parametern
int func2 (void);    // Deklaration einer Funktion ohne Parameter (in C und C++)
```

2.8 Eingabefunktionen in C und C++

Die Funktion *scanf* ist in C das Gegenstück zu *printf* und dient zur formatierten Eingabe von Variablen. Der Funktionskopf ist analog zu dem von *printf* und lautet:

```
int scanf (char *format, arg1, arg2, … )
```

In der Formatzeichenkette stehen wieder Formatangaben, diesmal von der Form

```
% [ * ]  [ Breite ]  [ Präfix ] Typ
```

Die in eckigen Klammern stehenden Angaben sind wahlfrei. Die Bedeutung der einzelnen Werte ist:

*:	der nächsten Variablen wird kein Wert zugewiesen (sehr selten verwendet)	
Breite:	maximal so viele Zeichen vom Eingabestrom werden eingelesen	
Präfix:	l, L, h	siehe printf, zusätzlich ‚lf' bei double-Variablen
Typ:	d, i, o, u, x:	siehe printf
	c:	das nächste Zeichen wird gelesen
	s:	alle Zeichen bis zum nächsten White Space werden gelesen
	e, f, g:	Gleitpunktzahl (double) wird eingelesen
	p	Zeigerwert
	%	entspricht dem Zeichen %

Die Verwendung der Formatzeichenkette in der Funktion *scanf* ist sehr gewöhnungsbedürftig. Es wird daher empfohlen, hier außer der Formatangabe nur Leerzeichen zu verwenden. Diese sind ohne Bedeutung. Andere vorkommende Zeichen werden in der gleichen Reihenfolge bei der Eingabe erwartet und dann einfach überlesen. Dies führt häufig zu einem unerwarteten Verhalten.

Nach ANSI-C werden beim Einlesen von Variablen (auch von Zeichenketten!) mittels *scanf* alle führenden White-Spaces überlesen. White-Spaces sind neben dem Leerzeichen der Tabulator, der Wagenrücklauf und das Neue-Zeile-Zeichen. Erfahrungsgemäß wird diese Regel jedoch von praktisch allen C-Compilern im Falle des Einlesens einzelner Zeichen (mittels %c) verletzt. Steht vor dem Prozentzeichen mindestens ein Leerzeichen, so werden White-Spaces wie gewünscht überlesen. Ansonsten liest *scanf* das direkt folgende Zeichen ein, eventuell also auch ein Leerzeichen.

Beim Einlesen von Gleitpunktzahlen muß im Gegensatz zum Ausgeben mit *printf* folgendes beachtet werden: Mit '%f' wird eine float-Zahl, mit '%lf' eine double-Zahl und mit '%Lf' eine long-double-Zahl eingelesen. Analoges gilt für den Typ e und g.

scanf ist eine Funktion wie jede andere. Um Variablen über Parameter zurückgeben zu können, müssen diese call-by-reference übergeben werden, also als Zeiger! Bei Fehlern oder beim Erreichen des Dateiendes liefert scanf den Wert EOF zurück. Betrachten wir ein Beispiel zur formatierten Eingabe:

```
#include <stdio.h>

int main ()
{  int i, tag, monat, jahr;
   float x;
   char ch;

   printf ("\nNacheinander Ganzzahl, Gleitpunktzahl\n");
   printf ("und ein Zeichen eingeben:\n");

   scanf ("%d%f %c", &i, &x, &ch );       /* Adressen !!! */

   printf ("Ausgabe: i=%d , x=%f , ch=%c\n", i, x, ch);
   printf ("\nDatum in der Form tt.mm.jjjj eingeben: ");

   scanf ("%d . %d . %d", &tag, &monat, &jahr);

   printf ("Ausgabe: Tag %d.%d.%d\n\n", tag, monat, jahr);
   return 0;
}
```

Programm: 2-einf/scanf.c

Die Punkte in der Formatzeichenkette zwischen Tag und Monat bzw. Monat und Jahr erlauben das Einlesen eines Datums in der gewohnten Schreibweise, etwa mittels ‚7. 12. 1994'. Doch seien Sie wegen möglicher Nebeneffekte sehr vorsichtig mit diesen Anwendungen.

Achtung:

➔ Häufiger Fehler: **Das Zeichen ‚&' wird im scanf vergessen!**

<u>Eingabefunktion **getchar**:</u>

Neben *scanf* gibt es auch andere Eingabefunktionen. Eine sehr häufig vorkommende Funktion ist das Einlesen eines einzelnen Zeichens. Dazu dient die Funktion *getchar*. Beispiel:

ch = getchar ( ) ;

Damit wird das nächste Zeichen in die int-Variable *ch* geschrieben. Es wird eine int-Variable zurückgeliefert, damit neben Zeichen auch andere Informationen zurückgegeben werden können. Die wichtigste solche Info ist die Mitteilung des Endes der Datei (EOF). Dies kann auch bei der Eingabe von der Tastatur simuliert werden. Auf praktisch allen Rechnern ist entweder Ctrl-D (Unix) oder Ctrl-Z (MS-Dos) das EOF-Zeichen.

Achtung:

➔ Häufiger Fehler: **Typ char statt int bei getchar( )!**

Eine int-Variable kann übrigens genauso zum Abspeichern von einzelnen Zeichen verwendet werden wie char-Variablen. Die Ausgabe erfolgt analog mit '`%c`', ebenso die Eingabe mit *scanf*.

Übrigens unterscheidet C Funktionen von Variablen dadurch, daß den Funktionen ein Klammerpaar folgt. Somit muß auch bei parameterlosen Funktionen das Klammerpaar geschrieben werden. Sonst hält C die Funktion für eine Variable gleichen Namens halten, die aber vermutlich nicht definiert wurde.

Ausgabefunktion **putchar**:

Das Gegenstück zu *getchar ( )* ist *putchar (ch)* ;

Das angegebene int-Zeichen *ch* wird ausgegeben. Ist *ch* ein char-Zeichen, so wird es zuvor automatisch in ein int-Zeichen verwandelt. Beachten Sie, daß nur einzelne Zeichen und keine Zeichenkette mit *putchar* ausgegeben werden können. Zur Unterscheidung zu Zeichenketten werden einzelne Zeichen in Hochkomma eingeschlossen und nicht mit Gänsefüßchen. Wir schreiben also: `putchar('\n');` , wenn eine neue Zeile begonnen werden soll.

Beispiel: Zeichenweises Kopieren der Standardeingabe in die Standardausgabe:

```
/* kopiert Input nach Output, "Pascal"-Version */
#include <stdio.h>
int main ()
{  int ch;      /* um auch EOF zu lesen: int-Variable */
   printf ("Geben Sie beliebig viele Zeichen ein.");
   printf ("Beenden mit CTRL-D bzw. CTRL-Z:\n");
   ch = getchar();
   while (ch != EOF)
   {  putchar(ch);
      ch = getchar();
   }
   return 0;
}
```

Programm: 2-einf/copy.c

Das folgende Beispiel unterscheidet sich funktional nicht vom obigen. Es zeigt jedoch eindrucksvoll die Philosophie in C: Jede Aktion ist ein Ausdruck, also auch alle Unterprogrammaufrufe oder Zuweisungen. Erst durch ein folgendes Semikolon wird ein Ausdruck zu einer Anweisung! Somit lassen sich, wie im folgenden geschehen, Zuweisungen überall dort weiterverwenden, wo Ausdrücke erlaubt sind. Dies geschieht hier innerhalb des While-Ausdrucks. Nicht nur wegen der im Vergleich zum vorhergehenden Beispiel deutlich kürzeren Schreibweise, erfreut sich dieser „neue" Programmierstil in der C-Welt großer Beliebtheit. Wir werden diesen Stil deshalb auch in der Vorlesung häufig verwenden.

```
/* kopiert Input nach Output, "C"-Version */
#include <stdio.h>
int main ()
{  int ch;      /* um auch EOF zu lesen: int-Variable */
   printf ("Geben Sie beliebig viele Zeichen ein.");
   printf ("Beenden mit CTRL-D bzw. CTRL-Z:\n");
   while ( (ch = getchar() ) != EOF)
      putchar(ch);
   return 0;
}
```

Programm: 2-einf/copy2.c

Übrigens wird in C die Ungleichheit mit '!=' und nicht durch '<>' angegeben. Die zusätzlichen Klammern innerhalb der While-Schleife sind erforderlich, da die Zuweisung schwächer bindet als Vergleichsoperatoren.

Wegen der Probleme der Eingabe (Adreßübergabe bei *scanf*, int-Variable bei *getchar*) ist auch bei der Eingabe der Eingabestrom von C++ wesentlich angenehmer. Wir haben bereits die Eingabe mit C++ kennengelernt. Wichtig ist, daß alle Datentypen korrekt erkannt, und die Eingaben entsprechend konvertiert werden. Die erste Eingabe aus dem Programm *scanf.c* lautet in C++ einfach:

```
cin >> i >> x >> ch ;
```

Die zweite Eingabe ist nicht so einfach, da ein automatisches Überlesen der Punkte in C++ nicht implementiert wurde. Wir überlesen stattdessen ein einzelnes Zeichen:

```
cin >> tag >> ch >> monat >> ch >> jahr;
```

Beachten Sie, daß in Eingabeströmen (*cin*) immer führende Whitespaces überlesen werden!

Nun besitzt C++ wie C aber noch eine Menge weiterer Ein- und Ausgabefunktionen. Vergleichbar mit *getchar* und *putchar* sind:

```
i = cin.get ( )     Liest ein int-Zeichen vom Eingabestrom
cin.get (ch)        Liest ein char-Zeichen vom Eingabestrom
cout.put (ch)       Schreibt ein int- oder char-Zeichen ch in den Ausgabestrom
```

Die While-Schleife im Programm *copy2.c* läßt sich nun wie folgt schreiben:

```
while ( (ch = cin.get() ) != EOF )
   cout.put (ch);
```

Ich rate jedoch von dieser Verwendung ab. Erstens muß *ch* eine Int-Zahl sein, obwohl doch Zeichen eingelesen werden; und zweitens darf man die Klammern im While-Ausdruck nicht vergessen. Wesentlich angenehmer ist:

```
while ( cin.get(ch) != 0 )
   cout.put (ch);                 // in C++ vorzuziehen
```

Hier wurde erfolgreich ausgenutzt, daß die Funktion *cin.get()* nur bei Fehlern, etwa beim Erreichen des Dateiendes, den Wert 0 zurückliefert. Es ist jedoch zwingend zu beachten, daß hier die Variable ch eine Char-Variable ist!

2.9 Weitere Kontrollstrukturen

<u>Die if-Anweisung:</u>

```
Syntax:     if ( Ausdruck )
               Anweisung
            [ else                       /* else-Zweig ist optional */
               Anweisung ]
```

Die im if-Ausdruck meist benötigten Vergleichsoperatoren lauten in C:

< , > , <= , >= , == , != .

Beachten Sie die Unterschiede gegenüber Pascal beim Vergleich auf Gleichheit und Ungleichheit.

C kennt keine Booleschen Variablen. Stattdessen erwartet C als Ergebnis eines If- oder While-Ausdrucks einen int-Wert. C interpretiert das Ergebnis dieses int-Wertes als wahr, falls der Ausdruck ungleich Null ist. Ist der Ausdruck hingegen gleich Null, so wird der Wahrheitswert falsch angenommen.

Auch Vergleiche liefern keinen Booleschen Wert zurück, sondern eine int-Zahl. Ist ein Vergleich wahr, so ist das Ergebnis des Vergleichs in C immer gleich 1, ansonsten gleich 0.

```
Beispiel:  i = 3 > 2 ;             /* in i wird der Wert 1 gespeichert */
```

Mit diesem Wissen können wir die Funktionsweise der If-Anweisung in C genau angeben: ist der If-

Ausdruck ungleich 0, so wird die folgende Anweisung ausgeführt. Ansonsten (Ausdruck ist 0) wird zum Else-Zweig gesprungen. Existiert dieser nicht, so wird zur folgenden Anweisung gesprungen.

Sollen mehr als eine Anweisung im If- oder Else-Zweig abgearbeitet werden, so sind wie in Pascal Blockanweisungen mittels geschweifter Klammern erforderlich.

Beispiel:

```
if ( a == b )
    cout << "Die Variablen a und b sind gleich.\n" ;
else
    cout << "Die Variablen a und b sind ungleich.\n" ;
```

Semikolon zwingend erforderlich, um Ausdruck in Anweisung zu verwandeln

Achtung:

➔ Häufiger Fehler: im if-Ausdruck wird **'='** statt **'=='** verwendet !

Was liefert obiges Beispiel im Falle von '(a = b)'?

Antwort: Der Wert von b wird auch in a gespeichert. Das Ergebnis dieser Zuweisung wird interpretiert. Ist also b ungleich 0, so wird das erste *printf* ausgegeben, ist b gleich 0, so das zweite! Insbesondere liegt kein syntaktischer Fehler vor!!! Glücklicherweise warnen viele Compiler.

Im If-Ausdruck werden häufig die Operatoren AND und OR benötigt. Sie heißen in C '&&' bzw. '||'.

Wichtig: Nochmals sei erwähnt, daß alle Operationen in C ein Ergebnis zurückliefern. Dieses Ergebnis kann weiterverwendet werden, muß aber nicht. Ein abschließendes Semikolon beendet den Ausdruck und erzeugt formal eine Anweisung. Beispiel:

```
statt:          i = j; j++;
abkürzend:      i = j++;
```

Mehrfachverzweigungen (switch):

Analog zur Case-Verzweigung in Pascal existiert in C die switch-Verzweigung:

```
switch  ( Ausdruck )
{   case  Konstante :  Anweisungen  break;
    case  Konstante :  Anweisungen  break;
    …
             default :  Anweisungen
}
```

Betrachten wir ein Beispiel, das in einem eingegebenen Text die Anzahl der darin enthaltenen Zahlen, White-Spaces und sonstigen Zeichen zählt und ausgibt:

```
#include <iostream.h>

int main ()
{  char ch;
   int zahl, frei, sonst;

   zahl = frei = sonst = 0;                 // init
   cout << "\n\nZeichen eingeben, mit Ctrl-D beenden\n";

   while ( cin.get(ch) != 0)
      switch (ch)
      {  case '0': case '1': case '2': case '3':
         case '4': case '5': case '6': case '7':
         case '8': case '9':                 zahl++; break;
         case ' ': case '\n': case '\t':     frei++; break;
         default:                            sonst++;
      }

   cout << "\n\nZahlen = " << zahl << "\nFreiraeume = "
        << frei << "\nSonstige = " << sonst << "\n\n\n";
   return 0;
}
```

Programm: 2-einf/switch.cpp

Zu beachten ist folgendes:

- In einer Switch-Verzweigung wird der entsprechende Zweig angesprungen. Kommt die Konstante nicht vor, so wird zu Default gesprungen. Ist kein Default vorhanden, so wird die Switch-Verzweigung verlassen.
- Wurde ein Sprungziel gefunden, so wird der Switch-Block bis zum nächsten *Break* durchlaufen. Folgt kein *Break*, so werden alle Anweisungen bis zum Ende des Switch-Blocks abgearbeitet.
- Break bricht innerhalb eines Switch-Blocks diesen Block ab. Es wird mit der nächsten Anweisung fortgefahren. *Break* ist auch innerhalb eines Schleifen-Blocks (for, while) erlaubt und bricht entsprechend die (innerste) Schleife ab.
- continue: Diese Anweisung ist nur innerhalb von Schleifen erlaubt. Im Gegensatz zu *Break* wird damit nicht die gesamte Schleife abgebrochen, sondern nur der aktuelle Schleifendurchlauf. Es wird sofort mit dem Überprüfen des nächsten Schleifenausdrucks fortgesetzt.

Schleife mit Überprüfung am Ende

Auch in C gibt es eine Schleife mit der Überprüfung am Ende des Schleifendurchlaufs. Bei diesem Schleifentyp wird die Schleife immer mindestens einmal durchlaufen:

```
do                                Pascal:   repeat
   Anweisung                                    Anweisungen
while  ( Ausdruck ) ;                       until  Ausdruck ;
```

Bei genauer Betrachtung stellen wir zwischen C und Pascal zwei größere Unterschiede fest. Erstens muß in C eine Blockanweisung (also geschweifte Klammern) verwendet werden, soll mehr als eine Anweisung innerhalb der Schleife ausgeführt werden. Zweitens fragt Pascal solange, bis etwas wahr wird, während die Schleife in C durchlaufen wird, solange etwas wahr ist. In C steht also im Ausdruck genau das Gegenteil vom Pascal-Ausdruck! Beispiel:

```
do                                          repeat
{   …                      entspricht:          …
}  while ( x > 0 ) ;                        until  x <= 0 ;
```

2.10 Variablen, Typen und Operatoren

Wir kennen bereits folgende Datentypen:

- Ganzzahlen: int, long, short, char
- Gleitpunktzahlen: float, double, long double
- Zeiger: Zeiger auf obige Datentypen

Wir lernen noch kennen:

- Felder
- Zeichenketten (spezielle Felder)
- Strukturierte Datentypen: struct, union (entspricht Record in Pascal)
- Dateidatentypen
- Zeiger auf diese Datentypen

Eigenschaften des Datentyps char:

In C ist der Datentyp **char** eine **1 Byte Ganzzahl**! Es gibt auch keine Funktionen *chr(i)* oder *ord(ch)*. In einer Variablen vom Typ *char* können sowohl kleine Zahlen oder auch ASCII-Zeichen gespeichert werden. Beim Einlesen mit *getchar* oder *scanf* mit der Formatangabe %c wird in einer Ganzzahl (int, long, short oder char) immer der ASCII-Wert des gelesenen Zeichens abgelegt. Entsprechend wird dasjenige Zeichen mit *putchar* oder *printf* mit der Angabe %c ausgegeben, dessen ASCII-Wert in der Varia-

blen gespeichert ist. In den Eingabeströmen in C++ wird automatisch der korrekte Typ erkannt. Daher wird in eine Char-Variable immer der ASCII-Wert eines Zeichens eingelesen. Die Ausgabe erfolgt entsprechend. Soll nun der ASCII-Wert einer Int-Variable ausgegeben werden, so ist sie explizit mit Hilfe des Cast-Operators (siehe weiter unten) in eine Char-Variable umzuwandeln.

Wird hingegen eine Ganzzahl mit *scanf* und der Formatangabe %d (oder %i, %o, %x, …) eingelesen, so wird die interne Ganzzahldarstellung dieser Zahl gespeichert. Analog wird mit *printf* und %d die Zahl ausgegeben, die der internen Ganzzahldarstellung entspricht. Im folgenden Beispiel wird nacheinander die Zeichendarstellung und der ASCII-Wert eines Zeichens ausgegeben:

```
char c;
c = getchar ( );
printf ( "Der ASCII-Code von %c ist %d.\n", c, c );
```

Aus diesem und weiteren Gründen ist es wichtig, daß in der Formatangabe von *printf* und *scanf* immer der gewünschte Typ richtig angegeben wird. Der Compiler wird hier nie einen Syntaxfehler feststellen, da wirklich alle Kombinationen erlaubt sind. Ob diese sinnvoll sind, ist eine andere Frage.

Wir verstehen jetzt, daß es immer möglich ist, statt char- auch int-Variablen zu verwenden. Auch der umgekehrte Weg ist möglich, vorausgesetzt es werden nur kleine Ganzzahlwerte benötigt. Es sei aber auf folgende Eigenarten von *scanf* nochmals dringend hingewiesen:

```
scanf ("%c", &zahl);     /* liest das nächste Zeichen, auch White-Space */
scanf (" %c", &zahl);    /* liest das nächste Zeichen ungleich White-Space */
```

In beiden Fällen wird das Zeichen in das <u>höchstwertige</u> Byte der Ganzzahl eingelesen. Dieses unverständliche Verhalten von *scanf* (nur beim Typ %c) führt dazu, daß praktisch nur Zeichen vom Typ *char* ordentlich weiterverarbeitet werden können; denn bei der Ausgabe (mit *printf* oder *putchar*) wird immer das niederwertige Byte ausgegeben!

In C++ haben wir es glücklicherweise einfacher. Betrachten wir nochmals obiges Beispiel in C++:

```
char c;
cin.get (c);
cout << "Der ASCII-Code von " << c << " ist " << int(c) ;     // int(c) wandelt c in int um
```

<u>Arithmetische Operatoren: + - * / %</u>

Diese sind von der Funktionsweise her bekannt. Die Bindungsstärke ist wie üblich „Punkt vor Strich“. Der Modulooperator ist ‘%’ und ist nur auf Ganzzahlen anwendbar. Eine Eigenheit weist in C die Division auf: Sind beide Operanden Ganzzahlen (int, short, long oder char), so ist das Ergebnis wieder eine Ganzzahl, entspricht demnach dem Operator DIV aus Pascal. Beispiele:

```
17 / 3      ergibt    5
17.0 / 3    ergibt    5.666667
17 % 3      ergibt    2
```

<u>Vergleichsoperatoren: < <= > >= == !=</u>

Wie bereits erwähnt, ist das Ergebnis von Vergleichen entweder die int-Zahl 1 oder 0, je nachdem der Vergleich wahr oder falsch ist. Das Vergleichsergebnis darf also innerhalb eines Ausdrucks überall dort weiterverwendet werden, wo int-Werte zugelassen sind.

<u>Boolesche (logische) Verknüpfungsoperatoren: && || !</u>

‘&&’ entspricht der Und-Verknüpfung, ‘||’ der Oder-Verknüpfung und ‘!’ ist die logische Verneinung. Interessant ist, daß die beiden Operatoren ‘&&’ und ‘||’ schwächer binden als die Vergleichsoperatoren, so daß Klammerungen wie in Pascal nicht erforderlich sind. Natürlich sind aber aus Übersichtlichkeitsgründen trotzdem jederzeit Klammern erlaubt. Das folgende Beispiel ist demnach logisch korrekt:

```
if ( x > 0 || y++ < 1 ) …
```

Zu beachten ist, daß in C die beiden Operatoren ‘&&’ und ‘||’ eine feste **Abarbeitungsreihenfolge**

von links nach rechts garantieren. Ist etwa im obigen Beispiel der Wert von x positiv, so ist der gesamte Ausdruck mit Sicherheit wahr. Somit wird der rechte Teil garantiert nicht mehr durchlaufen. Der Wert von y wird hier demnach genau dann um 1 erhöht, wenn x nicht positiv ist!!

Der Nicht-Operator '!' arbeitet wie folgt:

$$!\,zahl = \begin{cases} 0 & falls \;\; Zahl \neq 0 \\ 1 & falls \;\; Zahl = 0 \end{cases}$$

Zuweisungsoperatoren: = += -= *= /= %= <<= >>= &= |= ^= ~=

Die Zuweisung '=' ist bekannt. Sie liefert als Ergebnis den Wert zurück, der der Variablen auf der linken Seite zugeordnet wurde.

In C gilt darüberhinaus folgende abkürzende Verwendung:

```
a  op=  b          ist äquivalent zu    a = a op ( b )
```

wobei 'op' ein arithmetischer Operator oder Bitoperator (siehe später) ist. Beispielsweise entspricht

```
i   /=   2 ;        der Zuweisung      i = i / 2 ;
```

Explizite und implizite Typumwandlung:

Enthalten bei binären Operatoren der linke und der rechte Operand Variablen unterschiedlichen Datentyps, so wird der „niederwertige" Typ automatisch in den „höherwertigen" umgewandelt. Anschließend erfolgt die Operation, die als Ergebnis den höherwertigen Typ zurückliefert.

Zu beachten gilt, daß in binären Operationen der „niederwertigste" Typ der Datentyp *int* ist, alle short- und char-Variablen werden in einem Ausdruck daher immer mindestens in einen *int* umgewandelt. Diesem „niederwertigsten" Typ *int* folgen *float*, *double* und *long double*. Die Behandlung von unsigned-Variablen ist etwas komplexer und wird hier nicht weiter verfolgt.

Bei Zuweisungen läßt C praktisch keine Wünsche offen: es ist alles erlaubt. Leider gibt es daher auch keinerlei Fehlermeldungen bei fehlerhafter Anwendung. Die Zuweisung:

```
Höherwertige Variable = Niederwertiger Ausdruck
```

stellt kein Problem dar. Doch auch die umgekehrte Form ist in C möglich:

```
Niederwertige Variable = Höherwertiger Ausdruck
```

Sind hier Variable und Ausdruck Gleitpunktzahlen, so werden eventuell Nachkommastellen verlorengehen. Kann der höherwertige Wert nicht in der niederwertigen Variable gespeichert werden, so gilt das Ergebnis als undefiniert (keine Fehlererkennung durch C!).

Ist die Variable eine Ganzzahl und der Ausdruck eine Gleitpunktzahl, so werden zunächst die Nachkommastellen abgeschnitten. Ist diese Zahl nicht als Ganzzahl darstellbar, so gilt das Ergebnis als undefiniert (keine Fehlererkennung durch C!).

Sind Variable und Ausdruck Ganzzahlen, so werden bei der Zuweisung in eine vorzeichenlose Variable gegebenenfalls die höherwertigen Bits abgeschnitten. Ansonsten ist das Ergebnis undefiniert, falls die Zahl in der Variable nicht vollständig darstellbar ist.

Dieses Verhalten von C ermöglicht bei unsauberer Programmierung das Einschleichen von Fehlern, die nur sehr schwer entdeckt werden können.

➔ Programmieren Sie strukturiert und defensiv und verwenden Sie bei Bedarf den Cast-Operator:

Cast-Operator:

Mittels des Cast-Operators erfolgt eine explizite Typumwandlung. Anwendung:

```
( Datentyp )  Ausdruck          /* in C und C++ */
Datentyp ( Ausdruck )           // nur in C++
```

Als Beispiel sei die Ausgabe der Sinuskurve aus dem Übungsblatt erwähnt. Wir können schreiben:

```
cout << setw ( int (12.0 * sin (i / 3.0) ) ) << '*' ;
```

Eine dringende Empfehlung:

➔ Verwenden Sie beim Umwandeln in einen niederwertigen Typ immer den Cast-Operator

Beispiel:

```
int i;  float x;
x = 3.1415;
i = (int) x;      /* hier wird klar, dass der Programmierer absichtlich die Umwandlung wollte */
```

Inkrement und Dekrement: ++ --

Diese beiden Operatoren können sowohl vor als auch hinter einer Variablen stehen:

```
i++    ++i    erhöhen den Wert von i um 1
i--    --i        vermindern den Wert von i um 1
```

Besondere Beachtung muß diesen Operatoren gezollt werden, wenn sie Teil eines größeren Ausdrucks sind. Dann gilt (analog für --):

```
i++    es wird i um 1 erhöht, allerdings wird der ursprüngliche Wert im Ausdruck verwendet
++i    es wird i um 1 erhöht, und dieser neue Wert wird im Ausdruck verwendet
```

Beispiele:

```
i = 5;
n = i++;              Inhalt der Variablen am Ende dieser Zeile: i=6, n=5
n = ++i;              Inhalt der Variablen am Ende dieser Zeile: i=7, n=7
```

Im Zusammenhang mit komplexeren Ausdrücken und den Inkrement- und Dekrement-Operatoren entstehen leicht undefinierte Ergebnisse. Es gilt in C nämlich analog zu Pascal, daß Resultate, die von der Reihenfolge der Abarbeitung abhängen, und diese Reihenfolge nicht explizit vorgegeben ist, undefiniert sind. Beispiele für undefinierte Anweisungen:

```
i = 2; k = (i++) + i;              k=4 oder k=5 ?
s[i] = i++;                        s[i]=i oder s[i+1]=i ?
s[i++] = i;                        s[i]=i oder s[i]=i+1 ?
```

Der Referenzoperator & in C++:

Wir haben den Referenzoperator bei der Parameterübergabe bereits kennengelernt. Er besitzt hier die Bedeutung des Var-Parameters in Pascal. Darüberhinaus kann er aber auch als Referenzvariable und zur Rückgabe von Referenzen in einer Funktion eingesetzt werden. Betrachten wir ein Beispiel:

```
int i = 1;
int &r = i;          // r ist eine Referenzvariable, r zeigt auf den gleichen Speicherinhalt wie i
r = 2;
cout << i;           // es wird der Wert 2 ausgegeben
```

Die Referenzvariable r ist genaugenommen ein konstanter Zeiger. Bei der Deklaration muß diese Referenzvariable daher mit einer existierenden Variable vorbelegt werden. Diese Vorbelegung ist nicht mehr veränderbar. Bei jeder Verwendung einer Referenzvariable erfolgt automatisch eine Dereferenzierung, so daß sich diese wie eine gewöhnliche Variable verhält. Im obigen Beispiel wurde r auf die Adresse von i gesetzt. Sowohl i als auch r beziehen sich damit auf den gleichen Speicherplatz. Eine Änderung dieses Speicherinhalts durch eine der beiden Variablen wirkt sich demnach immer auf beide Variablen aus. Salopp kann man sagen, daß eine Referenzvariable ein weiterer Variablenname ist, die sich den Speicher mit einer bereits existierenden Variable teilt. Und genau dies nützen wir im Falle einer Call-by-reference-Paramterübergabe aus. Hier erfolgt die Zuordnung zu einer existierenden Variable implizit durch den Funktionsaufruf.

Betrachten wir noch eine besondere Anwendung von Referenzen, um ihre Mächtigkeit zu erahnen: Referenzen dürfen auch als Funktionsergebnis zurückgeliefert werden. Das Ergebnis einer solchen Funktion ist daher nicht nur ein Ausdruck, sondern im Sinne der Syntax von C++ eine Variable. Schließlich identifiziert sich eine Referenz ja immer mit einer Variablen. Wir können auf den Funktionsaufruf daher sogar den Inkrementoperator '++' anwenden:

```
int & plus (int &a)
{  return  ++a;
}
int main ()
{  int i;
   cout << "Geben Sie bitte eine Ganzzahl ein: ";
   cin  >> i;
   cout << "Ergebnis: " << ++ plus(i) << '\n';
   return 0;
}
```

Programm: 2-einf/plus.cpp

Beachten Sie bei der Rückgabe des Funktionsergebnisses als Referenz, daß dieses Funktionsergebnis eine Variable mit eindeutig definiertem Inhalt sein muß! Der Bezeichner ‚++a' entspricht dieser Vorgabe: es handelt sich um die Variable a, die vorher um den Wert 1 erhöht wurde. Die Angabe ‚a++' hingegen wäre fehlerhaft. Hier handelt es sich um keinen sogenannten L-Value, ein Wert, der links vom Gleichheitszeichen stehen darf. Weitere Beispiele werden wir erst im Zusammenhang mit Klassen kennenlernen.

Weitere Operatoren:

In C sind auch die Klammern '(' und ')' und die Indizierung '[' und ']' Operatoren. Weitere Operatoren sind:

- * und &: Zeigeroperatoren
- -> und . (Punkt): Strukturoperatoren bei struct und union
- :: (2 Doppelpunkte): wird in Klassen verwendet. Damit können aber auch globale Variablen direkt angesprochen werden (nur in C++ vorhanden).
- ->* und .* : Operatoren in Klassen (nur in C++ vorhanden)
- new, delete: Dynamische Speicherverwaltung (nur in C++ vorhanden)
- , (Komma): Der Kommaoperator trennt zwei Ausdrücke voneinander, ähnlich dem Semikolon
- ‚? :' (Fragezeichen/Doppelpunkt): kann statt einer einfachen If-Anweisung in Ausdrücken verwendet werden. Beispiel:

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

- Bitoperatoren: Diese Operatoren sind in einer Systemprogrammiersprache notwendig. Damit können Ganzzahlvariablen bitweise manipuliert werden. Es gibt folgende Operatoren:

  << : Shiftet die Variable bitweise nach links (z.B. k = i << 2; /* Shift um 2 Bits nach links */)
  >> : Shiftet die Variable bitweise nach rechts
  & : Bitweise UND-Verknüpfung
  | : Bitweise ODER-Verknüpfung
  ^ : Bitweise XOR-Verknüpfung
  ~ : Bitweise Negation

Ein Beispiel zur XOR-Verknüpfung ist die Vertauschung zweier Variablen:

```
   /* Implementierung des Vertauschens mit XOR */
#include <iostream.h>
void vertausche (int &a, int &b)
{  a ^= b;
   b ^= a;
   a ^= b;
}
```

```
int main()
{  int i = 10, j = 15;
   cout << "\n\ni = " << i << ", j = " << j << '\n';

   vertausche ( i, j);

   cout << "i = " << i << ", j = " << j << endl;
   return 0;
}
```

Programm: 2-einf/xor.cpp

– sizeof-Operator: gibt die Bytes an, die der Operand an Speicher belegt. Er kann sowohl auf Variablen als auch auf Datentypen angewendet werden. Im letzteren Fall ist der Datentyp in Klammern zu setzen und es wird ermittelt, wieviel Speicher eine Variable dieses Typs belegen würde.

Zusammenfassung der Operatoren mit Hierarchie

	Operator	Ass.
Spezielle Op.	( ) , [ ] , -> , . , :: (nur C++)	LR
Unäre Operatoren	! , ~ , ++ , -- , + , - *(Vorzeichen)* , * , & *(Zeiger)* , (cast) , sizeof , new (nur C++) , delete (nur C++)	RL
Klassenop.	->* (nur C++) , .* (nur C++)	LR
Arithm. Op.	* , / , %	LR
Arithm. Op.	+ , -	LR
Shift	<< , >>	LR
Vergleiche	< , <= , > , >=	LR
Vergleiche	== , !=	LR
Bit Op.	&	LR
Bit Op.	^	LR
Bit Op.	|	LR
Logischer Op.	&&	LR
Logischer Op.	||	LR
If	? :	RL
Zuweisungen	= , += , -= , *= , /= , %= , &= , ^= , |= , <<= , >>=	RL
Komma	, (*Komma*)	LR

In obiger Tabelle sind alle C++-Operatoren angegeben, geordnet nach Bindungsstärke, beginnend bei den am stärksten bindenden Operatoren. Die Angaben LR bzw. RL geben die Ausführungsreihenfolge (Assoziativität) innerhalb der gleichen Bindungsstärke an (RL = Assoziativität von rechts nach links, LR = Assoziativität von links nach rechts). Assoziativität darf nicht mit Bewertung verwechselt werden. Nur die Operatoren '&&', '||', '?:' und ',' (Komma) garantieren eine Bewertung von links nach rechts!

Trotz dieser vielen Operatoren und Hierarchiestufen sind die Grundregeln einfach. Neben den Klammern, Feld- und Strukturoperatoren binden die unären Operatoren am stärksten, gefolgt von den arithmetischen Operatoren. Vergleiche binden stärker als die logischen Operatoren. Mit am schwächsten binden die Zuweisungen. Beachten Sie, daß nur unäre Operatoren und Zuweisungen (und ?:) von rechts nach links zusammengefaßt werden. Die Assoziativität kann natürlich mittels Klammerung geändert werden.

Manchmal kommt es vor, daß zwei oder mehr Operatoren direkt hintereinander stehen. Werden zwischen diesen Operatoren keine Trennzeichen (White-Spaces) geschrieben, so faßt C von links nach rechts so viele Zeichen zu einem Operator zusammen, solange dies Sinn ergibt. Beispiel:

```
c += a +++ b;      bedeutet:      c+= (a++) + b;
```

Umgekehrt ist der folgende Ausdruck ein Fehler:

```
c += a ++ b;      /* Fehler */
```

Denn C faßt die beiden Pluszeichen zum Inkrementoperator zusammen. Es gilt also <u>nicht:</u>

```
c += a + (+b);
```

2.11 Weitere Möglichkeiten in C++

Wir haben bereits gesehen, daß C++ eine echte Erweiterung von C ist. Allerdings lassen sich „schlampig" geschriebene C-Programme wegen der strengeren Typüberprüfung in C++ nicht immer leicht nach C++ konvertieren. Besonders empfehlenswert ist es, die Ein-/Ausgabeströme von C++ zu nutzen, aber auch der Referenzoperator ist in C++ eine wertvolle Unterstützung. Doch es gibt weitere wertvolle Erweiterungen in C++, die noch nichts mit Objektorientierung zu tun haben:

Variablendeklarationen:

In C++ dürfen Variablen nicht nur am Anfang eines Blocks deklariert werden. Dies kann leider auch mißbraucht werden, so daß der Code sogar schwerer lesbar wird. Es sollte nur dann davon Gebrauch gemacht werden, wenn eine Variable nur an einer ganz bestimmten Stelle nur sehr lokal benötigt wird. Dies kann etwa für einen Schleifenzähler der Fall sein. In C++ kann dann geschrieben werden:

```
for ( int i = 0; i< N; i++ )   …              // kein C-Code!
```

Es sei aber vor dieser Anwendung gewarnt. Es ist nämlich in C++ nicht zweifelsfrei definiert, ob eine so definierte Variable i nur innerhalb der For-Schleife oder global definiert ist. Hier können also bei der Portierung eines Programms von einem zum anderen Compiler Fehler auftreten!

Inline-Funktionen:

Wird eine Funktion mit der Direktive 'inline' versehen, so ersetzt der Compiler den Funktionsaufruf durch die Funktion selbst. Dies bedeutet, daß im erzeugten Code kein Funktionsaufruf, sondern die Funktion selbst an diese Stelle kopiert wird. Enthält ein Programm mehrere dieser Funktionsaufrufe, so wird jedes Mal die Funktion an diesen Platz kopiert. Dies erhöht den Programmcode, reduziert aber die Laufzeit, da die laufzeitintensiven Funktionsaufrufe entfallen.

Es ist allerdings ein Fehler, die Direktive 'inline' zu verwenden, wenn nicht alle Funktionsaufrufe in dieser Funktion selbst Inline-Funktionen sind. Außerdem dürfen Inline-Funktionen keine rekursiven Aufrufe enthalten. Betrachten wir ein Beispiel:

```
inline int max (int a, int b)          // Maximum zweier Zahlen ermitteln
{   return (a>b) ? a : b;
}
…
x = max (3, 2);      // wird ersetzt durch den (Maschinen-)Code     x = (3>2) ? 3 : 2 ;
y = max (i+j, k);    // wird ersetzt durch den (Maschinen-)Code     y = (i+j>k) ? i+j : k ;
```

Die Anwendung von Inline-Funktionen vor allem im Schreiben kleiner Funktionen, wobei der teure Funktionsaufruf vermieden wird.

Vorbelegen von Parametern

Gelegentlich werden Funktionsaufrufe mit einer variablen Zahl von Parametern gewünscht. Dies wird auch von C unterstützt, hat aber den Nachteil, daß der Compiler die Parameterübergabe dieser C-Funktionen nicht auf korrekte Datentypangaben überprüfen kann. Denken wir nur an *printf*. Dessen Deklaration lautete:

```
int printf (char * s,  … );
```

Eine einfache und handliche Möglichkeit bietet nun C++, Parameter einer Funktion vorzubelegen. Werden diese Parameter beim Funktionsaufruf nicht angegeben, so werden einfach die Vorbelegungswerte gewählt. Beispiel:

```
int add (int i, int j, int k=0, int l=0)
{  return i+j+k+l;
}
```

```
int main ()
{   int u = 5;
    int v = 10;
    int w = 3;
    cout << "u = " << u << ", v = " << v << ", w = "
         << w << "\n\n";
    cout << "      u+v = " << add (u,v)       << '\n';
    cout << "    u+v+w = " << add (u,v,w)     << '\n';
    cout << "u+v+w+w*w = " << add (u,v,w,w*w) << '\n';
    return 0;
}
```

Programm: 2-einf/add.cpp

In diesem Beispiel sind die ersten beiden Parameter nicht vorbelegt, die anderen beiden sind es. Ein Aufruf der Funktion *add* umfaßt daher in diesem Beispiel mindestens zwei und maximal vier Parameter.

Wichtig ist, daß beim Definieren von Funktionen alle nichtvorbelegten Parameter vor den vorbelegten aufgezählt werden müssen. Folgende Funktion wäre demnach fehlerhaft:

```
int add ( int i, int j, int k=0, int l=0, int m );        // Fehler, m ist nicht vorbelegt!
```

Mit Hilfe der Vorbelegung haben wir in C++ also die Möglichkeit, eine Funktion mehrfach zu verwenden, sozusagen zu „überladen“. Weitere Möglichkeiten werden wir in der objektorientierten Programmierung kennenlernen.

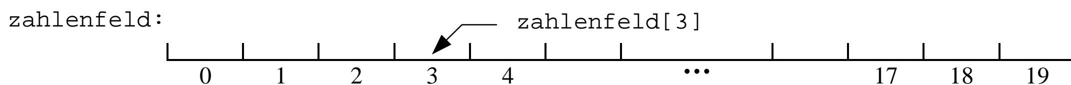
3 Zeiger und Felder

3.1 Felder

Beginnen wir gleich mit der Definition eines Feldes von 20 int-Zahlen:

```
int zahlenfeld [20] ;
```

Mit dieser Definition wird folgender Speicher angelegt:



In C beginnt der Index immer bei 0, das letzte Feldelement ist demnach zahlenfeld[19]! Ansonsten ist die Ähnlichkeit zu Pascal sehr hoch. Denken wir nur an die Indizierung mit den eckigen Klammern.

Beispielprogramm: Es sollen bis zu 20 Zahlen eingelesen, sortiert und dann wieder ausgegeben werden. Dieses Programm enthält eine Neuigkeit:

- Als Ausnahme werden Felder (und Zeichenketten) immer call-by-reference übergeben. Es entfällt die Übergabe der Adresse mittels des ‘&’-Zeichen bzw. die Call-by-Reference Übergabe in C++. Ebenso ist die Angabe der Größe des Feldes im Unterprogramm nicht erforderlich, also nicht ausführlich ‘int feld[20]’. Wird die Größe doch übergeben, so wird sie von C ignoriert.

Die Funktion *sort* wurde vorab deklariert, indem nur der Funktionskopf, gefolgt von einem Semikolon geschrieben wurde. C erwartet dann die Definition dieser Funktion in einem anderen Modul oder weiter unten in der gleichen Datei. Die Funktion darf im folgenden sofort benutzt werden.

Das folgende Hauptprogramm sollte weiter keine Schwierigkeiten bereiten. Interessant ist vielleicht neben der Verwendung von Feldern noch das Abbruchkriterium in der Do-While-Schleife. Die Programmierung ist zugegebenermaßen trickreich, behandelt aber jeden nur möglichen Fall korrekt. Es sei als Übung empfohlen, dieses Abbruchkriterium genau nachzuvollziehen.

```
#include <iostream.h>

void sort ( int feld[], int max );
// sortiert feld, Definition erst spaeter

int main()
{  int zaehler = 0;
   int zahlenfeld[20];    /* Zahlenfeld */
   int i;
   cout << "\n\nSortierung von Zahlen\n\n"
        << "Geben Sie Ganzzahlen ein. Abbruch durch 0.\n";
   do
   {  cout << "Zahl " << zaehler+1 << "d: ";
      cin >> zahlenfeld[zaehler];
   } while ( zahlenfeld[zaehler] != 0 && ++zaehler < 20 );
                              // Abbruch sehr geschickt !
   sort (zahlenfeld, zaehler);      // nicht &zahlenfeld !
   cout << "Sortierte Ausgabe:\n";
   for (i=0; i<zaehler; i++)
      cout << zahlenfeld[i] << '\n';
   return 0;
}
```

Programm: 3-felder/sort.cpp

Betrachten wir noch die wichtige Sortierfunktion in diesem Programm:

```
// Bubblesort:

void sort (int feld[], int max)
{  int i, j, var;

   for (i=1; i<max; i++)
      for (j=max-1; j>=i; j--)
         if (feld[j] < feld[j-1])
         {                                /* vertausche: */
            var        = feld [j-1];
            feld [j-1] = feld [j];
            feld [j]   = var;
         }
}
```

Programm: 3-felder/sort.cpp

Wir finden hier den Bubblesort wieder. Wir sehen, daß die Programmierung von Feldern bisher fast identisch zu Pascal erfolgt. Weitere Möglichkeiten in C sind:

– Initialisieren von Feldern, etwa

  char hexfeld[16] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };

– Mehrdimensionale Felder (siehe Abschnitt 3.4), etwa

  int matrix [10] [8];

– Zeichenketten sind char-Felder, die mit der binären Null '\0' enden, siehe nächster Abschnitt.

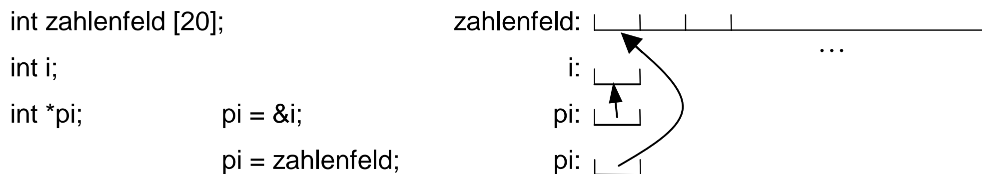
Wegen der gewöhnungsbedürftigen Felderindizierung, wie hier von 0 bis 19 bei 20 Elementen, empfehle ich in For-Schleifen dringend die Programmierung mit asymmetrischen Grenzen. Wir haben sie bereits verwendet. Zur Vertiefung sei folgendes Beispiel der Ausgabe einer Matrix angegeben:

```
int matrix [10] [8];
for ( i=0; i < 10; i++ )          /* nicht  i <= 9  !!! */
{   for ( j=0; j < 8; j++ )       /* nicht  j <= 7  !!! */
        cout << matrix [i] [j] ;
    cout << '\n' ;
}
```

Nur dieser Stil garantiert den Bezug zu den oberen Grenzen des Feldes, hier 10 und 8!

Wir hätten damit die wichtigsten Grundkenntnisse zu Feldern erarbeitet. Die enorme Flexibilität von Feldern in C erhalten wir erst dadurch, daß wir mit Zeigern auf Felder zugreifen können. Vergleichen wir

folgende Deklarationen:



Es gilt in UNIX (in MS-DOS wegen der 2 Byte-int-Länge die Hälfte):

```
sizeof zahlenfeld = 80;        // Speicher für das Feld
sizeof i = 4;                  // Speicher für die int-Variable
sizeof pi = 4;                 // Speicher für die Adresse
```

Um das gesamte Zahlenfeld zu kennen, muß C eigentlich nur die Adresse des Beginns des Feldes und die Größe des Feldes kennen. Aus diesem Grund können wir die Variable *zahlenfeld* auch als die konstante unveränderliche Adresse des Feldes ansehen, die zusätzlich die Feldgröße beinhaltet. Enthält diese Variable tatsächlich eine Adresse, so wäre sie mit *&i* oder *pi* zu vergleichen. Folglich müßte **zahlenfeld* den Inhalt des ersten Elements des Feldes repräsentieren. **zahlenfeld* und *zahlenfeld[0]* müßten demnach äquivalente Ausdrücke sein. Dies ist tatsächlich der Fall. Es gilt nämlich:

Mit der Deklaration eines Feldes wird ein entsprechend großes Feld reserviert. Der Name dieses Feldes (etwa *zahlenfeld*) repräsentiert in C eine nicht veränderbare Adresse, die auf den reservierten Bereich zeigt. Wir bezeichnen daher den Namen eines Feldes ab jetzt auch als <u>Feldzeiger</u>. Zusätzlich wird mit diesem Namen noch die Größe des Feldes identifiziert.

Wir haben bereits gesehen, daß gilt:

```
zahlenfeld [0]    ≡    * zahlenfeld
```

Doch wir können auch alle anderen Feldelemente allein über Zeiger ansprechen. C besitzt dazu eine Zeigerarithmetik: zu einem Zeiger darf eine Ganzzahl addiert oder subtrahiert werden, ebenso dürfen zwei Zeiger voneinander subtrahiert oder miteinander verglichen werden.

Zeigerarithmetik:

1) Addition (Subtraktion) eines Zeigers *p* mit einer Ganzzahl *i*: Das Ergebnis ist ein Zeiger, der die Adresse *p* zuzüglich (abzüglich) *i* Felder enthält. Beachten Sie, daß zu *p* nicht einfach die Zahl *i* addiert (subtrahiert) wird, sondern *i*sizeof(Typ von p)*, bei int-Zeigern also beispielsweise *4*i*.
2) Subtraktion zweier Zeiger: beide Zeiger müssen vom gleichen Datentyp sein. Als Ergebnis erhalten wir die Anzahl der Elemente, die zwischen beiden Adressen Platz haben.
3) Vergleich von Zeigern: Alle Vergleichsoperatoren sind erlaubt. Ein Zeiger besitzt einen um so größeren Wert, je höher die Adresse ist.

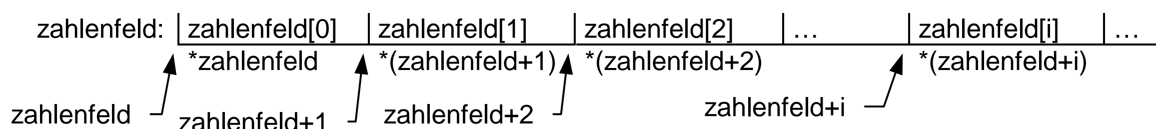
Mit dieser Arithmetik und den Feldzeigern erhalten folgende Zuweisungen einen Sinn:

```
i = 2;
pi = zahlenfeld + i;      /* pi zeigt auf das 3. Element von zahlenfeld */
pi++;                     /* entspricht pi = pi+1;  pi zeigt auf das 4. El. von zahlenfeld */
pi[-1] = 2;               /* das Vorgängerelement (3. El.) erhält den Wert 2 */
i = pi - zahlenfeld;      /* i=3 */
if (pi >= zahlenfeld) …
```

Für uns sehr wichtig ist, daß folgende beiden Ausdrücke vollkommen <u>äquivalent</u> sind:

Wichtig:

➔ zahlenfeld [i] = * (zahlenfeld + i)
➔ & zahlenfeld [i] = zahlenfeld + i



Wir können folglich immer die Indexschreibweise vollständig durch die Zeigerschreibweise ersetzen und umgekehrt. Betrachten wir ein Beispiel:

```
#include <iostream.h>

/* Summe von Zahlen mit Feldindizierung: */

int summe1 ( int feld [] )
{  int i, sum = 0;
   for ( i=0; i<10; i++ )
      sum += feld[i];         /* Addition der 10 Zahlen */
   return sum;
}

/* Summe von Zahlen mit Zeigerschreibweise: */

int summe2 ( int *feld )
{  int *pend, sum = 0;          /* pend speichert Feldende */
   for ( pend = feld + 10; feld < pend; feld++ )
      sum += *feld;
   return sum;
}
```

Programm: 3-felder/summe.cpp

Beide Funktionen *summe1* und *summe2* sind gleichwertig. Beachten Sie in der Funktion *summe2*, daß der Zeiger *feld* eine Kopie des Feldzeigers *zahlenfeld* ist, und somit verändert werden darf. Vorher wird ein Hilfszeiger auf das Ende des Feldes gesetzt. Dabei legt ANSI-C fest, daß die Adresse direkt hinter einem Feld (im Programm *feld+10*) immer existiert. Es darf aber keine Dereferenzierung mehr erfolgen (verboten: **(feld+10)*). In der Funktion *summe2* wird nun durch Fortschalten des Zeigers *feld* das Zahlenfeld durchlaufen, und zwecks Abbruchkriterium werden zwei Zeiger miteinander verglichen.

Der Aufruf der beiden Funktionen ist identisch:

```
sum1 = summe1 (zahlenfeld);
sum2 = summe2 (zahlenfeld);
```

Da *zahlenfeld* bereits ein Zeiger (Feldzeiger) ist, darf nicht '&zahlenfeld' übergeben werden. Die Variablen *sum1* und *sum2* enthalten jeweils den gleichen Wert.

Wir haben bisher nur gesehen, daß Felder wie Zeiger verwendet werden können. Umgekehrt kann ein Zeiger auch auf Felder zeigen, ebenso kann bei Zeigern natürlich auch die Indexschreibweise verwendet werden. Um den dennoch vorhandenen kleinen Unterschied zwischen Zeigern und Feldern etwas hervorzuheben, betrachten wir die 6 Möglichkeiten, Zeiger oder Felder zu deklarieren:

```
int a [ 20 ] ;                  // Konstantes Feld mit 20 int-Elementen
int b [ ] ;                     // Konstantes Feld mit 0 int-Elementen, sinnlos
int * c ;                       // nur Zeiger, kein dazugehöriges Feld!
int d [ 20 ] = {1, 2, 4 } ;     // Konstantes Feld mit Wertzuweisung: 1, 2, 4, 0, 0, 0, …
int e [ ] = {1, 2, 4 } ;        // Konstantes Feld mit 3 int-Elementen, Wertzuweisung
int * f = {1, 2, 4 } ;          // Zeiger zeigt auf Feld, das aus den 3 El. besteht
```

Wir sehen, daß beim Feld *e* genauso viele Elemente reserviert werden, wie in der Aufzählung angegeben sind. Zwischen Feldern (*a*, *b*, *d*, *e*) und Zeigern (*c*, *f*) bestehen gewisse Unterschiede. Zum einen durch die Anwendung des Sizeof-Operators, zum andern wegen der Unveränderbarkeit der Feldzeiger. Fassen wir zusammen:

Vergleich zwischen Feldzeiger *fp* und Zeiger *p*:

- „echter" Zeiger *p*: *p* ist eine Variable oder Parameter, besitzt also einen Speicherplatz. Es kann die Adresse dieses Speicherplatzes verwendet werden (*&p*), ebenso darf *p* verändert werden (p=…). Der Zeiger *p* muß immer erst initiiert werden, bevor er dereferenziert werden darf (**p*).
- Feldzeiger *fp*: *fp* repräsentiert ein Feld (*sizeof fp* ist die Feldlänge) und enthält bereits bei der Deklaration die Feldadresse. *fp* selbst ist jedoch keine Variable und für den Benutzer auch nicht zu-

gänglich. Daher sind Zuweisungen (fp=…) und die Adresse von *fp* (*&fp*) nicht möglich und führen zu einem Syntaxfehler!

– Gemeinsamkeiten: *fp* und *p* sind Zeiger auf einen Datentyp (z.B. auf *int* oder *float*). Zeigerarithmetik darf verwendet werden, also auch die Indizierung.

Diese Unterschiede und Gemeinsamkeiten werden beim Vergleich der Variablen *e* und *f* erkennbar. Ausdrücke der Form *e[1]* und *f[1]* sind jeweils erlaubt (Zeigerarithmetik), wobei jeweils das zweite Element angesprochen wird (Gemeinsamkeiten). Andererseits kann nur *f* verändert werden, etwa durch *f++*. Dafür liefert *sizeof e* die Länge des Feldes.

Der Operator *sizeof* liefert bei *a* und *d* den Wert 80, bei *b* den Wert 0, bei *e* den Wert 12 und bei *c* und *f* den Wert 4 zurück. Als einziger der obigen 6 (Feld-)Zeiger besitzt *c* keinen definierten Inhalt. *c* muß demnach erst eine Adresse erhalten, bevor zugegriffen wird.

Jetzt verstehen wir, warum Felder automatisch als call-by-reference in Funktionen übergeben werden, schließlich übergeben wir ja nur die Zeiger und nicht das Feld selbst. Auch die Parameter der Funktion enthalten demnach nur Zeiger. Betrachten wir wieder unsere 6 obigen Variablen *a* bis *f*. Da bei Parametern keine Wertzuweisung erfolgt, kommen als Parameter nur die Variablen *a*, *b* und *c* in Frage. Weiter interessiert bei der Parameterübergabe nur die Adresse und der Datentyp. Eine Angabe der Größe des Feldes ist bei der Parameterübergabe daher sinnlos. Es bleiben daher nur *b* und *c* übrig. Beide sind als Parameterdeklarationen im Funktionskopf äquivalent und repräsentieren einen Zeiger auf den Datentyp.

Betrachten wir zur Übung noch einige Beispiele:

```
c = a + 3;        // Zeiger c zeigt auf das 4. El. von Feld a
c[-2] = 4;        // Das 2. El. von a erhält den Wert 4
* (d+4) = 9;      // Das 5. El. von d erhält den Wert 9
f++;              // f zeigt jetzt auf das 2. El.
// e++;           wäre falsch, da e nicht verändert werden kann, e ist Feld!
```

Haben Sie jetzt alles über Zeiger verstanden? Wenn Sie glauben, daß ja, so betrachten Sie doch als quasi Abschlußtest noch folgendes obskure Beispiel mit *i* als int-Variable und *a* als Zeiger:

```
a [ i ]  ≡  * ( a + i )  ≡  * ( i + a )  ≡  i [ a ]
```

Frage: Wo ist der Fehler? Oder ist doch alles korrekt? Antwort: in der Vorlesung.

3.2 Zeichenketten

Definition: Eine Zeichenkette ist ein char-Feld, das mit einer binären Null (‘\0’) abgeschlossen wird.

Folgende beiden Felder sind demnach gleichwertig:

```
"hallo\n"        { 'h', 'a', 'l', 'l', 'o', '\n', '\0' }
```

Wir können jetzt endlich den Unterschied zwischen "A" und 'A' erläutern. Das erste ist die Zeichenkette {'A', '\0'}, das zweite das einzelne Zeichen 'A'. Das erste ist demnach ein Feld aus zwei Elementen!

Die Mächtigkeit der Zeichenketten ergibt sich ähnlich zu Turbo-Pascal daraus, daß wir Zeichenketten vergleichen, zuweisen, und anderweitig verknüpfen können. Doch im Gegensatz zu Turbo-Pascal gehören diese Möglichkeiten nicht zum Sprachumfang von C. Sie werden vielmehr über Funktionen realisiert, die in der Headerdatei *string.h* deklariert sind. Doch beginnen wir bei der Deklaration. Wie bei Feldern stehen 6 Möglichkeiten zur Auswahl:

```
char a [ 20 ] ;             // Konstantes Feld mit 20 char-Elementen
char b [ ] ;                // Konstantes Feld mit 0 char-Elementen, sinnlos
char * c ;                  // nur Zeiger, kein dazugehöriges Feld!
char d [ 20 ] = "hallo" ;   // Wertzuweisung: hallo
char e [ ] = "hallo" ;      // Konstantes Feld mit 6 (!) char-Elementen, Wertzuweisung
char * f = "hallo" ;        // Zeiger zeigt auf Feld, das aus den 6 El. besteht
```

In den letzten beiden Fällen werden jeweils 6 char-Elemente reserviert, da wir '\0' nicht vergessen dürfen. Beachten wir wieder den wichtigen Unterschied zwischen den Variablen *e* und *f*. *e* ist nicht veränderlich und liefert bei *sizeof* den Wert 6 zurück. Der Zeiger *f* hingegen kann verändert werden und liefert bei *sizeof* den Wert 4 zurück (Speicher für die Adresse). Kommen wir jetzt zu einigen Zeichenkettenfunktionen:

```
#include <string.h>
char * strcat ( char *s1, char *s2 )     // hängt s2 an s1 an
char * strcpy ( char *s1, char *s2 )     // kopiert s2 nach s1
int strcmp ( char *s1, char *s2)         // = 0 , falls s1 gleich s2
                                         // < 0 , falls s1 kleiner als s2 (nach ASCII-Code)
                                         // > 0 , falls s1 größer als s2
size_t strlen ( char *s )                // Länge des Strings (ohne '\0')
```

Beachten Sie, daß beim Kopieren oder Konkatenieren im Feld, auf das *s1* zeigt, genügend Speicher reserviert ist. Dies wird von C nicht überprüft und führt gegebenenfalls zu Fehlern wegen Speicherüberschreibungen. Kopiert wird inklusive der binären Null, bei der Konkatenierung wird bei *s1* vorher die binäre Null entfernt. Der Datentyp *size_t* ist in der Headerdatei *string.h* definiert. Er entspricht in der Regel dem Typ *unsigned int*.

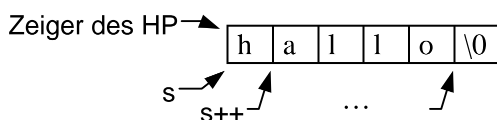
Die Variablen *d*, *e* und *f* aus obigem Beispiel liefern als Stringlänge jeweils den Wert 5 zurück. Es interessiert für die Stringlänge demnach nicht die Länge des definierten Feldes oder die Form der Definition, sondern die Länge bis zum Auftreten von ‘\0’.

Betrachten wir ein Beispiel zur Zeichenkettenverarbeitung, eine Implementierung der Funktion *strlen*:

```
/* strlen: gibt die Laenge des Strings s zurueck */

int strlen (char *s)
{
   int n;

   for (n=0; *s != '\0'; s++)
      n++;

   return n;
}
```

Programm: 3-felder/strlen.cpp

Betrachten wir das Verhalten dieser Funktion *strlen*, wenn die Zeichenkette "hallo" übergeben wird.



Es wird der übergebene Zeiger *s* solange fortgeschaltet, bis die binäre Null erreicht wird. Der mitlaufende Zähler *n* wird am Schluß zurückgegeben.

Im folgenden Beispiel wandeln wir eine int-Zahl *n* in eine Zeichenkette *s* um. Die Funktion *itos* erhält die Zahl *n* und gibt den String *s* zurück. Die in der Funktion *itos* am Schluß verwendete Hilfsfunktion *reverse* dreht den Inhalt eines Strings um. In beiden Funktionen wurde diesmal die Index-Schreibweise verwendet. Dies verdeutlicht nochmals die Gleichwertigkeit beider Schreibweisen. Je nach Bedarf und eigenem Stil kann die Zeiger- oder Indexschreibweise gewählt werden. In diesem Programm ist noch dringend zu beachten, daß die Funktion *itos* davon ausgeht, daß das aufrufende Programm hinreichend Speicher für die Zeichenkette *s* zur Verfügung stellt. Wenn nicht, so liegt ein Fehler vor, der zum Überschreiben von anderweitig reservierten Speicherbereichen führt.

```
void itos (char s[], int n)
{
   int i = 0;
   int vorzeichen;

   if ( (vorzeichen = n) < 0 )     // merkt das Vorzeichen
      n = -n;

   do                          // erzeugt umgekehrten String
      s[i++] = n % 10 + '0';
   while ( (n /= 10) > 0 );          // Abbruch

   if (vorzeichen < 0)
      s[i++] = '-';

   s[i] = '\0';                // explizites Stringende

   reverse (s);                // dreht String um
}
```

Programm: 3-felder/itos.cpp (Funktion *itos*)

Betrachten wir jetzt noch die in der Funktion *itos* verwendete Funktion *reverse*:

```
/* dreht einen String um: */

void reverse (char s[])
{
   int c, i, j;

   for (i=0, j=strlen(s)-1; i<j; i++, j--)
      c= s[i], s[i] = s[j], s[j]=c;
}
```

Programm: 3-felder/itos.cpp (Funktion *reverse*)

Betrachten wir noch ein Beispiel, aus der die Möglichkeiten der Zeigerdarstellung erst richtig erkennbar werden. Im folgenden Programm wurden gleich vier Strcpy-Funktionen definiert. Alle vier Funktionen sind vollkommen gleichwertig. Hier erkennen wir erstmals deutlich den Vorteil der Zeigerschreibweise.

```
/* strcpy1: Kopieren in der Feldschreibweise: */
void strcpy1 (char s1[], char s2[])
{  int i;
   for (i=0; s2[i] != '\0'; i++)
      s1[i] = s2[i];
   s1[i] = '\0';         /* abschliessende binaere Null */
}

// strcpy2-4: Kopieren in der Zeigerschreibweise:
void strcpy2 (char *s1, char *s2)
{  while ( (*s1 = *s2) != '\0' )
   {  s1++; s2++;
   }
}

void strcpy3 (char *s1, char *s2)     // Kurze Schreibweise
{  while ( (*s1++ = *s2++) != '\0' )
   ;
}

void strcpy4 (char *s1, char *s2)     // noch kuerzer
{  while ( *s1++ = *s2++ )
   ;
}
```

Programm: 3-felder/strcpy.cpp

Zum Verständnis des Ausdrucks in der While-Schleife von *strcpy3* und *strcpy4* müssen wir in Erinnerung rufen, daß unäre Operatoren von rechts nach links bewertet werden. **s1++* bedeutet demnach **(s1++)*. Das Inkrement bezieht sich demnach auf den Zeiger und nicht auf den Inhalt. Der Ausdruck in *strcpy4* bedeutet folglich:

```
*s1 = *s2;        /* Kopieren der Inhalte */
s1++; s2++;       /* Inkrement der Zeiger: fortschalten zum nächsten Zeichen */
s1 != '\0'        /* Überprüfen auf Stringende */
```

Dies ist aber das Gleiche wie in *strcpy2* oder *strcpy3*. Es sei nochmals auf den Unterschied zwischen dem Zeichen '0' und dem Zeichen '\0' hingewiesen. Ersteres ist der ASCII-Wert des Zeichens Null, das zweite ist in C eine Darstellung für den ASCII-Wert Null, also die binäre Null! Wir wissen weiter, daß C die binäre Null als FALSE interpretiert und alle anderen Zahlen als TRUE. Daher ist die verkürzte Schreibweise von *strcpy3* zu *strcpy4* legitim.

Wichtige Ein-/Ausgabefunktionen in C (stdio.h) für Zeichenketten sind:

- **printf** mit Formatangabe '%s': gibt die angegebene Zeichenkette aus (ohne '\0').
- **scanf** mit Formatangabe '%s': liest eine Zeichenkette in die Zeigervariable ein (keinen Adreßoperator & verwenden!). Es wird ein '\0' angehängt. *scanf* liest eine Zeichenkette nur bis zum nächsten White-Space!
- **puts (char *s)** gibt die Zeichenkette s aus, wobei '\0' in ein '\n' verwandelt wird
- **gets (char *s)** liest eine ganze Zeile in die Zeichenkette *s* ein, wobei '\n' in '\0' verwandelt wird.

Wichtige zusätzliche Ein-/Ausgabefunktionen in C++ (iostream.h) für Zeichenketten sind:

- **cout** erkennt automatisch Zeichenketten und gibt sie aus (ohne '\0').
- **cin** liest immer eine Zeichenkette in eine Zeichenkettenvariable (Zeiger oder Feld) ein. Es wird ein '\0' angehängt. *cin* liest eine Zeichenkette nur bis zum nächsten White-Space!
- **cout.write (char *s, int n)** gibt genau n Zeichen der Zeichenkette s aus.
- **cin.get (char *s, int n)** liest eine ganze Zeile in die Zeichenkette *s* ein; hat die Zeile mehr als n-1 Zeichen, so werden nur n-1 Zeichen gelesen. Läßt das Returnzeichen '\n' im Dateipuffer!
- **cin.getline (char *s, int n)** entfernt im Unterschied zu *cin.get* das Zeichen '\n' aus dem Dateipuffer! Beim Einlesen von mehreren Zeilen hintereinander sollte unbedingt *getline* verwendet werden.

Betrachten wir zwei weitere Beispiele. Wir wollen zum einen alle Leerzeichen am Ende eines Strings und zum anderen alle Leerzeichen am Anfang eines String löschen. Beginnen wir mit dem Löschen am Ende. Hier ist der Algorithmus schon die halbe Miete:

Gehe zum Ende des Strings !!!
Gehe rückwärts zum nächsten Zeichen ungleich Leerzeichen, maximal bis Dateianfang !
Schreibe '\0' in das folgende Zeichen

Das Programm *loeschende* ergibt sich jetzt zu:

```
#include <iostream.h>
#include <string.h>         /* enthaelt String-Funktionen */

void loeschende (char *s)
{  char *p;

   // Beginne mit der Suche beim Stringende !
   p = s + strlen(s) - 1;

   while (p >= s && *p == ' ')
       p--;

   *(++p) = '\0';   // '\0' hinter letztem Zeichen einfuegen
}
```

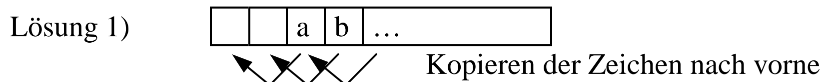
Programm: 3-felder/loesch.cpp (Funktion *loeschende*)

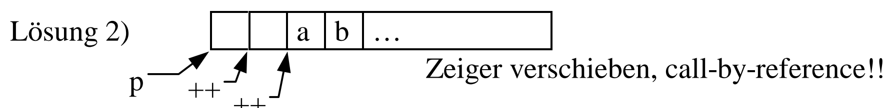
Beachten Sie, daß wir bei einer Zeichenkette, die nur aus Leerzeichen besteht, nicht über den linken Rand hinauslaufen. Die Schleife in der Funktion *loeschende* überprüft daher auf Leerzeichen und auf

Zeichenkettenanfang. Wir sehen, daß auch kleine Funktionen schon recht unangenehme Fallstricke enthalten können. Überlegen Sie selbst, daß folgende Variante (Funktion *loeschende* ab der While-Schleife) dann fehlerhaft ist, wenn die gesamte Zeichenkette nur aus Leerzeichen besteht:

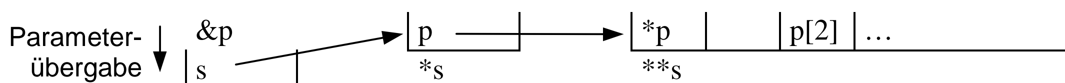
```
while (*p = ' ')
    p--;
*(++p) == '\0';
```

Im zweiten Beispiel, dem Löschen von führenden Leerzeichen, bieten sich zwei Lösungen an, zum einen das Suchen des ersten Zeichens ungleich Leerzeichen und des folgenden zeichenweisen Nach-Vorne-Kopierens. Zum anderen könnte der Zeiger einfach auf das erste Zeichen ungleich Leerzeichen gesetzt werden. Dies setzt allerdings voraus, daß ein variabler Zeiger (also kein Feldzeiger) vorliegt. Weiter gehen eventuell die führenden Leerzeichen als Speicherplatz verloren.





Lösung 2 birgt ein gewisses Problem. Der Zeiger selbst wird verändert. Dieser muß demnach call-by-reference übergeben werden. Wir übergeben also die Adresse eines Zeigers! Ist demnach *p* ein Zeiger auf ein char-Feld, so wird an die Funktion *&p* übergeben. Ist andererseits in der Funktion selbst die Variable *s* ein Zeiger auf eine Zeichenkette, so ist **s* die Zeichenkette, also ein Zeiger auf ein Zeichen. Schließlich ergibt ***s* das erste Zeichen der Zeichenkette!



Glücklicherweise besitzen wir mit C++ wieder eine Call-by-Reference-Übergabe. Wir geben daher für Lösung 2 zwei Varianten an, eine reine C-Lösung und die elegantere C++-Lösung. Betrachten wir jetzt die drei entsprechenden Funktionen *loeschanfang1*, *loeschanfang2a* und *loeschanfang2b*:

```
void loeschanfang1 (char *s)       // Version 1!
{  char *string;           // Hilfsstring

   for (string = s; *string == ' '; string++)
   ;
   // fuehrende Blanks sind entfernt, jetzt nach s kopieren:

   strcpy (s, string);
}

void loeschanfang2a (char **s)    // Version 2a (C)
{  while (**s == ' ')
      (*s)++;
}              // fuehrende Blanks + Speicher gehen verloren

void loeschanfang2b (char* &s)    // Version 2b (C++)
{  while (*s == ' ')
      s++;
}              // fuehrende Blanks + Speicher gehen verloren
```

Programm: 3-felder/loesch.cpp (Funktionen *loeschanfang1/2a/2b*)

Betrachten wir noch das dazugehörige Hauptprogramm *main*:

```
int main ()
{  char *p2, *p3, zeile1[100], zeile2[100], zeile3[100];

   cout << "\n\nBitte eine Zeile eingeben\n";
   cin.get (zeile1, 99);                    // Zeile einlesen
   cout << "\nOriginal: >" << zeile1 << "<\n";
```

```
    loeschende (zeile1);
    strcpy (zeile2, zeile1);   strcpy (zeile3, zeile1);
    p2 = zeile2;               p3 = zeile3;
    loeschanfang1 (zeile1);    // Version1
    loeschanfang2a (&p2);      // Version2a
    loeschanfang2b ( p3);      // Version2b

    cout << "\nErgebnis1 : >" << zeile1 << '<'
         << "\nErgebnis2a: >" << p2     << '<'
         << "\nErgebnis2b: >" << p3     << "<\n";

    return 0;
}
```

Programm: 3-felder/loesch.cpp (Hauptprogramm)

Zum Schluß dieses Abschnitts sei noch vor einem häufigen Fehler gewarnt.

Achtung:

➔ Verwenden Sie zum Kopieren einer Zeichenkette s1 in die Zeichenkette s2 den Funktionsaufruf strcpy (s2, s1);
und nicht s2 = s1; /* Fehler !!! */

Letzteres würde nur die Zeigeradressen kopieren, nicht jedoch deren Inhalt.

3.3 Einschub: Konstanten und selbstdefinierte Datentypen

Häufig möchte man etwa für Feldgrößen nicht Zahlen, sondern Konstanten verwenden. Bei konsequenter Verwendung solcher Konstanten kann im Bedarfsfall die Feldgröße leicht durch einfaches Ändern der Konstante angepaßt werden, ohne daß weitere Änderungen im Programm erforderlich wären. Diese Definition von Konstanten ist in C leider nicht so schlüssig wie in Pascal. Wir verwenden dazu Präcompiler-Anweisungen. Solche Präcompiler-Anweisungen beginnen immer mit einem Nummern-Zeichen ('#'). Wir kennen bereits die #include-Anweisung. Die Konstanten-Definition lautet:

```
#define  <Name>  <Ersetzung>
```

Mit dieser Anweisung ist die Konstante <Name> definiert und wird im Programm ab dieser Zeile überall durch die angegebene Ersetzung ersetzt. Dabei trennen der Name und die Ersetzung mindestens ein Leerzeichen. Die Ersetzung darf beliebige Zeichen, auch Leerzeichen enthalten. Dieser Ersetzungsstring endet mit dem Ende der Zeile. Beispiele:

```
#define  MAX  80
#define  MIN  -MAX
#define  GRUSS   "Halli Hallo"
```

Nun existieren die Konstanten MAX mit dem Wert 80, MIN mit -80 und GRUSS mit "Halli Hallo".

Achtung:

➔ Der Zahl darf kein Semikolon folgen, ansonsten wäre beispielsweise MAX gleich '80;' !

➔ Konstanten sind ab ihrer Definition bis Dateiende bekannt (unabhängig von Funktionen)!

Konvention: Es hat sich in C eingebürgert, daß Konstanten groß geschrieben werden. Dies ist zwar kein Muß, ist aber heute fast schon ein Standard.

Wir erkennen, daß das Arbeiten mit Konstanten in C erstens umständlich und zweitens fehlerträchtig ist (keine saubere Typüberprüfung). Wir wenden uns daher ab sofort an die Möglichkeiten in C++. Hier sind Konstanten explizit in der Sprache selbst definierbar. Einige Beispiele:

```
const int MAX = 10;
const float PI = 3.141593;
const NEIN = 0;                     // automatisch vom Typ int
```

```
const char * STR1 = "hallo";        // die Zeichenkette "hallo" ist konstant
char * const STR2 = "jawohl";       // der Zeiger STR2 ist konstant
```

Grundsätzlich gilt in C++, daß jeder Variablendeklaration das Wort *const* vorangestellt werden darf. In diesem Fall achtet der Compiler darauf, daß dieser Bezeichner im ganzen Programm nicht verändert werden darf. Einer Konstanten wird direkt bei der Deklaration ein Wert zugewiesen, da dies ja später nicht mehr möglich ist. Übrigens gibt es in C++ keinen Konstanten-Deklarationsteil wie in Pascal. Überall dort wo eine Variable deklariert werden darf, darf auch eine Konstante deklariert werden.

Wird bei der Konstante kein Datentyp angegeben, so wird automatisch der Typ int angenommen. Bei Zeichenketten ist noch zu beachten, daß sowohl der Zeiger als auch die Zeichenkette selbst als unveränderlich markierbar sind. Im obigen Beispiel darf STR1 verändert werden, ebenso die Zeichenkette "jawohl", nicht jedoch STR2 und "hallo"! Erlaubt wäre also:

```
STR1 = "neu";    // STR1 selbst ist nicht konstant.
*STR2 = 'J';     // Die Zeichenkette selbst darf manipuliert werden.
```

Auch bei Funktionsaufrufen darf 'const' bei den Parameterübergaben verwendet werden. Damit wird garantiert, daß im Unterprogramm diese Variable nicht geändert wird. Dies könnte dann so aussehen:

```
xyz ( const char *str);    // der Inhalt des Strings str kann in xyz nicht verändert werden
```

Wir lösen damit ein Problem, das bereits in Pascal bestand: Wollen wir nur aus Performancegründen eine Variable als Call-by-Reference übergeben, und wollen wir diese Variable im Unterprogramm nicht ändern, so übergeben wir sie als konstante Referenz, etwa:

```
beispiel (const float & x);         // konstante Referenz x
```

Wie in Pascal können wir auch in C eigene Datentypen definieren. Dies war in Pascal häufig auch zwingend erforderlich, da beispielsweise in Unterprogrammen nur Variablen von definierten Datentypen übergeben werden konnten. Hier bietet C weniger Einschränkungen, wir haben ja bereits gesehen, daß Felder und Zeichenketten einfach als Zeiger übergeben werden. Trotzdem ist es ein guter Stil und dient der besseren Veranschaulichung, mit selbstdefinierten Datentypen zu arbeiten. Wollen wir etwa einen Datentyp *string* definieren, so geschieht dies mit:

```
typedef char * string;
```

String ist damit ein Synonym für char * . *String* und char * besitzen nun eine gleichwertige Bedeutung. Die Funktionsweise von *typedef* läßt sich relativ einfach erklären: deklarieren wir eine Variable, so wird durch ein vorangestelltes *typedef* statt dieser Variablen der entsprechende Datentyp definiert.

Beispiel: Ein Feld aus N float-Zahlen mit der Konstante N=50 ist zu definieren:

```
const int N = 50;
typedef float ffeld [N];            /* Der Datentyp ffeld wird definiert */
ffeld A, B;                         /* zwei Felder A und B werden deklariert */
```

Beachten Sie dringend den Unterschied zwischen Konstanten- und Typdefinition:

```
typedef char * string;          und          #define string char *
```

haben nicht die gleiche Wirkung. Denken Sie nur an die folgende Deklaration:

```
string s1, s2;
```

Im ersten Fall werden wirklich zwei Zeiger auf *char* definiert, im zweiten ist hingegen s1 ein Zeiger, s2 aber nur eine char-Variable! Die Konstantendefinition hat nur die Wirkung, daß das Wort 'string' durch den Ausdruck 'char *' ersetzt wird! Solange wir also mit C++ arbeiten, vergessen wir die Präprozessoranweisung ‚#define'!

3.4 Mehrdimensionale Felder (Matrizen)

Mehrdimensionale Felder (Matrizen) werden analog zu Pascal verwendet. Hier empfiehlt sich auch

die Verwendung der Indexschreibweise. Zu beachten ist allerdings wieder, daß Felder in C von Null an indiziert sind. In Schleifen empfiehlt sich außerdem dringend die Verwendung von asymmetrischen Grenzen! Betrachten wir ein Beispiel zur Matrizenrechnung:

```
/* Matrizen lesen, multiplizieren und schreiben */

#include <iostream.h>

const int DIM = 10;
typedef float matrixtyp [DIM] [DIM];     // 10 mal 10 Matrix

// einlesen einer dim x dim - Matrix:
void liesmatrix (matrixtyp A, int dim)
{  int i, j;

   for (i=0; i<dim; i++)          // asymmetrische Grenzen!
   {  cout << "Zeile " << i+1 << ": ";
      for (j=0; j<dim; j++)
         cin >> A[i][j];
   }
}
```

Programm: 3-felder/matrix.cpp (Funktion liesmatrix)

```
void schreibmatrix (const matrixtyp A, int dim)
{  int i,j;
   for (i=0; i<dim; i++)
   {  for (j=0; j<dim; j++)
         cout << A[i][j];
      cout << '\n';
   }
}

void multmatrix(const matrixtyp A, const matrixtyp B,
                      matrixtyp C, int dim)
{  int i, j, k;
   float summe;

   for (i=0; i<dim; i++)
      for (j=0; j<dim; j++)
      {  summe = 0;
         for (k=0; k<dim; k++)     // Matrizenmultiplikation
            summe += A[i][k] * B[k][j];
         C[i][j] = summe;
      }
}
```

Programm: 3-felder/matrix.cpp (Funktionen schreibmatrix, multmatrix)

Wir erkennen den Vorteil der Verwendung von Konstanten, da wir bei Bedarf den Wert von DIM jederzeit ändern können. Den Datentyp *matrixtyp* hätten wir nicht definieren müssen, doch es wirkt unübersichtlicher und nicht so verständlich, hätten wir beispielsweise statt 'matrixtyp A' überall die Deklaration 'float A [DIM] [DIM]' geschrieben. Die Matrizen werden automatisch call-by-reference übergeben (Feld = Zeiger!). Diese call-by-reference-Übergabe ist in der Funktion *liesmatrix* notwendig und spart in *schreibmatrix* Laufzeit. Sicherheitshalber haben wir in letzterer Funktion das Feld als Konstante übergeben. Sehen wir uns das dazugehörige Hauptprogramm an:

```
int main ()
{  matrixtyp A, B, C;

   cout << "****** Matrizenmultiplikation ********\n"
        << "Gib 2x2-Matrix A ein:\n";

   liesmatrix (A, 2);
   cout << "Gib 2x2-Matrix B ein:\n");
   liesmatrix (B, 2);
   multmatrix (A, B, C, 2);

   cout << "Ergebnis der Multiplikation A x B:";
   schreibmatrix (C, 2);
   return 0;
}
```

Programm: 3-felder/matrix.cpp (Funktion main)

Zur weiteren Vertiefung betrachten wir ein kleineres Feld, etwa das Feld *m*, deklariert durch

float m [2] [3];

Dieses Feld *m* belegt in C garantiert Speicher direkt hintereinander. Wie in Pascal sind die einzelnen Elemente zeilenweise abgespeichert, also:

m[0][0]	m[0][1]	m[0][2]	m[1][0]	m[1][1]	m[1][2]

Diese Speicherung gilt analog auch für höhere Dimensionen. Auch die Initiierung mehrdimensionaler Felder ist kein Problem:

float m [2] [3] = { {1, 2, 3}, {4, 5, 6} };

Es existiert auch die kürzere, aber schlampige, und nicht ungefährliche Schreibweise:

float m [2] [3] = {1, 2, 3, 4, 5, 6};

Beachten Sie etwa den Unterschied zwischen { {1, 2}, {3, 4} } und {1, 2, 3, 4 } . In beiden Fällen werden fehlende Zahlen durch Null ergänzt, doch jeweils an verschiedenen Stellen.

Auch bei mehrdimensionalen Feldern gibt es einen äquivalenten Zugang mittels Zeiger. Wir vermuten daher wieder richtig, daß *m* ein Zeiger ist. Die weitere Vermutung, daß dann **m* das erste Element identifiziert, ist allerdings falsch! Um dies zu verstehen, müssen wir weiter ausholen. Wir müssen verstehen, welcher Art dieser Zeiger ist. Schließlich wissen wir, daß ein Zeiger erst durch zwei Eigenschaften festgelegt ist: durch die Adresse <u>und</u> den Datentyp, auf den der Zeiger verweist.

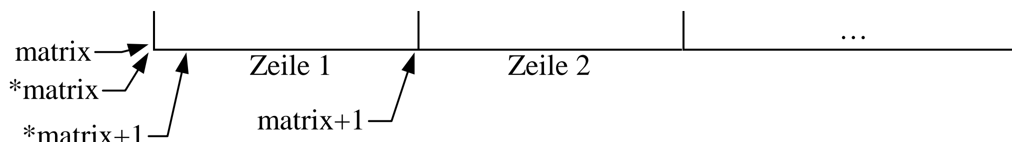
Betrachten wir das Feld *matrix*, deklariert durch

float matrix [M] [N];

Diese Matrix enthält M Zeilen, jede Zeile besteht aus N Spalten von float-Werten. Wir hätten diese Deklaration auch nacheinander aufbauen können:

```
typedef float zeile [N];          /* zeile enthält N Spalten vom Typ float */
zeile matrix [M];                 /* die Matrix enthält M Zeilen */
```

Mit der letzten Deklaration wird klar, daß die Variable matrix ein Zeiger auf den Datentyp *zeile* ist! Folglich repräsentiert '*matrix' die gesamte erste Zeile. Weiter ist 'matrix+1' ein Zeiger auf die zweite Zeile, oder allgemeiner 'matrix+i' ein Zeiger auf die i+1-te Zeile.



Eine einzelne Zeile ist aber wieder ein Feld, somit ist '*matrix' ebenfalls genaugenommen ein Zeiger, diesmal aber ein Zeiger auf float. Mit der nochmaligen Dereferenzierung '**matrix' sprechen wir demnach erst das erste Element der ersten Zeile an. Suchen wir jetzt etwas allgemeiner die dritte Spalte der zweiten Zeile, so müssen wir zunächst zur zweiten Zeile gehen. 'matrix+1' ist ein Zeiger auf diese Zeile

und ‘*(matrix+1)’ repräsentiert damit diese Zeile. Die dritte Spalte erreichen wir durch die Zeigerarithmetik ‘*(matrix+1)+2’. Durch die Dereferenzierung ‘*(*(matrix+1)+2)’ haben wir schließlich das gesuchte Element gefunden. Dies läßt sich verallgemeinern:

matrix [i] [j]   ≡   * (* (matrix + i) + j)

Zur Übung sei noch folgendes Beispiel angegeben:

```
float *p, matrix [M] [N];
p = matrix;              // falsch, da verschiedene Datentypen
p = matrix [0];          // korrekt
p = * matrix;            // korrekt
p = &matrix [0] [0];     // korrekt
```

Wir haben bereits bei eindimensionalen Felder gesehen, daß C zur Parameterübergabe alle Zeigerinformationen benötigt, also den Zeiger zusammen mit dem Datentyp, nicht aber die Feldgröße. Dies gilt ganz analog auch für mehrdimensionale Felder. Wir müssen demnach nicht die gesamte Matrix als Parameter übergeben. Die Deklaration der folgenden Funktion *liesmatrix* ist zwar korrekt, enthält aber eine überflüssige Feldangabe:

```
void liesmich ( float A [M] [N] );
```

Leider ist das vollständige Weglassen beider Feldgrenzen ein Fehler:

```
void liesmich ( float A [ ] [ ] );       /* Fehler!!!!! */
```

Dies liegt daran, daß A ein Zeiger auf eine Zeile ist. Diese Zeile muß vollständig festgelegt sein, die Anzahl der Spalten ist demnach erforderlich:

```
void liesmich ( float A [ ] [N] );       /* korrekt */
```

Schließlich muß C schon zur Übersetzungszeit wissen, welche Adresse beispielsweise A+1 ergibt. Etwas klarer und mit eindimensionalen Feldern vergleichbarer wird dies, wenn wir unsere Typdefinition der Zeile nochmals zugrundelegen:

```
void liesmich ( zeile A [ ] );
```

Wie bereits bei eindimensionalen Feldern besprochen, ist demnach gleichwertig:

```
void liesmich ( zeile * A );
void liesmich ( float (*A) [N] );        /* korrekt, aber ungebräuchlich */
```

In der letzten Zeile darf die innere Klammer nicht weggelassen werden, da eckige Klammern stärker binden als die Dereferenzierung. Mit

```
float *A [N]
```

wird nämlich ein Feld aus N Elementen vom Typ (float *) definiert. Jedes Element enthält also einen Zeiger auf *float*. Wir kommen im übernächsten Abschnitt darauf zurück.

3.5 Dynamische Speicherverwaltung in C

Ähnlich wie in Pascal gibt es auch in C Funktionen zum Reservieren und Freigeben von dynamisch reservierten Speicherbereichen (auch Heap genannt). Diese Funktionen sind zum einen mächtiger als in Pascal, dafür sind sie aber relativ komplex handhabbar. Die Erweiterungen in C++ hingegen erinnern wieder sehr an Pascal. Beginnen wir mit den Möglichkeiten von C. Die Funktionen zur Speicherverwaltung lauten hier:

```
malloc        calloc        realloc        free
```

Die ersten Funktionen belegen Speicher im Heap, die Funktion *free* gibt diesen Speicher wieder frei. Diese Funktionen sind in der Headerdatei **stdlib.h** deklariert. Im einzelnen gilt:

void * malloc (size_t size)

Diese Funktion reserviert Speicher der Größe *size* und gibt einen Zeiger auf diesen Speicherbereich als Funktionswert zurück. Der Datentyp *size_t* ist in *stdlib.h* definiert und ist in der Regel vom Typ *long int*.

Malloc gibt einen Zeiger zurück. Wir erinnern uns, daß ein Zeiger immer aus zwei Informationen besteht, aus einer Adresse und dem Datentyp, auf den die Adresse zeigt. *Malloc* ist so allgemein implementiert, daß es nur die Anzahl von Bytes reserviert, unabhängig von einem Datentyp. Aus diesem Grund kann *Malloc* zwar eine Adresse, aber keinen speziellen Datentyp dazu zurückliefern. Als Ausweg existiert in C der allgemeine Zeigertyp ‘void *’. Dieser Zeigertyp ist mit jedem anderen Zeigertyp verträglich. Es wird aber dringend angeraten, bei der Zuweisung dieses allgemeinen Zeigers an einen bekannten Zeiger den Cast-Operator explizit zu verwenden.

Ist p ein Zeiger auf den Datentyp *element*, so wird Speicher für ein Element wie folgt reserviert:

```
p = (element *) malloc ( sizeof (element) ) ;
```

In Pascal hätte *new (p)* genügt. Dafür ist *Malloc* aber sehr mächtig. So können wir auch Speicher für Zeichenketten reservieren. Ist etwa ein String *s1* gegeben, und wollen wir diesen String in ein weiteres Feld kopieren, und ist *s2* ein Zeiger auf *char*, so geht dies mit

```
s2 = (char *) malloc (strlen (s1) + 1);          /* Speicher einschließlich ‘\0’ */
strcpy (s2, s1);
```

Weitere Funktionen sind:

void * calloc (size_t anzahl, size_t size)

Hier wird Speicher für *anzahl* Elemente der Größe *size* reserviert und ein allgemeiner Zeiger auf den Anfang dieses Speicherbereichs wird zurückgegeben. Zusätzlich wird der Speicher mit binären Nullen vorbelegt. Letzteres gilt nicht für Malloc.

void * realloc (void * p, size_t size)

Der Speicherbereich, auf den der Zeiger *p* zeigt, wird auf die Größe *size* geändert. Der Zeiger auf den eventuell neuen Speicherbereich wird zurückgegeben. Ist *size* kleiner als der bisher belegte Speicher, so bleibt der bisherige Inhalt anteilig erhalten. Ist *size* größer oder gleich, so bleibt der bisherige Inhalt komplett erhalten. Der Inhalt des neu hinzugekommenen Speichers ist undefiniert. Zeiger *p* darf auch der NULL-Zeiger sein. Mißlingt die Speicheranforderung, wird NULL zurückgeliefert; der bisherige Speicher bleibt dann erhalten. Dieser Befehl wird verwendet, wenn der bisherige Speicher nicht ausreicht und zusätzlicher Speicherplatz benötigt wird.

void free (void *p)

Diese Funktion entspricht der Prozedur *dispose* in Pascal und gibt den Speicherbereich frei, auf den *p* zeigt, und der durch *malloc*, *calloc* oder *realloc* angefordert wurde. Ist *p* der NULL-Zeiger, so geschieht keine Aktion.

Übrigens liefern die Speicherreservierungsfunktionen den NULL-Zeiger zurück, falls der benötigte Speicher nicht reserviert werden konnte. Malloc kann daher beispielsweise wie folgt aufgerufen werden:

```
if ( (p= (element *) malloc ( sizeof(element) ) ) == NULL )
{   puts ("Nicht genügend Speicher vorhanden");  return 0;
}
```

3.6 Dynamische Speicherverwaltung in C++

In C++ sind selbstverständlich weiterhin alle Möglichkeiten von C zur dynamischen Speicherverwaltung verfügbar. Darüberhinaus existieren in C++ aber auch die beiden Operatoren **new** und **delete**, die die dynamische Speicherverwaltung deutlich vereinfachen. Doch sehen Sie selbst. Im folgenden Beispiel wollen wir einen Speicherplatz für 100 Ganzzahlen reservieren:

```
/* in C und C++: */                                // nur in C++:

int *p;                                            int *p;
p = (int *) malloc (100 * size of (int));          p = new int [100];
```

Der Operator *new* ersetzt die schwerfällige Funktion *malloc*. New erkennt automatisch den Speicherbedarf des Datentyps *int*, reserviert diesen Speicher und liefert einen Zeiger (dieses Datentyps) auf diesen Speicherbereich zurück. Die Wirkung ist also ähnlich wie bei der Pascal-Prozedur *new*, nur daß der Operator *new* vielseitiger einsetzbar ist. Ein weiteres Beispiel:

```
char * c = new char [100];    // es werden 100 char-Elemente dynamisch angefordert
```

Zu beachten ist, daß der Operator *new* den NULL-Zeiger zurückliefert, wenn der angeforderte Speicher nicht reserviert werden kann.

Einmal angeforderter Speicher bleibt so lange reserviert, bis er wieder explizit freigegeben oder das Programmende erreicht wird. Die Gültigkeit des Speichers ist demnach unabhängig von irgendwelchen Kontexten, d.h. innerhalb einer Funktion reservierter Speicher ist auch beim Verlassen dieser Funktion weiter vorhanden (hoffentlich auch der Zeiger, der auf diesen Speicher zeigt!). Dies galt ebenso in Pascal und C.

Das Freigeben von Speicher geschieht mit dem Operator *delete*, etwa

```
delete p;          // gibt den Speicher für die 100 Ganzzahlen wieder frei
delete c;          // gibt den Speicher für die Zeichenkette wieder frei
```

Beachten Sie, daß *new* und *delete* Operatoren sind. Wir müssen also die folgenden Zeigervariablen nicht in Klammern setzen und benötigen auch nicht mehr die Headerdatei *stdlib.h*.

3.7 Zeigerfelder

Wir haben uns bei eindimensionalen Feldern gefreut, daß wir die Feldgröße nicht mit als Parameter übergeben mußten. Leider gilt dies nicht mehr uneingeschränkt bei zwei Dimensionen. Ferner liegt nicht immer der Idealfall von Matrizen vor, wo alle Zeilen gleich lang sind. Hier könnte man durch mehr Flexibilität in vielen Anwendungsfällen Speicher sparen.

Wollen wir also Speicher sparen, und wollen wir weder Zeilen- noch Spaltengrößen an Parameter übergeben, so sollten wir Zeigerfelder verwenden.

Um flexible Zeilengrößen zu bekommen, dürfen wir die Zeilenlänge nicht festlegen. Wir müssen deshalb jede einzelne Zeile durch einen „echten“ Zeiger repräsentieren. Wir definieren deshalb ein Feld von Zeigern, die für sich auf eine zunächst unbekannte, also variabel lange Zeile zeigen. Betrachten wir im folgenden char-Elemente, so läßt sich eine variabel lange Zeile wie folgt definieren:

```
typedef char * zeile;
```

Wollen wir jetzt 10 Zeilen definieren, wobei die i-te Zeile N+i Zeichen enthält, so geht dies wie folgt:

```
const int N = 80;
zeile text [10];
for (int i = 0; i<10; i++)
    text [ i ] = new char [ N+i ] ;        // Speicher für jede Zeile dynamisch angefordert
```

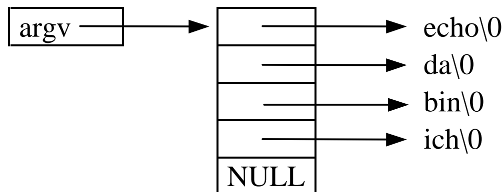
Nun ist *text* ein Feld aus 10 Elementen, wobei jedes Element eine Zeile, also einen Zeiger auf char enthält. Jeder dieser 10 Zeiger verweist auf einen Speicherbereich von N und mehr Zeichen. Kommen wir jetzt noch zu einer interessanten Anwendung von Zeigerfeldern, den Parametern der Funktion *main*.

Bisher haben wir die Funktion *main* wie eine parameterlose Funktion behandelt. Sie ist es aber nicht. Folgender Funktionskopf wäre in C (nicht in C++) fehlerhaft:

```
int main (void)
```

Denn die Funktion *main* besitzt tatsächlich zwei Parameter (nach ANSI-C). Der erste Parameter wird

meist mit *argc* bezeichnet und ist eine int-Zahl. Diese Zahl gibt die Anzahl der Argumente beim Aufruf des Programms an. Der zweite Parameter heißt meist *argv* und ist ein Zeigerfeld, das alle diese Argumente namentlich enthält. Heißt unser Programm etwa 'echo', und rufen wir dieses auf mit 'echo da bin ich', so enthält *argc* den Wert 4 und *argv* repräsentiert folgendes Zeigerfeld:



argv zeigt demnach auf ein Feld von (char *)-Zeigern. Der letzte Zeiger enthält den Wert NULL. Dieser Wert entspricht NIL aus Pascal. Damit wird ein Zeiger definiert, der auf keinen Speicherbereich zeigt.

Betrachten wir zunächst Zugriffe mit *argv*: *argv* ist ein Zeiger auf ein Zeigerfeld. Dann ist '**argv*' das erste Element des Feldes, also ein Zeiger auf eine Zeichenkette. Folglich ist '***argv*' das erste Zeichen dieses Zeichenfeldes, demnach das Zeichen '*e*' von '*echo*'. Wollen wir jetzt das zweite Zeichen des dritten Arguments ansprechen, so zeigt '*argv+2*' auf das dritte Element des Zeigerfeldes, '**(argv+2)*' zeigt damit auf die dritte Zeichenkette und '**(*(argv+2)+1)*' ist schließlich das gewünschte Element.

Aber wir können auch umgekehrt mit Indizes arbeiten: *argv[2]* ist das dritte Element des Feldes, und damit ist *argv[2][1]* das zweite Zeichen der dritten Zeichenkette. Wieder sind beide Schreibweise (Index und Zeiger) gleichwertig. Wir erkennen noch mehr: ein Zeigerfeld unterscheidet sich bei Zugriffen auf die einzelnen Elemente nicht von einer Matrix.

Der Unterschied zwischen einer Matrix und einem Zeigerfeld liegt demnach nicht in den Zugriffen, sondern in der Speichertechnik: eine Matrix wird direkt hintereinander abgespeichert. In einem Zeigerfeld werden nur die Adressen hintereinander gespeichert. Die Daten (der einzelnen Zeilen) befinden sich an beliebigen anderen Stellen. Gleichzeitig wird bei Zeigerfeldern Speicher für die Adressen benötigt, es liegen somit auch veränderbare Zeiger vor, nicht so bei Matrizen.

Wie bereits weiter oben erwähnt, könnte *argv* wie folgt definiert werden:

```
char * argv [20];
```

Damit wird ein Feld mit 20 char-Zeigern reserviert. Bei der Übergabe dieser Variable an Funktionen interessiert die Größe des Feldes nicht, so daß *argv* als Parameter in *main* wie folgt deklariert wird:

```
int main ( int argc, char * argv [] )
```

oder

```
int main ( int argc, char ** argv )
```

Die letzte Darstellung ist, wie wir bereits bei eindimensionalen Feldern erwähnten, äquivalent zur ersten. Wir verwenden die erste Darstellung, wenn wir den Feldcharakter betonen, die zweite, wenn wir die Zeigerschreibweise hervorheben wollen.

Betrachten wir noch ein Beispiel, das ich direkt einem UNIX-Buch entnommen habe:

```
/* Was macht dieses Programm?         */

#include <stdio.h>

int main (int argc, char *argv[])
{
   while ( --argc > 0)
      cout << *++argv << (argc>1) ? ' ' : '\n';

   return 0;
}
```

Programm: 3-felder/echo.cpp

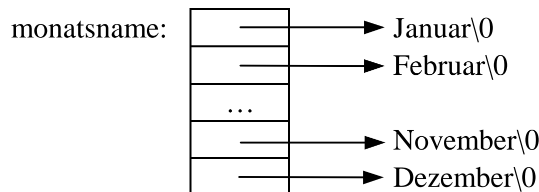
Diese Funktion bildet das Echo-Kommando nach: die Argumente werden mit Ausnahme des Pro-

grammnamens nacheinander ausgegeben. Bitte vollziehen Sie die Funktionsweise dieses Programms im einzelnen nach.

Übrigens lassen sich auch Speicherfelder komplett initiieren. Betrachten wir dazu als Beispiel die Definition der zwölf Monatsnamen:

```
char * monatsname [ ] =  { "Januar", "Februar", "Maerz", "April", "Mai", "Juni", "Juli",
                           "August", "September", "Oktober", "November", "Dezember" };
```

Hier ist die Variable *monatsname* ein Feldzeiger auf ein festes Feld von 12 Zeigern. Jeder dieser 12 Zeiger zeigt auf eine eigene Zeichenkette, beginnend bei „Januar" bis hin zu „Dezember". Es wird also Speicher für 12 Zeiger <u>und</u> die dazugehörigen Monatsnamen reserviert. Dies sieht dann so aus:



Zusammengefaßt können wir sagen, daß wir nicht zuviel versprochen haben: wir können zweidimensionale Felder so definieren, daß wir in der gewohnten Matrizenschreibweise zugreifen können. Es muß jedoch nicht jede Zeile gleich lang sein, aber vor allen Dingen muß keine Angabe zu den Feldgrößen an Unterprogramme weitergegeben werden. Dadurch gewinnen solche Unterprogramme eine große Wiederverwertbarkeit, da sie nicht für jede einzelne Feldgröße neu geschrieben werden müssen.

4 Dateibearbeitung

Analog zu allen anderen Programmiersprachen geschieht das Arbeiten mit Dateien in vier Schritten:

- Deklarieren einer Dateivariablen
- Öffnen der gewünschten Datei
- Eigentliches Arbeiten mit dieser Datei
- Schließen der Datei

Bereits die Eingabe von der Tastatur und das Schreiben auf den Bildschirm unterscheiden sich in C und bei den Dateiströmen in C++ vollständig. Dies setzt sich natürlich auch bei Dateien fort. Wir betrachten daher beide Möglichkeiten getrennt voneinander.

4.1 Dateibearbeitung in C

Beginnen wir mit der Deklaration: in C gibt es wie in Pascal einen eigenen Datentyp für Dateien. Der Datentyp heißt FILE und ist in stdio.h deklariert. Im Programm selbst arbeiten wir mit Dateizeigern vom Typ 'FILE *'. Beispielsweise lautet die Deklaration der Dateizeiger *datei*, *fin* und *fout* wie folgt:

```
FILE * datei, * fin, * fout ;
```

Im gesamten Programm werden die Dateien durch ihre Dateizeiger repräsentiert. Beim Öffnen müssen wir allerdings den tatsächlichen Dateinamen kennen. Das Öffnen und die Zuordnung zwischen internem Dateizeiger und externem Dateinamen geschieht mit der Funktion *fopen*. Sie ist in *stdio.h* wie folgt deklariert:

```
FILE * fopen (char * dateiname, char * modus)
```

Diese Funktion *fopen* öffnet die Datei *dateiname* in dem angegebenen Modus (Lesen oder Schreiben),

baut eine interne Dateiverwaltungsstruktur vom Typ FILE auf und liefert den Zeiger auf diese Struktur als Funktionsergebnis zurück. Mißlingt aus irgendwelchen Gründen das Öffnen, so wird der Nullzeiger NULL zurückgegeben. Die wichtigsten Modi sind:

"r": Öffnen zum Lesen. Die zu öffnende Datei muß existieren. Das Programm zeigt auf den Anfang der Datei.

"w": Öffnen zum Schreiben. Eine bereits existierende Datei wird überschrieben.

"a": Öffnen zum Schreiben. Existiert die Datei bereits, so wird an das Ende der Datei angehängt.

"r+", "w+", "a+": eine im angegebenen Modus geöffnete Datei darf auch anderweitig bearbeitet werden (gemischtes Lesen und Schreiben)

"rb": Wie "r", nur daß die Datei als Binärdatei geöffnet wird. Das Zeichen 'b' darf auch mit allen obigen Modi kombiniert werden, etwa "w+b". In diesem Fall wird die angegebene Datei zunächst im Schreibmodus binär geöffnet. Sie darf sowohl beschrieben als auch gelesen werden.

Bei Dateimodi mit dem Zusatz '+' oder 'b' gibt der Modus 'r', 'w' und 'a' nur an, wie die Datei zunächst geöffnet wird, ob also eine Datei vorher existieren muß, und wohin das Programm innerhalb der Datei zeigt.

Beginnen wir mit einem einfachen Beispiel zum Öffnen einer Datei mit Überprüfung auf Erfolg:

```
datei = fopen ("liesmich.txt", "r");
if (datei == NULL)
  puts ("Fehler beim Oeffnen\n");
```

oder abkürzend:

```
if ( (datei = fopen ("liesmich.txt", "r") ) == NULL )
  puts ("Fehler beim Oeffnen\n");
```

In der Headerdatei *stdio.h* werden drei Dateizeiger vordefiniert. Dies sind

```
stdin           /* Standardeingabe (von Tastatur) */
stdout          /* Standardausgabe (auf Bildschirm) */
stderr          /* Standardfehlerausgabe (auf Bildschirm) */
```

Die Ein-/Ausgabe in diese Dateien geschieht standardmäßig von Tastatur bzw. auf den Bildschirm, außer im Betriebssystem werden diese Ein-/Ausgaben umgelenkt (in UNIX: mit '< Dateiname' für die Eingabe, '> Dateiname' für die Ausgabe und '2> Dateiname' für die Fehlerausgabe).

Das Schließen einer geöffneten Datei erfolgt mit

```
fclose ( dateizeiger );
```

Das eigentliche Arbeiten mit Dateien geschieht mit Hilfe von zahlreichen Funktionen, die alle in *stdio.h* deklariert sind. Wir wollen hier nur die wichtigsten ansprechen:

FILE * fp ;	/* Deklaration des Dateizeigers fp */
fprintf (fp, formatstring, …)	formatierte Ausgabe in die Datei fp fprintf (stdout, …) ≡ printf (…)
fscanf (fp, formatstring, …)	formatierte Eingabe von Datei fp fscanf (stdin, …) ≡ scanf (…)
fputc (zeichen, fp)	Zeichen in die Datei fp schreiben fputc (ch, stdout) ≡ putchar (ch)
putc (zeichen, fp)	wie fputc, ist aber als Makro realisiert
zeichen = fgetc (fp)	Zeichen von Datei fp lesen fgetc (stdin) ≡ getchar ( )
zeichen = getc (fp)	wie fgetc, ist aber als Makro realisiert
ungetc (zeichen, fp)	zuletzt gelesenes Zeichen wird in den Dateipuffer zurückgelegt, um nochmals gelesen werden zu können. Das Zurücklegen nur eines Zeichens wird garantiert.

fputs (string, fp)	String in die Datei fp schreiben fputs (s, stdout) ≡ puts (s)
fgets (string, anzahl, fp)	String von Datei fp lesen. Im Unterschied zu gets muß die maximale Anzahl zu lesender Zeichen mit angegeben werden
zahl = feof (fp)	Zahl ist 1, wenn das Dateiende erreicht wurde, sonst 0
zahl = ferror (fp)	Zahl enthält den letzten Ein-/Ausgabefehler zu der Datei fp

Es gibt noch weitere Funktionen zur Dateibearbeitung, darunter sehr hardwarenahe wie *fread* oder *fwrite* oder Such- und Positionierfunktionen wie *fseek*.

Ein wichtiger Hinweis ist, daß beim Lesen aus einer Datei bei jedem Lesezugriff sichergestellt sein muß, daß nicht über das Dateiende hinaus gelesen wird. Nach ANSI-C sind solche Zugriffe undefiniert. Neben der Funktion *feof* liefern alle Lesefunktionen Rückgabewerte zurück, aus denen das Dateiende ersichtlich wird. Dies sind

EOF	bei fscanf, scanf, fgetc, getc, getchar
NULL	bei fgets, gets

Es wird dringend empfohlen, bei allen verwendeten Eingabefunktionen immer den Rückgabewert entsprechend zu überprüfen. Zu beachten ist allerdings, daß EOF kein Zeichen, sondern eine Int-Zahl ist! Beim folgenden Beispiel wird eindrucksvoll gezeigt, daß bei jedem Lesen auf EOF abzuprüfen ist.

Dazu betrachten wir ein Programm, das den Inhalt einer Datei (Argument 1) an eine andere anhängt (Argument 2). Fehlt das zweite Argument, so wird die Datei auf *stdout* ausgegeben.

```
#include <stdio.h>

int main (int argc, char *argv[])
{  FILE *fin, *fout;
   void filecopy (FILE *, FILE *);

   if (argc == 1)     /* keine Argumente */
      puts("append: keine Argumente");
   else
      if ( (fin = fopen(*++argv, "r")) == NULL )
         printf("append: Datei %s nicht lesbar\n", *argv);
      else
         if (argc == 2)   /* nur ein Argument */
            filecopy ( fin, stdout );
         else
         {  fout = fopen(*++argv, "a");
            filecopy ( fin, fout );
         }
   return 0;
}
```

Programm: 4-datei/append.c (Hauptprogramm)

Betrachten wir noch die Funktion *filecopy*, in der wie angekündigt bei jedem Dateizugriff auf das Dateiende abgeprüft wird:

```
/* filecopy kopiert infile nach outfile: */

void filecopy (FILE* infile, FILE* outfile)
{
   int c;
   while ((c = getc(infile)) != EOF)
      putc (c, outfile);
}
```

Programm: 4-datei/append.c (Funktion *filecopy*)

4.2 Dateibearbeitung in C++

Die Dateibearbeitung mit Strömen ist noch umfangreicher als mit den Funktionen aus stdio.h. Insbesondere mit Klassen entfalten diese Ströme ihre Leistungsfähigkeit. Wir werden beim Arbeiten mit Klassen gezielt darauf eingehen. Zunächst betrachten wir aber die „normalen“ Möglichkeiten. Die für die Dateibearbeitung wichtigen Ströme heißen:

- ostream: Ausgabestrom
- istream: Eingabestrom
- ofstream: Ausgabestrom in Dateien
- ifstream: Eingabestrom von Dateien

Die Dateiströme *ifstream* und *ofstream* unterscheiden sich von *istream* und *ostream* vor allem durch die zusätzliche Definition von Funktionen zum Arbeiten mit Dateien (z.B. Öffnen und Schließen). Ansonsten ist die Dateibearbeitung analog zu der Bildschirmbearbeitung. Zu beachten ist, daß jetzt die Headerdatei ***fstream.h*** verwendet wird.

Wir kennen bereits die Ströme *cout*, *cerr* und *cin*. Diese sind standardmäßig vordefiniert. Dateiströme müssen hingegen deklariert werden:

```
ifstream eindat;                    // Eingabestrom zum Lesen aus einer Datei
ofstream ausdat;                    // Ausgabestrom zum Schreiben in eine Datei
```

Weiter existieren die Funktionen *open* und *close*, um Dateien zu öffnen und bei Bedarf wieder zu schließen. Das Arbeiten mit Dateien sieht dabei wie folgt aus:

```
ifstream infile;
infile.open ("datei.txt");          // die Datei datei.txt wird zum Lesen geoeffnet
infile >> x ;       // …            // die Variable x wird aus der Datei eingelesen usw.
infile.close ();
```

Das Arbeiten mit Dateiströmen funktioniert also völlig analog wie das Arbeiten mit *cin* und *cout*. Wir müssen nur *cin* bzw. *cout* durch den Namen des Dateistroms ersetzen. Es stehen alle Funktionen und die Operatoren ‚<<‘ und ‚>>‘ zur Verfügung. Natürlich dürfen Eingabeströme nur für Eingaben und Ausgabeströme nur für Ausgaben verwendet werden.

C++ macht die Dateibearbeitung sogar noch einfacher: die Deklaration einer Variable und das Öffnen der dazugehörigen Datei dürfen verschmolzen werden. Beispiele:

```
ifstream eindat ("datei.ein");           // Eingabestrom oeffnen, verbinden mit datei.ein
ofstream ausdat ("datei.aus");           // Ausgabestrom oeffnen, verbinden mit datei.aus
ofstream andat ("datei.an", ios::app);   // Ausgabestrom zum Anhaengen (append) oeffnen
```

Neben dem Modus *ios::app* spielt auch *ios::binary* noch eine wichtige Rolle. Diese Modi stehen als zweiter Parameter bei der Deklaration oder bei der Funktion *open*. Der erste Modus öffnet eine Datei zum Anfügen, der zweite öffnet eine Datei im Binärmodus. Sollen beide Modi gelten, so muß der binäre Oder-Operator verwendet werden. Bisher nicht behandelte weitere interessante Funktionen sind:

i = eof ( )	gibt den Wert 1 zurück, falls das Dateiende erreicht wurde, sonst 0
putback (ch)	schreibt das zuletzt gelesene Zeichen ch wieder in den Eingabepuffer zurück
i = peek ( )	schaut das nächste Zeichen an, ohne es aber einzulesen
ignore (i, ch)	ignoriert Eingaben bis zum Zeichen ch, aber nie mehr als i Zeichen

Die Überprüfung auf das Dateiende kann auch direkt durch Abfragen der Funktionsergebnisse erfolgen. Wird das Dateiende nämlich erreicht, so wird folgendes zurückgeliefert:

EOF	bei den Funktionen get() und peek()
0	bei den Funktionen get(ch), get(str, laenge), getline(str, laenge) und Operator >>

Aber auch der Dateistrom selbst kann auf Fehler überprüft werden. Konnte etwa eine dieser Dateien nicht geöffnet werden, so liefert ‘eindat’ bzw. ‘ausdat’ bzw. ‘andat’ den Wert 0. Auch bei einem erfolgreichen Öffnen und einem späteren Fehler oder Dateiende wird der Wert 0 zurückgeliefert. Beispiel:

```
if (!eindat) return 0;              // Dateiende oder anderer Einlesefehler
```

Betrachten wir abschließend noch das Beispiel aus dem letzten Abschnitt, jetzt aber mit Strömen:

```
#include <fstream.h>

int main (int argc, char *argv[])
{  if (argc == 1)          // keine Argumente
      cout << "append: keine Argumente\n";
   else
   {  ifstream fin ( *++argv );

      if ( !fin )
         cout << "append: " << *argv << " nicht lesbar\n";
      else
         if (argc == 2)     // nur ein Argument
            filecopy ( fin, cout );
         else
         {  ofstream fout ( *++argv, ios::app );
            filecopy ( fin, fout );
         }
   }
   return 0;
}
```

Programm: 4-datei/append.cpp (Hauptprogramm)

Es fällt auf, daß die Ströme nicht am Anfang des Hauptprogramms deklariert wurden. In C müssen Deklarationen zu Beginn einer Funktion oder eines Blockes deklariert werden. In C++ gilt diese Einschränkung nicht. Wir können also Ströme an beliebigen Stellen des Programms deklarieren. Ob dies sinnvoll ist, ist allerdings eine andere Frage. Betrachten wir zum Schluß auch noch die Funktion *filecopy*. Sie enthält einige Schwierigkeiten. Zunächst wollen wir im Unterprogramm mit den Originaldateien weiterarbeiten. Wir müssen die Dateiströme also Call-by-reference übergeben. Im Hauptprogramm übergeben wir aber in einem Fall auch den Ausgabestrom *cout*. Dieser Strom ist nicht vom Typ *ofstream*. Wählen wir also im Funktionskopf von *filecopy* den Typ *ofstream*, so gibt es eine Fehlermeldung, da *cout* nicht an *ofstream* übergeben werden kann. Doch diese Schwierigkeit läßt sich lösen. Wir wählen einfach den Typ *ostream*. Dies schadet hier keineswegs, da wir in der Funktion *filecopy* keine Klassenfunktionen zum Öffnen oder Schließen von Dateien benötigen. Und in fast anderen Fällen stimmen diese Klassen ja überein.

```
void filecopy (ifstream& infile, ostream& outfile)
{  char c;
   while ( infile.get(c) )
      outfile.put(c);
}
```

Programm: 4-datei/append.cpp (Funktion *filecopy*)

5 Modulare Programmentwicklung in C

5.1 Gültigkeitsbereich von Bezeichnern in einer Datei

Bezeichner in C sind Variablennamen, Funktionsnamen, Datentypnamen und in C++ auch Konstanten. Bezeichner dürfen in der Regel entweder extern, also außerhalb von Funktionen, oder intern zu Beginn eines Blocks (insbesondere zu Beginn eines Funktionsblocks) deklariert werden. Es gilt:

- eine Funktion ist außerhalb eines Blocks zu definieren und ist immer extern,
- ein externer Bezeichner gilt ab der Zeile der Deklaration bis zum Dateiende,
- ein interner Bezeichner gilt innerhalb des Blocks, wo er deklariert wird,

- ein Parameter einer Funktion gilt innerhalb des Funktionsblocks, wo er deklariert wird,
- ein interner Bezeichner überdeckt einen globalen (außerhalb dieses Blocks definierten) Bezeichner gleichen Namens. In C++ kann allerdings eine globale außerhalb einer Funktion definierte Variable angesprochen werden, auch wenn eine lokale Variable gleichen Namens existiert. In diesem Fall muß der globalen Variable der ‚::'-Operator vorangestellt werden. Existiert etwa in einem Block eine lokale Variable *a*, und existiert eine globale Variable gleichen Namens, so wird mit *a* die lokale und mit *::a* die globale angesprochen.

In C können Funktionen an beliebiger Stelle innerhalb einer Datei plaziert werden, also insbesondere vor oder hinter der Funktion *main*. Soll aber bereits weiter oben in der Datei auf diese Funktion zugegriffen werden, so ist die Funktion zumindest vorher zu deklarieren. Dies geschieht an beliebiger Stelle durch Schreiben des Funktionskopfs, abgeschlossen durch ein Semikolon. In diesem Falle dürfen die Parameternamen weggelassen werden, nicht jedoch die Parametertypen (ANSI-C und C++). Funktionsdeklarationen unterliegen den obigen Gültigkeitsbereichen von Bezeichnern.

Beispiel:

```
…
int beispiel (int *);          /* nur Deklaration der Funktion beispiel */
int main ()
{   …
    a = beispiel (pointer);    /* erlaubt, da Funktion bereits vorher deklariert */
    …
}
…
int stack [ 1000 ];            /* in main nicht sichtbar */
int beispiel ( int * p )       /* hier folgt die Definition der Funktion beispiel */
{   …
}
```

Wir erkennen, daß wir den Definitionsbereich externer Bezeichner, etwa den der Variablen *stack*, durch eine geschickte Plazierung einschränken können.

5.2 Speicherklassen

In C gibt es zwei Speicherklassen und vier Schlüsselwörter, die Variablen diesen beiden Speicherklassen zuordnen. Die Speicherklassen sind:

automatisch	Variablen sind nur innerhalb bestimmter Programmteile gültig. Sie werden bei Betreten dieses Programmteils dynamisch erzeugt und beim Beenden wieder entfernt
statisch	Variablen werden bereits zur Compilierzeit angelegt und sind von Programmstart bis Programmende vorhanden. Diese Variablen werden immer mit binären Nullen vorbelegt

Der Programmcode aller C-Funktionen wird zusammen mit dem Funktionsnamen ebenfalls bereits zur Compilierzeit angelegt. Wir können daher die Definition einer Funktion quasi als statisch bezeichnen.

Variablen, die außerhalb von Funktionen deklariert sind, werden immer der Speicherklasse statisch zugeordnet. Variablen, die innerhalb von Funktionen deklariert werden, gehören standardmäßig der Speicherklasse automatisch an. Diese Zugehörigkeit zu einer Speicherklasse ist unabhängig von der Sichtbarkeit von Variablen. Diese Zugehörigkeit kann durch bestimmte Schlüsselwörter geändert werden. Die vier Schlüsselwörter lauten:

auto	Lokale Variablen werden der Speicherklasse automatisch zugeordnet
static	Lokale Variablen werden der Speicherklasse statisch zugeordnet
register	Lokale Variablen und Parameter werden der Speicherklasse automatisch zugeordnet und, wenn möglich, in einem Register gehalten
extern	Verweis von lokalen Variablen und Funktionsdeklarationen auf außerhalb deklarierte statische Variable

Eines dieser vier Schlüsselwörter darf einer Variablendeklaration vorangestellt werden. In der Praxis wird das Schlüsselwort *auto* aber kaum verwendet, da alle innerhalb von Funktionen deklarierten Variablen standardmäßig bereits der Speicherklasse automatisch angehören. Auch die Parameter einer Funktion gehören zur Speicherklasse automatisch. Der Sinn von automatischen Variablen liegt darin, rekursive Funktionen zu ermöglichen. Jeder Aufruf einer solchen rekursiven Funktion arbeitet mit einem eigenen Satz von Parametern und Variablen!

Soll aber ausnahmsweise eine interne Variable nicht nach jedem Verlassen eines Block entfernt und beim nächsten Durchlaufen wieder neu angelegt werden, so kann sie als *static* deklariert werden. Solche interne Variablen werden wie externe Variablen schon zur Compilierungszeit angelegt und sind während des gesamten Programmlaufs vorhanden. Die Sichtbarkeit dieser Variablen wird dadurch jedoch nicht beeinflußt. Sie können weiter nur innerhalb des Blocks, in dem sie definiert sind, angesprochen werden. Betrachten wir folgendes Beispiel:

```
int zaehler (void)
{   static int n=0;
    return ++n;
}
```

Wir sehen, daß das Schlüsselwort vor den Datentyp geschrieben wird. In unserem Beispiel liefert der erste Aufruf von zaehler() den Wert 1, der zweite den Wert 2 usw. zurück. Beim Verlassen der Funktion bleibt nämlich der Speicherplatz von n erhalten. Beim nächsten Aufruf wird genau dieser Speicherplatz wieder belegt. Static-Variablen werden dabei nur einmal – zu Programmstart – initiiert. Würde das Schlüsselwort *static* weggelassen, so würde immer der Wert 1 zurückgeliefert werden.

Wie das Schlüsselwort *static* ist auch das Schlüsselwort *register* nur innerhalb von Funktionen anwendbar. Der Compiler wird in diesem Fall versuchen, die Variable dauerhaft in einem Register zu halten. Dies beschleunigt die Zugriffe auf diese Variable. Wir können damit das Leistungsverhalten des Programms beeinflussen. Da die Anzahl der Register meist sehr klein ist, sollten nur wenige Variablen vom Typ *register* sein. Zu beachten ist, daß zu Registervariablen keine Adressen existieren. Jede Referenzierung solcher Variablen ist demnach ein Fehler. Den Parametern einer Funktion darf als einziges Schlüsselwort dieses Wort *register* zugewiesen werden. Die Wirkung ist identisch mit der bei Variablen.

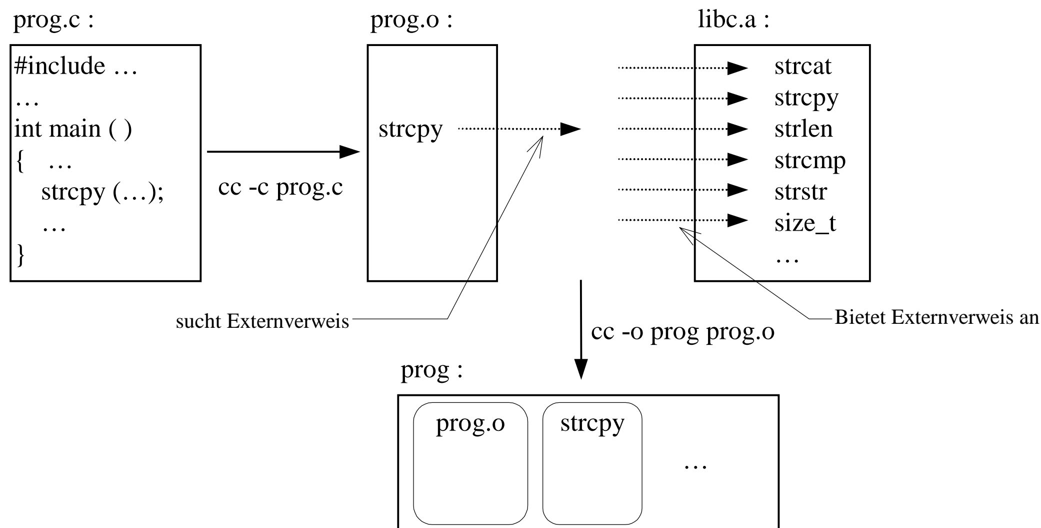
Die Schlüsselwörter besitzen weitere Bedeutungen bei der getrennten Übersetzung. Wir kommen gleich darauf zurück.

5.3 Getrennte Übersetzung

In C ist es problemlos möglich, ein umfangreiches Programm auf mehrere Teile, und damit in mehrere Dateien, aufzuteilen. Jeder Teil kann für sich übersetzt und als Modul gespeichert werden. Der Linker baut schließlich die einzelnen Module zu einem ablauffähigen Programm zusammen. Damit dies reibungslos funktioniert, müssen allerdings einige Regeln eingehalten werden.

Betrachten wir dazu die Funktionsweise von Compiler und Linker etwas ausführlicher. Beginnen wir mit nur einer Datei. Gegeben sei ein Programm *prog.c*. Dieses ruft die Funktion *strcpy* auf. Beim Übersetzen wird die Syntax von *strcpy* vom Compiler überprüft, da die Headerdatei *string.h* in das Programm *prog.c* kopiert wurde. Diese enthält alle (Funktions-)Deklarationen zur Stringbearbeitung.

Mittels der Option '-c' weist man dem C-Compiler an, das Programm nur zu übersetzen. Dieser erzeugt bei einem fehlerfreien Programm die Datei *prog.o* . Dieses Modul ist für sich nicht ablauffähig. Es fehlt das Laufzeitsystem von C und die Implementierung der Funktion *strcpy*. Beides befindet sich in der Bibliothek *libc.a* . Die Aufgabe des Linkers ist es nun, die benötigten Teile zum Modul *prog.o* dazuzubinden. Dazu besitzt der Modul *prog.o* sogenannte Externlisten. Diese enthalten die Namen der Bezeichner, die gesucht werden. Umgekehrt bietet die Bibliothek eine Menge von Namen an. Der Linker sucht nun gleiche Namen, verknüpft diese und baut schließlich mittels dieser Verknüpfungen ein lauffähiges Programm zusammen.



Die Option '-o Dateiname' erzeugt statt *a.out* eine Datei mit dem angegebenen Dateinamen. Wird dem Befehl *cc* statt eine Source ('.c') ein Modul ('.o') mitgegeben, so entfällt das Compilieren, nur der Linker wird aufgerufen. Grafisch ist dieses Vorgehen in der obigen Übersichtsskizze dargestellt.

Nur geringfügig komplizierter wird dieses Vorgehen, wenn wir statt einer Datei mehrere Source-Dateien besitzen. Betrachten wir dazu nochmals unser Matrizenprogramm. Die Funktionen *liesmatrix*, *schreibmatrix* und *multmatrix* wollen wir in einem eigenen Modul verwalten. Das Hauptprogramm steht in einer eigenen Datei. Dies könnte wie folgt aussehen:

Datei1 (prog1.c):

```
#include <iostream.h>
const int DIM = 10;
typedef float matrixtyp [DIM] [DIM];
void liesmatrix (…);
void schreibmatrix (…);
void multmatrix (…);
int main ( )
{     …
      liesmatrix (A, 2);
      liesmatrix (B, 2);
      multmatrix (A, B, C, 2);
      schreibmatrix(C, 2);
      …
}
```

Datei2 (prog2.c):

```
#include <iostream.h>
const int DIM = 10;
typedef float matrixtyp [DIM] [DIM];
void liesmatrix (…)
{     …
}
void schreibmatrix (…)
{     …
}
void multmatrix (…)
{     …
}
```

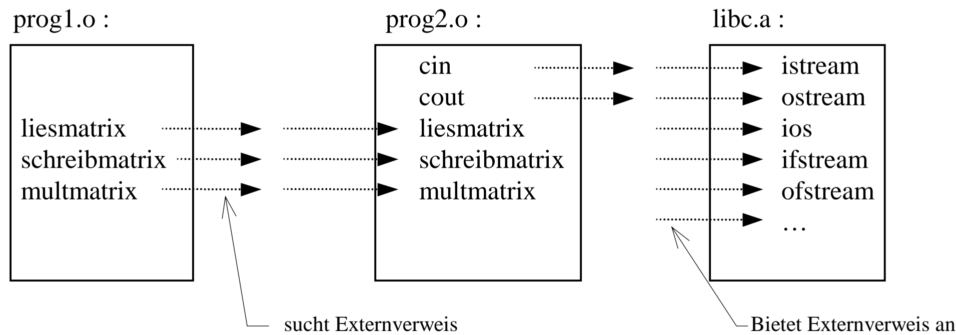
Wir haben in *prog1.c* die Matrixfunktionen nur deklariert. Diese Deklaration genügt dem Compiler, um das Programm zu übersetzen. Wie wir noch sehen werden, ist es Aufgabe des Linkers, die Implementierung dieser Funktionen zu suchen. Die beiden Teilprogramme lassen sich mittels eines einzigen Befehls komplett übersetzen:

```
cc -o matrix prog1.c prog2.c
```

Doch betrachten wir die Arbeitsweise von Compiler und Linker wieder getrennt. Mittels der Befehle

```
cc -c prog1.c
cc -c prog2.c
```

wird jedes Teilprogramm für sich übersetzt. Der Linker muß jetzt die beiden erzeugten Module *prog1.o* und *prog2.o* zusammen mit der Bibliothek *libc.a* zu einem Programm zusammenbinden. Wieder geschieht dies über die Externlisten, wobei diesmal *prog2.o* sowohl nach externen Funktionen sucht, als auch welche anbietet:



Der Linker wird wie folgt aufgerufen, um die ausführbare Datei *matrix* zu erzeugen:

```
cc -o matrix prog1.o prog2.o
```

Das bisherige Vorgehen ist zwar hinreichend, um ein ausführbares Programm zu erhalten. Es sind jedoch mehrere Fallstricke enthalten, die bei Änderungen einer der beiden Programmteile zu unverhergesehenen Effekten führen. Ändern wir zum Beispiel in *prog2.c* die Konstante DIM zu 6, um Speicher zu sparen, und vergessen wir diese Änderung in *prog1.c* nachzuvollziehen, so wird zwar eine ausführbare Datei erzeugt, die aber sicherlich nicht das Gewünschte ausführt. Den gleichen Effekt erzielen wir, wenn wir uns nur verschreiben. Auch bei Änderungen im Funktionskopf der Matrixfunktionen in nur einer der beiden Dateien wird ein ausführbares Programm erzeugt. Dieses Verhalten ist nicht im Sinne einer sauberen Programmierung. Ganz zu schweigen davon, daß der erzeugte Code mit großer Wahrscheinlichkeit verheerende „Nebenwirkungen" vorweist. Glücklicherweise läßt sich mit minimalem Aufwand ein Programm auf mehrere Dateien aufteilen, so daß der Compiler, unterstützt vom Betriebssystem, alle bisher entdeckten Fehler auch weiterhin findet.

Doch betrachten wir zunächst das grundsätzliche Verhalten von C bei getrennter Übersetzung:

- ist eine Funktion innerhalb einer Datei nur deklariert und nicht definiert, so nimmt der Compiler an, daß die Funktionsdefinition in einem anderen Modul steht. Er akzeptiert diese Funktion mit dem angegebenen Rückgabewert und den angegebenen Parametern.
- Jede definierte Funktion und jede außerhalb von Funktionen deklarierte Variable einer Programmdatei wird als Externverweis vermerkt. Der Linker wird auf diese Externverweise zurückgreifen, wenn er sie zum Binden benötigt.

Wir erkennen, daß praktisch jedes Modul sowohl externe Verweise anbietet als auch sucht. Der Linker wird (hoffentlich!) die richtigen Teile zusammenbinden. Dieses Weiterreichen von Externverweisen an den Linker kann beeinflußt werden. Dazu stehen die Schlüsselwörter *static* und *extern* zur Verfügung.

Wird das Schlüsselwort *static* vor Funktionen oder vor Variablen, die außerhalb von Funktionen deklariert sind, geschrieben, so wird kein Externverweis nach außen erzeugt. Diese Funktionen und Variablen sind demnach nur innerhalb der Datei sichtbar. Dies ist eine sehr gute Möglichkeit, nur lokal benötigte Variablen und Funktionen zu verstecken, was im Sinne der strukturierten Programmierung ist.

Wird das Schlüsselwort *extern* vor globalen Variablen geschrieben, so wird keine Variable deklariert, sondern nur ein Verweis definiert. Findet der Compiler diese Variable nicht innerhalb der Datei, so betrachtet er die dazugehörige Variable als außerhalb der Datei deklariert, und der Variablenname kann wie jeder andere Variablenname verwendet werden. Der Externverweis wird den Linker anweisen, diese Variable in einer anderen Übersetzungseinheit zu suchen. Beispiel:

```
in Datei1:    int n;          /* Deklaration, Speicherreservierung */
in Datei2:    extern int n;   /* nur Verweis auf eine außerhalb dekl. Variable */
```

Fassen wir zusammen:

- Alle Funktionen sind *extern* und besitzen einen Externverweis, außer sie werden mit dem Schlüsselwort *static* versehen.
- Funktionsdeklarationen, bei denen nur der Funktionskopf angegeben wird, sind Verweise auf Funktionen, die woanders definiert sind. Sie geben dem Compiler einen Hinweis auf die Syntax

des Funktionsaufrufs. Funktionsaufrufe dürfen ab dieser Zeile verwendet werden, auch wenn die eigentliche Funktionsdefinition erst später oder außerhalb der Datei erfolgt.

- Alle Variablen, die außerhalb von Funktionen deklariert sind, besitzen einen Externverweis, außer sie sind mit dem Schlüsselwort *static* versehen. Sie gehören immer der Speicherklasse statisch an.
- Variablendeklarationen mit dem Zusatz *extern* sind keine Deklarationen, sondern verweisen nur auf eine Variable des gleichen Namens. Wird *extern* außerhalb von Funktionen verwendet, so wird auf eine Variable verwiesen, die auch in einer anderen Übersetzungseinheit stehen kann.
- Alle Variablen, die innerhalb von Funktionen deklariert sind, besitzen grundsätzlich keinen Externverweis. Diese Variablen gehören standardmäßig der Speicherklasse automatisch an. Durch Hinzufügen des Schlüsselworts *register* wird der Compiler versuchen, diese Variable im Register zu halten. Durch Hinzufügen des Schlüsselworts *static* wird diese Variable der statischen Speicherklasse zugeordnet.

5.4 Headerdateien

Um Variablendeklarationen und Funktionsdefinitionen auch über mehrere Übersetzungseinheiten hinweg korrekt überprüfen zu können, müssen sogenannte **Headerdateien** oder Headerfiles verwendet werden. Diese Headerdateien enthalten alle Externverweise und globalen Typ- und Konstantendefinitionen einer oder mehrerer Übersetzungseinheiten und werden mittels einer Include-Anweisung den erforderlichen Dateien hinzugefügt. Es ist üblich, daß Headerfiles die Endung '.h' besitzen. In unserem Beispiel zur Matrizenmultiplikation könnte die Headerdatei folgenden Inhalt aufweisen:

```
const int DIM = 10;
typedef float matrixtyp [DIM] [DIM];
void liesmatrix ( matrixtyp, int );
void schreibmatrix ( matrixtyp, int );
void multmatrix ( matrixtyp, matrixtyp, matrixtyp, int );
```

Ist der Name der Headerdatei gleich *matrix.h*, so ändern sich unsere beiden Dateien zu:

Datei1 (prog1.c):

```
#include <iostream.h>
#include "matrix.h"
int main ( )
{    …
     liesmatrix (A, 2);
     liesmatrix (B, 2);
     multmatrix (A, B, C, 2);
     schreibmatrix (C, 2);
     …
}
```

Datei2 (prog2.c):

```
#include <iostream.h>
#include "matrix.h"
void liesmatrix (…)
{    …
}
void schreibmatrix (…)
{    …
}
void multmatrix (…)
{    …
}
```

In der Include-Anweisung können die Headerdateien in spitzen Klammern oder in Gänsefüßchen gesetzt werden. Der Unterschied ist wie folgt:

#include < … >	Datei wird in dem Standard-Includeverzeichnis des Systems gesucht
#include " … "	Datei wird zunächst im lokalen Verzeichnis gesucht. Wird sie nicht gefunden, so wird auch das Standard-Includeverzeichnis durchsucht

Wir erkennen jetzt, daß jede einseitige Änderung der Syntax der Funktionen, ob im Aufruf in *prog1.c*, in der Deklaration in *matrix.h* oder in der Definition in *prog2.c* vom Compiler erkannt wird, entweder beim Compilieren von *prog1.c* oder von *prog2.c* . Wir können eine getrennte Übersetzung mit Hilfe von Headerdateien demnach genauso sicher machen wie ohne Aufsplittung in mehrere Dateien. Wir müssen folgendes beachten:

Jede Datei, die Dienste (Externverweise) anbietet, bietet diese über eine Headerdatei an. Diese Hea-

derdatei muß auch von allen Programmteilen, die diese Dienste in Anspruch nehmen, mittels *#include* eingebunden werden. In eine Headerdatei gehören:

– alle Funktionsdeklarationen, die in einer Datei definiert werden und global sichtbar sind
– alle Konstanten und Datentypdefinitionen, die auch außerhalb verwendet werden sollen
– alle Verweise auf Variablen, die auch außerhalb verwendet werden sollen

In eine Headerdatei gehören nicht:

– Variablendeklarationen
– komplette Funktionsdefinitionen
– Include-Anweisungen (gilt nur bedingt, siehe auch Abschnitt 10.2.)

Ich empfehle zu jeder Datei, die externe Verweise anbietet, eine eigene Headerdatei zu verwenden. Manchmal wird auch für ein Projekt insgesamt nur eine Headerdatei benutzt. Dies hat den Nachteil, daß der Überblick bei größeren Projekten leicht verlorengeht, und daß bei einer geringfügigen Änderung dieser Headerdatei alle Programmteile nochmals übersetzt werden müssen.

5.5 Getrennte Übersetzung an einem Beispiel

Wir haben bereits einige Funktionen zum Zugriff auf Matrizen und einige Funktionen zur Zeichenkettenverarbeitung geschrieben. Jedes Mal, wenn wir diese Funktionen verwenden wollen, müssen wir diese in das entsprechende Programm hineinkopieren. Um dies zu vermeiden, wollen wir zwei Module und dazu je eine Headerdatei erstellen:

a) Stringmodul:

```
#include "strpriv.h"
// strlength: gibt die Laenge des Strings s zurueck:
int strlength (char *s)
{   int n;
    for (n=0; *s != '\0'; s++)
       n++;
    return n;
}

// strcpy: kopiert einen String in einen anderen:
void strcpy (char *s1, const char *s2)
{   while ( *s1++ = *s2++)
    ;
}
```

Modul: 5-modul/strpriv.cpp

```
/* Headerdatei strpriv.h: */

/*strcpy: kopiert einen String in einen anderen: */
void strcpy (char *, const char *);

/* strlength: gibt die Laenge des Strings s zurueck: */
int strlength (char *);
```

Headerdatei: 5-modul/strpriv.h

b) Matrizenmodul:

```
#include "matrizen.h"
// Multipliziert Matrizen:
void multmatrix(const matrixtyp A, const matrixtyp B,
                      matrixtyp C, int dim)
{   int i, j, k;
    float summe;
```

```
   for (i=0; i<dim; i++)
      for (j=0; j<dim; j++)
      {  summe = 0;
         for (k=0; k<dim; k++)    // Matrizenmultiplikation
            summe += A[i][k] * B[k][j];
         C[i][j] = summe;
      }
}
// addiert Matrizen:
void addmatrix (const matrixtyp A, const matrixtyp B,
                      matrixtyp C, int dim)
{  int i, j;
   for (i=0; i<dim; i++)
                                    /*    usw.   */
```

Modul: 5-modul/matrizen.cpp

```
/* Headerdatei: Matrizenmultiplikation */
const int DIM = 10;
typedef float matrixtyp [DIM] [DIM];  /* 10 mal 10 Matrix */

/* addiert Matrizen: */
void addmatrix( const matrixtyp, const matrixtyp,
                      matrixtyp, int);

/* Multipliziert Matrizen: */
void multmatrix( const matrixtyp, const matrixtyp,
                       matrixtyp, int);
```

Headerdatei: 5-modul/matrizen.h

c) Hauptprogramm:

```
#include "strpriv.h"
#include "matrizen.h"
#include <stdio.h>
int main ()
{  int i,j;
   matrixtyp A = { {1, 2}, {3, 4}},
             B = { {1, 1}, {2, 2}},              C;
   char s1[80] = "ein schoener Tag",             s2[80];
   addmatrix (A, B, C, 2);     /* in matrizen.h */
   for (i=0; i<2; i++)
   {  for (j=0; j<2; j++)
         printf ("%10.2f", C[i][j] );
      putchar('\n');
   }
   strcpy (s2, s1);            /* in strpriv.h */
   printf("String >%s< hat %d Zeichen\n", s2,strlength(s2));
   return 0;
}
```

Programm: 5-modul/hauptprg.cpp

Das gesamte Programm läßt sich mit dem Befehl

cc hauptprg.c strpriv.c matrizen.c

übersetzen. Dies hat jedoch den Nachteil, daß bei geringfügigem Ändern in einer der drei Dateien immer alle Teile komplett neu übersetzt werden müssen. Stattdessen können auch alle Programme einzeln und unabhängig voneinander übersetzt werden:

cc -c hauptprg.c
cc -c strpriv.c
cc -c matrizen.c

Mit dem Befehl

cc -o haupt hauptprg.o strpriv.o matrizen.o

werden schließlich die einzelnen Module zum Programm *haupt* zusammengebunden. Leider birgt sich in dieser Vorgehensweise eine neue Gefahrenquelle. Aus Versehen könnten ältere nicht mehr aktuelle

Module mit neueren Modulen zusammengebunden werden. Der Linker selbst kennt nur die Externverweise. Er weiß nicht, ob diese Verweise auch wirklich korrekt zusammenpassen. Zur Sicherstellung, daß auch wirklich alle Komponenten eines größeren Projekts den neuesten Stand besitzen, gibt es (nicht nur) unter Unix den Make-Befehl und ein sogenanntes *Makefile*.

5.6 Makefile

In die Datei mit dem Namen *Makefile* oder *makefile* werden alle Abhängigkeiten eines Programms von seinen Programmteilen vermerkt. Gleichzeitig wird angegeben, welche Aktionen gegebenenfalls durchzuführen sind, um das Programm zu aktualisieren. Der Befehl *make* wird ein *Makefile* durchsuchen und die notwendigen Aktionen entsprechend den Vorgaben im *Makefile* durchführen.

Folgende Aktionen kennt *make* standardmäßig:

cc -c name.c	um eine Datei namens name.c in das Modul *name.o* überzuführen
cc -o name name.o	um ein Modul *name.o* in das ausführbare Programm *name* überzuführen

Der Aufruf

make name

wird demnach die Datei *name.c* nach obigen Regeln übersetzen und binden, auch wenn kein *Makefile* existiert. Für C++ Programme nützen uns die Voreinstellungen des Make-Befehls leider nichts. Doch das Schreiben eines nur wenige Zeilen umfassenden Makefiles erlaubt uns die Verwendung eines Makefiles auch für C++:

```
CC = g++                         # aktueller Compiler
.SUFFIXES: .cpp .cc .c .o        # diese Endungen unterstuetzen!
# alle Dateien, die mit .cpp enden, werden so uebersetzt:
.cpp:
        $(CC) –o $* $<
```

Makefile: 5-modul/makefile (Auszug)

Dieses Makefile sieht unverständlich aus. Betrachten wir daher Zeile für Zeile. Zunächst ist das Nummernzeichen (‚#') der Beginn eines Kommentars. Das Kommentarende ist automatisch das Zeilenende. In der ersten Zeile steht eine Variablendeklaration, erkennbar an der Zuweisung. Ab sofort steht also ‚CC' für die Ziffern ‚g++'. Ein Aufruf von Variablen an späterer Stelle erfordert ein voranstehendes Dollarzeichen (‘$’). Besteht die Variable aus mehr als einem Zeichen, so ist der Variablenname in Klammern zu setzen. Somit verstehen wir bereits den Ausdrucks ‚$(CC)' in der letzten Zeile. Er entspricht einfach dem Aufruf des C++ Compilers. Jede Variablendefinition muß in einer eigenen Zeile erfolgen.

Die zweite Zeile gibt an, welche Endungen der Make-Befehl bearbeiten soll. Diese Zeile ist notwendig, da der Makebefehl standardmäßig zwar die Endungen *cc*, *c* und *o* kennt, nicht jedoch *cpp*! Beachten wir die Groß- und Kleinschreibung!

Nach der Kommentarzeile folgen die zwei wesentlichen Zeilen. Die Zeile mit dem Doppelpunkt gibt an, wann etwas zu tun ist; die folgende Zeile gibt an, was zu tun ist. Wir lesen diese Zeilen wie folgt: eine Datei mit der Endung ‚.cpp' ist in eine Datei ohne diese Endung überzuführen. Leisten muß dies der Befehl in der letzten Zeile. Diesem Befehl geht immer ein Tabulatorzeichen voraus! Müssen mehrere Befehle ausgeführt werden, so folgen diese zeilenweise, immer mit einem beginnenden Tabulatorzeichen. Der Befehl in diesem Makefile verwendet einige vordefinierte Variablen, die im folgenden erklärt werden:

$<	diese Variable steht in Vorgaberegeln für die Dateinamen, die diese Aktion auslösen
$*	diese Variable steht in Vorgaberegeln für die Dateinamen, die diese Aktion auslösen, wobei die Endung weggelassen ist

$? diese Variable steht für die Dateinamen, die diese Aktion auslösen und jünger als die Zieldatei sind

Schreiben wir etwa

make prog

so steht ‚$<‘ für *prog.cpp* und ‚$*‘ für *prog*. Die letzte Zeile lautet demnach nach dem Ersetzen der Variablennamen:

g++ -o prog prog.cpp

Dies ist aber nichts anderes als der gesuchte Befehl, der eine Quellprogrammdatei in eine ausführbare Datei übersetzt, wobei bei der ausführbaren Datei nur die Endung entfernt wird. Nach so viel Vorarbeit wagen wir uns an ein umfangreicheres Makefile:

```
# Variablendefinitionen:
OBJ   = hauptprg.o matrizen.o strpriv.o
SCR   = hauptprg.cpp matrizen.cpp strpriv.cpp matrizen.h \
        strpriv.h
CFLAG = -g                # Debug-Info

.SUFFIXES: .cpp .cc .c .o
.cpp.o:
        g++ -c $(CFLAG) $<

hauptprg : $(OBJ)
        g++ -o hauptprg $(OBJ)

hauptprg.o matrizen.o : matrizen.h
hauptprg.o strpriv.o  : strpriv.h

# alle geaenderten Quellprogramme werden gedruckt:
# '$?' steht fuer Dateien, juenger als die Zieldatei

printall:
        lp $(SCR) > printneu
printneu: $(SCR)
        lp $? > printneu
```

Makefile: 5-modul/makefile

Die meisten Teile dieses Makefiles kennen wir schon. Wir müssen uns im wesentlichen nur noch um die Zeilen kümmern, die die Abhängigkeiten bzw. Vorgaben angeben. Dies sind diejenigen Zeilen, die Doppelpunkte enthalten. Grundsätzlich gilt: Links vom Doppelpunkt stehen die Dateinamen, die erzeugt werden sollen, und rechts davon stehen diejenigen, aus denen sie erzeugt werden. Wir erkennen beispielsweise, welche Headerdateien in welchen Modulen verwendet werden. Wir sehen auch, von welchen Dateien das Hauptprogramm abhängt. Die Abhängigkeit der Module mit der Endung ‚.o‘ von den Quelldateien beschreibt die Zeile ‚.cpp.o:‘. Dies besagt, daß die Module von Dateien mit der Endung ‚.cpp‘ abhängen. Wie diese überführt werden, beschreibt dann die Folgezeile.

Der Aufruf *make* ohne Parameter führt übrigens die erste Abhängigkeit aus, die im Makefile steht, und aktualisiert deshalb unser Programm aus diesem Kapitel. Der Aufruf *make printall* gibt alle Quellprogramme auf Drucker aus, der Aufruf *make printneu* nur die seit dem letzten Ausdrucken geänderten. In den letzten beiden Fällen wird das Druckerprotokoll in die Datei *printneu* geschrieben. Mittels dieser Datei erkennt *make*, ob eine Datei gedruckt werden muß oder nicht. Wir sehen, daß ein Makefile nicht nur zu Übersetzungszwecken angewendet werden kann. Die Abhängigkeit *printall* besitzt rechts vom Doppelpunkt keine Angabe und wird demnach immer ausgeführt.

Der Make-Befehl erkennt all diese Abhängigkeiten natürlich am letzten Änderungsdatum der einzelnen Dateien, und nicht existierende Dateien werden natürlich immer erzeugt.

6 Datenstrukturen (struct und union)

6.1 Die Datenstruktur struct

Der Datentyp *struct* entspricht im wesentlichen den Records ohne varianten Teil in Pascal. Mit

```
struct person
   {   char name [20];
       char vorname [15];
       int alter;
   } ;
```

wird in C der Datentyp *struct person* und in C++ der Datentyp *person* definiert. Dies entspricht in Pascal einer Type-Definition. Mit

```
struct person  hans, erna ;    /* C-Version */
```

bzw.

```
person hans, erna ;            // C++ Version
```

werden schließlich zwei Variablen *hans* und *erna* vom Datentyp *struct person* (*person*) deklariert. Beide obige Anweisungen können in C und C++ auch zusammengefaßt werden:

```
struct person
   {   char name [20];
       char vorname [15];
       int alter;
   } hans, erna ;
```

Damit sind sowohl die Variablen *hans* und *erna*, als auch der Datentyp *struct person* (*person*) deklariert. Wir erkennen, daß in C der Datentyp nicht *person*, sondern *struct person* lautet. Dies ist jedoch sehr umständlich. Wir werden uns im weiteren daher an die C++ Version halten und den Namen *struct* immer weglassen. Bedenken Sie jedoch, daß dies bei einem reinen C-Compiler zu Fehlermeldungen führen wird. Ergänzen Sie daher gegebenenfalls diese Datentypen mit dem Wort *struct*.

Zu beachten ist dringend, daß eine Strukt-Definition immer mit einem Semikolon endet, auch wenn keine Variablen deklariert werden sollen. Weiter kann der Struktur-Name (hier: *person*) auch weggelassen werden, wenn weiter kein entsprechender Datentyp verwendet werden soll.

Zu beachten ist weiter, daß es sich bei Strukturen nicht wie bei Feldern um Zeiger handelt. Folgende Anweisung kopiert tatsächlich die gesamten Daten in die neue Variable *sepp*:

```
person sepp;
sepp = hans;          // Kopie
```

Der von *sizeof* zurückgelieferte Wert (hier ca. 40) gibt den belegten Speicher der Struktur an. Neben einem Gesamtzugriff auf eine Struktur können wir auch auf die einzelnen Elemente zugreifen. Der dazugehörige Operator ist wie in Pascal der Punkt ('.'). Weiter sind selbstverständlich auch Strukturen von Strukturen oder Felder von Strukturen oder Strukturen von Feldern denkbar. Weiter können Funktionen auch Strukturen als Rückgabewert besitzen. Betrachten wir ein Beispiel, wo eine Funktion eine Struktur mit einer Vorbelegung zurückliefert:

```
person vorbelegen (void)
{   person temp;
    temp.name[0] = temp.vorname[0] = '\0';
    temp.alter = 0;
    return temp;
}
```

Strukturen können auch initiiert werden. Wir können unsere kleine Funktion daher kürzer fassen:

```
person vorbelegen (void)
{   person temp = { " ", " ", 0 };
    return temp;
}
```

Ein Aufruf dieser Funktion, etwa durch

```
person peter = vorbelegen ( );
```

reserviert und belegt Speicher für *temp*, und kopiert diesen beim Beenden der Funktion in die Struktur *peter* und gibt sofort danach den Speicher für *temp* wieder frei. Betrachten wir nun folgendes Feld:

```
person einwohnerkartei [10000];
```

Damit ist ein Feld, bestehend aus 10000 Elementen vom Typ *person* deklariert. Das Setzen des Alters der 1017. Person auf den Wert 21 ist mit einer der beiden folgenden Zuweisungen möglich:

```
einwohnerkartei [1017] . alter = 21;
(*(einwohnerkartei + 1017) ) . alter = 21;
```

Da der Punkt stärker als der Stern bindet, sind im zweiten Beispiel Klammern zwingend notwendig!

Sollen Strukturen als Parameter an Funktionen übergeben werden, so empfiehlt es sich aus Laufzeitgründen häufig, nur die Zeiger auf diese Strukturen weiterzureichen. Aber vor allem bei den Listen- und Baumstrukturen werden regelmäßig Zeiger auf Strukturen verwendet. Ein Zeiger auf eine Struktur kann wie folgt deklariert werden:

```
person * p_hans;
```

Wurde dem Zeiger *p_hans* ein Speicherbereich zugewiesen, so wird der Familienname mit

```
(*p_hans).name
```

angesprochen. Wieder sind wegen der hohen Bindungsstärke des Punktes Klammern erforderlich. Da diese Zugriffsform häufig vorkommt, das Schreiben der Klammern aber lästig ist, existiert in C eine Abkürzung, unter Verwendung des Operators '->' (Pfeil). Obiger Ausdruck ist dann äquivalent zu:

```
p_hans->name
```

Auch hier ist zu beachten, daß Pfeil wie Punkt die höchste Bindungsstärke besitzt. Im Ausdruck

```
++p_hans->alter
```

wird das Alter und nicht der Zeiger auf die Struktur erhöht. Wir werden die Strukturen im Kapitel über dynamische Strukturen noch intensiv verwenden.

Zum Schluß sehen wir uns in diesem Abschnitt zum Vergleich die Möglichkeiten zum Reservieren von dynamischen Speicher für Strukturen in C und C++ an:

```
/* in C und C++: */

struct list
{   struct list * next;
    int inhalt;
};
struct list *p;

p = (struct list *) malloc (size of (struct list));

p->next = NULL;
p->inhalt = 1;
```

```
// nur in C++:

struct list
{   list * next;
    int inhalt;
};
list *p;

p = new list;

p->next = NULL;
p->inhalt = 1;
```

Es fällt auf, daß in C++, wie bereits bekannt, das Wort *Struct* außer in der Deklaration immer weggelassen werden darf. Das Hauptaugenmerk liegt bei diesem Beispiel jedoch auf dem Reservieren von Speicher. Wir erkennen deutlich, daß diese Speicherreservierung mit dem Operator *new* wesentlich eleganter geschieht als mit der schwerfällige *Malloc*-Funktion.

6.2 Unions

In Pascal haben wir auch variante Records kennengelernt. Diese sind nicht mit Hilfe von *struct* realisierbar. Hier gibt es in C den Datentyp *union*. Der Datentyp *union* besitzt die identische Syntax wie der Datentyp *struct*. Als einziger Unterschied werden mit *struct* die angegebenen Daten hintereinander im Speicher angelegt, bei *union* hingegen überlappend. Die Aussagen für C++ gelten bei *union* analog. Auch hier kann das Wort *union* beim Datentyp einfach weggelassen werden.

Interessiert uns etwa bei einer verheirateten Person der Partnername, bei geschiedenen hingegen nur das Scheidungsjahr und bei ledigen nur der Hinweis, daß die Person ledig ist, so wäre es Speicherverschwendung, diese drei Daten mittels eines *struct* hintereinander im Speicher abzulegen. Wir verwenden statt dessen eine *union*:

```
union
{   int scheidungsjahr;
    char partnername [20];
    char fam_stand [6];
} partner ;
```

Mittels des Operators *sizeof* wird man feststellen, daß die Variable *partner* nicht mehr als 20 Byte Speicher belegt. Zum Zugriff auf Unions können wie bei Structs die Operatoren '.' (Punkt) und '->' (Pfeil) verwendet werden.

Eine häufige Anwendung finden Unions in der systemnahen Programmierung. Hier werden oft Hardwareschnittstellen je nach Anwendung unterschiedlich belegt. Diese Belegungsmuster können mittels Unions übereinander gelegt werden. Jede Anwendung kann so sein eigenes Muster verwenden.

Wir wollen hier mit Hilfe von Unions zeigen, wie Short-Variablen gespeichert werden. Dazu überlappen wir eine 2-Byte Variable vom Typ *short* mit zwei vorzeichenlosen Zeichen.

short a:	char b[0]	char b[1]

```
/* Ueberlagerung mit Unions */

#include <iostream.h>

void uniondemo (const short zahl)
{  union
   {   short a;                  // 16 Bit Zahl
       unsigned char b [2];      // 2  8-Bit Zahlen
   }  ;

   a = zahl;    // Zuweisung an a

   // Ausgabe von b:
   cout.fill('0');
   cout << "Ausgabe byteweise: " << hex << setw(2)
        << int(b[0]) << ' ' << setw(2) << int(b[1]) << endl;
}
```

Programm: 6-struct/union.cpp

Wir erkennen an diesem Beispiel, daß wir in C immer auf jede Variante zugreifen können. Einschränkungen gibt es nicht, die Verantwortung liegt beim Programmierer. Darüberhinaus haben wir in diesem Beispielprogramm noch eine weitere Besonderheit verwendet, die nur in C++ existiert:

In C++ gibt es nämlich die Möglichkeit, bei der Deklaration einer Union weder Datentyp noch Variable anzugeben. In diesem Fall sind die im Union deklarierten Variablen direkt verfügbar, also nicht über eine Unionvariable, gefolgt von einem Punkt. Damit sind in diesem Programm *a* und *b* zwei „normale" Variablen, die allerdings den gleichen Speicherplatz belegen.

7 Rekursion

Die Rekursion ist eine Fertigkeit, die bereits in Pascal geübt wurde. Die Handhabung der Rekursion in C entspricht voll der in Pascal. Grundsätzlich dürfen in C alle Funktionen rekursiv aufgerufen werden (außer *main*). Betrachten wir als Einführungsbeispiele die Fakultätsfunktion und die Fibonacci-Zahlen:

```
// Fakultaet:
long fak ( const int n )
{  return n>0 ? n*fak(n-1) : 1;
}

// Fibonacci-Zahlen:
long fibonacci ( const int n )
{  return n>1 ? fibonacci(n-1)+fibonacci(n-2) : n;
}
```

Programm: 7-rekurs/fakultat.cpp

Wir wollen uns jetzt zur Vertiefung an ein komplexeres Beispiel wagen, das **Acht-Damen-Problem**:

Auf einem Schachbrett mit 8x8 Feldern sind acht Damen so zu plazieren, daß sie sich nicht gegebenseitig schlagen können. Eine Dame kann eine andere Figur nach den Schachregeln dann schlagen, wenn sie das Feld, auf dem die andere Figur steht, mit einem Zug erreichen kann. Die Dame darf in einem Zug beliebig weit waagrecht, senkrecht oder diagonal ziehen.

Obwohl es zunächst unmöglich erscheint, die acht Damen passend zu plazieren, gibt es dennoch 92 Lösungen. Diese Lösungen gibt auch unser Programm *dame.c* aus, das wir am Anfang der Vorlesung kennengelernt haben, wenn wir nach dem Programmstart den Wert 8 (für die Feldgröße 8x8) eingeben. Wir wollen hier jedoch ein eigenes strukturiertes Programm herleiten.

Da keine Formel für das Aufstellen der Damen bekannt ist, müssen wir die einzelnen Lösungen durch systematisches Probieren bestimmen. Hier bietet sich das sogenannte **Backtracking**-Verfahren an:

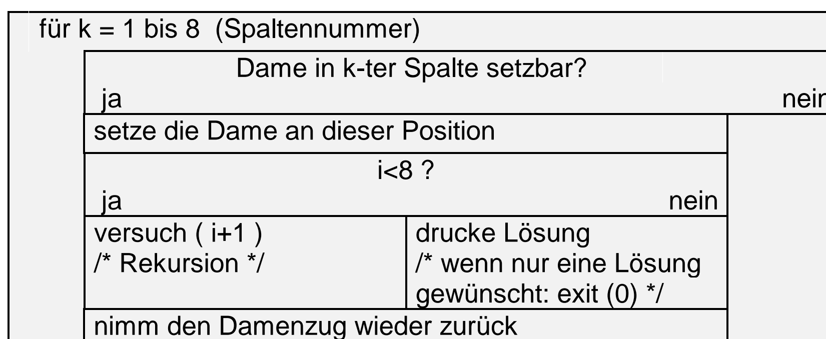
Hier wird nach einer vorgegebenen Regel ein Zug nach dem anderen ausgeführt, bis entweder eine Lösung gefunden wird oder alle Zugmöglichkeiten erschöpft sind. Ein einzelner Zug besteht dabei immer aus dem gleichen Algorithmus:

Ist ein neuer Zug möglich?
ja: ziehe diesen Zug
nein: nimm den letzten Zug zurück

Wir wollen dieses Schema nachvollziehen, wobei wir alle Lösungen des Acht-**Damen**-Problems ausgeben wollen. Unsere wichtigste Funktion wird die Funktion

void versuch (int i)

sein, die den nächsten Zug durchführt, in unserem Falle also die i-te Dame setzt. Die Funktion wird rekursiv programmiert. Das Struktogramm ist angegeben:



Typisch für die Rekursion ist wieder, daß wir uns nur einen einzigen Fall genau überlegen müssen. Alles andere übernimmt die Rekursion. Der Algorithmus ist damit bereits vollständig angegeben. In der

Praxis bereiten allerdings verbale Ausdrücke wie „Dame in k-ter Spalte setzbar“ technische Schwierigkeiten. Es empfiehlt sich nicht, tatsächlich ein 8x8-Feld zu reservieren und dort die Damen an die entsprechende Position zu setzen. Wir würden zu viele redundante Informationen speichern, was den Programmieraufwand und die Laufzeit unnötig vergrößern.

Da eine Dame waagrecht beliebig weit ziehen kann, darf je Zeile nur eine Dame stehen. Somit setzen wir die erste Dame in die erste Zeile, die zweite in die zweite usw. Wir müssen uns somit nur die Spaltenposition merken, an der die i-te Dame steht. Wir brauchen daher nur ein Feld aus 8 int-Elementen für unsere acht Damen. Probleme bereitet aber noch zu erkennen, welche Felder von bereits gesetzten Damen beherrscht werden. Bei jedem Zug zu überprüfen, ob eine bereits gesetzte Dame dieses Feld senkrecht oder diagonal erreichen kann, ist zu rechenintensiv. Es empfiehlt sich daher, für die Spalten und die /-Diagonalen und die \-Diagonalen je ein weiteres Feld zu definieren. Es gibt 8 Spalten und jeweils 15 Diagonalen auf einem 8x8-Schachbrett. Die Diagonalen sind gekennzeichnet durch die Eigenschaften:

Zeilennummer + Spaltennummer = const.	(/-Diagonale)
Zeilennummer - Spaltennummer = const	(\-Diagonale)

Bei jedem Setzen einer Dame wird demnach die entsprechende Spaltennummer in das Damenfeld eingetragen und die Einträge zu der dazugehörigen Spalte und den Diagonalen auf 1 gesetzt. Beim Zurücknehmen eines Damenzugs, werden die Spalte und die beiden Diagonalen wieder auf 0 zurückgesetzt. Dies erfordert eine Initiierung der entsprechenden Felder mit dem Wert 0. Da in C alle Felder mit dem Wert 0 beginnen, laufen die Spalten- und Zeilennummern von 0 bis 7. Für die drei Merkfelder gilt:

Spalte:	0 . . 7	
/-Diagonale	0 . . 14	(Zeile + Spalte)
\-Diagonale	0 . . 14	(Zeile - Spalte + 7)

Es folgt nun die implementierte Funktion *versuch*:

```
void drucke (const int *);        // gibt die Loesungen aus

void versuch (int i)
{  static int spalte[8] = {0},    // statisch vorbelegt
              d1[15]    = {0},
              d2[15]    = {0},
              dame[8];            // Feld fuer die 8 Damen

   for (int k=0; k<8; k++)
      if (spalte[k]==0 && d1[i+k]==0 && d2[i-k+7]==0)
      {                                 // Dame ist setzbar
         dame[i] = k;  /* Dame wird gesetzt */
         spalte[k] = d1[i+k] = d2[i-k+7] = 1;
         if (i < 7)   versuch (i+1);    // Rekursion
           else       drucke(dame);
         spalte[k] = d1[i+k] = d2[i-k+7] = 0;
      }
}
```

Programm: 7-rekurs/achtdame.cpp

Das Hauptprogramm ruft im wesentlichen nur die Funktion *versuch(0)* auf. Weiter ist noch eine geeignete Funktion *drucke* zum Ausgeben einer Lösung zu schreiben. Im Programm *7-rekurs/ndame.cpp* ist darüberhinaus auch eine Lösung des n-Damenproblems angegeben. Es wird empfohlen, dieses Programm zunächst selbst zu erstellen.

8 Dynamische Datenstrukturen

8.1 Lineare Listen

Lineare Listen sind aus der Vorlesung „Programmieren in Pascal“ bekannt. Wir müssen jetzt nur die

Programmiertechniken von C verwenden, algorithmisch kommt nichts neues hinzu. Zunächst müssen wir die Datenstruktur eines Listenelements festlegen. Ist der gespeicherte Inhalt ein Zeichenkettenfeld, so können wir diese Struktur wie folgt definieren:

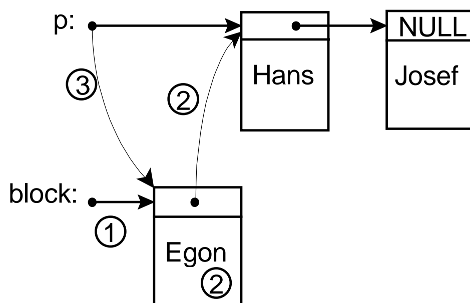
```
const int N = 80 ;
struct list
{   list *next;
    char element [N];
} ;
```

Beachten Sie, daß *sizeof (list)* den Wert 84 zurückliefert, 4 Bytes für den Zeiger und 80 Bytes für die Zeichenkette. Beginnen wir damit, ein Element *block* obigen Typs am Anfang einer Linearen Liste *p* einzuketten. Wir gehen dazu in folgenden Schritten vor:

(1) Ein neues Element wird erzeugt (mit *block* als Zeiger auf dieses Element)

(2) *Block* mit Inhalt versorgen, also Text in das Zeichenkettenfeld *element* eintragen und den Zeiger auf den bisherigen Listenanfang zeigen lassen

(3) Der bisherige Listenanfang *p* wird auf das neue Element block umgebogen



Betrachten wir die Realisierung dieses Einkettens mit Hilfe der Sprache C++:

```
void einketten ( const char *name, list* &p)
{
   list *block;

   block = new list;                  // 1: Speicher belegen
   strcpy(block->element,name);       // 2: Inhalt eingeben
   block->next = p;                   // 2: verketten
   p = block;                         // 3: neu einhaengen
}
```

Programm: 8-dynamic/liste.cpp (Funktion *einketten*)

Zu beachten ist folgendes:

– Eine Zuweisung der Form ‘block->element = name’ ist nur eine Zeigerzuweisung und wäre hier ein Syntaxfehler, da *block->element* ein (konstanter) Feldzeiger ist. Wir müssen im Gegensatz zu Pascal in C immer die Funktion *strcpy* verwenden. Vor allem der Anfänger macht hier häufig Fehler.

– Die Funktion *einketten* arbeitet auch bei zunächst leerer Liste. Die Liste muß allerdings mit ‘*p=NULL;*’ initiiert worden sein.

– Wir legen uns bei der Speicherreservierung von Anfang an auf C++ fest.

– Der Zeiger *p* auf die Lineare Liste muß Call-by-Reference übergeben werden, damit der Zeiger *p* mit dem aufrufenden Zeiger des Hauptprogramms identifiziert wird.

Wir wollen anhand des Hauptprogramms den Aufruf der Funktion *einketten* aufzeigen:

```
int main()
{  list *listpointer = NULL;    // korrekt initiiert
   char name[N];

   cout << "Listenprogramm\n\nElementgroesse: "
        << sizeof (list) << '\n';
   cout << "Namen eingeben. Ende ist leerer String!";
   cin.getline (name, N) ;
   while ( strlen(name) != 0 )
   {  einketten (name, listpointer);
      cin.getline (name, N);
   }
   cout << "Liste enthaelt " << anzahl(listpointer)
        << " Elemente\n";
   ausgeben (listpointer);
   return 0;
}
```

Programm: 8-dynamic/liste.cpp (Funktion *main*)

Betrachten wir noch die in *main* vorkommenden Funktionen *anzahl* und *ausgeben*, die die Anzahl der Elemente einer Liste ermitteln bzw. alle Elemente der Liste ausgeben:

```
int anzahl (const list *p)
{  int i=0;
   while (p != NULL)      /* kuerzer:  while (p) */
   {  p = p->next;
      i++;
   }
   return i;
}

void ausgeben (const list *p)
{  while (p != NULL)
   {  cout << p->element << '\n';
      p = p->next;
   }
}
```

Programm: 8-dynamic/liste.cpp (Funktionen *anzahl* und *ausgeben*)

Dieses Beispiel läßt erkennen, daß das Arbeiten mit Listen in C++ und in Pascal bis auf die C++-typischen Konstrukte übereinstimmen. Zur Wiederholung des Stoffes der Pascal-Vorlesung wollen wir ein komplexeres Listenbeispiel ausarbeiten, das Suchen und Einfügen in einer geordneten Liste.

Um die Aufgabe überschaubar zu halten, trennen wir das Erzeugen und Vorbelegen eines neuen Elements mittels einer eigenen Funktion *erzeuge* ab. Der Funktionskopf lautet:

list * erzeuge (char *name)

Die Zeichenkette wird an die Funktion übergeben, der Zeiger auf den Speicherbereich wird zurückgeliefert. Die Funktion wird also in der Form

block = erzeuge (name);

aufgerufen. Es sei zur Übung dringend empfohlen, diese Funktion zunächst selbständig auszuarbeiten, bevor das Lesen im Skript fortgesetzt wird. Wir müssen in dieser Funktion zunächst Speicherplatz reservieren, dann den Namen in das Strukturelement eintragen, den Zeiger in der Struktur auf NULL setzen und anschließend diese Struktur zurückgeben.

```
list * erzeuge ( const char * name)
{  list * block = new list;

    // vorbelegen:
   strcpy ( block->element, name );
   block->next = NULL;

   return block;
}
```

Programm: 8-dynamic/suche.cpp (Funktion *erzeuge*)

Wir kommen nun zur zentralen Funktion *suche*. Dieser Funktion werden zwei Parameter übergeben, ein Zeiger auf den Listenanfang und ein Zeiger auf das neue Element. Aufgabe der Funktion ist es, das neue Element geordnet in die Liste einzuhängen. Leider kann dieses neue Element gelegentlich auch an den Anfang der Liste plaziert werden. Die Folge ist, daß dann der übergebene Listenzeiger geändert wird. Dies muß sich auch auf den Listenzeiger des Hauptprogramms auswirken, der Listenzeiger ist demnach call-by-reference aufzurufen. Ist *listpointer* der Listenzeiger und *block* ein Zeiger auf einen neuen Block, so lautet der Funktionsaufruf:

```
suche (listpointer, block);
```

Der Kopf der Funktion *suche* lautet:

```
void suche ( list * &root, list *block ) ;        // 1. Parameter ist Call-by-Reference
```

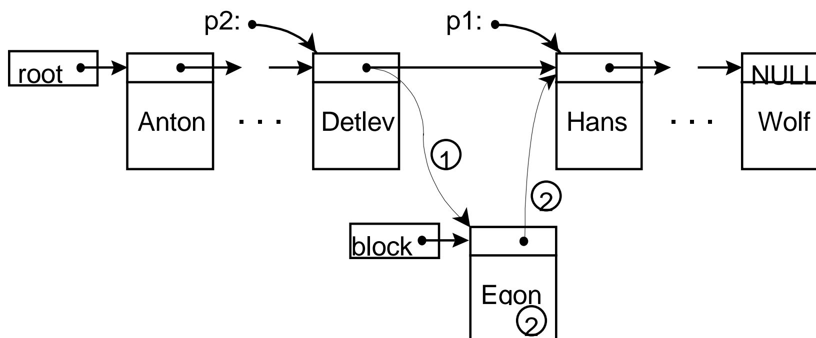
Wir kommen jetzt zur Herleitung des Algorithmus. Es sind bei der Suche drei Fälle zu unterscheiden:

1) Die Liste ist leer (daher list * &)

2) Der neue Block wird am Anfang eingekettet (daher list * &)

3) sonst: Durchsuchen des richtigen Platzes für das neue Element

Im letzten Fall müssen wir die Liste durchsuchen. Unser Problem bei der Suche ist allerdings, daß wir beim Vergleich der Zeichenkette eines Listenelements mit der einzuhängenden Zeichenkette immer nur entscheiden können, ob das neue Element davor einzuhängen ist. Um aber vor einem Element einzuketten, benötigen wir einen Zeiger auf das Vorgängerelement. Es empfiehlt sich daher, mit zwei Zeigern *p1* und *p2* zu arbeiten, wobei *p2* dem Zeiger *p1* immer um ein Element hinterherläuft. Die folgende Grafik veranschaulicht, wie ein Element in eine geordnete Lineare Liste einzuhängen ist. Die Laufzeiger *p1* und *p2* geben in der Grafik die Position an, die sie belegen, wenn der Platz gefunden wurde, wo das gewünschte Element (hier: *Egon*) einzuhängen ist.



Und nun zur Implementierung:

```
void suche ( list * & root, list *block )
{  if (root == NULL)                     // 1. Fall
       root = block;
   else
    if ( strcmp (root->element, block->element) > 0 )
    {  block->next = root;               // 2. Fall
       root        = block;
    }
    else                                 // 3. Fall
    {  list * p1, *p2;               // Laufzeiger
       p1 = root;
       do                            // Fortschalten
       {   p2 = p1;    p1 = p1->next;
       }
       while ( p1 != NULL &&
               strcmp (p1->element, block->element) <= 0 );
       p2 -> next = block;           // einketten
       block -> next = p1;
    }
}
```

Programm: 8-dynamic/suche.cpp (Funktion *suche*)

Beachten Sie insbesondere das Abbruchkriterium der Do-While-Schleife. Wegen der garantierten Bewertung eines Ausdrucks von links nach rechts bei Verwendung des &&-Operators, können wir zunächst einen Zeiger auf NULL überprüfen und dann auf den Inhalt zugreifen.

Der Vergleich von Zeichenketten ist in C nicht gerade elegant und erfolgt im Programm mittels der Funktion *strcmp*.

8.2 Bäume

Wir beschränken uns auf Binärbäume. Die Struktur eines Baumelements habe folgenden Aufbau:

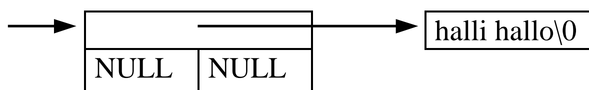
```
struct knoten
    {   knoten *links, *rechts;
        char * inhalt;                  /* diesmal nur ein Zeiger auf einen String!!! */
    } ;
```

Bereits aus dem 1. Semester ist das Erzeugen eines Elements und das Einfügen und Suchen in einem Baum bekannt. Wir wollen dies kurz wiederholen. Beginnen wir mit dem Erzeugen eines neues Elements. Wieder wird dringend empfohlen, diese einfache Funktion zunächst selbst zu erarbeiten, bevor mit dem Lesen fortgesetzt wird. Beachten Sie allerdings, daß wir die Funktion malloc zweimal aufrufen müssen, einmal zum Reservieren von Speicher für den Knoten und zweitens zum Reservieren von Speicher für die als Parameter übergebene Zeichenkette.

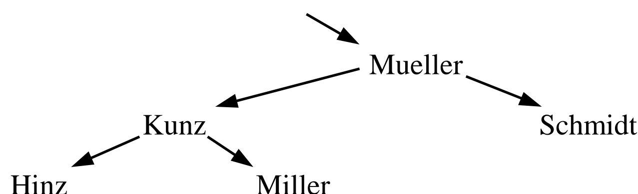
```
knoten * erzeugeknoten ( const char * name )
{
   knoten *p;
   p = new knoten;
   p->links = p->rechts = NULL;
   p->inhalt = new char [strlen(name) + 1];
                         // Speicher fuer String anfordern
   strcpy(p->inhalt, name);
   return p;
}
```

Programm: 8-dynamic/baum.cpp (Funktion erzeuge)

Der Vorteil des dynamischen Reservierens von Speicher für die Zeichenkette ist die hohe Flexibilität: Die Zeichenkette darf beliebig lang sein, und gleichzeitig wird immer nur der gerade benötigte Speicherplatz reserviert. Ist die Zeichenkette ‘halli hallo’, so sieht der reservierte Speicher nach dem Anfordern wie folgt aus:



Zum geordneten Eintragen im Baum verwenden wir die Funktion *suche*. Sie kann sowohl iterativ als auch rekursiv formuliert werden. Wir verwenden die rekursive Form. Die Idee ist folgende. Wir überprüfen das einzufügende Element mit der Wurzel des Baums. Ist das einzufügende kleiner als die Wurzel, so muß es irgendwo links von dieser Wurzel einzufügen sein. Wir geben damit das Problem auf das links folgende Element rekursiv weiter. Irgendwann ist der entsprechende linke oder rechte Nachfolger gleich NULL. Und genau an dieser Stelle ist das Element dann einzuhängen. Soll im folgenden Baum der Name *Meyer* eingefügt werden, so geschieht dies links von *Miller*.



Die Funktion *suche* hat zwei Parameter: die zu suchende Zeichenkette und einen Zeiger auf ein Baumelement. Da in der Rekursion letztlich der Nullzeiger übergeben wird und dieser verändert werden muß, ist dieser Zeiger call-by-reference zu übergeben.

```
void suche ( const char *name, knoten * & baum )
{  int i;
   if (baum == NULL)        // Ende der Rekursion
   {  baum = erzeugeknoten (name);
      return;
   }
   if ( (i=strcmp (name, baum->inhalt)) < 0 )
      suche ( name, baum->links );        // Suche links
   else if (i>0)
      suche ( name, baum->rechts );       // Suche rechts
   else
      cout << "Name " << name << " ist bereits vorhanden\n";
}
```

Programm: 8-dynamic/baum.cpp (Funktion suche)

Als eine interessante Übung sei die iterative Lösung der Funktion *suche* empfohlen. Wir wollen hier noch ein relativ einfaches rekursives Beispiel angeben, nämlich das Ausgeben eines Binärbaums. Wieder sollte der Leser diese Funktion zunächst selbst erarbeiten, bevor er mit dem Lesen fortfährt. Das folgende Beispiel gibt auch noch die Baumstruktur aus. Hierzu wird ein zweiter Parameter verwendet, in dem die Rekursionstiefe gemerkt wird.

```
/* Ausgeben des gesamten Baums mit Anzeige der Tiefe: */

void ausgeben (const knoten * baum, const int tiefe)
{
   if (baum)       //    != NULL
   {
      ausgeben ( baum -> links, tiefe + 1 );
      cout << setw(5*tiefe) << " " << baum->inhalt << '\n';
      ausgeben ( baum -> rechts, tiefe + 1 );
   }
}
```

Programm: 8-dynamic/baum.cpp (Funktion ausgeben)

Zum Abschluß dieses Abschnitts wollen wir das **Löschen** eines Baumelements kennenlernen. Die Bedingung beim Löschen ist, daß der Baum dabei geordnet bleibt. Es gilt, drei Fälle zu unterscheiden:

1. Fall: Der zu löschende Knoten ist ein Endknoten: Der Knoten ist einfach zu entfernen, der Zeiger auf diesen Knoten ist auf NULL zu setzen.

2. Fall: Der Links- oder Rechtszeiger des zu löschenden Knotens ist NULL: Es genügt, den Zeiger auf diesen Knoten auf das einzige Nachfolgeelement umzubiegen. Anschließend kann der Knoten entfernt werden.

3. Fall: Beide Zeiger des zu löschenden Knotens sind ungleich NULL: Der Knoten kann nicht einfach entfernt werden, da mindestens einer der beiden Nachfolgezeiger in der Luft hängen würde. Die Idee ist, das zu löschende Element durch ein Ersatzelement zu ersetzen. Als Ersatz kommt entweder das nächstkleinere oder nächstgrößere Element in Frage.

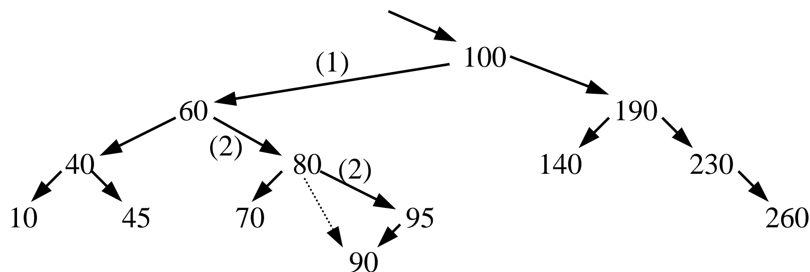
Betrachten wir den Fall, daß wir das nächstkleinere Element als Ersatzelement wählen (der andere Fall ist analog). Dieses Element besitzt die Eigenschaft, daß es keinen rechten Nachfolger besitzt. Es kann also gemäß Fall 1 oder Fall 2 problemlos ausgehängt werden.

Unser Algorithmus besitzt demnach grob folgenden Aufbau: zunächst suchen wir im Baum das zu löschende Element. Ist Fall 3 gegeben, so suchen wir außerdem das nächstkleinere Element. Wir hängen dieses nächstkleinere Element gemäß Fall 1 oder Fall 2 aus und hängen es an die Stelle des zu löschenden Elements wieder ein. Wir entfernen das zu löschende Element. Die Suche des nächstkleineren Elements vom zu löschenden Element aus geschieht wie folgt:

(1) gehe zum linken Nachfolger (dieser existiert!)

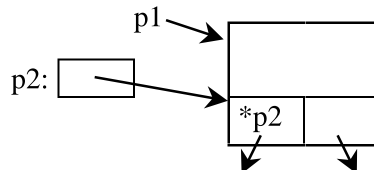
(2) solange der rechte Nachfolger ungleich NULL ist, gehe zum rechten Nachfolger

Betrachten wir dieses Vorgehen anhand eines Zahlenbaums:



In dem Baum ist das Aushängen des Ersatzelements 95 gestrichelt eingezeichnet. Betrachten wir die Funktion *loesche*, die zunächst das zu löschende Element sucht (siehe Teil1 des folgenden Listings) und dann nach obigem Algorithmus aushängt (siehe Teil2 des folgenden Listings).

Beachten Sie bitte im Teil2 des Listings den „Call-by-Reference“-Zeiger *p2*. Dieser zeigt direkt auf den Zeiger eines Knotenelements, und zwar immer auf den Zeiger, der auf den nächsten Knoten verweist. Somit können wir deshalb direkt das Ersatzelement aushängen. Mit Hilfe dieser Methode können wir ganz allgemein in C und C++ auf einen Nachlaufzeiger verzichten. Zeigen wir kurz die Arbeitsweise auf, indem wir die Situation aufzeigen, wenn p2 auf den Linkszeiger des Knotens, auf den p1 zeigt, gesetzt wird:




```
void loesche ( const char * name, knoten * & baum )
{  int i;
   knoten * p1;        // merkt das zu loeschende Element
   knoten **p2;        // sucht Ersatzelement (call by ref.)

  // suchen des zu loeschenden Elements:
   if (baum == NULL)       // Ende der Rekursion
   {  cout << "Element wurde nicht gefunden\n";
      return;
   }
   if ( (i=strcmp (name, baum->inhalt ) < 0 )
      loesche ( name, baum->links );
   else if (i>0)
      loesche ( name, baum->rechts );
   else
      // zu ersetzender Knoten ist gefunden, ausfuegen:
   {  p1 = baum;
      if (p1->rechts == NULL)        // Fall 1 oder 2
         baum = p1->links;
      else if (p1->links == NULL)    // Fall 1 oder 2
         baum = p1->rechts;

      else       // komplexer Fall, kein Nachfolger ist NULL
      {          // Ersatzelement suchen
         p2 = &(p1->links);          // linker Nachfolger
         while ( (*p2)->rechts != NULL)
            p2 = &(*p2)->rechts;     // jetzt nach rechts

         // *p2 zeigt auf Ersatzelement, neu verketten:
         baum = *p2;
         *p2 = (*p2)->links;         // umbiegen
         baum->links  = p1->links;
         baum->rechts = p1->rechts;
      }
      // Element loeschen:
      delete p1->inhalt;       // Speicher fuer String!!!
      delete p1;               // Speicher fuer Element
   }
}
```

Programm: 8-dynamic/baum.cpp (Funktion *loesche*)

Achtung:

> Beachten Sie beim **Entfernen** von Elementen aus einem Baum oder einer Liste immer, daß auch der für dieses Element angelegte Speicher restlos entfernt wird.

Dies sind in unserem Fall der Speicher für die Zeichenkette und der Speicher für das Knotenelement.

Beim Anlegen eines Baumes (mittels *suche*) und beim Löschen von Elementen (mittels *loesche*) können sehr schieflastige Bäume entstehen. Dies kann im schlimmsten Fall dazu führen, daß der Baum zu einer Liste entartet. Um dies zu verhindern, gibt es mehrere Möglichkeiten, die allerdings alle mehr oder weniger aufwendig sind. Hier sei auf ausgeglichene Bäume und B-Bäume verwiesen. etwa in Wirth: Algorithmen und Datenstrukturen oder Ottmann/Widmayer mit dem gleichnamigen Titel. Auch in der Vorlesung *Datenorganisation* wird dieses Thema ausführlich behandelt.

9 Sonstige Möglichkeiten mit C

9.1 Arbeiten mit Bitoperatoren (Wiederholung)

Wir haben bereits ein Beispiel mit Bitmanipulationen kennengelernt. Zur Vertiefung wollen wir ein weiteres Beispiel bringen. Besonders in der Systemprogrammierung werden häufig nur einige Bits einer int-Zahl benötigt. Dies kommt daher, daß die Hardware bestimmte Übertragungsmuster besitzt. Wir benötigen deshalb eine Funktion, die aus einer Bitleiste (int-Zahl) nur bestimmte Bits auswählt. Wir erstellen daher eine Funktion *getbits*, die aus einer int-Zahl genau *n* Bits ab der Bitposition *pos* zurückliefert. In der Hardware wird in der Regel die Bitposition von rechts nach links gezählt, beginnend bei 0. Dies wollen wir berücksichtigen.

Versuchen wir die Aufgabe an einem Beispiel zu verstehen. Betrachten wir eine (hier nur 16 Bit) große vorzeichenlose int-Zahl. Wir wollen ab der Position *pos=8* genau *n=3* Bits zurückliefern.

```
                 3 Bits
i:          0 1 1 0 1 0 0 1 0 1 1 0 0 1 1 0
                          pos=8            pos=0
Rückgabe:   0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
```

Die Funktion *getbits* wird demnach zunächst die Bitleiste um *pos+1-n* Bits nach rechts shiften und anschließend die überflüssigen Bits ausmaskieren. Im folgenden Programmlisting ist auch noch eine Funktion *print_bits* aufgeführt, die das Bitmuster der Bitleiste ausgibt. Wir kommen darauf zurück.

```
unsigned getbits (const unsigned i,
                  const int pos, const int n)
{  return ( i >> (pos+1-n))  &  ~(~0 << n) ;
}

void print_bits (const unsigned i)
{  int k;
   for (k=8*sizeof(unsigned)-1; k>=0; k--)
      cout.put ( i & (1<<k) ? '1' : '0' );
}
```

Programm: 9-sonst/bits.cpp

Die Berechnung der Rückgabeleiste in der Funktion *getbits* sieht recht komplex aus. Betrachten wir diesen Ausdruck daher etwas genauer:

- Der Ausdruck ‘i >> (pos+1-n)’ verschiebt die Zahl *i* um die gewünschten Bits nach rechts
- ‘~0’ ist eine Zahl, die nur aus binären Einsen besteht

- Der Ausdruck '~0<<n' ist eine Zahl, die in den rechten *n* Bits Nullen und sonst Einsen enthält
- Der Ausdruck '~(~0<<n)' ist eine Zahl, die in den rechten *n* Bits Einsen und sonst Nullen enthält. Ist *n=3*, so besitzt sie folgendes Aussehen: 000…0000111. Dies ist eine Und-Maske für den Und-Operator &
- Der Und-Operator setzt bis auf die letzten *n* Bits alle Bits auf Null. Die letzten *n* Bits bleiben unverändert.

Um das Programm zu testen, wäre es schön, eine Funktion zu schreiben, die das Bitmuster einer Zahl ausgibt. Wir bräuchten dazu nur unsere Funktion *itob* aus den Übungen auf unsigned-Zahlen zu erweitern. Oder wir verwenden die im obigen Listing mit angegebene Funktion *print_bits*. Die Angabe *sizeof (unsigned)* in dieser Funktion ermöglicht die Portabilität zwischen verschiedenen Rechnerplattformen. Als Maske wurde diesmal die Zahl eins gewählt, die entsprechend nach links geshiftet wird.

9.2 Präprozessoranweisungen

Die Compilierung (ohne Linker!) sieht wie eine Einheit aus. In Wirklichkeit besteht sie aus mehreren Teilen. Ein Präprozessor bereitet zunächst das Quellprogramm für den Compiler vor. Der Compiler selbst besteht aus einem Parser, der das Quellprogramm untersucht, Variablen bestimmt und Ausdrücke berechnet und dem eigentlichen Compiler, der das C-Programm in Maschinensprache umsetzt.

Der Präprozessor ist eine Besonderheit in C (gegenüber beispielsweise Pascal). Dieser Präprozessor läuft immer vor dem eigentlichen C-Compiler und sucht nach Präprozessoranweisungen. Jede Zeile, die mit dem Zeichen '#' beginnt (vorhergehende Leerzeichen sind erlaubt), wird als Präprozessoranweisung interpretiert. Die wichtigsten dieser Anweisungen sind:

a) # include
b) # define , # undef
c) # if , # ifdef , # ifndef , # elif , # else , # endif

Der Präprozessor interessiert sich ausschließlich für diese Anweisungen. Aus diesem Grund sind diese Präprozessoranweisungen auch völlig unabhängig von der sonstigen Struktur des Programms. Diese Anweisungen wirken daher immer in, ab oder bis zur jeweiligen Zeile.

Wie bereits bekannt, folgt der Include-Anweisung ein Dateiname, der entweder in Gänsefüßchen ("…") oder spitzen Klammern (<…>) geklammert ist. Im ersten Fall wird zunächst nach der Datei im aktuellen Verzeichnis gesucht. Wird eine Datei dieses Namens nicht gefunden, dann wird auch das Include-Verzeichnis des Rechners durchsucht. Im zweiten Fall entfällt das Durchsuchen des lokalen Verzeichnisses. Die gefundene Datei wird nun anstelle der Include-Zeile in die Programmdatei komplett eingefügt. Enthält auch diese Datei Include-Anweisungen, so werden auch diese Dateien kopiert.

Vor allem das letzte Verhalten kann dazu führen, daß zu viele Headerdateien kopiert werden. Dies führt zu sehr langen Compilierungszeiten. Doch viel gravierender ist, daß dadurch eventuell einige Headerdateien mehrfach existieren, was zu mehrfachen Deklarationen von Konstanten und Funktionen und damit zu Syntaxfehlern beim Compilieren führen kann. Gebrauchen Sie daher in Headerdateien möglichst keine Include-Sätze, außer Sie richten sich nach den weiter unten angegebenen Richtlinien!

Mittels der Define-Anweisungen werden Konstanten und Makros (siehe nächster Abschnitt) definiert. Die Undef-Anweisung vergißt diese ab dieser Zeile wieder. Glücklicherweise können wir in C++ fast völlig darauf verzichten.

Die If-Anweisung erlaubt eine bedingte Compilierung. Nach den Zeichen *#if* darf ein konstanter Ausdruck stehen. Ist dieser unwahr, so werden alle Zeilen bis zum nächsten *#elif*, *#else* oder *#endif* entfernt. Ist der Ausdruck wahr, so werden alle Zeilen ab dem nächsten *#elif* oder *#else* bis zum *#endif* entfernt. Hierbei ist *#elif* eine Abkürzung für *#else*, direkt gefolgt von *#if*.

Die beiden Anweisungen *#ifdef* und *#ifndef* sind eine Abkürzung für *#if defined* bzw. *#if !defined*, wobei *defined* ein Präprozessoroperator ist.

Eine geschickte Verwendung dieser If-Anweisung ermöglicht die Compilierung nur des gewünschten Codes. Dies ist bespielsweise für die Verwendung eines Programms auf verschiedenen Rechnerplattformen recht wünschenswert. Die meisten Headerfiles definieren rechnerspezifische Konstanten (UNIX, MSDOS, VMS, …). So könnten etwa für MSDOS folgende Anweisungen interessant sein:

```
#ifdef _MSDOS_
clrscr ( );
#else
puts("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
#endif
```

Unter Unix wird hier wegen der fehlenden Implementierung von *clrscr* eine Latte von Neue-Zeile-Zeichen ausgegeben. Für uns sehr wichtig ist, daß wir mit Präprozessoranweisungen das mehrfache Einlesen der gleichen Headerdateien grundsätzlich verhindern können. Betrachten wir unser Beispiel zur Matrizenberechnen. Wir müssen unsere Headerdatei nur wie folgt erweitern:

```
#ifndef _MATRIX_H
#define _MATRIX_H
/* Hier kommen jetzt die Headereinträge: */
…
#endif
```

Wird jetzt diese Headerdatei mehrfach eingebunden, so wird der Code nur beim ersten Mal durchlaufen, da danach die Konstante _MATRIX_H definiert ist. Wir sehen weiter, daß eine Konstante keinen Wert besitzen muß. Allein die Existenz wurde hier ausgenutzt. Dies ist praktisch die einzige Anwendung der Define-Anweisung in einem guten C++ Programm.

Achtung:

➔ Wir verwenden ab jetzt Headerdateien nur in der obigen Form mit bedingter Compilierung.

9.3 Makros

Makros sind eine Erweiterung der Konstantendefinition. Wir kennen bereits Konstanten. Beispiele:

```
#define MAX  10
#define MIN  5
#define ERR  "Fehler in Zeile __LINE__ in Datei __FILE__ vom __DATE__  __TIME__"
#define MINMIN  ( ( MIN > 0 || -MIN < MAX )      ?       -MAX    \
                                                 :       MIN  )
```

Die dritte Zeile zeigt Möglichkeiten, vordefinierte Konstanten des Präprozessors zu verwenden. Damit kann die aktuelle Zeilennummer, der Dateiname, das Datum und die Uhrzeit der Übersetzung im Programm verwendet werden. Die letzte Definition zeigt die Möglichkeit, bereits definierte Konstanten weiterzuverwenden. Der Backslash '\' dient dabei als Fortsetzungszeichen in die nächste Zeile.

Grundsätzlich gilt: Der Name nach *#define* wird durch die folgenden Zeichen der Zeile ersetzt, und zwar ab dieser Zeile bis zum nächsten *#undef* oder bis Dateiende. Der Compiler wird dann diesen Code nach dem Lauf des Präprozessors interpretieren. In unserem Beispiel wird dringend das Setzen der Klammern empfohlen. Wir kommen darauf zurück.

Wir weisen nochmals darauf hin, keine Datentypen mit *#define* zu definieren. Dazu steht der Bezeichner *typedef* zur Verfügung ('#define string char *' entspricht nicht 'typedef char * string;' !).

Wir können mit der Define-Anweisung auch ganze Funktionen, sogenannte Makros, definieren. Betrachten wir die Absolut-Funktion *abs*, die den Absolutbetrag eines Wertes zurückliefert:

1 oder mehrere Bezeichner, durch Kommata getrennt

```
# define  abs(x)   ( ( (x) >= 0 )  ?  (x) : -(x) )
```

hier keine Leerzeichen!

Kommt jetzt in einer der folgenden Zeilen etwa

```
abs (zahl)
```

vor, so wird dieser Ausdruck durch die rechte Seite der obigen Definition ersetzt, wobei *x* durch den Übergabewert *zahl* ersetzt wird. Aus Sicht des Programms wirken diese Makros wie Funktionsaufrufe. Sie haben sogar den Vorteil, daß der Präprozessor diesen Aufruf expandiert, so daß dadurch zwar größerer Code generiert wird, dafür aber kein echter Funktionsaufruf erfolgt. Dies reduziert die Laufzeit eines Programms. Nicht nur aus diesem Grund sind Makros in C sehr beliebt.

In C++ besitzen wir die Direktive *inline*, so daß die Verwendung von Makros überflüssig wird. Ein Nachteil von Makros ist beispielsweise, daß der C-Compiler die übergebenen Datentypen nicht auf Korrektheit überprüfen kann.

Aber auch sonst besitzen Makros gefährliche Fallen. Um diese zu erkennen, betrachten wir unser Beispiel etwas genauer. Warum wurden so viele Klammern verwendet? Hätten wir diesen Makro nicht kürzer definieren können, etwa zu:

```
# define  abs(x)  x >= 0  ?  x : -x
```

Ist jetzt die Variable *zahl* gleich -4, so liefert *abs(zahl)* den korrekten Wert 4 zurück. Doch folgendes Beispiel zeigt den Fehler der letzten Definition auf:

```
a = 1; b = 2; c = abs (a-b);
```

Die Expansion des Makros liefert:

```
a = 1; b = 2; c = a-b >= 0 ? a-b : -a-b;
```

Wir erkennen, daß nun *c* den Wert -3 enthält anstelle von +1. Um diesen oder ähnliche Fehler zu vermeiden, kommen wir demnach nicht umhin, jeden Parameter für sich einzuklammern. Wir erhalten:

```
# define abs(x)   (x) >= 0  ?  (x) : -(x)
```

Doch auch dies genügt noch nicht. Hätten wir in unserem Beispiel noch die Zuweisung

```
d = abs (b) - 1;
```

so liefert die Makroexpansion

```
d = (b) >= 0  ?  (b) : -(b) - 1;
```

Und damit wird an *d* der Wert 2 statt der beabsichtigten 1 zugewiesen. Wir müssen, um diesen Fehler zu vermeiden, demnach auch den gesamten Ausdruck klammern. Wir sind somit schon fast wieder bei unserer ursprünglichen Definition angekommen. Wir merken:

Achtung:

➔ In Makrodefinitionen sind alle Parameter und der gesamte Ausdruck zu klammern.

Doch auch unsere erste nicht weiter zu verbessernde Makrodefinition besitzt noch eine gefährliche Falle. Ist etwa ein Feld *A* mit *n* Elementen gegeben und wollen wir die Anzahl der Elemente dieses Feldes bestimmen, die betragsmäßig größer als eine Variable *b* sind, so funktioniert folgender Programmcode leider nicht zufriedenstellend:

```
i = groesser = 0;
while (i<n)
   if ( abs(a[i++]) > b ) groesser ++ ;
```

Der Fehler liegt darin, daß im Makro der übergebene Parameter zweimal ausgewertet wird, einmal bei der Abfrage und ein zweites Mal bei der Zuweisung. Betrachten wir nur die Expansion der letzten Zeile:

```
if ( ( (a[i++]) >= 0)  ?  (a[i++]) : -(a[i++]) ) > b )    groesser ++ ;
```

Nun erkennen wir, daß *i* nach dieser Zeile um 2 und nicht nur um 1 erhöht wurde. Übrigens ist der Ausdruck nicht undefiniert, da der '?:'-Operator garantiert, daß zunächst der Abfrageausdruck vollständig bewertet wird, bevor die Zuweisung ausgeführt wird.

Dieses Dilemma ist auch der Grund, warum das ehemalige Makro *toupper* heute als Funktion realisiert wird. Denn die erste der folgenden beiden Definitionen liefert nur für Kleinbuchstaben korrekte Ergebnisse, die zweite hingegen liefert falsche Ergebnisse, wenn der Inkrement- oder Dekrementoperator

bei der Parameterübergabe verwendet wird:

```
# define  toupper(c)  ( (c) + 'A' - 'a' )
# define  toupper(c)  ( ( (c) >= 'a' && (c) <= 'z' )  ?  (c) + 'A' - 'a' : (c) )
```

Als C++ Programmierer vergessen wir daher sofort wieder die Verwendung von Makros. Die einzige sinnvolle Anwendung von *#define* bleibt deren Verwendung in Headerfiles wie in 9.2 beschrieben.

9.4 Aufzählungen

In C können mittels *enum* auch Aufzählungen definiert werden. Die Syntax ist ähnlich zu der von *struct* und *union*. Beispiele:

```
enum boolean { FALSE, TRUE } ;
enum tage { MO, DI, MI, DO, FR, SA, SO } wochentag ;
```

Mit diesen beiden Definitionen existieren die Datentypen *enum boolean* und *enum tage* und eine Variable *wochentag* vom Typ *enum tage*. Außerdem sind damit die Konstanten FALSE=0, TRUE=1, MO=0, DI=1, MI=2, DO=3, FR=4, SA=5 und SO=6 definiert. Wir erkennen, daß die Werte der Konstanten bei Null beginnen, und dann um 1 hochgezählt werden. Ist diese automatische Wertzuweisung nicht gewünscht, so können auch direkte Zuweisungen erfolgen. Beispielsweise liefert

```
enum tage { MO=1, DI, MI, DO, FR, SA=10, SO } wochentag ;
```

die Konstantenwerte MO=1, DI=2, MI=3, DO=4, FR=5, SA=10 und SO=11. Wir haben auf einfache Art und Weise Konstanten definiert. Steht in der Konstantenaufzählung keine Zuweisung, so wird immer der Vorgängerwert plus eins genommen. Der Vorteil dieser Konstanten ist, daß sie den Gültigkeitsregeln von C unterliegen. Sind diese Konstanten beispielsweise lokal definiert, so sind sie auch nur hier sichtbar.

Die Nachteile dieser Aufzählungstypen sind, daß die Konstanten nur Ganzzahlen annehmen dürfen (natürlich einschließlich einzelner Zeichen) und daß es keine Bereichsüberprüfungen für Variablen dieser Aufzählungstypen gibt. Die obige Variable *wochentag* darf beispielsweise jeden int-Wert annehmen! Dies ist der Hauptgrund, warum Aufzählungstypen meist nur zur schnellen Definition mehrerer Konstanten verwendet wird.

9.5 Zeiger auf Funktionen

Wir kennen bereits die Möglichkeit, eine Funktion aufzurufen, der ein komplexer Ausdruck als Parameter übergeben wird. Dieser Ausdruck darf dabei ohne weiteres eine Funktion enthalten, etwa

```
x = sin ( 0.5 * sqrt (0.5) ) ;
```

Hier wird der Compiler den Ausdruck bewerten und dann an die Sinusfunktion übergeben. Es liegt demnach keine Funktionsübergabe an die Sinusfunktion vor. Anders sieht es hingegen bei der numerischen Berechnung eines bestimmten Integrals aus, etwa

$$I = \int_a^b f(x)\, dx$$

Hier soll zu einer beliebigen Funktion f das Integral von a nach b berechnet werden. Aus diesem Grund benötigen wir einen Aufruf der Form

```
i = integral (a, b, f);
```

Dies ist ein korrekter C-Code, falls a und b definierte Variablen und f eine definierte Funktion repräsentieren. Analog zu Feldern übergibt C automatisch nicht die Funktion selbst, sondern einen Zeiger auf diese Funktion. Die Deklaration der Funktion *integral* erfordert zur strengen Typüberprüfung auch alle

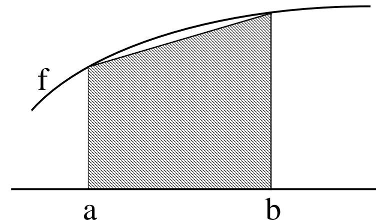
Parameter der Funktion *f*. Die Deklaration könnte daher folgendes Aussehen besitzen:

```
float integral ( int a, int b, float (*f) (float x) );
```

Zu beachten ist das Einklammern von **f*. Ohne Klammerung wäre *f* eine Funktion, die einen Zeiger auf *float* zurückliefert. Mit den Klammern ist *f* ein Zeiger auf eine Funktion, die einen *float* zurückliefert. Analog zu Call-by-reference-Parametern müssen wir in der Funktion *integral* immer **f* schreiben, wenn wir die übergebene Funktion verwenden wollen.

Wir verwenden zur Berechnung des Integrals die Trapezformel. Sie lautet

$$i = (b-a)\,\frac{f(a)+f(b)}{2}$$



Diese Trapezformel wird mit der Zahl der Stützstellen immer genauer. Folgendes C-Beispielprogramm ist dem Buch *Press u.a.: Numerical Recipes in C, Cambridge University Press* entnommen. Die Trapezfunktion ist etwas eigentümlich gewählt. Sie wird zunächst mit 2 Stützstellen, dann mit 4, dann mit 8 usw. Stützstellen aufgerufen. Das dazugehörige Testprogramm vollzieht dies in einer Schleife nach. Die Ergebnisse kommen dem tatsächlichen Wert des Integrals ziemlich nahe.

```
#define FUNC(x) ((*func)(x))
float trapzd( float (*func)(float), float a, float b, int n)
{  float x,tnm,sum,del;
   static float s;
   static int it;
   int j;
   if (n == 1) {
      it=1;
      return (s=0.5*(b-a)*(FUNC(a)+FUNC(b)));
   } else {
      tnm=it;
      del=(b-a)/tnm;
      x=a+0.5*del;
      for (sum=0.0,j=1;j<=it;j++,x+=del) sum += FUNC(x);
      it *= 2;
      s=0.5*(s+(b-a)*sum/tnm);
      return s;
   }
}
```

Modul: 9-sonst/trapzd.c

Das Programm benutzt eine Konstantendefinition zur Definition der Funktion FUNC, um sich die etwas lästige Zeigerschreibweise der übergebenen Funktion zu sparen.

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

float trapzd ( float (*func)(float), float, float, int );

#define NMAX 12
#define PIO2 1.5707963

/* Test function: */
float func ( float x )
{  return (x*x) * (x*x-2.0) * sin(x);
}

/* Integral of test function: */
float fint ( float x)
{  return 4.0*x*(x*x-7.0)*sin(x)
                    -(pow(x,4.0)-14.0*(x*x)+28.0) * cos(x);
}
```

```
main()
{  int i;
   float a=0.0, b=PIO2, s;
   printf ("\nIntegral of func with 2^(n-1) points\n");
   printf ("actual value of integral is %20.6f\n",
                          fint(b) - fint(a) );
   printf ("%6s %24s\n", "n", "approx. integral" );
   for (i=1 ;i<=NMAX; i++)
   {  s = trapzd (func, a, b, i);
      printf ("%6d %20.6f\n", i, s);
   }
   return 0;
}
```

Modul: 9-sonst/trapztst.c

Im Testprogramm *trapztst.c* wird eine Funktion *func* definiert und auch die dazugehörige Stammfunktion angegeben. Es werden dann das exakte Integral und anschließend in einer Schleife Näherungswerte ausgegeben, wobei die Anzahl der Stützstellen je Durchlauf verdoppelt wird. Die Ausgabe des Programms sieht in etwa wie folgt aus:

```
Integral of func with 2^(n-1) points
actual value of integral is            -0.479159
     n         approx. integral
     1             0.905772
     2            -0.020945
     3            -0.361461
     4            -0.449584
     5            -0.471756
     6            -0.477308
     7            -0.478696
     8            -0.479044
     9            -0.479130
    10            -0.479149
    11            -0.479153
    12            -0.479154
```

Ausgabe des Programms: 9-sonst/trapzd.out

10 Anwendungen und Vertiefung des Stoffs

10.1 Einige nützliche Funktionen in C

Wir wollen hier nur einige weitere Funktionen ansprechen, die das eine oder andere Mal recht nützlich sein können. Dies sind: *setjmp*, *longjmp*, *fflush*, *fseek, seekg*.

setjmp und longjmp (deklariert in setjmp.h)

In Programmen mit sehr vielen verschachtelten Funktionsaufrufen (häufig in Verbindung mit rekursiven Aufrufen) führt ein Fehler zu einem kaskadierenden Beenden aller aktiven Funktionen. Wenn die dazu benötigte Laufzeit auch nicht immer ins Gewicht fällt, so heißt dies für den Programmierer, daß er in jeder Funktion einen Fehler aus einem Unterprogrammaufruf an das übergeordnete Programm weiterreichen muß. Dies bläht den Programmcode unnötig auf und trägt zur Unübersichtlichkeit bei.

Obwohl es eigentlich der strukturierten Programmierung widerspricht, ist es in solchen Fällen doch empfehlenswert, aus einem tief geschachtelten Unterprogramm direkt zu einem Wiederaufsetzpunkt zu springen. Dies wird in C mit den beiden Funktionen *setjmp* und *longjmp* unterstützt. Dabei setzt ein Aufruf von *setjmp* einen Wiederaufsetzpunkt. Jeder Aufruf von *longjmp* wird jetzt zum Setjmp-Aufruf zurückspringen (natürlich nur, wenn die Funktion, in der der Setjmp-Aufruf abgesetzt wurde, noch aktiv

ist). Das Funktionsergebnis von *setjmp* ist Null, wenn ein Wiederaufsetzpunkt eingerichtet wird, er ist ungleich Null, wenn *setjmp* über einen Longjmp-Aufruf angesprungen wurde. Wir benötigen:

```
#include <setjmp.h>
jmp_buf  restartpoint;          /* Puffer für das Wiederaufsetzen */
setjmp (restartpoint);          /* Aufruf von setjmp mit Puffervariable */
longjmp (restartpoint, code);   /* Rücksprung mit Puffervariable und int-Rückgabewert */
```

Der in *longjmp* übergebene Wert von *code* (ungleich Null) wird von *setjmp* zurückgeliefert. Ein Beispielprogramm von Prof. Dr. Jobst zur Berechnung des Wertes von arithmetischen Ausdrücken findet sich unter *10-anwend/ausdruck.c*.

fflush, fseek (deklariert in stdio.h) bzw. flush, seekg (deklariert in iostream.h)

Diese Funktionen dienen zur internen Beeinflussung von Ein- und Ausgaben. Ein- und Ausgaben erfolgen in der Praxis aus Laufzeitgründen gepuffert. Dies bedeutet bei Programmfehlern, daß bestimmte Ausgaben noch nicht in die Datei oder auf Bildschirm geschrieben sind, obwohl sie das Programm bereits ausführte. Wir können diese Ausgabe aber immer mittels der Funktion *fflush (Dateizeiger)* bzw. *Dateistrom.flush()* erzwingen. In diesem Fall wird der Dateipuffer sofort auf Platte bzw. Bildschirm geleert.

Wem es nicht genügt, sequentiell in eine Datei zu schreiben oder sequentiell zu lesen, der sollte die Funktion *fseek* (in C) und *seekg* (in C++) genauer betrachten. Diese Funktionen ermöglichen, insbesondere bei Binärdateien, das beliebige Positionieren innerhalb der Datei. Wir benötigen:

```
#include <stdio.h>
int ursprung = SEEK_SET;          /* Anfang der Datei; oder SEEK_END: Ende der Datei; */
                                  /* oder SEEK_POS: augenblickliche Position */
long offset = 0;                  /* beliebiger Wert, in Textdateien muß offset Null sein */
FILE * datei; …                   /* und dann Öffnen der Datei !! */
fseek (datei, offset, ursprung);  /* springt an den Ursprung + Offset (in Bytes) der Datei */
```

Und in C++ sieht dies wie folgt aus:

```
#include <iostream.h>
int ursprung = ios::beg;          // Anfang der Datei; oder ios::end: Ende der Datei;
                                  // oder ios::cur: augenblickliche Position
long offset = 0;                  // beliebige Ganzzahl, auch in Textdateien
seekg (offset);                   // Absolute Adressierung mittels der long-Variable offset
seekg (offset, ursprung);         // Relative Adressierung mit ursprung + Offset (in Bytes)
```

Der Aufruf erfolgt wie gewohnt mit Voranstellen des Dateistroms, also etwa:

```
dat.seekg (100, ios::cur);        // Gehe 100 Byte ab Cursorposition weiter
```

10.2 Curses

Curses ist ein Programmpaket, das auf fast allen Plattformen zur Verfügung steht. Curses ist nicht in ANSI-C enthalten. Es sei hier nur deshalb erwähnt, weil es ungepufferte Eingabe, das Unterdrücken eines Echos (Eingabe von Paßwörtern) und eine Semi-Fenstertechnik unterstützt. Wer sich dafür interessiert, der sei auf Literatur (Manuale der C-Compiler-Hersteller, Public-Domain-SW) oder das Online-Manual auf der SUN hingewiesen. Hier nur ein kleines einführendes Beispiel ohne weiteren Kommentar:

```
 #include "curses.h"
WINDOW * win;              /* Struktur des Curses-Fensters */
int main (void)
{   int c;
    initscr ();                        /* oeffnen von Curses */
    cbreak ();                        /* ungepuffertes Lesen */
    noecho ();                      /* keine Bildschirmechos */
    box (stdscr, '*', '*');     /* Fensterrand mit Sternen */
    mvprintw (1, 30, "Beispielprogramm");
                    /* schreibt in Zeile 1, Spalte 30 Text */
    win = subwin (stdscr, 7, 42, 5, 20 );     /* Subwindow */
    box ( win, 0, 0 );    /* Umrandung mit Standardzeichen */
    mvwprintw (win, 1, 10, "%-10s", "normal" );    /* Text */
    wattrset  (win, A_REVERSE );                 /* Invers */
    mvwprintw (win, 3, 10, "%-10s", "invers");     /* Text */
    wattroff  (win, A_REVERSE );   /* Attribut ruecksetzen */
    mvwprintw (win, 5, 5, " Pfeil oben oder Taste e" );
    refresh ();          /* Aenderungen auf Ausgabe zeigen */
    keypad (stdscr, TRUE);             /* symbolische Namen */
    while ( (c=getch()) != KEY_UP && c != 'e') ;
    endwin ();                               /* end curses */
    return 0;
}
```

Programm: 10-anwend/cursbsp.c

10.3 Hash-Verfahren

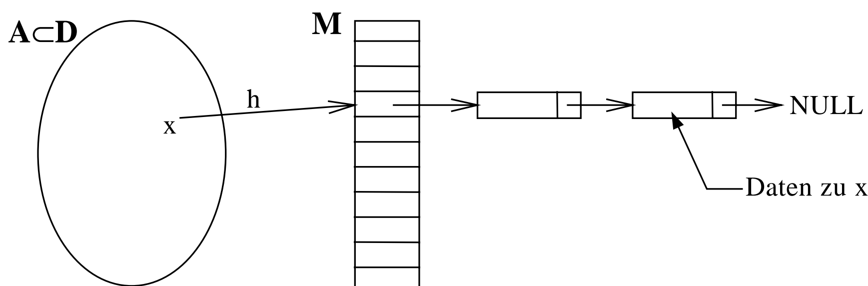
Im Pascal-Skript wird die Idee des Hashing erklärt. Weiter findet sich ein erstes Beispiel zu Hashfunktionen. Wiederholen wir kurz das Hashverfahren. Dieses wird meist dann angewendet, wenn ein sehr großer Definitionsbereich D (z.B. die Menge aller Namen) auf eine überschaubare Menge M (z.B. Speicherbereich eines Feldes) abgebildet werden soll. Eine Funktion von D in M heißt Hashfunktion h.

Da die Mächtigkeit der Menge D in der Regel wesentlich größer als die Mächtigkeit der Bildmenge M ist, kann die Abbildung *h* nicht eineindeutig sein, d.h. daß es gibt Elemente der Menge D, die das gleiche Bildelement besitzen. Auch wenn wir die Funktion *h* auf einen sehr kleinen Definitionsbereich A (z.B. die Menge aller Studenten der FH) einschränken, können wir die Eineindeutigkeit nicht erzwingen.

Ein Hashverfahren besteht daher aus zwei wesentlichen Punkten: aus

a) der Wahl einer geeigneten Hashfunktion h: $A \subset D \rightarrow M$
b) einer Ausnahmebehandlung, wenn *h* zwei Elemente aus A auf das gleiche Bildelement abbildet.

Im Pascal-Skript wurde das Problem b) dadurch gelöst, daß Einträge, die auf das gleiche Bildelement abbilden, in einer Linearen Liste verkettet wurden. Dieses Verfahren wird gerne dann gewählt, wenn nur wenige Änderungen im Laufe der Zeit zu erwarten sind. In diesem Fall fällt dann das aufwendige Anfordern von dynamischen Speicher nicht ins Gewicht. Erinnern wir uns:



Im Feld M sind die Anfangszeiger von Listen gespeichert. Die Hashfunktion *h* liefert die Position im Feld M, an der eine Liste beginnt, in der die Daten zu einem Element abgespeichert sind. Sei beispielsweise *x* ein Name eines Studenten, so liefert *h(x)* die Position des Listenzeigers im Feld M. Demnach ist

M[h(x)] der Anfangszeiger auf die Liste, in der die Daten zum Studenten *x* gespeichert sind.

Wir wollen hier eine zweite Möglichkeit kennenlernen, das Speichern aller Einträge in einem fest vorgegebenen Feld. Dies ist der Hauptanwendungsfall und setzt voraus, daß die Mächtigkeit von M größer ist als die Mächtigkeit von A (sonst hätten ja nicht alle Einträge Platz).

Betrachten wir die Technik dieses Hash-Verfahrens an einem Beispiel. Gegeben seien Personen, die durch die Sozialversicherungsnummer eindeutig identifiziert sind. Zwecks einfacherer Handhabung entfernen wir aus dieser Nummer den einzelnen Buchstaben (nehmen wir an, daß dadurch die Eindeutigkeit nicht verletzt wird). Weiter sei ein Feld aus 1000 Elementen gegeben, in der die Daten zu den einzelnen Personen eingetragen werden. Als Hashfunktion *h* wählen wir:

```
h(x) = x % 1000
```

Dies ist die naheliegendste Funktion, die garantiert jeder Sozialversicherungsnummer *x* einen Platz im Feld zuweist. Die Modulo-Funktion spielt die Rolle des Zerhackens einer großen Zahl auf kleine Bereiche, der Name 'Hash' hat daher auch seinen Namen. Die Sozialversicherungsnummer 58090454091 würde durch *h* demnach auf die Zahl 91 abgebildet. Wir definieren jetzt die Hashtabelle wie folgt:

```
const int DIM = 1000;
const int N = 500;
struct hashtab
      {  long key;
         char info [N];          /* enthält die Daten zu den einzelnen Personen */
      };
hashtab  hashfeld [ DIM ] ;    /* Hashtabelle */
```

Die Daten der Person mit obiger Nummer würden demnach in *hashfeld[91].info* zu finden sein. Oder ausführlicher: ist *x* eine Sozialversicherungsnummer, so stehen die Daten zu dieser Person in

```
hashfeld [ h(x) ] . info
```

Probleme gibt es leider dann, wenn $h(x) = h(y)$ für unterschiedliche Nummern *x* und *y* gilt. Betrachten wir als Beispiel folgende Nummern, die sich um die Zahl 261 häufen:

37081236261   11121124261   40190446262   81020171261

Hier treten an Position 261 Kollisionen auf. Bis auf den ersten Eintrag muß allen anderen 261er-Einträgen ein anderer Speicherplatz zugewiesen werden. Als einfachste Lösung bietet sich an, den anderen Einträgen einfach den nächsten freien Platz anzubieten. Wie unser obiges Beispiel zeigt, wird dadurch unser Kollisionsproblem leider nicht entschärft. Würden obige Einträge nacheinander erfolgen, so würde sich um den Speicherplatz 261 folgende Situation einstellen:

Index	key	info
260	0	
261	37081236261	Meyer …
262	11121124261	Huber …
263	40190446262	Müller …
264	81020171261	Schmidt …
265	0	

In der Regel liefert die Modulo-Funktion eine gute Hashfunktion. Aber gerade bei Namen kommt es immer wieder vor, daß sich Häufungen an einzelnen Positionen nicht vermeiden lassen. Die Wahl des Ersatzspeicherplatzes im Falle von Kollisionen geschieht mit Hilfe einer weiteren Funktion h_2, die auch als Hashfunktion 2. Ordnung bezeichnet wird. In unserem Fall wurde die Funktion

$$h_2 (x) = x + 1$$

gewählt. Wie hoffentlich erkannt wird, gelang es dieser Funktion nicht, die Kollision zu entschärfen. Auch eine Funktion mit einem größeren Versatz löst das Problem nicht, im Gegenteil: die Funktion

$$h_2 (x) = (x + 500) \% 1000$$

würde zu jedem Eintrag nur maximal 2 Ersatzplätze erlauben. Wir würden für Frau Schmidt keinen Speicherplatz mehr finden (der obige Modulooperator ist erforderlich, um den Bereich 0…999 nicht zu verlassen).

Unser Ziel ist es, eine Versatzfunktion h_2 zu finden, die allen gleichen Nummern verschiedene Ersatzplätze zuweist. Es hat sich daher bewährt, nicht nur den Index (hier 261), sondern den gesamten Schlüssel (Versicherungsnummer) in der Versatzfunktion zu verwenden. Gute Ergebnisse liefert beispielsweise

$$h_2(x) = (x / 1000) \% 1000$$

Dies garantiert, daß bei Kollisionen jeder Nummer ein anderer Ersatzspeicher zugeordnet wird. Dies entzerrt eventuelle Häufungen enorm. Weiter empfiehlt es sich, als Feldgröße keine geraden Zahlen zu verwenden, sondern stattdessen Primzahlen. Dies verhindert, daß eventuell nur wenige Ersatzspeicherplätze existieren.

Bisher haben wir offen gelassen, wie erkannt wird, ob ein Speicherplatz leer ist oder nicht. Hier empfiehlt sich eine Vorbelegung des gesamten Feldes mit der Zahl Null. Wir haben im wesentlichen zwei Funktionen zu programmieren, die Funktion *eintragen* und die Funktion *suche*. Erstere trägt ein neues Element ein, die zweite sucht die Daten zu einem gespeicherten Eintrag. Der Algorithmus lautet grob:

Eintragen: ermittle die Speicherstelle mittels der Hashfunktion h. Ist die Stelle noch frei, so trage ein. Sonst ermittle solange eine Ersatzspeicherstelle mittels der Hashfunktion h_2, bis ein freier Platz gefunden wurde. Trage dann das Element ein.

Suchen: ermittle die Speicherstelle mittels der Hashfunktion h. Ist die Speicherstelle leer, so existiert der gesuchte Eintrag nicht. Ansonsten vergleiche den Eintrag (key) mit der gesuchten Nummer. Stimmen sie überein, so lies die Daten aus. Sonst suche solange mittels der Hashfunktion h_2 einen Ersatzspeicher, bis diese Stelle leer ist oder der gesuchte Eintrag gefunden wurde. Ist die Speicherstelle leer, so existiert der gesuchte Eintrag nicht. Sonst lies die gefundenen Daten aus.

Wir wollen diese theoretischen Überlegungen nun an einem Beispiel nachvollziehen. Wir speichern die Telefonnummern aller Dozenten des Sammelgebäudes ab. Wird die korrekte Telefonnummer eingegeben, so werden die dazugehörigen Dozenten ausgegeben.

```
const int DIM = 53;           // sollte eine Primzahl sein!
const int N   = 100;

struct hashtab
{   long key;
    char info [N];
} ;

int ablegen ( hashtab [], const long, const char * );
               /* speichert ein Element in Hashtabelle */

int suchen ( const hashtab [], const long );
               /* sucht ein Element in Hashtabelle */

void ausgeben ( const hashtab [] );
               /* gibt die komplette Hashtabelle aus */
```

Programm: 10-anwend/hash.cpp (Deklarationen)

```
int suchen (hashtab tab[], const long key )
{  int index, i;
   long schluessel, versatz;

   // Primaersuche mit Hash-Funktion:
   index = (int) (key % DIM);
   schluessel = tab [index] . key ;

   // Sekundaersuche:
   for (i=1;i<DIM && schluessel!=0 && schluessel!=key; i++ )
   {  versatz = key / DIM;
      index = (int) ((index + versatz) % DIM);
      schluessel = tab [index] . key ;
   }

   // Position zurueckgeben, wenn nicht gefunden, dann -1:
   return schluessel != key ? -1 : index;
}
```

Programm: 10-anwend/hash.cpp (Funktion *suche*)

Zu beachten ist, daß die Konstante DIM eine Primzahl sein sollte. Nur so ist gewährleistet, daß beim Eintragen immer alle Elemente der Hashtabelle durchsucht werden, und damit der volle Speicher zur Verfügung steht. Wir sehen ferner, daß uns die Indizierung von Feldern in C, die immer bei Null beginnt, in diesem Anwendungsfall ausnahmsweise sehr entgegenkommt.

10.4 Fallstricke

Nur kurz sollen hier einige häufige Fehler erwähnt werden. Wir kennen bereits:

- In der Funktion *scanf* müssen als Parameter Zeiger angegeben werden (nicht bei C++ Strömen).
- Im If-Ausdruck ist '==' und nicht '=' zu verwenden.
- Nach dem Ende des If-, For-, While- und Switch-Ausdrucks gehört kein Semikolon.
- Beim Kopieren von Zeichenketten muß die Funktion *strcpy* verwendet werden.
- Am Ende von Struct-, Union- und Enum-Deklarationen, ebenso am Ende von Initialisierungen von Feldern und Strukturen muß ein Semikolon stehen.

Überhaupt sind viele Fehler Leichtsinnsfehler, meist mit unverhofftem Ausgang, wie etwa in

```
if (n < 3)
    return
n = 5;
```

Das fehlende Semikolon nach dem Return verändert den gewollten Programmfluß. Oder ein Beispiel aus *Koenig: Der C-Experte*:

```
struct logrec
{   int date, time, code; }
main ( )
{   …  }
```

Wegen des fehlenden Semikolons ist *main* eine Funktion mit dem angegebenen Struct als Rückgabewert. Zitat: Die Auswirkung … bleibt krankhaften Geistern als Übung überlassen.

Hätten wir 'int main ( )' geschrieben, so wären wir durch einen Syntaxfehler gewarnt worden!

Doch auch semantische Fallstricke lauern überall. Um einen int-Overflow etwa bei Addition zweier großer Zahlen zu vermeiden, könnte man versuchen, diesen wie folgt abzufangen, wobei *fehler* eine gegebene Funktion und *a* und *b* zwei int-Zahlen seien:

```
if ( a+b < 0 )
    fehler ( );
```

Doch wir können nicht davon ausgehen, daß jeder Compiler bei int-Overflow das Vorzeichenbit setzt. Nach ANSI-C ist der Ausdruck bei Overflow undefiniert. Wir können jedoch ausnützen, daß es bei Unsigned-Variablen keinen Overflow gibt. Das Ergebnis ist hier fest definiert als modulo 2^n mit n als die Bitanzahl der int-Variable. Führende Bits werden also einfach abgeschnitten. Wir können das obige Ergebnis also korrekt wie folgt schreiben:

```
if ( (unsigned) a + (unsigned) b > INT_MAX )
    fehler ( );
```

Dabei ist INT_MAX in *limits.h* vordefiniert. Noch einfacher ist

```
if ( a > INT_MAX - b )
    fehler ( );
```

10.5 Eine komplexe Funktionsdefinition

Die hardwarenahe Programmierung führt manchmal zu eigenartigen Blüten. Häufig ist das Bootprogramm eines Rechners an der Speicherstelle Null abgelegt. Wenn ein Reboot erfolgen soll, so muß dieses

Programm aufgerufen werden. Doch wie schafft man es, ein Programm aufzurufen, das an der Speicheradresse Null abgelegt ist und weder Parameter noch Rückgabewerte besitzt?

```
Lösung:        ( * ( void ( * ) ( ) ) 0 ) ( ) ;
```

Diese Hieroglyphen wollen wir verstehen lernen. Dazu wiederholen wir, was ein Funktionszeiger überhaupt ist. Beginnen wir dazu mit Zeigern auf Variable:

```
float * f;
```

Hier ist *f* ein Zeiger. Ist *x* eine float-Variable, so wird nach der Zuweisung

```
f = &x;
```

der Zeiger *f* auf eine float-Variable zeigen. Ganz analog verhält es sich mit Funktionszeigern. Zunächst unterscheidet sich eine Funktion von einer Variablen, daß dem Namen ein Klammerpaar folgt. Wir definieren deshalb einen Funktionszeiger *g*, der auf eine Funktion zeigt, die eine float-Variable zurückliefert und keine Parameter besitzt zu:

```
float (*g) ( );       /* genauer: float (*g) (void)  */
```

Die Klammern sind wichtig, da Funktionsklammern stärker als der Operator '*' binden. Wir können obige Zeile auch lesen, daß **g* eine Funktion und demnach *g* selbst ein Zeiger auf diese Funktion ist.

Wollen wir mehrere Funktionszeiger definieren, so sollte ein entsprechender Datentypen eingeführt werden, etwa mittels

```
typedef float (*fptr) ( );
```

Der Datentyp *fptr* entspricht dabei dem Typ

```
float (*) ( )
```

Beachten Sie, daß wieder die Klammern um den Stern notwendig sind, da 'float * ( )' der Datentyp einer Funktion mit Rückgabewert 'float *' ist. Jetzt erkennen wir erste Ähnlichkeiten mit unserem obigen Funktionsaufruf. Er besitzt die Form:

```
( * ( Datentyp ) 0 ) ( ) ;
```

Der Datentyp ist dabei

```
void ( * ) ( )
```

also ein Funktionszeiger, der auf eine parameterlose Funktion zeigt, die keinen Rückgabewert besitzt.

Ein Funktionszeiger ist tatsächlich nichts anderes als ein anderer Zeiger auch. Er zeigt auf eine Adresse, wo der Funktionscode einer Funktion steht. Wäre jetzt *fnull* ein Funktionszeiger, der auf die Adresse Null zeigt, so würde der Aufruf

```
( * fnull ) ( ) ;
```

das Gewünschte leisten. Leider ist die Zahl 0 kein solcher Funktionszeiger. Der Ausdruck '*0' würde daher einen Syntaxfehler verursachen, da der Operator * einen Zeiger erwartet. Doch mit Hilfe des Cast-Operators können wir die Zahl 0 in einen geeigneten Datentyp verwandeln. Schreiben wir also

```
typedef void (* fnulltyp) ( ) ;
```

so liefert

```
( * ( fnulltyp ) 0 ) ( ) ;
```

den gewünschten Aufruf. Ersetzen wir fnulltyp noch durch den äquivalenten Typ 'void (*) ( )', so erhalten wir unseren ursprünglichen Aufruf. Wir erkennen wieder, daß ein Zeiger immer zwei Informationen speichert, zum einen eine Adresse und zum anderen den Datentyp, auf den die Adresse zeigt.

Wir merken uns:

- ein Funktionszeiger zeigt auf den Beginn des Funktionscodes,
- ein Variablenzeiger zeigt auf den Anfang eines Variablenbereichs.

Ansonsten besteht kein Unterschied zwischen beiden Datentypen. Rein theoretisch könnten wir auch die Zeigerarithmetik auf Funktionszeiger anwenden. Doch überlassen wir dies (noch) den C-Profis.

11 Objektorientierte Programmierung und Klassen

Wir haben bereits seit Beginn unserer C-Programmierung auch C++ Code verwendet. Laufend haben wir auf die C++ Ströme zugegriffen, aber auch der Referenzoperator und hat uns das eine oder andere Mal das Programmieren erleichtert. Wir wollen auch die Operatoren zur dynamischen Speicherreservierung nicht vergessen. Doch jetzt wollen wir endlich in die eigentliche Welt von C++ einsteigen, in die objektorientierte Programmierweise.

11.1 Objektorientierte Programmierung

Ganz knapp gesagt, ist die objektorientierte Programmierung nichts anderes als die konsequente Umsetzung der strukturierten Programmierung. In der strukturierten Programmierung teilen wir ein Programm aufgabenbezogen in Unterprogramme ein. Diese Unterprogramme sind in C die Funktionen. Diese Funktionen arbeiten mit eigenen lokalen Variablen. Die Schnittstellen zwischen diesen einzelnen Funktionen sind die Funktionsaufrufe und die dazugehörigen Parameter.

In der objektorientierten Programmierung geschieht genau das Gleiche! Nur in der Umsetzung dieser Theorie sind wir (hoffentlich) konsequenter. Es gibt nämlich häufig Probleme, wenn lokale Variablen von mehr als einer Funktion verwendet werden. Die Weitergabe dieser Variablen über Parameter führt zu einer nur schwer handhabbaren Anzahl von Parametern. Eventuell helfen Strukturen als Parameter noch etwas weiter. Meist greift man dann jedoch auf globale Variablen zu. Der Nachteil dieser Methode ist jedoch, daß auch andere Programmteile diese Variablen sehen und verwenden können.

Dies ist bei größeren Programmen tatsächlich ein wesentlicher Nachteil. Denn jedes Programm muß regelmäßig angepaßt oder erweitert werden. Je mehr Verzahnungen es innerhalb eines solchen Programms gibt, um so komplizierter wird dieses Vorhaben. Jede Änderung könnte Seiteneffekte provozieren. Im schlimmsten Fall ist das Programm wegzuwerfen und neu zu schreiben.

Dieses Abkapseln von Daten kann bereits in C-Programmen hinreichend erreicht werden. Dazu werden alle Teilprogramme in eigenen Modulen geschrieben, alle nicht notwendigerweise nach außen sichtbaren globalen Variablen und Funktionen werden mit dem Schlüsselwort static versehen. Damit hätten wir die Idee der objektorientierten Programmierung bereits zum großen Teil nachvollzogen.

Hauptidee der objektorientierten Programmierung:

> Jedes (umfangreiche) Programm läßt sich in logisch zusammengehörige Teile zerlegen. Ein solches Teil besteht aus Variablen, die manipuliert werden und aus Funktionen, die diese Manipulationen durchführen. Diese Teile bilden eine logische Einheit, nur die außerhalb notwendigen Variablen und Funktionen werden explizit sichtbar gemacht.

Ein solches zusammengehöriges Teil heißt in der objektorientierten Programmierung ein **Objekt**. Als Erweiterung zum Datentyp struct enthält ein solches Objekt nicht nur Eigenschaften (Variablen), sondern auch Methoden (Funktionen), die angeben, wie diese Eigenschaften verändert werden sollen.

1. Beispiel: Verwaltung eines Stacks

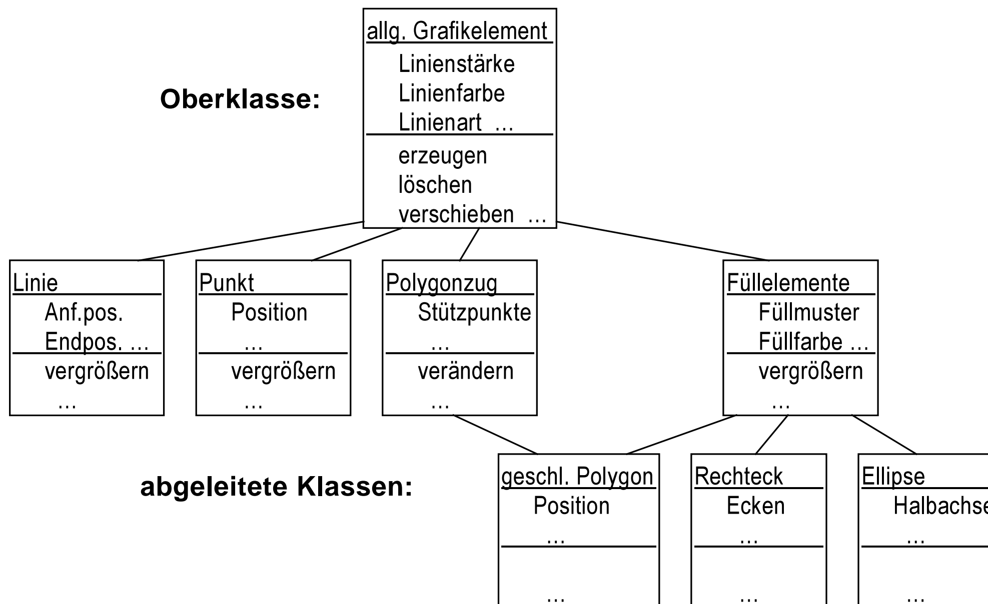
   Eigenschaften: Größe des Stacks, gespeicherte Daten je Stackelement
   Methoden: push, pop, init

2. Beispiel: Darstellung eines Kreises in Grafiken

   Eigenschaften: Position des Mittelpunkts, Radius, Kreisrandstärke, Kreisrandfarbe, Füllfarbe, Füllmuster, …
   Methoden: erzeugen, löschen, vergrößern, verkleinern, verschieben, …

Das letztgenannte Beispiel zeigt weitere Möglichkeiten der Objektorientierung auf. Denn auch die anderen Grafikobjekte besitzen ähnliche Eigenschaften und Methoden wie Kreise. Wir können daher ge-

meinsame Eigenschaften und Methoden in einer Oberklasse zusammenfassen und daraus dann nur noch die besonderen Eigenschaften und Methoden der einzelnen (Unter-)klassen ableiten. Betrachten wir dazu das folgende (unvollständige) Bild zu Grafikelementen:



Die Oberklasse „allgemeines Grafikelement“ stellt Basisvariablen und -funktionen zur Verfügung. Die abgeleiteten Unterklassen definieren weitere spezifische Eigenschaften der angegebenen Objekte. Die Eigenschaften der Oberklassen werden dabei an die Unterklassen vererbt. In unserem Beispiel erbte die Klasse „geschlossenes Polygon“ Eigenschaften von sogar zwei Oberklassen.

Bei großen Projekten ist es alles andere als trivial, ein Programm optimal in Objekte und diese wiederum in Hierarchien einzuordnen. Bei einer verkehrten Wahl der Klassenhierarchien treten später auch in der objektorientierten Programmierung Probleme der effizienten Erweiterung dieser Programme auf, da sich die neuen Aufgaben dann kaum in die bestehende Hierarchiestruktur einbinden lassen.

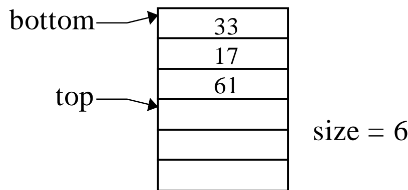
Bei der objektorientierten Programmierung gilt daher noch mehr als bisher:

erst denken, dann programmieren!

Abschließend können wir sagen, daß die objektorientierte Programmierung zusammengehörige Teile eines Programms noch besser miteinander verbindet. Da nur über vordefinierte Schnittstellen zwischen den einzelnen Teilen kommuniziert werden kann, werden nicht zusammengehörige Daten optimal voneinander abgeschotet.

11.2 Objektorientierte Implementierung eines Stacks

Zur Nachbildung eines Stacks benötigen wir einen Stackbereich (ein Feld oder eine Liste), Zeiger auf diesen Bereich und die Funktionen push und pop. Um einen möglichst flexiblen Stack zu erhalten, soll die Größe des Stacks als Parameter angegeben werden können. Wir wählen daher ein Feld, das dynamisch mit der gewünschten Stackgröße angelegt wird. Wir benötigen insgesamt drei Variablen: *bottom* ist der Zeiger auf den Anfang des Stacks, *top* zeigt auf den ersten freien Platz und *size* speichert die Größe des Feldes.



Die beiden Funktionen *push* und *pop* müssen auf alle drei Variablen und den Stackbereich zugreifen können. Um dies zu ermöglichen, werden diese Variablen in der konventionellen Programmierung daher meist als globale Variablen deklariert. Damit kann der Programmierer aber, unter Umgehung der Funktionen *push* und *pop*, auf diese Variablen direkt zugreifen. Vielleicht fragt man sich, warum man dies nicht zulassen sollte. Wollen wir etwa überprüfen, ob der Stack bereits voll ist, so wäre dies ganz einfach mittels der folgenden Abfrage möglich:

```
if (top - bottom == size)                    // top zeigt auf das erste freie Element
   cout << "Stack ist voll\n" ;
```

Dieses Beispiel zeigt jedoch, daß bei dieser Programmierweise viele Internas zur Stackimplementierung verwendet werden, etwa muß *top* auf das erste freie Feld zeigen und der Stack muß als Feld realisiert sein. All dieses Wissen muß unser außenstehender Programmierer mitbringen. Doch erst richtig verheerend (im Sinne der Wiederverwendbarkeit von Software) wirkt sich aus, daß bei diesem Programmierstil die interne Stackimplementierung nicht verändert werden darf. Ansonsten müßten auch alle obigen und ähnlichen Stellen im gesamten Programm angepaßt werden.

Mit der objektorientierten Programmierung können wir jetzt einen Datentyp *stack* erzeugen, der alle benötigten Variablen und Funktionen in sich vereinigt, und der alle Internas versteckt. Somit kann der Programmierer seine interne Implementierung jederzeit umstellen. Er muß nur dafür sorgen, daß die „öffentlichen" Teile seines Stacks unverändert bleiben. Betrachten wir eine mögliche Implementierung:

```
class stack
{  private:       // standardmaessig, daher nicht notwendig
      int * top;            // Spitze des Stacks
      int * bottom;         // Anfang des Stacks
      int size;             // max. Groesse des Stacks

   public:
      stack (int s)                     // Konstruktor
      {  top = bottom = new int [size = s]; }

      ~stack ()                         // Destruktor
      {  delete bottom; }

      void push (int i)
      {  if ((top - bottom) < size )
            *top++ = i;
      }

      int pop (void)
      {  if (top > bottom)   return *(--top);
         else                return 0;
      }
};                     // Ende der Klassendefinition
```

Programm 11-class/stack.cpp

Wir haben damit einen Datentyp, die Klasse *stack*, erzeugt. Eine Variable *stapel* dieses Datentyps erhalten wir etwa mit folgender Anweisung:

```
stack stapel (n);    // erzeugt einen Stack namens stapel, bestehend aus n Elementen
```

Zugegriffen wird auf diesen Stapel mittels der in der Klasse implementierten Funktionen push und pop. Soll etwa die Zahl 17 gespeichert und anschließend in die Variable i wieder ausgelesen werden, so geschieht dies wie folgt:

```
stapel.push (17);      // fügt die Zahl 17 dem Stapel hinzu
i = stapel.pop ();     // entfernt das oberste Element des Stapels, und weist es i zu
```

Wir wollen jetzt dieses Beispiel verstehen lernen. Zunächst ist die Klassendefinition nichts weiter als eine echte Erweiterung der Strukturen. Neu hinzugekommen ist, daß in einer Klasse auch Funktionen aufgenommen werden können, so daß jetzt Funktionen (Methoden) und die dazugehörigen Variablen (Eigenschaften) in einer einzigen Struktur gemeinsam verwaltet werden. Klassenfunktionen werden ana-

log wie Struct-Variablen aufgerufen, also mittels des Namens des Klassenelements, gefolgt vom Punktoperator und dem Funktionsnamen.

Weiter können die Direktiven *private* und *public* verwendet werden. Sie können an beliebiger Stelle innerhalb der Klassendefinition stehen und geben an, daß die folgenden Variablen und Funktionen nur innerhalb der Klasse sichtbar sind (bei der Direktive *private*), oder daß diese auch von außerhalb zugreifbar sind (bei der Direktive *public*). Wird am Anfang keine Direktive gesetzt, so wird standardmäßig *private* eingestellt. Unser Beispiel zeigt, daß die Variablen nicht von außen zugreifbar sind. Der Stack kann daher nur über die beiden Funktionen *push* und *pop* beeinflußt werden, die beide *public* sind. Folgender Code, außerhalb der Klasse geschrieben, wäre demnach ein Syntaxfehler:

```
if ( stapel.top == stapel.bottom )        // Fehler: private Elemente einer Klasse
    cout << "Stack ist leer\n" ;
```

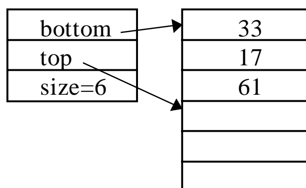
Wird eine solche Abfrage benötigt, so empfiehlt sich stattdessen die zusätzliche Implementierung einer Public-Funktion ‘isempty’ innerhalb der Klasse.

Ohne bisher erwähnt worden zu sein, enthält unsere Klasse neben *push* und *pop* noch zwei weitere Funktionen, die als **Konstruktor** und **Destruktor** bezeichnet wurden. Beim Erzeugen einer Klassenvariablen wird automatisch ein Konstruktor gestartet, und analog wird beim Entfernen der Destruktor aufgerufen. Dies erlaubt ein elegantes Initiieren und ein sauberes Aufräumen beim Beenden.

Ein Konstruktur wird bei statischen Variablen zu Programmbeginn, bei automatischen Variablen beim Betreten des Programmblocks und bei dynamischen Variablen beim Anlegen des dynamischen Speicherbereichs aufgerufen. Entsprechendes gilt für den Destruktor. Ein Konstruktor besitzt den gleichen Namen wie die Klasse, beim Destruktor wird davor noch das Zeichen ‘~’ geschrieben.

Der im Beispiel vorliegende Konstruktor ersetzt demnach eine Funktion init. Statt eine Klassenvariable, häufig auch kurz als **Objekt** bezeichnet, zu deklarieren und anschließend mittels eines Init-Funktionsaufrufs dieses Objekt vorzubelegen, genügt wegen des impliziten Aufrufs allein die Definition des Konstruktors. In unserem Beispiel wird im Konstruktor dynamischer Speicher für das Stackfeld angefordert. Dies bereitet beim Entfernen eines Objektes Probleme. Zwar wird standardmäßig der Speicherplatz, den das Objekt selbst belegte, freigegeben, nicht jedoch zusätzlicher, dynamisch angeforderter Speicher. Dies ist nur mit Hilfe des Destruktors möglich.

Wir verstehen nun unser Beispiel vollständig. Es ist ein Konstruktor mit einem Parameter definiert. Demnach erfolgt das Deklarieren eines Klassenobjekts *stapel* zusammen mit der Angabe eines entsprechenden Parameterwertes. Es wird dann Speicher für das Objekt angelegt, und der Konstruktor wird gestartet, der dynamischen Speicherplatz für den eigentlichen Stack anfordert. Mit *push* und *pop* kann jetzt auf diesen Stack zugegriffen werden. Beim Entfernen des Objekts muß auch der dynamische Speicher wieder freigegeben werden. Dies geschieht implizit mit Hilfe eines geeigneten Destruktors.



Betrachten wir einige weitere Besonderheiten: Jede Klasse besitzt mindestens einen Konstruktor und genau einen Destruktur: ist ein Konstruktor oder Destruktor nicht explizit definiert, so wird je ein Standardkonstruktor und -destruktor erzeugt, der die Elemente des Klassenobjekts anlegt bzw. wieder entfernt. Der Standardkonstruktur legt Speicher für alle in der Klasse deklarierten Variablen an und ist parameterlos. Der Standarddestruktor gibt genau diesen Speicher wieder frei. Sollen mehrere Konstruktoren definiert werden, so müssen sie sich in den Parametern unterscheiden (in der Anzahl oder zumindest in einem Datentyp). Je nachdem, mit welchen Parameterwerten ein Klassenobjekt deklariert wird, wird der entsprechende Konstruktor aufgerufen. Da der Destruktor üblicherweise nicht explizit aufgerufen wird, ist er parameterlos, so daß nur genau ein Destruktor definiert werden kann.

Beispiel: In obiger Stackklasse soll bei einer parameterlosen Deklaration ein Stack mit genau 10 Elementen erzeugt werden. Dies geschieht durch Hinzufügen eines weiteren Konstrukturs in der Klasse:

```
stack (void)      { top = bottom = new int [ size = 10 ]; }
```

Wir hätten aber auch unseren bestehenden Konstruktor einfach durch Vorbelegung erweitern können:

```
stack (int s = 10)    { top = bottom = new int [ size = s ]; }
```

Mittels der folgenden Deklarationen werden je nach Bedarf entsprechend große Objekte erzeugt:

```
stack stapel_A;        // Stack mit 10 Elementen
stack stapel_B (15);   // Stack mit 15 Elementen
stack stapel_C = 20;   // bei einparametrigen Konstruktoren alternativ erlaubt: 20 Elemente
```

In Klassen gibt es keine Einschränkung bezüglich der erlaubten Variablen und Funktionen. Insbesondere dürfen in Klassen auch andere Klassenobjekte als Elemente aufgenommen werden. Beim Erzeugen der Klassen werden auch alle enthaltenen Klassenobjekte erstellt.

Weiter dürfen auch Variablen mit dem Schlüsselwort **static** versehen werden. Diese Variablen existieren dann je Klasse nur einmal, gleichgültig wie viele Klassenobjekte deklariert werden. Eine Manipulation einer solchen Variablen beeinflußt alle Objekte dieser Klasse. Intern wird daher eine static-Variable extra gespeichert. Analog geschieht dies auch für die Funktionen einer Klasse. Auch hier genügt es, diese Funktionen für alle Objekte nur einmal anzulegen. Zur Unterscheidung von „normalen" Variablen werden static-Variablen und Funktionsdefinitionen nicht durch einen Punkt, sondern durch zwei Doppelpunkte ('::') vom Klassennamen getrennt. Wir kennen diese Schreibweise bereits von den Flags der Ein-/Ausgabeströme. Wir sehen jetzt, daß es sich bei diesen Flags um statische Konstante der Klasse ios handelt.

Eine häufige Anwendung dieser zwei Doppelpunkte finden wir bei Funktionsdefinitionen. Denn insbesondere bei großen Klassen mit vielen Funktionen wird die Definition der Klasse unübersichtlich. Um dem entgegenzuwirken, müssen nur die Funktionsköpfe in der Klasse angegeben werden. Die Funktionsdefinitionen selbst dürfen außerhalb der Klasse stehen. Um zu erkennen, zu welcher Klasse diese Funktionen gehören, muß die Klasse, gefolgt von den beiden Doppelpunkten, dem Funktionsnamen vorangestellt werden. Betrachten wir unser entsprechend abgeändertes Stackbeispiel.

```
class stack
{      int * top;            // Spitze des Stacks
       int * bottom;         // Anfang des Stacks
       int size;             // max. Groesse des Stacks
   public:
       stack (int);                     // Konstruktor
       ~stack ();                       // Destruktor
       void push (int);
       int pop (void);
};                     // Ende der Klassendefinition

stack::stack (int s)       // Konstruktor, kein Rueckgabewert
{  top = bottom = new int [size = s]; }

stack::~stack ()           // Destruktor, kein Rueckgabewert
{  delete bottom; }

void stack::push (int i)
{  if ((top - bottom) < size )  *top++ = i; }

int stack::pop (void)
{  return (top > bottom) ? *(--top) : 0; }
```

Programm: 11-class/stack2.cpp

Hier haben wir alle vier Funktionen (Konstruktor, Destruktor, push und pop) außerhalb der Klasse definiert. Beachten Sie nochmals, daß vor den beiden Doppelpunkten immer ein Klassentyp und nicht eine Klassenvariable stehen muß. Außerdem sei darauf hingewiesen, daß sowohl beim Konstruktor als auch beim Destruktor kein Rückgabewert angegeben wird.

Ein kleiner Unterschied zwischen der Definition von Funktionen innerhalb und außerhalb einer Klasse sei nicht verschwiegen: Funktionen, die innerhalb einer Klasse definiert werden, werden automatisch als **inline**-Funktionen implementiert. Ist dies auch bei Funktionen erwünscht, die außerhalb der Klasse definiert werden, so muß der Bezeichner *inline* explizit angegeben werden.

Klassen verhalten sich bei ihrer Verwendung wie Strukturen, etwa werden bei einer Zuweisung immer

standardmäßig alle Elemente kopiert. Genaugenommen wurden in C++ die Strukturen erheblich erweitert, so daß zwischen Strukturen und Klassen nur ein einziger Unterschied besteht:

- In Klassen sind alle Variablen und Funktionen standardmäßig als *private* deklariert
- In Strukturen sind alle Variablen und Funktionen standardmäßig als *public* deklariert

Wir haben gesehen, daß in einer Klasse sowohl Private- als auch Public-Variablen und -Funktionen gespeichert werden können. Die privaten Daten sind außerhalb der Klasse nicht zugreifbar. In unserem Stackbeispiel konnten wir dadurch die interne Implementierung des Stacks verstecken. Zugriffe sind nur über die Funktionen *push* und *pop* möglich.

Die Möglichkeiten der Verwendung von Klassen sind aber noch weit umfangreicher als bisher gezeigt. Beispielsweise können auch Klassenobjekte dynamisch mittels des Operators *new* erzeugt werden. Natürlich können wir auch hier die Größe des Stackfeldes mit übergeben:

```
stack * p_stapel = new stack (20);    // erzeugt ein dynamisch angelegtes Objekt
p_stapel -> push (17);                // speichert die Zahl 17 in diesem Stack
```

11.3 Friend- und Memberfunktionen

Alle in einer Klasse deklarierten Funktionen gehören dieser Klasse fest an. Sie heißen daher auch **Member** der Klasse. Im Stackbeispiel waren dies die Funktionen stack, ~stack, push und pop. Memberfunktionen haben gemeinsam:

- Die Funktionsköpfe sind innerhalb der Klasse angegeben
- Die Funktionsdefinitionen stehen entweder innerhalb der Klasse, oder sie werden mit dem Namen ‘Klasse::Funktion’ außerhalb definiert
- Memberfunktionen werden mittels ‘Klassenobjekt.Funktion’ aufgerufen
- Memberfunktionen dürfen auf alle privaten Funktionen und Variablen der Klasse zugreifen

Es gibt leider einige Anwendungen, wo diese Memberfunktionen nicht ausreichen. Betrachten wir nur das Beispiel, daß eine Funktion auf private Daten zweier Klassen zugreifen soll. Dies ist mit Memberfunktionen grundsätzlich nicht möglich, schließlich kann eine solche Funktion nur Member einer Klasse sein.

Als Lösung bietet sich an, daß Funktionen, die nicht zur Klasse gehören, auf private Daten der Klasse zugriffen dürfen, falls es ihnen in der Klasse explizit gestattet wird. Das Zauberwort hierfür heißt **friend**!

Zu beachten ist, daß natürlich die Konstruktoren und der Destruktor immer Member ihrer Klasse sind. Alle anderen Funktionen können je nach Wunsch und Geschmack als Member oder Friend definiert werden. Betrachten wir nochmals die Klasse stack, wobei wir diesmal die Funktionen *push* und *pop* als Friend-Funktionen definieren wollen:

```
class stack
{     int * top;            // Spitze des Stacks
      int * bottom;         // Anfang des Stacks
      int size;             // max. Groesse des Stacks
   public:
      stack (int);                        // Konstruktor
      ~stack ();                          // Destruktor
      friend void push (stack &, int);    // Friendfunktion
      friend int pop (stack &);           // Friendfunktion
};                    // Ende der Klassendefinition

stack::stack (int s)                     // Konstruktor
{  top = bottom = new int [size = s]; }

stack::~stack ()                         // Destruktor
{  delete bottom; }
```

```
void push (stack &s, int i)             // Call by reference
{  if ((s.top - s.bottom) < s.size )  *s.top++ = i; }

int pop (stack &s)                      // Call by reference
{  return (s.top > s.bottom) ? *(--s.top) : 0; }
```

Programm: 11-class/stckfrnd.cpp

Beachten Sie, daß *push* und *pop* jetzt eigenständige Funktionen sind. Im Aufruf der Funktionen merkt man sofort, ob es sich um Member- oder Friend-Funktionen handelt:

```
stapel.push (27);            // Memberfunktion
push (stapel, 27);           // Friendfunktion
```

Ist push eine Memberfunktion, so steht das Klassenobjekt, getrennt durch einen Punkt, vor dem Funktionsnamen. Bei einer Friendfunktion ist dies nicht der Fall, so daß in einem eigenen Parameter der Name des Objekts angegeben werden muß, auf das die Funktion wirkt.

➔ **Achtung**:

Eine Friendfunktion besitzt immer einen Parameter mehr als eine gleichwertige Memberfunktion. Dieser zusätzliche Parameter gibt an, auf welches Objekt der Klasse die Funktion wirken soll.

11.4 Das Schlüsselwort „this“

Es gibt Anwendungsfälle, wo eine Memberfunktion auf das eigene Klassenobjekt als Ganzes zugreifen möchte. Zumindest möchte man die Adresse des eigenen Objekts kennen. Aus diesem Grund gibt es in C++ das reservierte Wort **this**. *this* ist als Zeiger auf das eigene Objekt realisiert. Betrachten wir als Beispiel Lineare Listen, die mit Hilfe von Klassen realisiert sind:

```
class liste
{    liste * nachfolger;
     liste * vorgaenger;
     int zahl;
   public:
     liste (int);                  // Konstruktor
     void einketten (liste* &);    // neues Element einketten
     liste * ausgeben (void);      // ausgeben des Inhalts
};

liste::liste (int i = 0)
{  nachfolger = vorgaenger = NULL;
   zahl = i;
}

void liste::einketten (liste* &root) //hinter root einketten
{  nachfolger = root;
   if (root != NULL)
   {  vorgaenger = root->vorgaenger;
      root->vorgaenger = this;   // zeigt auf dieses Element
   }
   root = this;              // root zeigt auf dieses Element
}
```

Programm: 11-class/this.cpp

Wir erkennen eine doppelt verkettete Liste, wo Elemente mittels der Memberfunktion liste::einketten in die Liste aufgenommen werden. Diese Funktion hängt ein Element hinter dem Zeiger *root* ein. Dazu muß der Vorgänger des nächsten Elements und die Variable *root* die Adresse des einzuhängenden Elements speichern. Wir müssen deshalb auf das Wort *this* zurückgreifen.

Wir merken uns, daß der Zeiger *this* für jede Klasse als Private-Element definiert ist. Eine Member- oder Friendfunktion darf daher diesen Zeiger verwenden. Natürlich könnten wir auch ‘this->nachfolger’ statt nur ‘nachfolger’ benutzen, doch letzteres ist doch einfacher.

12 Überladen von Operatoren und Funktionen

Das Überladen von Funktionen wird von vielen Compilern implizit verwendet, denken wir nur an Pascal, wo die Prozedur *writeln* zwei Formate besaß:

```
writeln ( x );       ( Schreiben auf Bildschirm )
writeln ( f, x );    ( Schreiben in die Datei f )
```

Pascal erkennt anhand des ersten Parameter (Dateityp oder nicht), welcher Funktionsaufruf gemeint ist. Auch C++ nutzt diese Möglichkeit der Mehrfachverwendung von Funktionen. In C++ wird diese Möglichkeit sogar an den Benutzer weitergegeben. Wir haben sie auch bereits verwendet, denken wir nur an die Konstruktoren einer Klasse. C++ geht sogar noch einen Schritt weiter und erlaubt auch die Mehrfachverwendung (fast) aller Operatoren. Auch hier haben wir bereits die Operatoren '<<' und '>>' verwendet. Normalerweise sind diese Operatoren Bitmanipulatoren (Shift nach links bzw. nach rechts). Ist der erste Operand jedoch ein Ausgabe- bzw. Eingabestrom, so wird er für Aus- bzw. Eingaben verwendet.

Dieses Mehrfachverwendung einer Funktion oder eines Operators für verschiedene Aufgaben heißt **Überladen**. Beginnen wir mit dem Überladen von Funktionen.

12.1 Überladen von Funktionen

Damit C++ eine Funktion gleichen Namens mehrfach verwenden kann, muß C++ die richtige Funktion beim Aufruf erkennen können. Bei gleichem Namen müssen daher die Anzahl der Parameter und/oder die Datentypen der Parameter unterschiedlich sein. In C++ beschränkt sich diese Aussage nicht nur auf Konstruktoren, sondern gilt für alle Funktionen.

Es ist in C++ daher erlaubt, jederzeit mehrere Funktionen gleichen Namens zu definieren, vorausgesetzt sie unterscheiden sich mindestens in einem Parameter, d.h. mindestens ein Parameter besitzt einen unterschiedlichen Datentyp.

Betrachten wir als Beispiel die Berechnung des Absolutbetrags. Es ist schwerfällig, Funktionen mit dem Namen *iabs*, *fabs* oder *dabs* zu definieren, je nachdem ob int-, float- oder double-Zahlen als Parameter verwendet werden. Andererseits wäre eine Funktion allein zu wenig, etwa *dabs*, da zwar eine int-Zahl als Parameter implizit in eine double-Zahl verwandelt würde, der Rückgabewert wäre aber dann vom Typ double. In C++ schreiben wir nun entsprechende Funktionen gleichen Namens, der Compiler ruft immer die richtige Funktion auf, abhängig vom übergebenen Datentyp.

```
inline int abs (int x)
{   return x>=0 ? x : -x;
}
inline float abs (float x)
{   return x>=0 ? x : -x;
}
int main()
{   int a;
    float z;
    cout << "Bitte eine Ganzzahl einlesen: "; cin >> a;
    cout << "Jetzt eine Gleitpunktzahl: ";     cin >> z;
    cout << "Der Betrag von " << a << " ist " << abs(a);
    cout << "Der Betrag von " << z << " ist " << abs(z);
    return 0;
}
```

Programm: 12-overld/abs.cpp

Nachdem wir bereits das Vorbelegen von Parametern kennengelernt haben, haben wir jetzt eine noch mächtigere Möglichkeit zur Hand, Funktionsnamen mehrfach zu verwenden. Es wird aber dringend ange-

raten, davon nur dann Gebrauch zu machen, wenn diese Funktionen gleichen Namens auch eine gemeinsame Bedeutung besitzen.

12.2 Überladen von Operatoren

Wie bereits erwähnt, dürfen in C++ auch Operatoren überladen werden. Der Sinn dieser Möglichkeit liegt eindeutig in der besseren Lesbarkeit von Programmen. Sind etwa A, B und C Matrizen, so ist sicherlich die folgende Schreibweise der konventionellen bisherigen vorzuziehen:

```
C = A + B;        // statt:  C = addmatrix (A, B);
cout << C;        // statt:  schreibmatrix (C);
```

In diesem Fall wurden der Additions- und Ausgabeoperator überladen, damit sie auch mit Matrizen verträglich sind. Beim Überladen von Operatoren sind gewisse Regeln einzuhalten. Diese sind:

– Bis auf 5 Operatoren (‘.’, ‘.*’, ‘::’, ‘?:’ und sizeof) dürfen alle Operatoren überladen werden. Häufig werden die Plus- und Minus- und die Ein- und Ausgabeoperatoren überladen, doch auch der Inkrementoperator (Postfix und Prefix) oder der Operator *new* dürfen jederzeit überladen werden.
– Die Bedeutung der Operatoren für Standarddatentypen kann grundsätzlich nicht geändert werden. Auch die Hierarchie der Operatoren kann nicht beeinflußt werden.
– Überladene Operatoren müssen entweder Mitglied einer Klasse sein, oder sie müssen als Friend einer Klasse definiert sein, wobei dann mindestens ein Operand eine Klasse sein muß.

Operatoren werden ähnlich wie Funktionen definiert. Der „Funktionsname“ des Operators *op* ist dabei „operator op“. Als Parameter werden dann die Operanden angegeben. Ein binärer Operator *op* würde dann wie folgt deklariert:

```
Ergebnistyp operator op (Datentyp Operand1, Datentyp Operand2);
```

Betrachten wir ein Beispiel. Wir wollen die Zeichenkettenverarbeitung verbessern. Wir definieren dazu eine Klasse string und überladen den Operator ‘+’ für die Konkatenierung:

```
class string
{    char  s[255];
     short laenge;
 public:
     string (void) { s[0]='\0'; laenge=0; } //Konstruktor

     string (const char * str)    // Zeichenketten-Konstruktor
     {  strcpy (s,str);
        laenge = strlen(str);
     }

     friend string operator+ (const string& str1,
                              const string& str2)
     {  string str = str1;        // Initiierung erlaubt!
        strcat (str.s, str2.s);
        str.laenge = str1.laenge + str2.laenge;
        return str;
     }

     friend ostream& operator << (ostream& s,const string& x)
     {  return s << x.s;
     }
};
```

Programm: 12-overld/string.cpp

Die Klasse string benutzt nur statischen Speicher, um die Zeichenketten zu speichern. Konstruktoren ermöglichen zum einen, diese Strings mit Zeichenketten zu initiieren. Zum anderen wird der entsprechende Konstruktor implizit aufgerufen, wenn beim ‘+’-Operator ein Operand vom Typ string, der andere aber vom Typ char* ist. Der char*-Operand wird also dank unseres Konstruktors automatisch in den Typ string umwandelt. Die Parameter des Plusoperators wurden als Konstante gewählt, um bei dieser Umwandlung das nutzlose zusätzliche Erzeugen von Zwischenobjekten zu unterbinden. Es sind jetzt

folgende Ausdrücke syntaktisch korrekt:

```
string s1 = "Abend", s2 = "essen";     // initiiert mit Konstruktor
s1 = s1 + s2;                          // verwendet friend-Operator
s1 = s1 + "gehen";          // wandelt zunaechst 2. Operand mittels Konstruktor in string um
s1 = "Abend" + "essen";                // Fehler!!!!!! kein Operator für char*
```

Die vorletzte Zeile hätten wir auch schreiben können als

```
s1 = s1 + string ("gehen");
```

Hier wird der Cast-Operator explizit verwendet, was den Aufruf des entsprechenden Konstruktors zur Folge hat. Ist der Konstruktor für diesen Datentyp nicht definiert, so liegt ein Syntaxfehler vor.

Betrachten wir noch den weiteren Friend-Operator ‘<<’ zur Ausgabe eines Stringobjekts. Dieser Operator muß als Friend definiert werden, da er sowohl Ströme als auch Strings als Parameter besitzt. Der erste Operand ist immer ein Element der Klasse ostream (bei Eingabe: istream), der zweite der Parameter der betrachteten Klasse. In unserem Beispiel wird die Zeichenkette ausgegeben. Um die Ausgabe wie gewohnt auch hintereinander mehrfach verwenden zu können, müssen wir als Ergebniswert den Ausgabestrom zurückliefern. Es genügt jedoch nicht, einfach eine Kopie dieses Stroms zu verwenden, wir brauchen den Originalstrom. Da dieser Strom als Referenz übergeben wurde, können wir ihn einfach als Referenz weiterreichen, nachdem wir unsere eigene Ausgabe getätigt haben. Die Wirkungsweise wird aus dem folgenden Bild hoffentlich noch klarer:

```
cout   <<   str1   <<   str2   <<  …
ostream&
      ostream&
              ostream&
```

Völlig analog wird auch eine eventuelle Eingabe von Zeichenketten mittels des ‘>>’-Operators ermöglicht. Kommen wir jetzt zum Plusoperator zurück. Wir hätten diesen auch als Mitglied der Klasse definieren können:

```
string string::operator + (const string& str2)
{   string str = *this;            // Initiiere str mit dem Memberobjekt
    strcat (str.s, str2.s);
    str.laenge +=  str2.laenge;
    return str;
}
```

Wie wir bereits wissen, besitzt ein Mitglied immer einen Parameter weniger als ein Friendobjekt. Bei Operatoren ist dabei automatisch der erste Operand das Klassenobjekt selbst. Wir erkennen damit auch schon den Nachteil eines Member-Operators: der erste Operand muß ein Klassenobjekt sein. Eine Anweisung der Form

```
s1 = "Abend" + s2;
```

wäre also ein Syntaxfehler. Schreiben wir den Operator als Friend, so ist diese Anweisung korrekt.

12.3 Standardfunktionen und –operatoren einer Klasse

Wir haben im letzten Abschnitt stillschweigend vorausgesetzt, daß ein Klassenobjekt bei der Deklaration gleich initiiert werden kann, und zwar mit einem bereits existierenden Objekt, so daß alle Werte dieses Objekts übernommen werden. Ein entsprechender Konstruktor wurde aber in der Klasse nicht definiert. Dies wirft nun die Frage auf, welche Funktionen und Operatoren nun eigentlich implizit bei der Definition einer Klasse angelegt werden, und wann diese im Bedarfsfall überladen werden müssen.

Beantworten wir gleich den ersten Teil der Frage. Bei der Definition einer Klasse *a* werden automatisch folgende Funktionen und Operatoren erzeugt:

- Parameterloser Konstruktor a::a (void);
- Copy-Konstruktor a::a (const a &);
- Destruktor a::~a ( )
- Zuweisungsoperator ‚=‘ a a::operator = (const a &)

Den parameterlosen Konstruktor und den Destruktor haben wir bereits kennengelernt. Beide besitzen keinen Rückgabewert, der Destruktor auch keinen Parameter, auch nicht ‚void‘! Zusätzlich existiert ein Copy-Konstruktor und der Zuweisungsoperator. Der Zuweisungsoperator wird immer dann aufgerufen, wenn ein Klassenobjekt einem anderen bereits existierenden Objekt zugewiesen wird.

Der Copy-Konstruktur wird hingegen implizit aufgerufen, wenn ein neues Klassenobjekt erzeugt und dieses mit einem bereits bestehenden Klassenobjekt initiert wird. Betrachten wir dazu ein Beispiel zur Klasse *string*:

```
string str1 = "ein schöner Tag";    // Aufruf des selbstdefinierten Zeichenkettenkonstruktors
string str2 = str1;                 // Aufruf des Copy-Konstruktors
string str3;                        // Aufruf des parameterlosen Konstruktors
str3 = str1;                        // Aufruf des Zuweisungsoperators
```

Copy-Konstruktor und Zuweisungsoperator ähneln sich, besitzen aber einen wesentlichen Unterschied: Beim Zuweisungsoperator existiert das Element auf der linken Seite bereits, beim Copy-Konstruktor hingegen noch nicht! Diesen feinen Unterschied werden wir im nächsten Abschnitt genauer beleuchten.

Daß genau diese 3 Funktionen und der Zuweisungsoperator vorgegeben werden, hat einen Grund. Bei Strukturen existieren nämlich ebenfalls genau diese Eigenschaften: eine Struktur wird bei der Deklaration erzeugt (parameterloser Konstruktor), und sie kann gleich initiert werden (Copy-Konstruktor). Eine Struktur wird beim Verlassen des Gültigkeitsbereichs oder bei Programmende wieder entfernt (Destruktor), ebenso kann einer Struktur ein Wert zugewiesen werden (Zuweisungsoperator). Weitere Vorgaben gibt es bei Strukturen nicht, also auch nicht bei Klassen.

Übrigens haben diese Standardfunktionen und –operatoren exakt die gleiche Wirkung wie bei Strukturen. Der parameterlose Konstruktor erzeugt alle Variablen der Klasse ohne Vorbelegungen, der Copy-Konstruktor kopiert den Inhalt des Klassenobjekts eins zu eins in das neu erzeugte Objekt. Genauso wirkt der Zuweisungsoperator, und der Destruktor gibt den Speicherplatz der Variablen des zu löschenden Objekts wieder frei.

Genügen diese Vorgaben den Erfordernissen nicht, so müssen die entsprechenden Funktionen oder Operatoren überladen werden. Die Standardeigenschaften gehen dadurch verloren. Natürlich wird aber bei den Konstruktoren immer der Speicher für die Variablen automatisch weiterhin angelegt, und der Destruktor gibt genau diesen Speicher auch immer wieder frei. Zu beachten ist noch, daß der parameterlose Konstruktor nicht mehr zur Verfügung steht, sollte irgendein anderer Konstruktor definiert werden. Im Stackbeispiel wäre die Deklaration

```
stack neuer_stapel;
```

demnach fehlerhaft, außer es wird auch ein parameterloser Konstruktur definiert.

Im nächsten Abschnitt werden wir jetzt am Beispiel sehen, wann diese Funktionen und Operatoren überladen werden müssen.

12.4 Klassen und dynamisch reservierter Speicher

Wir ändern jetzt unser String-Beispiel geringfügig um. Wir reservieren nicht mehr festen Speicher für die Zeichenkette, sondern fordern den notwendigen Speicher hierfür dynamisch an. Damit sind Strings beliebiger Länge realisierbar. Die Standardfunktionen und –operatoren fordern dynamischen Speicher nicht automatisch an. Auch wird dieser dynamische Speicher nicht automatisch wieder freigegeben. Ebenso legt die Zuweisung und die Initierung nicht automatisch neuen dynamischen Speicher an. Wir müssen daher alle Standardvorgaben überladen.

Betrachten wir diese neue Klasse mit den entsprechenden Erweiterungen:

```
class string
{   char *s;                     // nur Zeiger!
    short laenge;
 public:
    string (void);               // parameterloser Konstruktor
    string (const char *);       // Zeichenketten-Konstruktor
    string (const string &);     // Copy-Konstruktor
    ~string ();                  // Destruktor
    string operator = (const string&);  // Zuweisungsop.
    friend string operator + (const string&, const string&);
                                 // Plusoperator
    friend ostream& operator << (ostream& s,const string& x)
};
```

Programm: 12-overld/string2.cpp (Konstruktoren und Plusoperator)

Der standardmäßig vorgegebene Zuweisungoperator arbeitet inkorrekt. Eine Zuweisung der Form

str1 = str2;

würde einfach den Zeiger str2.s auf str1.s kopieren. Beide Objekte würden daher den gleichen dynamischen Speicher verwenden. Der ursprüngliche dynamische Speicher von str1 wäre nicht mehr ansprechbar, und durch das Zugreifen von zwei Objekten auf den gleichen Speicherplatz treten nicht gewünschte Nebeneffekte auf. Analoges gilt für den Copy-Konstruktor.

Und nun die Implementierung der in der Klasse string deklarierten Funktionen und Operatoren:

```
string::string (void) {s=NULL; laenge=0;}  // Konstruktor
string::string (const char * str)   // Zeichenketten-Konstr.
{  laenge = strlen(str);
   s      = new char [laenge+1];
   strcpy (s,str);
}

string::string (const string& str)        // Copy-Konstruktor
{  laenge = str.laenge;
   s      = new char [laenge+1];
   strcpy (s,str.s);
}

string::~string ()  { delete s; }         // Destruktor

ostream& operator << (ostream& s,const string& x)
{  return s << x.s;
}
```

Programm: 12-overld/string2.cpp (Konstruktoren, Destruktor, Operator <<)

```
string operator + (const string& str1, const string& str2)
{  string str;
   str.laenge = str1.laenge + str2.laenge;
   str.s      = new char [str.laenge+1];
   strcpy (str.s, str1.s);
   strcat (str.s, str2.s);
   return str;
}

string string::operator = (const string& str)   // Zuweisung
{  if (&str != this)         // keine Kopie auf sich selbst
   {  delete s;              // bisherigen Speicher freigeben
      laenge = str.laenge;
      s      = new char [laenge+1];
      strcpy (s,str.s);
   }
   return *this;
}
```

Programm: 12-overld/string2.cpp (Operatoren = und +)

Der Destruktor ist relativ einfach, er gibt nur den dynamisch reservierten Speicher frei. Der Zuweisungsoperator wurde als Member geschrieben. Wie wir bereits wissen, besitzt ein Mitglied immer einen

Parameter weniger als ein Friendobjekt, und bei Operatoren ist der erste Operand das Klassenobjekt selbst. In unserem Beispiel ist der Parameter str der Operand auf der rechten Seite des Gleichheitszeichens, das Klassenobjekt ist die Variable links vom Gleichheitszeichen, und das Funktionsergebnis ist der Ergebniswert, der weiter verwendet werden kann, z.B. bei verketteten Zuweisungen (s1=s2=s3;).

Der Copy-Konstruktor gleicht dem Zuweisungsoperator. Der feine Unterschied liegt darin, daß beim Copy-Konstruktor das Klassenobjekt erst erzeugt wird, während es beim Zuweisungsoperator bereits existiert. Beim Zuweisungsoperator müssen wir zusätzlich durch eine Abfrage verhindern, daß nicht ein Wert auf sich selbst kopiert wird. In diesem Fall würde nämlich durch das Freigeben des Speichers dieser nicht mehr ansprechbar sein, der Aufruf von strcpy wäre folglich fehlerhaft! Mit Hilfe dieser zusätzlichen Operatoren und Funktionen arbeitet jetzt diese Stringklasse einwandfrei.

Betrachten wir im Überblick nochmals die standardmäßig vorhandenen Konstruktoren, Destruktor und Zuweisungsoperator, wann sie aufgerufen werden und welche Aktionen standardmäßig gestartet werden:

	Aufruf	Standardaktion	Überladen
Standardkonstruktor (parameterlos)	bei Deklaration eines Klassenobjekts ohne Parameter	reserviert Speicher für das Klassenobjekt	wenn zusätzliche Aktionen oder weitere Konstruktoren benötigt werden
Copy-Konstruktor	bei Deklaration eines Klassenobjekts mit Objektzuweisung	reserviert Speicher für das Klassenobjekt und initiiert das Objekt mittels Kopie	wenn zusätzliche Aktionen benötigt werden
Destruktor	beim Erlöschen der Gültigkeit des Objekts	gibt Speicher des Objekts frei	wenn zusätzliche Aktionen benötigt werden
Zuweisungsoperator	wenn Werte an Objekt zugewiesen werden	kopiert das Ergebnis eines Ausdrucks in Objekt	wenn zusätzliche Aktionen benötigt werden

Auch sei nochmals erwähnt, welch unterschiedliche Objektvariablen definiert werden können: statische, mit Lebensdauer von Programmstart bis -ende; automatische, mit Lebensdauer innerhalb eines Programmblocks; dynamische, mit Lebensdauer von New bis Delete. Jeweils zu Beginn des Lebenszyklus wird der passende Konstruktor und am Ende der Destruktor aufgerufen. In dieser Übersicht ist auch von Aktionen die Rede. Die mit Abstand wichtigste ist das Reservieren oder Freigeben von dynamisch reserviertem Speicher. Aber auch die Modifizierung von statischen Variablen kann eine Aktion sein. Wir können also auf jeden Fall sagen:

➔ **Achtung**:

Arbeitet eine Klasse intern mit dynamischer Speicheranforderung, so sind praktisch immer der Destruktor der Copy-Konstruktor und der Zuweisungsoperator zu überladen. Fast immer werden auch weitere Konstruktoren benötigt.

12.5 Klassen und statische Variable

Wie bereits erwähnt, können in Klassen auch statische Variablen verwendet werden. Hierzu ist in der Klassendefinition vor die entsprechenden Variablen nur der Bezeichner *static* anzugeben. Betrachten wir ein Beispiel, indem wir am Anfang der Klasse *string* die folgende Zeile einfügen:

```
static int zaehler;
```

Die Besonderheit ist, daß diese Variable nur einmal existiert, unabhängig davon wieviele Klassenobjekte erzeugt werden. Ändert demnach ein Objekt den Wert dieser Variablen, so sehen diese Änderung auch alle anderen Objekte. Diese Variable *zaehler* kann wie folgt verwendet werden:

```
Innerhalb von Memberfunktionen:   zaehler
Sonst:                            objekt.zaehler
                  oder            string::zaehler
```

wobei ‚objekt' ein definiertes Klassenobjekt der Klasse *string* sei. Der Schreibweise *string::zaehler*

ist dabei eindeutig der Vorzug zu geben. Es unterstreicht die Besonderheit dieser Variable. Eine Static-Variable kann sowohl im Public- als auch im Private-Teil definiert werden. Im letzteren Fall kann sie natürlich nur innerhalb von Friend- und Memberfunktionen aufgerufen werden.

Wir wollen nun eine interessante Anwendung von Static-Variablen kennenlernen, das Zählen aller gerade existierenden Klassenobjekte einer Klasse. Dazu wird beim Erzeugen eines neuen Objekts einfach die Static-Variable *zaehler* um 1 erhöht und beim Entfernen um 1 erniedrigt. Wir erkennen, daß wir dazu alle Konstruktoren und den Destruktor geeignet überladen müssen. Der Zuweisungsoperator hingegen ist nicht involviert. Betrachten wir die entsprechenden Funktionen:

```
class string
{   static int zaehler;
    char  s[255];
    short laenge;
 public:
    string (void) { s[0]='\0'; laenge=0; zaehler++; }

    ~string () { zaehler--; }   // Destruktor

    string (const char * str)   // Zeichenketten-Konstruktor
    {  strcpy (s,str);
       laenge = strlen(str);
       zaehler++;
    }

    string (const string& str)  // Copy-Konstruktor
    {  strcpy (s,str.s);
       laenge = str.laenge;
       zaehler++;
    }

    static int anzahl (void) { return zaehler; }
 ...

};

int string::zaehler = 0;       // Initiierung der stat. Var.
```

Programm: 12-overld/static.cpp

In diesem Programm ist die Variable *zaehler* ein privates Element der Klasse *string*. Aus diesem Grund kann darauf außerhalb der Klasse auch nicht zugegriffen werden. Wir betten daher diese Variable mittels der ebenfalls statischen Funktion *anzahl* ein. Statische Funktionen dürfen nur auf statische Variablen zugreifen. Sie gehören nicht zu einem bestimmten Klassenobjekt. Der Aufruf erfolgt etwa wie folgt:

cout << "Die Anzahl der Stringobjekte ist: " << string::anzahl ( ) << endl ;

Ein Problem ist noch die Initiierung der statischen Variable zaehler. Innerhalb der Klasse ist dies nicht möglich, da sonst die Variable bei jedem Erzeugen eines Objekts wieder zurückgesetzt würde. In C++ existiert daher die Möglichkeit, diese Variable außerhalb jeder Funktion (auch außerhalb von *main*) zu deklarieren und vorzubelegen. Dies ist im obigen Programm angegeben. Dieser einmalige externe Zugriff ist auch für private statische Elemente einer Klasse erlaubt.

12.6 Ein umfassendes Beispiel: Die Klasse der rationalen Zahlen

Wir wollen eine Klasse *ratio* einführen, die mit rationalen Zahlen rechnet. Im privaten Teil der Klasse sind der Zähler und der Nenner als int-Variablen gespeichert. Im Public-Teil stehen die wichtigsten benötigten Operatoren und Funktionen zur Verfügung:

+, -, *, /	arithmetische Operatoren
==, !=, < …	Vergleichsoperatoren
-	Vorzeichen
+=, -=, *=, /=	spezielle Zuweisungen
<<, >>	Ein- und Ausgabe

abs, kuerzen, Zaehler, Nenner Funktionen

Um diese Klasse der rationalen Zahlen möglichst einfach einsetzen zu können, definieren wir die Klasse in einem eigenen Headerfile namens *ratio.h*. Dieses Headerfile muß dann nur in diejenigen Programme eingebunden werden, die die rationalen Zahlen verwenden.

```
class ratio
{  int zaehler, nenner;
   ratio kuerzen (void);              // Kuerzt Brueche
 public:
   ratio (int =0, int =1);            // Konstruktor

   // Zaehler und Nenner ausgeben:
   int Zaehler (void) const;
   int Nenner  (void) const;

   // Arithmetische Operatoren:
   friend ratio operator + (ratio, ratio);
   friend ratio operator - (ratio, ratio);
   friend ratio operator - (ratio);           // Vorzeichen
   friend ratio operator * (ratio, ratio);
   friend ratio operator / (ratio, ratio);

   // Zuweisungen:
   ratio operator += (ratio);
   ratio operator -= (ratio);
   ratio operator *= (ratio);
   ratio operator /= (ratio);

   // Vergleiche:
   friend int operator == (ratio, ratio);
   friend int operator != (ratio, ratio);
   friend int operator <  (ratio, ratio);
   friend int operator <= (ratio, ratio);
   friend int operator >  (ratio, ratio);
   friend int operator >= (ratio, ratio);

   // Ein- und Ausgabe:
   friend ostream& operator << (ostream&, ratio);
   friend istream& operator >> (istream&, ratio& );

   friend ratio abs (ratio);
};
```

Programm: 12-overld/ratio.h (Definition der Klasse)

Die Funktionen *Zaehler* und *Nenner* wurden als konstante Memberfunktionen deklariert. Sie besitzen die Eigenschaft, daß sie keine Variablen der Klasse ändern dürfen. Eine Memberfunktion ist konstant, wenn sowohl bei der Deklaration als auch bei der Definition direkt hinter der Parameterleiste der Zusatz ‚const' angegeben wird. Versucht eine konstante Memberfunktion eine Klassenvariable zu ändern, so merkt dies der Compiler. Um unter allen Umständen Änderungen von Variablen zu unterbinden, ist in konstanten Memberfunktionen nur der Aufruf weiterer konstanter Memberfunktionen erlaubt.

In unserem Beispiel geben die Funktionen *Zaehler* und *Nenner* nur Werte aus. Es ist daher korrekt, diese als konstante Memberfunktionen zu deklarieren. Ganz allgemein ist es empfehlenswert, jede Memberfunktion, die keine privaten Werte eines Klassenobjekts ändert, als konstante Funktion zu deklarieren. Damit wird eine weitere mögliche Fehlerquelle ausgeschlossen.

Kommen wir jetzt zu den Definitionen der in der Klasse deklarierten Funktionen. Wir wollen hier nur einige herauspicken. Beginnen wir mit einigen technischen Funktionen. Zum Kürzen benötigen wir die Funktion kuerzen, diese wiederum benötigt das ggT. Zur Ermittlung des ggT zweier Zahlen verwenden wir einen bereits den Griechen bekannten rekursiven Algorithmus, das Sieb des Erastothenes:

```
int ggT (int a, int b)  // Sieb des Erastothenes
{  if (a==0 || b==0) return a+b;
   if (a==b)         return a;
   if (a<b)          return ggT (b-a,a);
     else            return ggT (a-b,b);
}
```

```
inline int abs (int a)
{  return (a>0) ? a : -a;  }

ratio ratio::kuerzen (void)
{  int teiler;
   if (nenner < 0)
   {  nenner  = -nenner;
      zaehler = -zaehler;
   }
   teiler = ggT (abs(zaehler), nenner);
   zaehler = zaehler / teiler;
   nenner  = nenner  / teiler;
   return *this;           // Rueckgabe der rationalen Zahl
}
```

Programm: 12-overld/ratio.cpp (technische Funktionen)

Kommen wir jetzt zum Konstruktor. Dessen Vorbelegung geschah bereits in der Klasse bei der Deklaration. Dort wurde die Zahl 0 vorbelegt (Zähler=0, Nenner=1).

```
ratio::ratio (int z, int n)         // Konstruktor
{  if (n==0)
   {  cout << "Nenner = 0! Wird durch 1 ersetzt.\n";
      n=1;
   }
   zaehler = z; nenner = n;
   kuerzen ();              // kuerzt den Bruch
}

inline int ratio::Zaehler (void) const
{  return zaehler;
}

inline int ratio::Nenner (void) const
{  return nenner;
}
```

Programm: 12-overld/ratio.cpp (Konstruktor, *Zaehler*, *Nenner*)

Betrachten wir jetzt die Implementierung einiger Standardoperatoren:

```
ratio operator - (ratio a)              // Vorzeichen!
{  return ratio (-a.zaehler, a.nenner);
}

int operator == (ratio a, ratio b)
{  return ((a.zaehler == b.zaehler) &&
                             (a.nenner == b.nenner));
}

int operator < (ratio a, ratio b)
{  // wegen Funktion kuerzen () ist Nenner immer positiv!
      return a.zaehler*b.nenner < a.nenner*b.zaehler;
}

ratio operator + (ratio a, ratio b)
{  ratio r (a.zaehler*b.nenner + a.nenner*b.zaehler,
            a.nenner*b.nenner);
   return r.kuerzen ();
}

ratio operator / (ratio a, ratio b)
{  if ( b.zaehler == 0)
   {  cerr << "Division durch Null\n";
      return 0;                  // Fehler
   } else {
      ratio r (a.zaehler * b.nenner, a.nenner * b.zaehler);
      return r.kuerzen ();
   }
}
```

```
inline ratio ratio::operator += (ratio a)
{  *this = *this + a;      // verwendet den Operator +
   return *this;
}
```

Programm: 12-overld/ratio.cpp (Vorzeichen, Vergleich, Addition, Division, Zuweisung)

Betrachten wir zuletzt noch die Funktion *abs* und insbesondere das Überladen der Eingabe. Die Eingabe wird in der Form ‘z / n’ erwartet, wobei z und n ganze Zahlen sind. Zwischen den beiden Zahlen steht das Zeichen ‘/’, zusätzliche White-Spaces spielen keine Rolle (falls das Flag skipws nicht explizit zurückgesetzt ist).

```
istream& operator >> (istream& i, ratio& a)
{  char ch;

   i >> a.zaehler >> ch >> a.nenner;
   if (ch != '/')
      cerr << "Syntaxfehler, '/' angenommen!\n";
   if (a.nenner == 0)
   {  cerr << "Nenner gleich null !!, durch 1 ersetzt.\n";
      a.nenner = 1;
   }
   a.kuerzen();
   return i;                     // Strom fortsetzen
}

inline ratio abs (ratio a)
{  return ratio( abs(a.zaehler), a.nenner );
}
```

Programm: 12-overld/ratio.cpp (Eingabe überladen, Funktion *abs*)

Dieser einmalige Aufwand ermöglicht jetzt das Verwenden rationaler Zahlen genauso wie das Verwenden von Standarddatentypen. Betrachten wir dies am Beispiel:

```
int main ()
{  ratio a, b;
   cout << "\nTest der rationalen Zahlen\n\n";
   cout << "Zwei rationale Zahlen eingeben:\n";
   cout << "Zahl 1: ";   cin >> a;
   cout << "Zahl 2: ";   cin >> b;
   cout << "Die Summe der Zahlen ist:       " << a+b << '\n';
   cout << "Die Differenz der Zahlen ist:   " << a-b << '\n';
   cout << "Das Produkt der Zahlen ist:     " << a*b << '\n';
   cout << "Die Division der Zahlen ist:    " << a/b << '\n';
   cout << "Die Negierung der 1. Zahl ist: " <<  -a << '\n';
   if (a < b)
      cout << "2. Zahl ist groesser als 1. Zahl\n" ;
   else
      cout << "1. Zahl ist groesser gleich als 2. Zahl\n";
   return 0;
}
```

Programm: 12-overld/ratio.cpp (Funktion *main*)

Wir erkennen die Möglichkeiten von C++, eigene Datentypen zu erzeugen, die wie die bereits vordefinierten verwendet werden können. Wir sehen aber auch, daß der (einmalige) Aufwand nicht ganz unerheblich ist.

Anmerkungen zu den rationalen Zahlen:

– Die Konstruktoren erlauben nicht nur ein Vordefinieren der rationalen Zahlen, sondern dienen auch als Cast-Operator beim impliziten oder expliziten Umwandeln von int-Zahlen in rationale Zahlen. Diese Konstruktoren werden etwa automatisch aufgerufen, wenn einer der Operanden der selbstdefinierten Operatoren ein int- und kein ratio-Wert ist. Die Konstruktoren wurden so gewählt, daß ohne Parameter der Wert 0 vorbelegt wird, bei Eingabe eines Parameters wird dieser als Zähler übernommen und der Nenner wird auf 1 gesetzt. Sind zwei Parameter vorhanden, so kennzeichnet der erste den Zähler und der zweite den Nenner (ungleich Null!).

- Die meisten Operatoren wurden als Friend definiert. Dies sichert uns, daß auch der erste Operand vom Typ *ratio* sein darf. Nur die Zuweisungsoperatoren wurden als Member deklariert. Hier ist der erste Operand links vom Gleichheitszeichen ja auch immer vom Datentyp *ratio*.
- Die Funktion *abs* wurde ebenfalls als Friend definiert. Es ist einfach natürlicher, *abs(x)* zu schreiben, anstelle von *x.abs()* .
- Die Ein- und Ausgabe funktioniert nach dem bereits aus dem letzten Abschnitt bekannten Muster.

Dieses Beispiel zeigt eindrucksvoll die Mächtigkeit von C++, aber auch die vielen kleinen Haken und Ösen, die erst einmal umschifft werden müssen. Fassen wir nochmals kurz zusammen:

- Konstruktoren dienen zum Initiieren von Klassenobjekten, aber auch als Cast-Operatoren zur Typkonvertierung.
- Destruktor, Copy-Konstruktor und Überladen des Zuweisungsoperators müssen bei dynamisch angeforderten Elementen einer Klasse explizit definiert werden, eventuell auch bei der Verwendung statischer Variablen.
- Memberfunktionen und -operatoren haben immer einen Parameter weniger als Friendfunktionen bzw. -operatoren. Man beachte auch die unterschiedliche Schreibweise bei Funktionen: *a.abs()* versus *abs(a)* . Auch muß bei Memberoperatoren der linke Operand immer ein Klassenobjekt sein.
- (Fast) alle Operatoren können überladen werden (siehe auch nächster Abschnitt).
- Das Überladen von Streams hat immer die Form:

```
ostream& operator << (ostream& , … );                    // >> analog
```

12.7 Überladen spezieller Operatoren

Wir wollen hier noch ein paar spezielle Operatoren überladen. Zum einen ist dies der Inkrementoperator ++, zum anderen die eckigen Klammern [] und letztlich der Cast-Operator. Analog lassen sich dann beispielsweise auch der Dekrementoperator -- und die runden Klammern ( ) bearbeiten.

Eine Besonderheit des Inkrementoperators ist, daß er sowohl als Postfix-, als auch als Prefixoperator existiert. Wir können in C++ tatsächlich beide Operatoren überladen. Beginnen wir mit dem Prefixoperator ++i. Dies ist ein unärer Operator, besitzt also nur einen Operanden. Realisieren wir ihn als Member, was bei unären Operatoren üblich ist, so haben wir eine Operatorfunktion ohne Parameter:

Problematisch wird es, wenn wir auch den Postfixoperator überladen wollen. Dieser muß sich schließlich vom Prefixoperator gleichen Namens unterscheiden. Aus diesem Grund wird dieser als binärer Operator realisiert, dessen zweiter Parameter aber nicht verwendet wird:

```
inline ratio ratio::operator ++ (void)     // Praefix
{  zaehler += nenner;
   return *this;
}
inline ratio ratio::operator ++ (int i)    // Postfix
{  zaehler += nenner;
   return *this -1;
}
```

Programm: 12-overld/ratio.cpp (Prä- und Postfixoperator ++)

Das Überladen der eckigen Klammern bereitet keinerlei Probleme. Der Operator heißt '[]'. Wollen wir beim Stringbeispiel auch das i-te Zeichen ausgeben, so wird dies durch die Definition dieses Operators unterstützt (siehe 12-overld/string3.cpp):

```
char string::operator [ ] (int i)
{  return  (i < laenge) ? s[i] : 0 ; }
```

Ist etwa *str* ein Objekt der Klasse string, so ist 'str[3]' das vierte Zeichen dieser Zeichenkette.

Das Überladen des Cast-Operators zeigen wir wieder am Beispiel der rationalen Zahlen. Dazu wan-

deln wir eine rationale Zahl in eine Double-Zahl um:

```
ratio::operator double ( )
{  return double (zaehler) / double (nenner) ;  }
```

Zu beachten ist, daß das Überladen der Operatoren [ ], ( ), -> und = immer als Member der Klasse geschehen muß. Bei den anderen Operatoren gilt diese Einschränkung nicht.

Zum Schluß noch ein Hinweis: wir haben in all unseren Beispielen die natürliche Erweiterung von Operatoren auf unsere selbstdefinierten Klassenobjekte gewählt. Natürlich wäre es syntaktisch auch richtig, den Operator '-' für die Addition zu verwenden. Doch ich glaube, jeder sieht ein, daß dies weder zweckmäßig noch vernünftig ist.

13 Weitere Möglichkeiten in C++

C++ ist auf den ersten Blick gar nicht so umfangreich wie vielfach gedacht und erwartet. Es fehlen auch tatsächlich nur noch wenige Themen, die bisher noch nicht behandelt wurden:

- Ausnahmebehandlung
- Templates
- Klassenhierarchie und Vererbung
- Virtuelle Funktionen

Das Hauptproblem in C++ ist, daß diese wenigen Erweiterungen gegenüber C so viele erst auf den zweiten Blick erkennbaren Möglichkeiten bieten, daß diese den Rahmen der Vorlesung bei weitem sprengen würden. Doch wir wollen nacheinander zumindest die Grundlagen dazu kennenlernen..

13.1 Ausnahmebehandlung

In Kapitel 10 haben wir die C-Befehle *setjmp* und *longjmp* kennengelernt. Sie dienten dazu, von fast beliebigen Stellen eines Programms auf einen definierten Aufsetzpunkt des Programms zurückzuspringen. Anwendung finden diese beiden Funktionen vor allen Dingen zur Behandlung von Fehlern: das Programm wird nicht durch das Weiterleiten von Fehlern an die übergeordneten Routinen überfrachtet. Andererseits widersprechen diese Jump-Befehle den Aussagen der strukturierten Programmierung.

Erfreulicherweise werden in C++ Mechanismen angeboten, die eine strukturierte Fehlerbehandlung ermöglichen. Wir wollen eine kleine Einführung in diese Fehlerbehandlungsmethoden geben. C++ trennt diese Ausnahmebehandlung in drei Teile:

den Code, wo ein Fehler auftreten kann	(Try-Teil)
den Code, der eine Ausnahme behandelt	(Catch-Teil)
den Code, der eine Ausnahme erzeugt	(Throw-Routine)

Betrachten wir die Implementierung in C++ schematisch:

```
try
{   // Programmcode, der untersucht wird
}
catch (  /* Fall 1 */  )
{   // Fehlerbehandlung zu Fall 1
}
catch (  /* Fall 2 */  )
{   // Fehlerbehandlung zu Fall 2
}
```

Im Block direkt nach dem Schlüsselwort *try* steht wie üblich der Programmcode. Tritt jetzt innerhalb dieses Blocks eine Ausnahme auf, so werden die darin folgenden *Catch*-Blöcke durchsucht, ob der momentan aufgetretene Ausnahmefall darin vorkommt. Wenn ja, so wird der entsprechende *Catch*-Block angesprungen. Die darin enthaltenen Befehle werden abgearbeitet. Anschließend wird mit dem nächsten Befehl nach dem letzten *Catch*-Block fortgesetzt. Wird der aufgetretene Fehlerfall in keinem *Catch*-Block abgefangen, so wird der Fehler an den übergeordneten Block weitergeleitet. Existiert kein übergeordneter *Try-Catch*-Block mehr oder wird dieser Fehler auch dort nicht behandelt, so löst das Laufzeitsystem einen Laufzeitfehler aus.

Try-Catch-Blöcke dürfen beliebig geschachtelt werden. Im Ausnahmefall werden zunächst die *Catch*-Blöcke durchsucht, die zum aktuell durchlaufenen *Try*-Block gehören. Behandelt kein *Catch*-Block diese Ausnahme, so werden die *Catch*-Blöcke des übergeordneten *Try-Catch*-Blockes durchsucht und so fort. Wir können also bestimmte Fehler erst global, andere wiederum gezielt lokal abfangen.

Ausnahmen können direkt vom System ausgelöst werden (Laufzeitfehler), aber auch der Programmierer selbst kann gezielt Ausnahmen erzögen; man spricht hier von einer Ausnahme werfen. Hierzu dient das Schlüsselwort *throw*.

Meist werden von Programmierern folgende Fehlerfälle abgefangen:

Division durch Null,
Überschreitung von Feldgrenzen,
Einlesefehler, insbesondere beim Lesen aus Dateien

Betrachten wir hierzu eine abgespeckte Version der Hashtabelle aus Kapitel 10. Wir lesen aus einer Datei abwechselnd eine Nummer und einen Namen ein. Als Fehler kann auftreten, daß die Kapazität des Feldes, in das die Werte eingelesen werden, nicht ausreicht, aber auch, daß das Einlesen einer Nummer mißlingt, weil in der Datei an dieser Stelle keine Ziffern stehen. Wir fangen beide Fehlerquellen mittels eines *Try-Catch*-Blockes ab:

```
const int DIM = 10;      // klein, um Ausnahme zu provozieren
const int N   = 100;

int main ()
{  struct { int nr;
            char name [N];
   } tabelle [DIM];           // Tabelle zum Abspeichern

   cout << "Demoprogramm zur Ausnahmebehandlung\n";

   ifstream fin ("tab.dat");  // Eingabestrom fin oeffnen
   if ( !fin )
   {  cerr << "Fehler beim Oeffnen\n"; return 1;
   }

   // Einlesen und Abspeichern:
   int i = 0;
   try
   {  while ( !fin.eof() )
      {  if (i>=DIM)           throw "zu viele Daten\n";
         fin >> tabelle[i].nr;
         if (!fin)             throw i;
         fin.getline (tabelle[i++].name, N);
      }
   }
   catch (int k)       // behandelt int-Ausnahme
   {  cerr << "Fehler beim Lesen der " << k << ". Zahl\n";
   }
   catch (char *s)     // behandelt char*-Ausnahme
   {  cerr << s;
   }

   // ...
```

Programm: 13-vererb/ausnahme.cpp

Der Ausnahmemechanismus wird aus diesem Programm auf einfache Weise deutlich. Im *Try*-Block werden zwei Ausnahmen geworfen, eine *Int*- und eine *Char**-Ausnahme. Die beiden *Catch*-Blöcke fangen genau diese zwei Ausnahmen auf. Der Wert, der hinter dem Throw-Bezeichner angegeben wird, wird

dabei an den *Catch*-Block als Parameter übergeben.

Wir erkennen, daß beim Werfen einer Ausnahme die Schleife sofort beendet wird. Hinter einem *Try*-Block dürfen beliebig viele *Catch*-Blöcke stehen. Voraussetzung ist, daß sich die Catchblöcke im Parametertyp unterscheiden. Die *Catch*-Blöcke werden nur im Ausnahmefall angesprungen. Ansonsten wird die Programmbearbeitung direkt hinter dem letzten *Catch*-Block fortgesetzt. Stellt sich bei der Behandlung in einem *Catch*-Block heraus, daß diese spezielle Ausnahme erst in einem übergeordneten *Try-Catch*-Block behandelt werden soll, so kann mittels *throw;* (keine Parameter) dieser Fehler weitergereicht werden. Dies soll als eine kleine Einführung genügen.

13.2 Templates

Bei Templates unterscheiden wir zwischen Funktions- und Klassentemplates. Templates helfen, daß nicht zu jedem einzelnen Datentyp eine eigene Funktion bzw. Klasse geschrieben werden muß. Betrachten wir zur Einführung die Funktion *vertausche*. Sollen sowohl int-, char-, float- und double-Variablen vertauschbar sein, benötigen wir 4 Funktionen mit den jeweiligen Datentypparametern. Einfacher geht es mit Templates:

```
template <class T> void vertausche (T &a, T &b)
{  T c = a;     // Hilfsvariable c
     a = b;
     b = c;
};

int main()
{  int i, j;
   float x, y;
   unsigned char c1, c2;

   cout << "Eingabe: 2 Int, 2 Float, 2 Zeichen: ";
   cin  >> i >> j >> x >> y >> c1 >> c2;

   vertausche (i, j);           // int-Zahlen
   vertausche (x, y);           // float-Zahlen
   vertausche (c1, c2);         // char-Zeichen

   cout << "i = " << i << "  und  j = " << j << '\n';
   cout << "x = " << x << "  und  y = " << y << '\n';
   cout << "c1= " << c1<< "  und  c2= " << c2<< '\n';
   return 0;
}
```

Programm: 13-vererb/template1.cpp

Wir haben im Programm einen Platzhalter namens *T* verwendet. Dieser kann jeden Standarddatentyp und jeden selbstdefinierten Klassentyp annehmen. Je nachdem mit welchen Parametern diese Funktion nun aufgerufen wird, wird der Compiler die entsprechende Funktion generieren. Das kleine Beispielprogramm demonstriert dies an Int-, Float- und Char-Zahlen.

Dieses Template-Prinzip läßt sich auch auf Klassen erweitern. Betrachten wir etwa unser Stackprogramm aus Kapitel 11. Die Verwendung von Klassen kapselt die internen Daten sauber gegen unerlaubte Zugriffe ab. Doch wir können in dieser Klasse nur Int-Werte abspeichern. Wünschenswert wäre es, alle vordefinierten und auch selbstdefinierten Datentypen in dieser Klasse aufzunehmen. Mittels Templates ist dieser Wunsch auf einfache Weise zu erfüllen, zumindest größtenweise. Wir können damit nämlich einen Stack definieren, der Werte eines beliebig vorgegebenen Datentyps aufnehmen kann (um innerhalb eines einzigen Klassenobjekts auch Elemente unterschiedlicher Datentypen aufnehmen zu können, sei auf die sogenannten Containerklassen von C++ verwiesen). Betrachten wir zunächst die Deklaration der allgemeinen Stackklasse:

```
template <class T> class stack
{  T * top;           // Spitze des Stacks
   T * bottom;        // Anfang des Stacks
   int size;          // max. Groesse des Stacks

  public:
   stack (int = 6);      // Konstruktor
   ~stack ();            // Destruktor
   void push (T);
   T pop (void);
};

template <class T> stack<T>::stack (int s)  // Konstruktor
{  top = bottom = new T [size = s]; }

template <class T> stack<T>::~stack ()      // Destruktor
{  delete bottom; }

template <class T> void stack<T>::push (T i)
{  if ((top - bottom) < size )
      *top++ = i;
}

template <class T> T stack<T>::pop (void)
{  return (top > bottom) ? *(--top) : 0 ;
}
```

Programm: 13-vererb/template2.cpp (Definition der Klasse)

Wieder erkennen wir, daß wir den Bezeichner *T* als Platzhalter verwenden. Es ist allerdings nicht ganz einfach, die einzelnen Funktionen extern zu definieren. Die Klasse heißt *stack<T>*, wobei *T* der Platzhalter ist. Weiter muß jedes Mal dieser Platzhalter explizit mittels des Ausdrucks *template <class T>* mit angegeben werden. Betrachten wir nun eine gekürzte Version des Hauptprogramms:

```
int main ()
{  int i, zahl;
   double dzahl;
   stack<int> keller(10);   // Keller mit 10 int-Elementen
   stack<double> stapel;    // Stapel mit 6 double-Elementen

   cout << "\n\nStackprogramm\n\n";
   cout << "Bitte eine int-Zahl eingeben: ";
   cin >> zahl;
   keller.push (zahl);

   cout << "\n\nund eine double-Zahl:\n";
   cin >> dzahl;
   stapel.push (dzahl);

   // Ausgabe:
   cout << "\nAusgabe der Int-Zahl: " << keller.pop()
        << "\nAusgabe der Double-Zahl: " << stapel.pop();
   return 0;
}
```

Programm: 13-vererb/template2.cpp (Ausschnitt aus Hauptprogramm)

Ganz eindrucksvoll erkennen wir, daß wir jetzt eine Fülle von Klassen benutzen können. Wir schreiben dazu einfach:

```
stack<Datentyp>
```

Als Datentyp ist dabei jeder vordefinierte Datentyp, aber auch jeder selbstdefinierte Typ erlaubt. Wir können beispielsweise auch einen Stack definieren, der ganze Strukt-Variablen oder auch Klassenobjekte aufnimmt.

Dies mag als eine kleine Einführung zu Templates genügen. Zu beachten ist, daß die übermäßige Verwendung von Templates auch Schattenseiten besitzt. Dies führt nämlich leicht dazu, daß der Compiler innerhalb des Programms nicht immer eindeutig erkennen kann, welche Funktion oder Klasse er anwenden muß, was Compilerfehler zur Folge hat. Mein Rat: Templates sollten sparsam eingesetzt werden, wobei der Cast-Operator im Programm bei Nichteindeutigkeit der Typzuordnung für Klarheit sorgt.

13.3 Klassenhierarchie und Vererbung

Betrachten wir nun etwas näher die Möglichkeiten der Vererbung. Wie bereits in Kapitel 11 im Abschnitt über objektorientierte Programmierung erwähnt, kommt es vor, daß bestimmte Klassen, etwa die Klasse der Rechtecke und die Klasse der Ellipsen, Gemeinsamkeiten aufweisen. Hier empfiehlt es sich, zunächst eine Oberklasse zu entwerfen, wo die gemeinsamen Eigenschaften definiert werden. Die davon abgeleiteten Unterklassen enthalten dann nur noch die Spezifika dieser Klassen. Der Vorteil ist, daß ein Teil des Codes nur einmal programmiert werden muß. Insbesondere bei Änderungen muß dann auch nur an einer Stelle Code angepaßt werden, was die Fehleranfälligkeit minimiert.

Betrachten wir als Beispiel die Klasse *string*. Möglicherweise gibt es nun eine Anwendung, die häufig die Anzahl der Großbuchstaben oder die Anzahl aller Buchstaben in der Zeichenkette benötigt. Es wäre dann vorteilhaft, diese Information direkt in der Klasse zu speichern. Wir können die Klasse erweitern. Dies hätte aber den Nachteil, daß auch Anwendungen, die diese Information nicht benötigen, diesen Overhead mit sich schleppen. Besser ist es, eine Klasse *astring* zu definieren, die aus unserer String-Klasse abgeleitet wird. In dieser abgeleiteten Klasse können alle bisherigen Eigenschaften geerbt werden, nur die Erweiterungen werden zusätzlich definiert.

```
class astring : public string
{   int gross;                // Anzahl der Grossbuchstaben
  public:
    astring ();               // Konstruktor
    astring (const char *);   // Konstruktor
    int anzahl_gross (void) const;
    int anzahl (void) const;    // kapselt die Laenge
    friend astring operator + (const astring&,
                               const astring&);
};
```

Programm: 13-vererb/string4.cpp (Definition der Klasse)

Wir erkennen die Ableitung der Klasse *astring* von der Klasse *string* an der ersten Zeile der Klassendefinition von *astring*. Das Wort *public* bedeutet, daß alle Public-Elemente der Klasse *string* auch in der abgeleiteten Klasse als Public-Elemente definiert werden. Standardmäßig würden sie als Private-Elemente übernommen werden.

Wir haben allerdings ein Problem. Die privaten Elemente der Klasse *string* sind auch von der abgeleiteten Klasse *astring* aus nicht zugreifbar. Wir können daher die Zeichenkette *s* und die Länge *laenge* der Klasse *string* nicht anfassen. Dies ist aber dringend erforderlich, um Konstruktoren, Funktionen und Operatoren in der abgeleiteten Klasse korrekt zu definieren. Wir müssen daher unsere Klasse *string* abändern. Es ist aber nicht erwünscht, alle Elemente allgemein zugänglich zu machen. Aus diesem Grund gibt es auch nicht nur zwei, sondern drei Schutzmöglichkeiten:

public:	// allgemein zugänglich
private:	// nur von Member- und Friend-Funktionen bzw. Operatoren verwendbar
protected:	// wie private, allerdings auch von abgeleiteten Klassen aus zugänglich

Ergänzen wir nun die Klasse *string* dahingehend, daß alle Privat-Elemente auf protected gesetzt werden, so können wir auf diese Elemente auch von der Klasse *astring* aus zugreifen, nicht jedoch von außerhalb dieser beiden Klassen.

Betrachten wir jetzt die Funktionen der Klasse *astring*:

```
astring::astring () : string (), gross (0)
{
}

astring::astring (const char *s) : string (s)
{  for (gross=0; *s!='\0'; s++)
      if (isupper(*s))    gross++;
}
```

```
int astring::anzahl_gross (void) const
{   return gross;
}

int astring::anzahl (void) const
{   return laenge;
}

astring operator + (const astring& str1,
                    const astring& str2)
{   astring str = str1;
    strcat (str.s, str2.s);
    str.laenge += str2.laenge;
    str.gross  += str2.gross;
    return str;
}
```

Programm: 13-vererb/string4.cpp (Konstruktoren und Funktionen)

Neu ist die Möglichkeit, auf die Konstruktoren der Oberklasse zuzugreifen, aber auch Variablen direkt zu initialisieren. In unserem Beispiel werden zunächst die angegebenen Konstruktoren der Oberklasse aufgerufen, anschließend wird die Variable *gross* mit dem Wert 0 initiiert. Im Implementierungsteil ist somit nichts mehr zu tun, er bleibt leer.

Der Kopf einer Konstruktor-Definition hat ganz allgemein folgendes Aussehen:

```
Klasse :: Klasse ( Parameterliste ) :  Basisklasse1 ( Parameterliste ) , … ,
                                       Klassenvariable1 ( Parameterliste ) , …
{ … }
```

Hinter dem Doppelpunkt folgen also zunächst alle Basisklassenkonstruktoraufrufe mit den notwendigen Parameterangaben und schließlich alle Klassenvariablen, die auf diesem Wege initiiert werden sollen. In der Parameterliste werden hier die Initiierungswerte angegeben. Sowohl die Angabe von Basisklassen als auch die von Klassenvariablen darf leer bleiben. Grundsätzlich dürfen also auch bei nicht abgeleiteten Klassen Klassenvariablen auf diese Art initiiert werden.

Da die Elementvariable *gross* als Privat-Element realisiert wurde, benötigen wir noch eine (konstante) Funktion *anzahl_gross*, die diese Variable ausgibt. Weiter müssen wir auch unseren Operator + neu definieren. Denn mittels dieses Operators muß auch die Variable *gross* neu gesetzt werden.

Anders verhält es sich mit dem Operator <<. Es wird hier nur die Zeichenkette ausgegeben und braucht für die neue Klasse *astring* nicht extra definiert zu werden. Schließlich ist der Operator << in der Klasse *string* als Public-Operator definiert und wird damit automatisch auch für abgeleitete Klassen übernommen (gilt auch für protected!). Nur wenn ein solcher Operator (oder eine solche Funktion) zusätzliche Eigenschaften aufweisen soll, wie etwa beim Plusoperator, muß er neu geschrieben werden.

Beachten Sie, daß C++ automatisch erkennt, welcher Plusoperator aufgerufen werden soll. Es gilt:

– Sind beide Operanden vom Typ *astring*, so wird der Plusoperator der Klasse *astring* aufgerufen.
– Sind beide Operanden vom Typ *string*, so wird der Plusoperator der Klasse *string* aufgerufen.
– Ist ein Operand vom Typ *astring*, der andere vom Typ *string*, so wird der Operand vom Typ *astring* in den Typ *string* konvertiert und der Plusoperator der Klasse *string* verwendet. Der Ergebniswert ist vom Typ *string*.
– Ist ein Operand vom Typ *astring*, der andere vom Typ *char**, so wird der Operand vom Typ *char** mittels des vorhandenen Konstruktors für *char** in den Typ *astring* überführt. Somit wird der Plusoperator der Klasse *astring* verwendet, und der Ergebniswert ist vom Typ *astring*.
– Ist ein Operand vom Typ *string*, der andere vom Typ *char**, so wird der Operand vom Typ *char** mittels des vorhandenen Konstruktors für *char** in den Typ *string* überführt. Somit wird der Plusoperator der Klasse *string* verwendet, und der Ergebniswert ist vom Typ *string*.

Es sei aber darauf hingewiesen, daß C++ nicht alle Kombinationen verarbeiten kann. Folgende beiden Fälle führen zu Syntaxfehlern:

– Sind beide Operanden vom Typ char*, so existiert kein geeigneter Operator.

– Ist ein Operand vom Typ *astring*, der andere vom Typ *string* und existiert in der Klasse *astring* ein Konstruktur für den Typ *string*, so weiß der Compiler nicht, ob er den Operanden vom Typ string in den Typ astring oder den Operanden vom Typ astring in den Typ string konvertieren soll. In diesem Fall muß der Programmierer einen Operanden mittels eines Cast-Operators explizit umwandeln.

Wir nennen das Vererben von Funktionen und Operatoren an abgeleitete Klassen und das automatische Erkennen der richtigen Funktion bzw. des richtigen Operators beim Aufruf auch **Polymorphie**. Weiter benötigen wir keinen Zuweisungsoperator oder Cast-Operator (Konstruktor), um eine Variable vom Typ *astring* in den Typ *string* zu verwandeln. C++ läßt in diesem Fall automatisch die Erweiterungen weg. Betrachten wir ein kleines Testprogramm, das unsere Strings verwendet:

```
int main()
{  astring s1 = "Abend";          // Parameter
   astring s2 ("Essen");          // Parameter
   cout << "\nTest des Ueberladens von Operatoren\n\n";
   cout << "String1: " << s1 << "\n";
   cout << "String2: " << s2 << "\n";

   // Konkatenierung:
   s1 = s1 + s2 + "Gehen";        // auch mit Zeichenkette!
   cout << "Zusammen: " << s1 << endl;
   cout << "Grossbuchstaben: " << s1.anzahl_gross() << endl;

   string s3 = "Jawohl, ";
   s3 = s3 + s1;                  // astring -> string
   cout << "String S3: " << s3 << endl;
   return 0;
}
```

Programm: 13-vererb/string4.cpp (Funktion *main*)

Ganz wichtig zu beachten ist, daß der Ausdruck *s3 + s1* vom Typ *string* ist. Der Versuch

s1 = s3 + s1;

würde daher eine Fehlermeldung nach sich ziehen, da der Ausdruck rechts nicht in den Datentyp *astring* konvertiert werden kann.

13.4 Virtuelle Funktionen

Die Polymorphie geht noch einen Schritt weiter; denn relativ oft liegt der Fall vor, daß eine Funktion in einer abgeleiteten Klasse genau die gleichen Parameter besitzen soll wie die gleichnamige Funktion in der Oberklasse. Denken wir nur an eine Funktion zum Zeichnen von Grafikobjekten, die in allen Klassen wie folgt definiert ist:

void draw (void);

Auch dies wird in C++ unterstützt. Allerdings muß es sich um eine Memberfunktion handeln. Wir wollen die Idee solcher Funktionen an einem Beispiel vorstellen. Dazu erweitern wir unsere Stringklassen *string* und *astring* um eine Funktion *puts*. Diese Funktion gibt neben dem String auch die Länge und im Falle der Klasse *astring* auch die Anzahl der Großbuchstaben aus. Betrachten wir die Erweiterungen:

```
void string::puts (void)
{ cout << "String: " << s << "; Laenge: " << laenge << "\n";
}

void astring::puts (void)
{
    cout << "String: " << s << "; Laenge: " << laenge <<
            ";  Grossbuchstaben: " << gross << "\n";
}
```

```
int main()
{  string *feld[2];
   feld[0] = new astring ("Und weiter geht’s");
   feld[1] = new string ("Und nochmal");
   feld[0]->puts();
   feld[1]->puts();
   return 0;
}
```

Programm: 13-vererb/string5.cpp (Ausschnitt)

Dieses Beispiel zeigt, daß wir zwei Memberfunktionen mit identischer Schnittstelle definiert haben. Weiter erkennen wir, daß wir einem Zeiger auf *string* auch einen Zeiger auf eine abgeleitete Klasse zuweisen können. Dies ist grundsätzlich möglich; denn Zugriffe mittels Methoden der Oberklasse sind auch immer in der abgeleiteten Klasse definiert. Im umgekehrten Fall liegt allerdings ein Fehler vor: wir können einem Zeiger auf *astring* keinen Zeiger der Basisklasse zuweisen.

In unserem Beispiel hat es der Compiler noch nicht allzu schwer. Er erkennt bereits zum Übersetzungszeitpunkt, daß sowohl *feld[0]* als auch *feld[1]* vom Typ *string** sind. Somit wird in beiden Fällen die Funktion *puts* der Klasse *string* aufgerufen. Die zusätzliche Information der Klasse *astring* wird also nicht ausgegeben. Wir sprechen hier von einer **statischen Bindung**; statisch deshalb, da die Zuordnung der Funktion *puts* zu den Objekten bereits beim Compilieren festgelegt wird.

Dieses Beispiel entspricht jedoch nicht dem, was wir eigentlich wünschen. Schließlich sollte im Falle eines Objektes vom Typ *astring* auch die entsprechende Memberfunktion aufgerufen werden. Um diesen Wunsch zu erfüllen, kann der Compiler jedoch nicht mehr bereits zum Zeitpunkt des Compilierens festlegen, welche Funktion *puts* (*string::puts* oder *astring::puts*) er aufrufen soll. Denn es ist jederzeit möglich, die Adresse *feld[0]* einmal einem Objekt vom Typ *astring* und einmal einem Objekt vom Typ *string* zuzuweisen. Dies erfordert eine sogenannte **dynamische Bindung**. Das Programm entscheidet erst zur Laufzeit, welche Funktion verwendet werden soll. Diese dynamische Bindung ist aufwendig und erfordert zusätzlichen Verwaltungsoverhead im Programm, wird aber von C++ voll unterstützt. Es muß dazu nur einmal bei der Deklaration der Funktion *puts* in der Basisklasse *string* das Wort ***virtual*** vorangestellt werden:

```
virtual void puts (void);
```

Dieses Zauberwort *virtual* führt dazu, daß obiges Programm folgende Ausgabe erzeugt:

```
String: Und weiter geht’s;  Laenge: 17;  Grossbuchstaben: 1
String: Und nochmal;  Laenge 11
```

Jetzt erkennt das Programm zur Laufzeit, daß das erste Feldelement tatsächlich auf ein Element vom Typ *astring* zeigt. Entsprechend wurde die Funktion *puts* dieser Klasse aufgerufen. Ohne dem Zusatz *virtual* in der Basisklasse hätte die Angabe der Anzahl der Großbuchstaben gefehlt. Das Wort *virtual* wird übrigens nur genau einmal geschrieben, und zwar bei der Deklaration der Funktion in der Basisklasse. Es erscheint nicht mehr bei der eigentlichen Definition (siehe oben) und auch nicht in der abgeleiteten Klasse.

Es sei nicht verschwiegen, daß das Verwenden virtueller Memberfunktionen für den Compiler und insbesondere das Laufzeitsystem einen wesentlich erhöhten Aufwand bedeutet. Das Laufzeitsystem legt sich intern Verwaltungsdaten an, aus denen es zweifelsfrei erkennen kann, welche der virtuellen Funktionen gegebenenfalls eingesetzt werden muß.

Nicht selten werden auch Destruktoren virtuell definiert. So wird sichergestellt, daß beim Entfernen eines Elements immer das Objekt der richtigen Klasse mitsamt dem dazugehörigen Speicher freigegeben wird. Demgegenüber dürfen Konstruktoren grundsätzlich nicht zu virtuellen Funktionen erklärt werden.

13.5 Abstrakte Basisklassen

Zur Vertiefung der Polymorphie und der virtuellen Funktionen wollen wir noch ein weiteres Beispiel

vorstellen. Dazu betrachten wir eine kleine Firma mit Mitarbeitern, die sich in die Kategorien Leitung, Programmierer und Sekretärin einteilen lassen. Zu diesen Mitarbeitern werden in einer Kartei Daten wie Name, Alter, Gehalt, aber auch Programmiersprachen- und Stenokenntnisse gesammelt. Wichtig ist, daß beispielsweise für eine Sekretärin kein Knowhow zu Datenbanken erforderlich ist und damit auch gar nicht gespeichert werden muß. Ähnlich ist es nicht nötig, für Programmierer abzulegen, wie schnell er Maschine schreiben kann. Aus diesem Grund wird eine Klasse *Person* angelegt, die all die Daten sammelt, die allen drei Kategorien gemeinsam ist. Anschließend werden in den von der Klasse *Person* abgeleiteten Klassen *Leitung*, *Programmierer* und *Sekretaerin* alle spezifischen Daten hinterlegt. Betrachten wir zunächst die Definition der Klasse *Person*:

```
class person
{ protected:
    int  persnr;
    char name[30];
    float einkommen;
  public:
    person (int, char *);            // Konstruktor
    virtual void ausgabe (void) = 0; // keine Implementierung
};
```

Programm: 13-vererb/person.h (Klasse *person*)

Wir erkennen aus dieser Klasse die internen Variablen für die Personalnummer, den Namen und das Einkommen, aber auch den Konstruktor, der gleich die Personalnummer und den Namen aufnimmt. Weiter existiert eine virtuelle Funktion *ausgabe*, die die gespeicherten Daten ausgibt. Sie ist virtuell definiert, damit immer genau die Daten der gerade angesprochenen Klasse ausgegeben werden.

In unserem Anwendungsbeispiel dient die Klasse *person* ausschließlich als Basis für die noch zu definierenden abgeleiteten Klassen. Aus diesem Grund werden keine Daten zur Klasse *person* ausgegeben. Die virtuelle Funktion *ausgabe* ist daher in dieser Klasse eigentlich noch gar nicht notwendig. Sie wird sozusagen nur als Platzhalter bereits in der Basisklasse definiert. Durch den Zusatz „= 0" machen wir aber darauf aufmerksam, daß diese virtuelle Funktion in der Basisklasse selbst nur deklariert, nicht aber definiert wird. Um zu verhindern, daß diese nicht implementierte Funktion in irgendeiner Weise aufgerufen wird, können keine Objekte zur Klasse *person* erzeugt werden. Eine solche Basisklasse heißt daher auch **abstrakte Basisklasse**.

Betrachten wir jetzt auch noch die drei abgeleiteten Klassen:

```
class leiter : public person
{  char abteilung[20];
  public:
    leiter (int, char *);              // Konstruktor
    void daten_lesen (float, char *); // Gehalt, Abt. lesen
    void ausgabe (void);               // gibt Daten aus
};

class programmierer : public person
{  char sprache1[20], sprache2[20];
    char netze, datenbank;             // 'j' oder 'n'
  public:
    programmierer (int, char *);       // Konstruktor
    void daten_lesen (float, char *, char *, char, char);
                // Gehalt, Sprache1, Sprache2, Netze, DB lesen
    void ausgabe (void);               // gibt Daten aus
};

class sekretaerin : public person
{  char steno, text;                   // 'j' oder 'n'
    int anschlaege_pro_min;
  public:
    sekretaerin (int, char *);         // Konstruktor
    void daten_lesen (float, char, char, int);
               // Gehalt, Steno, Textverarb., Anschlaege lesen
    void ausgabe (void);               // gibt Daten aus
};
```

Programm: 13-vererb/person.h (Klassen *leiter*, *programmierer*, *sekretaerin*)

Wir erkennen die zusätzlichen Daten der einzelnen Mitarbeiter, etwa Abteilungsname für den Leiter oder die Programmiersprachen und die Netz- Datenbankkenntnisse für den Programmierer und die Anschläge auf der Schreibmaschine und die Steno- und Textverarbeitungskenntnisse für die Sekretärin. Jede dieser Klassen besitzt außerdem einen Konstruktor, der wie in der Basisklasse bereits die Personalnummer und den Namen aufnimmt. Weiter gibt es je eine Funktion *daten_lesen*, die die speziellen Daten der einzelnen Klassen einliest. Zuletzt kommen wir auf die bereits in der Basisklasse definierte virtuelle Funktion *ausgabe*. Wir finden Sie in allen drei abgeleiteten Klassen wieder.

Wir wenden uns nun an die Implementierung dieser Klassen, wobei wir uns nur auf die Klassen *person* und *leiter* konzentrieren wollen. Die beiden anderen Klassen sind analog definiert:

```
// Konstruktor von person, belegt Persnr und Name vor:
person::person (int nr, char * s) : persnr(nr)
{  strcpy (name, s);  }

// Konstruktor von leiter:
leiter::leiter (int nr, char * s) : person (nr, s) { }

void leiter::daten_lesen (float gehalt, char * abt)
{  einkommen = gehalt;
   strcpy (abteilung, abt);
}

void leiter::ausgabe (void)     // gibt Daten des Leiters aus
{  cout.setf (ios::fixed);
   cout.precision (2);
   cout << "Leiter " << name << ", Persnr: " << persnr
        << "\n          Einkommen: " << einkommen
        << ", Abteilung: " << abteilung << "\n\n";
}
```

Programm: 13-vererb/person.cpp (Implementierung der Klassen *person*, *leiter*)

Wir sehen, daß die Implementierung der Konstruktoren der abgeleiteten Klassen einfach ist: es wird einfach nur der Konstruktor der Basisklasse aufgerufen. Die Implementierung der beiden Prozeduren *daten_lesen* und *ausgabe* unterscheiden sich natürlich etwas in den anderen beiden Klassen, es werden ja schließlich die klassenspezifischen Daten eingelesen und ausgegeben. Wie bereits weiter oben erwähnt, existiert hier keine Implementierung der Prozedur *ausgabe* für die Basisklasse.

Wir kommen nun zum Hauptprogramm, das diese soeben definierten Klassen benutzt. Wir definieren einfachheitshalber nur ein Feld, in das die Mitarbeiterdaten abgelegt werden. Im Programm werden insgesamt vier Mitarbeiter in dieses Feld gespeichert und zum Schluß in einer Schleife ausgegeben. Doch betrachten wir das Programm selbst:

```
int main ()
{  const int N = 20;
   person * personal [N] = {0};    // Personal-Feld

   cout << "\nPersonalprogramm.\n\n";

   leiter * l = new leiter (13, "Egon Maus");
   l->daten_lesen (6400.10, "Verkauf");
   personal[0] = l;         // Leiter

   programmierer * p = new programmierer (5, "Fritz Faul");
   p->daten_lesen (2555.55, "Pascal", "C", 'n', 'n');
   personal[1] = p;         // Programmierer 1

   p = new programmierer (9, "Maria Fleissig");
   p->daten_lesen (4990.90, "C++", "Java", 'j', 'j');
   personal[2] = p;         // Programmierer 2

   sekretaerin * s = new sekretaerin (26, "Dorothea Tipp");
   s->daten_lesen (4000.0, 'j', 'j', 350);
   personal[3] = s;         // Sekretaerin
```

```
    // Ausgabe aller Daten:
    for (int i=0; personal[i] != NULL; i++)
       personal[i]->ausgabe ();      // virtuelle Funktion!!!

    return 0;
}
```

Programm: 13-vererb/persprog.cpp

Hier interessieren weniger die speziellen Eingaben. Vielmehr ist uns wichtig, daß sichergestellt ist, daß die einzelnen Mitarbeiterdaten auch vollständig ausgegeben werden. Obwohl das Personalfeld nur aus Zeigern auf *person* besteht, erkennt das Programm dank der virtuellen Funktion *ausgabe* immer, welche Ausgabe gerade gemeint ist.

Dieses Wissen über die korrekt zugeordnete virtuelle Funktion geht auch dann nicht verloren, wenn mittels Verzweigungen und Schleifen einem Element des Personalfeldes mal der eine oder andere Mitarbeiter zugeordnet wird. Immer findet das Laufzeitsystem dank der dynamischen Bindung die korrekte Ausgabefunktion.

13.6 Konstanten innerhalb von Klassen

Zum Schluß zeigen wir noch, daß wir in Klassen auch Konstanten definieren können. Diese Konstanten werden mit der Erzeugung eines Klassenobjekts festgelegt und sind unveränderlich bis zu dessen Zerstörung. Die Definition einer solchen Konstante und deren Initiierung betrachten wir am einfachsten an einem Beispiel. Dazu nehmen wir unsere Stackklasse aus Kapitel 11 und definieren jetzt die Variable *size* (die Größe des Stacks) als Konstante. Diese Konstante muß von allen Konstruktoren vorbelegt werden. Da eine Wertzuweisung an die Konstante grundsätzlich untersagt ist, muß diese Vorbelegung bereits im Kopf der Konstruktoren stattfinden, dort wo auch Basisklassenkonstruktoren aufgerufen und Variablen initiert werden können. Betrachten wir die Änderungen in der Stackklasse:

```
class stack
{     int * top;            // Spitze des Stacks
      int * bottom;         // Anfang des Stacks
      const int size;       // Stackgroesse als Konstante
   public:
      stack (int = 10);                // Konstruktor
      ~stack ();                       // Destruktor
      void push (int);
      int pop (void);
};

stack::stack (int s) : size (s)  // Konstruktor, initiert
{  top = bottom = new int[size]; }            // Konstante

stack::~stack ()                 // Destruktor
{  delete bottom; }

void stack::push (int i)
{  if ((top - bottom) < size )  *top++ = i }

int stack::pop (void)
{  return (top > bottom) ? *(--top) : 0; }
```

Programm: 13-class/stackc.cpp

Sicherlich wurde bemerkt, daß C++ viele Aufgaben automatisch übernimmt. Dies ist eine große Entlastung für den Programmierer. Sicherlich wurde aber auch schon festgestellt, daß wir relativ genau wissen müssen, wann der Compiler was durchführt. Die Spielregeln sind zwar exakt festgelegt, sind aber leider sehr umfangreich (die ANSI-Norm ist über 600 Seiten dick). Ich hoffe, daß Ihnen das Programmieren in C++ dadurch aber nicht verleidet wird. Denn eines gilt auch für C++: Übung macht den Meister. Ich wünsche Ihnen viel Erfolg auf diesem Weg!