Programmieren in C

Vorlesungsskript zu Programmieren 1

Edwin Schicker
Fachhochschule Regensburg

Inhaltsverzeichnis

1 Einleitung

Prüfung: Keine Zulassungsvoraussetzungen. Zugelassen in der Prüfung sind dieses Skript, eigene Mitschriften zur Vorlesung, eigene Formelsammlungen, nicht jedoch Programmcode, der nicht in der Vorlesung besprochen wurde.

Übungen: In den Räumen U511, U514 und U521 befinden sich je 20 PCs mit den Betriebssystemen WindowsXP, Windows Vista und Unix. Die Übungen werden unter WindowsXP vorgeführt. Als Compiler stehen unter anderem zur Verfügung: Visual Studio C++ von Microsoft (sehr umfangreiches Werkzeug), Dev-C++ von Bloodshed (reicht für den Anfang vollkommen aus!)

Kennung: Zum Arbeiten auf den Rechnern werden die NDS-Benutzerkennungen benötigt.

Übungsbetrieb: In den Übungen werden Aufgaben gestellt und besprochen. Musterlösungen liegen vor.

Vorlesungsinhalt: Prozedurale Programmierung mit C (und etwas C++). Inhalt: Dieses Skript.

Vorlesungsvoraussetzungen: Keine, aber Spaß am Programmieren ist mehr als nur erwünscht.

Literatur: Kernighan/Ritchie: Programmieren in C, Hanser Verlag (ca. 32 Euro)
Wolf: C von A bis Z, Galileo Computing, (39,90 Euro)
Breymann: C++, eine Einführung, Hanser Verlag, enthält aber viel C++
RRZN-Skripte zu C, C++, UNIX (je ca. 5 Euro, im Uni-Rechenzentrum)

Erstellen von C/C++-Programmen:

Objektorientierte Programmierung erfolgt erst im zweiten Semester, wo dann C++ Pflicht wird! Doch von Anfang an wird auf handliche Werkzeuge von C++ wie die Ein- und Ausgabeströme und Referenzen eingegangen, so dass für die Vorlesung ein C++-Compiler erforderlich ist, auch wenn im Skript häufig nur von der Sprache C die Rede sein wird.

Jeder handelsübliche C/C++-Compiler kann verwendet werden. Spitze ist der unter GNU-Lizenz angebotene Compiler *gcc*! Eine grafische Oberfläche (IDE: integrated development environment, integrierte Entwicklungsumgebung) ist wünschenswert. Hier wird die IDE von Bloodshed für den Anfang empfohlen. Dieses Programm (Dev-C++) wird mit IDE geliefert und ist kostenlos.

Ein Spitzenwerkzeug ist natürlich Visual Studio C++ von Microsoft. Es kann von Studierenden ebenfalls kostenlos bezogen werden. Das Werkzeug ist aber so mächtig, dass es den Anfänger mehr verwirrt als nützt. Im November werden wir auf Visual Studio umsteigen.

Beide Compiler sind natürlich in den Rechnerpools verfügbar.

2 Einführung in C

2.1 Wozu Programme?

Frage: Warum brauchen wir einen Taschenrechner?
Antwort: Damit wir schnell und sicher rechnen können!

Frage: Warum brauchen wir Programme?
Antwort: Damit wir viele Rechenschritte automatisch nacheinander ausführen können!

Beispiel: In einem Betrieb gibt es noch Tausende alte Beträge, die in DM ausgewiesen sind. Sind diese Beträge elektronisch gespeichert, so können wir mit Hilfe eines Programms die Umstellung auf Euro in Bruchteilen einer Sekunde ausführen, mit einem Taschenrechner werden wir Stunden brauchen, ohne

Hilfsmittel sicherlich Tage. Mit höchster Wahrscheinlichkeit wird am Ende das Programm die wenigsten Fehler aufweisen!

2.2 Was ist ein (Computer-)Programm?

Ein **Computerprogramm** ist eine Folge von Befehlen (Anweisungen), die auf einem Computer zur Ausführung gebracht werden können, um damit eine bestimmte Funktionalität (z.B. Textverarbeitung) zur Verfügung zu stellen.

Die möglichen Befehle eines Computerprogramms sind fest vorgegeben und unterscheiden sich je nach Computersprache. Eine Computersprache ist anders als menschliche Sprachen exakt definiert, die Syntax ist festgelegt und erlaubt damit keine Interpretation.

In dieser Vorlesung wird die Computersprache C mit einigen Zusätzen in C++ verwendet. Bis auf Winzigkeiten ist die Sprache C++ eine echte Erweiterung von C. Dies heißt also:

Ein C-Programm ist immer auch ein C++-Programm. Dies gilt nicht umgekehrt!

Die Sprache C wurde von Kernighan und Ritchie Anfang der 70er Jahren entwickelt. Etwa 10 Jahre später erfolgte durch Stroustroup die Weiterentwicklung zur Sprache C++. Die Sprache C ist eine höhere Programmiersprache, die aber eine sehr maschinennahe Programmierweise unterstützt. Die Sprache C++ setzt voll auf C auf und unterstützt hervorragend die objektorientierte Programmierung.

Wer mehr über Computerprogramme, insbesondere deren Geschichte erfahren will, möge doch einen Blick auf Wikipedia mit dem Stichwort *Computerprogramm* werfen!

2.3 Das erste C-Programm

Betrachten wir unser erstes C-Programm. Dieses Programm gibt die Zeile „Hello World“ aus.

```
/* Unser erstes Programm */
#include <stdio.h>
int main ()
{
   printf("Hello world\n");
   return 0;
}
```

Programm 02-einf\prog1.c

Erklärungen:

- Ein Programm besitzt einen bestimmten Grundaufbau. Dazu gehört, dass jedes C-Programm eine oder mehrere Funktionen enthält. Innerhalb einer Funktion werden Anweisungen (Befehle) ausgeführt. Jedes C-Programm enthält die Funktion *main*, so auch obiges Miniprogramm. Jedes C-Programm beginnt in der ersten Zeile der Funktion *main* mit seiner Ausführung.
- Eine Funktion ist eine Programmeinheit, die ein oder mehrere Anweisungen (Befehle) enthält. Typischerweise endet jede Anweisung mit einem Semikolon. Unsere Funktion *main* enthält zwei Anweisungen.
- Eine Funktion besteht aus einer Funktionsdefinition, gefolgt von der Angabe der Anweisungen. Es gilt dabei immer folgende Syntax zu beachten:

Ergebnistyp Funktionsname (Funktionsparameter) { Anweisungen }

- Die hier verwendeten Klammern sind Bestandteil der Sprache C! In unserem Beispiel gilt:

  Ergebnistyp: ***int*** (int bedeutet, dass die Funktion eine Ganzzahl zurückgibt)
  Funktionsname: ***main***
  Funktionsparameter: keine, daher einfach weggelassen
  Anweisungen: zwei

- Der Befehl ***printf*** ist eine vordefinierte Ausgabefunktion. Die in Klammern stehenden Angaben werden auf die Konsole (den Bildschirm) ausgegeben.
- Ein in Gänsefüßchen eingeschlossener Text ist eine sogenannte ***konstante Zeichenkette***. In unserem Beispiel ist es die Zeichenkette "*Hello world\n*". Eine Zeichenkette ist ein Text, der in einem Programm verarbeitet werden kann. Das Zeichen "*n*" ist eines von mehreren Spezialzeichen und bedeutet Zeilenvorschub. Die Ausgabe wird hier also nach der Ausgabe noch in die nächste Zeile wechseln.
- Die **Return**-Anweisung beendet eine Funktion. Der hier anschließend angegebene Wert 0 bedeutet, dass die Zahl 0 als Funktionsergebnis zurückgegeben wird. Eine Return-Anweisung in der Funktion *main* beendet nicht nur die Funktion, sondern gleichzeitig das gesamte Programm.
- Ein C-Programm kennt von sich aus keine Funktionen, auch nicht die Ausgabefunktion *printf*. Die in der Sprache C vordefinierten Funktionen werden in Bibliotheken verwaltet. Mit Hilfe einer Include-Anweisung (*#include*) muss daher zunächst die Bibliothek, in der diese Funktion definiert ist, bekannt gegeben werden. Die wichtigsten Ein- und Ausgabefunktionen sind in der Bibliothek *stdio.h* definiert. Aus diesem Grund wird die entsprechende Include-Anweisung verwendet. Der Bibliotheksname *stdio.h* ist dabei in spitze Klammern zu setzen!
- Kommentare werden in /* … */ eingeschlossen. In C++ und in der ISO-Norm C99 dienen auch zwei Schrägstriche (‚//‘) als Kommentarbeginn. Letztere Kommentare enden automatisch am Zeilenende und benötigen daher kein explizites Kommentarende. Ein Kommentar wurde in diesem Programm in der ersten Zeile verwendet. Bitte geizen Sie nicht mit Kommentaren. Nur so werden Programme wirklich lesbarer.
- C erlaubt einen sehr freien Aufbau. Einrücken des Programmtextes oder Zeilenvorschübe sind überall erlaubt (natürlich nicht innerhalb von Bezeichnern und Namen!). Wo ein Leerzeichen erlaubt ist, dürfen beliebig viele stehen, ebenso Tabulatoren und Zeilenvorschübe. Ein Programm sollte so geschrieben sein, dass es übersichtlich und damit leicht lesbar ist.
- C unterscheidet zwischen Groß- und Kleinschreibung! Wird *Main* statt *main* geschrieben, so liegt ein Syntaxfehler vor.

Bitte beachten Sie vom ersten Tag an, dass ein Programm lesbar geschrieben wird! Auch in der Prüfung wird darauf Wert gelegt! Folgendes Programm leistet exakt das Gleiche wie unser erstes Programm. Natürlich können wir es noch lesen, aber sobald Programme umfangreicher werden, haben wir mit so einem Programmstil verloren. Wir werden dies bald sehen.

```
#include <stdio.h>
int main(){printf("Hello world\n");return 0;}
```

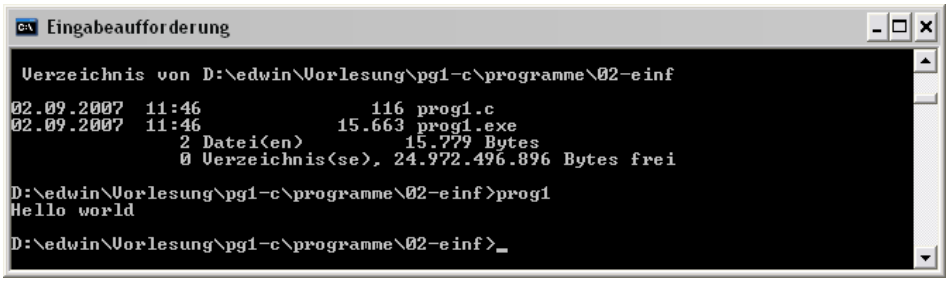
Programm 02-einf\prog2.c

Jetzt wollen wir natürlich unsere erste Gesellenleistung auch bewundern. Dies ist nun gar nicht so einfach. Ein C-Programm erfordert einen **C-Compiler**. Ein Compiler überprüft den gesamten geschriebenen Code auf Syntax-Fehler und übersetzt bei Fehlerfreiheit dieses Programm in eine maschinenlesbare Version. Dieser Maschinencode hat für sich allein noch wenig Wert. Erst ein **Linker** verbindet diesen Code mit dem Code der verwendeten Bibliotheken und einem Laufzeitsystem. Der Linker erzeugt letztlich ein ausführbares Programm. Unter Windows sind diese Programme an der Endung *.exe* erkennbar.

Arbeiten wir mit der IDE von Bloodshed, so wollen wir dort natürlich unser erstes Programm auch direkt starten. Tatsächlich wird durch Klick auf ein Icon links oben das Programm übersetzt (mit dem Compiler), gebunden (mit dem Linker) und dann sofort ausgeführt. Doch wir erkennen nur, das ein schwarzes Konsolenfenster geöffnet und sofort wieder geschlossen wird. Die IDE von Bloodshed hat unser Programm übersetzt und eine ausführbare Datei namens

prog1.exe

erzeugt. Anschließend wurde ein Konsolenfenster geöffnet, und dieses Programm ausgeführt. Nach der Beendigung des Programms wird allerdings das Konsolenfenster sofort wieder geschlossen. Dumm nur, dass wir mit unseren Augen da nicht mitkommen. Wir können dies aber zu Fuß nachholen. Wir öffnen selbst ein Konsolenfenster und wechseln in das Verzeichnis, wo sich unser Programm befindet. Wir finden jetzt neben der Datei *prog1.c* auch eine Datei *prog1.exe* vor. Wir geben nun *prog1* ein. Das Programm wird wie gewünscht ausgeführt:



Ausführung von 02-einf\prog1.c im Konsolenfenster

Hier wird übrigens deutlich die Wirkung von ‚\n' sichtbar, eine Leerzeile wurde eingefügt!

Schöner wäre es, wenn wir unser Ergebnis direkt mit der IDE bewundern könnten. Dies ist natürlich möglich. Verwenden wir vor der Return-Anweisung die Funktion *system* aus der Bibliothek *stdlib.h*, so bleibt das Programm vor der Beendigung stehen und wartet noch auf eine Tastatureingabe:

```
/* Unser drittes Programm */
#include <stdio.h>
#include <stdlib.h>
int main ()
{
   printf("Hello world\n");
   system("pause");
   return 0;
}
```

Programm 02-einf\prog3.c

Noch ein wichtiger Hinweis: C ist eine Sprache, die Disziplin erfordert. Für den Anfänger drohen fast überall Fallen. Aus diesem Grund wird dringendst empfohlen, Programme in Dev-C++ nur mit folgenden Compileroptionen zu übersetzen:

-ansi -Wall -Werror

Diese Optionen sind im Menüpunkt Werkzeuge unter Compileroptionen einzutragen. Dies ist keine Schikane, sondern ein Muss beim Programmierenlernen.

2.4 Allgemeine Informationen zu C

C wurde von Kernighan und Ritchie in den Bell Laboratories um 1972 entwickelt, und zwar als Systemprogrammiersprache für UNIX. Zunächst ausschließlich unter UNIX verwendet, fand C ab etwa 1980

eine immer größere Verbreitung. C wurde 1989 normiert (ISO-Norm C89) und 1999 nochmals leicht überarbeitet (ISO-Norm C99). Diese Norm ist umfangreich genug, um praktisch allen praktischen Anforderungen zu genügen. Diese ISO-Norm wird heute auf praktisch allen Plattformen eingesetzt. Jedes fehlerfreie C Programm kann daher jederzeit zwischen UNIX, Windows oder Mac ausgetauscht werden.

Ab etwa 1980 entwickelte Bjarne Stroustrup in den Bell Laboratories die Sprache C++ als eine Erweiterung der Sprache C um objektorientierte Komponenten. Einen ersten Abschluss fand diese Entwicklung erst Anfang 1998 mit der Verabschiedung der ISO-C++ Norm.

Es sei empfohlen, etwa in **Wikipedia** ein wenig zu C und C++ nachzulesen. Wir finden dort zusätzliche Informationen nicht nur zur Geschichte dieser Programmiersprachen.

Heute gibt es kaum noch reine C-Compiler. Die meisten Compiler übersetzen auch C++-Programme mühelos. C++ bietet neben der Objektorientierung einige zusätzliche Erweiterungen an. Wir werden diese bald einsetzen, um uns von altem und nicht einfach zu verstehendem C-Ballast zu befreien.

Die Vorlesung basiert also auf C, genauer auf C89. Auf verwendete Erweiterungen in C99 und C++ wird explizit hingewiesen. Auf C99 wird deshalb nicht aufgesetzt, da sich C99 und C++ in einigen wenigen Punkten widersprechen, und C99 von den meisten Compilern daher nicht komplett unterstützt wird.

Einige weit verbreitete 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?)

Um einige dieser Aussagen zu untermauern, folgt ein abschreckendes Programm-Beispiel. Dieses gibt alle Lösungen des sogenannten n-Damen-Problems aus. Dieses Problem lautet: Wie viele Möglichkeiten gibt es, auf einem nxn-Feld n Damen (aus dem Schachspiel) so zu positionieren, dass diese sich nicht gegenseitig schlagen können. Dieses Programm hat sogar einen Preis gewonnen. Es war bei der Ausschreibung allerdings gefordert worden, ein interessantes Programm abzuliefern, das gleichzeitig so kurz wie möglich sein sollte. Es ist fast unglaublich, dass dieses völlig unverständliche Wirrwarr an Zeichen tatsächlich ein hochkomplexes Problem löst. Doch sehen wir selbst:

```
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])
       ;
}

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

Programm 02-einf\dame.c

Gerne kann jemand versuchen, das Programm so darzustellen, dass es lesbar wird. Es wird mit Sicherheit Tage dauern, viel Erfahrung mit C vorausgesetzt!

2.5 Funktionen: ein kleiner Überblick

Wie wir schon wissen, enthält ein Programm Anweisungen. Einige Anweisungen sind Funktionsaufrufe, so beispielsweise die Funktionen *printf* oder *system*. Obwohl es über hundert vordefinierte Funktionen gibt, benötigen wir zum übersichtlichen Programmieren auch selbst definierte Funktionen. Fangen wir doch gleich damit an.

Frage: Was ist also eine Funktion?
Antwort: Das Zusammenfassen mehrerer Anweisungen zu einer Einheit.

Betrachten wir etwa ein praktisches Beispiel: Um nicht ziellos in den Tag hineinzuleben, machen wir einen Tagesplan. Dieser könnte ansatzweise so aussehen:

- Einkaufen gehen
- IKEA-Regal zusammenbauen und aufstellen
- Mittagessen kochen
- Übungsaufgaben zu PG1 bearbeiten

Jede dieser Aufgaben wäre im Sinne der Programmierung eine Funktion, die aus Einzelaufgaben besteht. Beispielsweise müssen wir für die zweite Aufgabe die Anleitung lesen, Hammer und Schraubenzieher holen und dann nach Anleitung Schritt für Schritt das Regal montieren. In Summe sind dies recht viele Einzelschritte. Diese Einzelschritte nennen wir in der Programmierung *Anweisungen*.

Natürlich könnten wir alle Einzelschritte auch auf unseren Merkzettel schreiben. Da kommen dann gut und gerne statt obiger vier vielleicht vierzig oder mehr Aufgaben drauf. Jetzt wird es aber unübersichtlich! Und die Zusammenhänge gehen auch verloren. Genau so ist es beim Programmieren! Wir verwenden daher von Anfang an Funktionen. Unser Hauptprogramm *main* enthält daher in Zukunft nur noch die wichtigen Schritte, also meist Funktionsaufrufe.

Die wichtigste Funktion in einem Programm, die Funktion *main* haben wir ja schon kennengelernt. Alle anderen Funktionen sind genauso aufgebaut. Hier nochmals die Definition:

Definition:

Eine ***Funktionsdefinition*** beschreibt exakt die durchzuführenden Schritte innerhalb der Funktion. Für die Syntax gilt:

Ergebnistyp Funktionsname (Funktionsparameter) { Anweisungen }

Jetzt sind wir allerdings nicht mehr auf den Namen *main* fixiert. Wir können den Funktionsnamen relativ frei wählen. In der Funktion *main* haben wir den Ergebnistyp *int* kennengelernt. Nicht jede Funktion muss ein Ergebnis zurückliefern. Hat eine Funktion keinen Ergebniswert, so schreiben wir als Ergebnistyp ***void***, auf deutsch *leer*, also kein Ergebniswert. Funktionsparameter werden wir etwas später behandeln. Jetzt müssen wir nur noch ein paar Anweisungen (Befehle) innerhalb der geschweiften Klammern schreiben.

Wir wissen auch schon, wie eine Funktion aufgerufen wird. Wir kennen ja schon die Aufrufe *printf* und *system*. Schreiben wir eine Funktion ohne Funktionsparameter, so müssen wir beim Aufruf dieser Funktion nur noch ein Klammerpaar schreiben. Eine Funktion ohne Rückgabewert und ohne Parameter wird also wie folgt aufgerufen:

Funktionsname ( ) ;

Unser Hauptprogramm soll jetzt zwei Sternenbäume ausgeben, einmal mit vier und einmal mit sechs Sternenzeilen. Das Ergebnis sollte also etwa wie folgt aussehen:

Ausführung von 02-einf\sternbaum.c

Wir benennen unsere zwei Funktionen mit *zeichneSternenbaum4* und *zeichneSternenbaum6* und können unser Hauptprogramm schon formulieren:

```
int main()
{
   printf("Vorlesung PG1: Ausgabe von Sternenbaeumen\n\n");
   printf("Sternenbaum mit vier Zeilen:\n");
   zeichneSternenbaum4();
   printf("Sternenbaum mit sechs Zeilen:\n");
   zeichneSternenbaum6();
   system("Pause");
   return 0;
}
```

Funktion *main* in 02-einf\sternbaum.c

Dieses Hauptprogramm ist kompakt und übersichtlich. Im Wesentlichen sind wirklich nur die „Hauptaufgaben“ formuliert. Damit dieses Programm vom Compiler übersetzt werden kann, benötigen wir neben den #include-Einträgen noch die beiden selbstdefinierten Funktionen, beginnen wir mit der ersten:

```
void zeichneSternenbaum4()
{
   printf("          *\n");
   printf("         ***\n");
   printf("        *****\n");
   printf("       *******\n");
}
```

Funktion *zeichneSternenbaum4* in 02-einf\sternbaum.c

Da war nun wirklich nichts dabei. Also schreiben wir gleich noch die Funktion *zeichneSternenbaum6*, indem wir einfach noch zwei Zeilen hinzufügen. Stopp! Halt! Natürlich könnten wir durch Paste and Copy diesen Code kopieren, die Zahl vier durch die Zahl sechs ersetzen und die zwei Zeilen hinzufügen. Davon rate ich ab. Warum sollten wir nicht bereits geschriebenen Code weiterverwenden? Sehen wir uns die elegantere Lösung an:

```
void zeichneSternenbaum6()
{
   zeichneSternenbaum4();
   printf("      *********\n");
   printf("     ***********\n");
}
```

Funktion *zeichneSternenbaum6* in 02-einf\sternbaum.c

Wir rufen einfach in der Funktion *zeichneSternenbaum6* die Funktion *zeichneSternenbaum4* nochmals auf. Jetzt erkennen wir einen weiteren Vorteil von Funktionen: Wir können sie jederzeit auch mehrfach verwenden. Das spart viele Tipparbeiten und erhöht die Übersicht wesentlich. Wir müssen jetzt nur noch

eine Frage beantworten, damit unser Programm komplett wird.

Frage: Wo müssen unsere beiden Funktionen im Programm platziert werden, vor oder nach der Funktion *main*?

Antwort: Eine Funktion muss vor dem ersten Benutzen bekannt sein. Wir müssen die beiden Funktionen also vor der Funktion *main* schreiben.

In der Praxis befriedigt diese Antwort nicht. Die wichtigste Funktion eines Programms steht damit quasi am Ende des Programms. Eigentlich sollte die Funktion *main* ganz am Anfang stehen! Dies ist glücklicherweise auch möglich, indem wir die Funktionen zu Beginn bekannt geben. Dies nennen wir eine **Deklaration einer Funktion**. Das Schreiben der eigentlichen Funktion heißt dagegen **Implementierung**. Eine Funktion deklarieren wir, indem wir die Funktion ohne die darin enthaltenen Anweisungen angeben, also direkt nach dem Schließen der runden Klammer ein Semikolon schreiben. Unser Programmstart könnte daher wie folgt aussehen:

```
/* Ausgabe von Sternenbaeumen */
#include <stdio.h>
#include <stdlib.h>
void zeichneSternenbaum4();        /* nur Deklaration */
void zeichneSternenbaum6();        /* nur Deklaration */
```

Programmstart in 02-einf\sternbaum.c

Damit gewinnt das Programm weiter an Übersicht. Alle benötigten Bibliotheken und Funktionen sind gleich zu Beginn des Programms übersichtlich angegeben. Die Reihenfolge der dann definierten Funktionen inklusive der Funktion *main* spielt nun keine Rolle mehr. Eine kleine Fehlerfalle ist damit ebenfalls beseitigt.

Vielleicht ist bereits aufgefallen, dass wir die Funktion *main* mit der Anweisung *return 0* beenden, unsere Sternenbaumfunktionen hingegen nicht. Der Unterschied ist, dass die Funktion *main* als Ergebnis einen Int-Wert zurückliefert, den das Programm auch an das Betriebssystem weitergibt! So können wir beispielsweise im Fehlerfall einen Wert ungleich Null zurückliefern. Die Funktion *main* muss also einen Rückgabewert angeben, zumindest wenn wir ein korrektes Programm wünschen. Die Return-Anweisung ist in der Funktion *main* also erforderlich! Dieser Rückgabewert kann vom Betriebssystem tatsächlich überprüft werden.

Unsere Sternenbaum-Funktionen hingegen sind Void-Funktionen, die keinen Rückgabewert liefern. Wir könnten als letzte Anweisung schreiben:

```
return;
```

Zu beachten ist, dass hier keine Zahl 0 steht, wir geben ja nichts zurück. Die Return-Anweisung bewirkt immer, dass die Funktion sofort beendet wird. Am Ende einer Funktion, also vor dem Schließen der geschweiften Klammer, wird eine Funktion aber automatisch beendet. Wir können uns diese Anweisung daher schenken.

2.6 Bezeichner und Variablen

Bis jetzt haben wir einige Beispiele vorgestellt, ohne beispielsweise genau zu erklären, welche Funktionsnamen überhaupt erlaubt sind. Programme gehorchen einer exakten Syntax. Wir müssen daher langsam ebenfalls exakt werden. Wir wollen gleich damit beginnen:

Starten wir mit den sogenannten **Bezeichnern**. Ein Bezeichner ist ein Name, etwa der Name einer Funktion wie *sternenbaum4* oder *printf* oder der Name eines vordefinierten Befehls wie *return* oder der Name eines Datentyps wie *int*. Natürlich gehören dazu auch der Name von Variablen oder Kontroll-

strukturen. Für Bezeichner gilt in C:

Definition:

Ein ***Bezeichner*** besteht aus ein oder mehreren Zeichen, wobei nur Buchstaben, Ziffern und das Unterstrichzeichen (‚_') erlaubt sind. Ein Bezeichner darf nicht mit einer Ziffer beginnen. Groß- und Kleinbuchstaben werden voneinander unterschieden. Alle anderen Zeichen, etwa ein Leerzeichen, trennen diesen Bezeichner von anderen Programmteilen ab.

Eine wichtige Rolle spielen in Programmen Leerzeichen. Sie dienen der Übersicht und auch als Trennzeichen zwischen den einzelnen Bezeichnern eines Programms. Die gleiche Funktion wie Leerzeichen erfüllen auch Leerzeilen oder Tabulatoren. Diese Zeichen werden zusammengenommen als *White-Spaces* bezeichnet.

Definition:

Ein ***White-Space*** ist entweder ein Leerzeichen, ein horizontaler Tabulatorzeichen (\t), ein vertikaler Tabulator (\v), ein Neue-Zeile-Zeichen (\n) oder ein Wagenrücklauf (\r).

Generell dürfen in jedem Programm an jeder beliebigen Stelle beliebig viele White-Spaces eingefügt werden, nur nicht innerhalb von Bezeichnern. Damit ist der gestalterische Aufbau eines Programms bereits erklärt. Unsere Programme enthalten alle Include-Anweisungen. Diese speziellen Anweisungen sind Präprozessoranweisungen und beginnen mit einem Nummernzeichen (#). In C-Programmen werden vor allem zwei dieser Präprozessoranweisungen verwendent.

Definition:

Eine ***Präprozessor-Anweisung*** ist eine generelle Anweisung an ein Programm, die noch vor dem Compilieren ausgeführt wird. Präprozessoranweisungen beginnen mit einem Nummernzeichen (#) und enden in der Regel mit dem Ende der Zeile. Diese Präprozessoranweisungen wirken ab der Zeile ihres Vorkommens. Die beiden wichtigsten sind:

#include	(fügt meist Bibliotheksdeklarationen hinzu)
#define	(definiert Konstanten und sogenannte Makros)

In C finden beide Präprozessor-Anweisungen Anwendung. In C++ kann die Define-Anweisung praktisch vollständig durch modernere Konstrukte ersetzt werden. Zu beachten ist, dass am Ende einer Präprozessoranweisung kein Semikolon steht!

Ein übersichtliches Programm enthält viele Kommentare. Es gilt:

Definition:

Ein ***Kommentar*** beginnt mit der Zeichenfolge ‚/*' und endet mit der Zeichenfolge ‚*/'. Ein Kommentar darf sich über mehrere Zeilen erstrecken. Alle innerhalb dieser Kommentarklammern enthaltenen Angaben werden vom Compiler grundsätzlich überlesen.

Zusätzlich gilt in **C++** und der ISO-Norm **C99**: Ein Kommentar beginnt mit der Zeichenfolge ‚//' und endet automatisch am Ende der gleichen Zeile.

Das Salz in der Suppe ist in Programmiersprachen die **Variable**. Eine Variable ist hier der Name eines Speicherplatzes. In diesem Speicherplatz wird der Wert einer Variable hinterlegt. Allerdings muss der Compiler wissen, welche Art von Variable gespeichert werden soll, etwa ein Ganzzahl oder eine Gleitpunktzahl.

Definition:

Eine ***Variable*** ist der Name eines Speicherplatzes. Jede Variable muss vor ihrer Verwendung de-

klariert werden. In der Deklaration wird einer Variable ein fester ***Datentyp*** zugewiesen. Bei jeder Verwendung einer Variable im Programm wird der Variablenname durch den gespeicherten Wert in diesem Speicherplatz ersetzt.

Die drei wichtigsten Grunddatentypen sind:

- **int** eine mindestens 32 Bit große Ganzzahl, z.B. 1017
- **float** eine meist 32 Bit große Gleitpunktzahl, z.B. 3.14159
- **char** ein einzelnes Zeichen, z.B. 'A' (die Hochkomma gehören dazu!)

In der Mathematik werden Variablen meist mit **Operatoren** eingesetzt. In C und C++ gibt es über 50 verschiedene Operatoren. Wir wollen zunächst nur ein paar bekannte Operatoren vorstellen:

- **Zuweisungsoperator:** **=**
- **Addition:** **+**
- **Subtraktion:** **-**
- **Multiplikation:** ***
- **Division:** /

Die arithmetischen Operatoren verhalten sich wie in der Mathematik gewohnt: Punkt geht vor Strich. Die Multiplikation und Division binden also stärker als Addition und Subtraktion. Der Zuweisungsoperator bewirkt, dass einer Variable ein Wert zugewiesen wird. Beispiel:

```
x = 17 + 5;
```

Der Compiler berechnet zunächst die Summe (22) und speichert diesen Wert an der für die Variable *x* reservierten Platz. Dieser Wert kann jederzeit wieder abgerufen werden. Soll etwa in der Variable *y* ein um 5 höherer Wert als in *x* gespeichert werden, so könnten wir dies wie folgt erreichen:

```
y = x + 5;
```

Wir haben eine neue Anweisung kennengelernt: die Ausführung einer Zuweisung. Es ist wieder zu beachten, dass eine Anweisung mit einem Semikolon endet.

Definition:

Eine ***Zuweisung*** hat folgendes Format:

Variable = Ausdruck

Rechts vom Gleichheitszeichen steht ein Ausdruck. Dies kann eine Konstante sein (z.B. 13), eine Variable (z.B. *x*) oder ein beliebig komplexer Ausdruck (z.B. x*y+z). Selbstverständlich sind in einem Ausdruck auch Klammern zulässig. Der Compiler berechnet diesen Ausdruck und weist das Ergebnis der Variablen links vom Gleichheitszeichen zu.

Zuletzt betrachten wir in diesem Abschnitt ein kleines Beispiel, in dem wir die Verwendung von Variablen demonstrieren. Dieses Programm ist schon etwas umfangreicher. Also teilen wir es in drei Teile, also drei Funktionen, auf:

```
void bearbeitenZeichen()
{
   /* die Deklarationen stehen vor den Befehlen! */
   char zeichen;   /* ein einzelnes Zeichen */
   /* Einfache Wertzuweisung: */
   zeichen = 'A';
   printf("Die Variable zeichen enthaelt: %c\n", zeichen);
   zeichen = 'z';
   printf("zeichen enthaelt jetzt: %c\n", zeichen);
}
```

Funktion *bearbeitenZeichen* in 02-einf\variable.c

Die Funktion *bearbeitenZeichen* zeigt uns den weiteren Aufbau von Funktionen. In der Regel stehen die in der Funktion verwendeten Variablendeklarationen zu Beginn der Funktion. Allerdings erlauben die C99-Norm und C++ Variablendeklarationen auch an beliebiger Stelle innerhalb einer Funktion. Anschließend folgen die einzelnen Anweisungen, die jeweils mit Semikolon abgeschlossen werden.

Definition:

Die ***Deklaration einer Variable*** hat folgenden Aufbau:

Datentyp Variablenname ;

Wir haben hier als Datentyp den Typ *char* gewählt. Damit wird für die Variable *zeichen* ein entsprechend großer Speicherplatz reserviert (hier: 1 Byte). Der Compiler wird weiter darauf achten, dass alle Zuweisungen und andere Operationen so erfolgen, dass sie mit diesem Datentyp *char* kompatibel sind. Eine Zuweisung der Form

```
zeichen = 1;        /* Fehler!!! */
```

wird der Compiler zwar akzeptieren, doch mit Sicherheit wird nicht die Zahl 1 im Speicher abgelegt! Vielmehr wird die Zahl 1 in ein Zeichen mit dem Code 1 umgewandelt. Dies kann gerne ausprobiert werden. Zeichen werden grundsätzlich in Hochkomma geschrieben. Zeichen dürfen nicht mit Zeichenketten verwechselt werden, die in Gänsefüßchen gesetzt werden. Die Unterschiede werden wir noch kennen lernen.

In der Funktion *bearbeitenZeichen* geben wir Variablen auch aus. Dies ist mit der Funktion *printf* möglich, erfordert jedoch neues Wissen. Wir geben den gewünschten Text aus und schreiben für die Variablen Platzhalter. Die vier wichtigsten Platzhalter sind:

%c	Platzhalter zur Ausgabe eines Zeichens vom Typ char
%d	Platzhalter zur Ausgabe einer Ganzzahl vom Typ int
%f	Platzhalter zur Ausgabe einer Gleitpunktzahl vom Typ float (und double)
%s	Platzhalter zur Ausgabe einer Zeichenkette

Die tatsächlichen Variablen werden dann nach dem Text, durch Komma getrennt, aufgelistet. Kommen wir gleich zu unserer nächsten Funktion:

```
void bearbeitenZahlen()
{
   int zahl;       /* Deklaration einer Ganzzahl */
   int i = 5;      /* Deklaration mit Wertzuweisung */
   int j, k = 0;   /* mehrere Deklarationen gleichzeitig */
   printf("Bitte geben Sie eine Ganzzahl ein: ");
   scanf("%d", &zahl);   /* Einlesen in die Variable zahl */
   j = zahl * zahl;      /* Quadrieren und speichern in j */
   k = j + i;            /* Addieren */
   printf("Quadriert und addiert mit 5 ergibt: %d\n", k);
}
```

Funktion *bearbeitenZahlen* in 02-einf\variable.c

In der Funktion *bearbeitenZahlen* haben wir natürlich wieder die Deklaration zuerst geschrieben. Hier zeigen wir weitere Möglichkeiten bei der Deklaration auf. Beispielsweise hat unsere Variable *zahl* zwar einen Speicherplatz, aber was der Compiler dort zufälligerweise abgelegt hat, wissen wir nicht. Leicht vergessen wir, der Variable dann einen Wert zuzuweisen. Wird dann die Variable vor einer vorherigen Zuweisung verwendet, so ist das Ergebnis rein zufällig. Um dies zu vermeiden, kann einer Variable gleich bei der Deklaration ein Wert zugewiesen werden. Die Variable *i* erhält beispielsweise gleich den Wert 5. In der Folgezeile werden zwei Variablen auf einmal deklariert, *j* und *k*. So können auch drei oder mehr Variablen deklariert werden. Zwischen den einzelnen Variablen muss nur ein Komma stehen. Hier wurde zusätzlich die Variable *k* mit der Zahl 0 vorbelegt.

Datenverarbeitung heißt, dass Daten eingelesen, verarbeitet und wieder ausgegeben werden. Wir wollen daher nun auch Daten einlesen. Das Gegenstück zur Funktion *printf* ist die Funktion ***scanf***. Als erster Parameter wird eine Zeichenkette mit mindestens einem Platzhalter (%c, %d, %f oder %s) erwartet, als zweiter Parameter folgt die Variable, in die ein Wert eingelesen werden soll. Dieser Variable ist der sogenannte Adressoperator (‚&') zwingend voranzustellen!

Achtung:

➔ Häufiger Fehler: Das Zeichen **‚&'** wird in der Funktion ***scanf*** vergessen!

Wir lesen also in die Variable *zahl* eine Ganzzahl ein, quadrieren diese und speichern das Ergebnis in der Variable *j* ab, addieren dann zu *j* den Wert von *i* dazu, speichern dies in *k* und geben schließlich *k* aus. Da sich der Compiler Zwischenergebnisse intern merkt, wären wir auch mit weniger Variablen ausgekommen. Wir hätten auf *j* und *k* verzichten und kurz schreiben können:

```
zahl = zahl * zahl + i;
```

Wir müssen beachten, dass das Gleichheitszeichen in C eine Zuweisung ist. Zunächst wird alles rechts von diesem Zeichen berechnet und intern zwischengespeichert, und dann wird erst das Ergebnis der Variable *zahl* zugewiesen! Natürlich muss jetzt die Variable *zahl* und nicht *k* ausgegeben werden, damit das Programm korrekt funktioniert. Kommen wir nun zur Bearbeitung von Gleitpunktzahlen:

```
void bearbeitenGleitkomma()
{
   float x;             /* Gleitpunktzahl */
   float pi = 3.14159;  /* Gleitpunktzahl mit Zuweisung */

   /* Gleitpunktzahlen: */
   printf("Bitte geben Sie eine Gleitpunktzahl ein: ");
   scanf("%f", &x);     /* Einlesen einer Gleitpunktzahl */
   printf("Kreisflaeche mit Radius %f ist %f\n", x, x*x*pi);
}
```

Funktion *bearbeitenGleitkomma* in 02-einf\variable.c

Zu beachten ist, dass ein C-Programm immer penibel zwischen Gleitpunktzahlen und Ganzzahlen un-

terscheidet. Eine Ganzzahl kann grundsätzlich nur ganzzahlige Werte aufnehmen, eine Gleitpunktzahl sowohl Ganzzahlen als auch Zahlen mit Nachkommastellen. Zu beachten ist allerdings die Rundungsgenauigkeit. Eine Float-Zahl rechnet nur mit etwa 6 Stellen!

Die Funktion *bearbeitenGleitkomma* enthält wenig neue Inhalte. Interessant ist hier nur, dass in der Funktion *printf* nicht nur Variablen, sondern ganze Ausdrücke übergeben werden dürfen. Es fällt auf, dass Gleitpunktzahlen standardmäßig immer mit 6 Nachkommastellen ausgegeben werden. Dies liegt am Platzhalter *%f* und Standardvorgaben. Übrigens sind die Ausgaben völlig falsch, wenn falsche Platzhalter verwendet werden.

Achtung:

➔ Häufiger Fehler: In der Funktion ***printf*** wird der falsche Platzhalter verwendet.

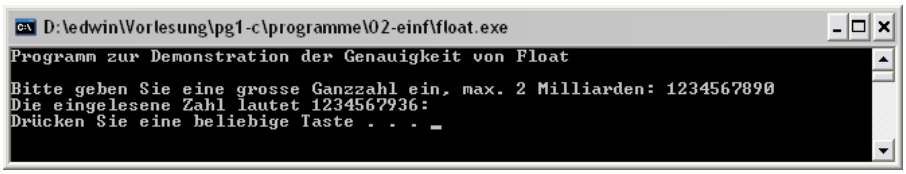
Das Hauptprogramm selbst braucht hier nicht mehr wiedergegeben zu werden. Es enthält einen kurzen Begrüßungstext, den Aufruf der drei obigen Funktionen, eine Systempause und einen Return-Befehl.

Zur Demonstration der Genauigkeit von Float-Variablen soll ein weiteres Programm dienen. Hier lesen wir eine Ganzzahl ein, wandeln diese in eine Float-Variable um und dann gleich wieder zurück in eine zweite Ganzzahl. Schließlich geben wir die zweite Ganzzahl aus. Lesen wir sehr große Werte im Bereich von einer Milliarde ein, so werden wir eine Überraschung erleben: Die eingelesene und die ausgegebene Zahl stimmen nicht überein! Übrigens ist es eine Besonderheit in C, dass Zuweisungen von einem Zahlentyp zum anderen prompt und automatisch durchgeführt werden. Leider treten dabei oft Nebeneffekte auf, wenn die „größere“ Zahl nicht in die „kleinere“ passt. C schneidet dann meist radikal ab, das Ergebnis ist undefiniert. Doch kommen wir zu unserem kleinen Programm zurück:

```
void bearbeitenFloat()
{
   float x;                /* Gleitpunktzahl */
   int zahl1, zahl2;       /* Ganzzahlen */
   printf("Grosse Ganzzahl eingeben, max. 2 Milliarden: ");
   scanf("%d", &zahl1);
   x = zahl1;       /* Ganzzahl einer Float-Zahl zuweisen */
   zahl2 = x;       /* Float-Zahl einer Ganzzahl zuweisen */
   printf("Die eingelesene Zahl lautet %d:\n", zahl2);
}
```

Funktion *bearbeitenFloat* in 02-einf\float.c

Dieses harmlos wirkende Programm hat eine Tücke. Bei großen Zahlen ist das Ergebnis nicht gleich der Eingabe. Zu beachten ist, dass keine Ganzzahl größer als 2 Milliarden (genauer: 2.147.483.647) eingegeben wird, da eine Int-Zahl größere Werte nicht aufnehmen kann. Zur Demonstration des Problems lassen wir das Programm einfach mal laufen und sehen, dass aber der 8. Ziffer Unterschiede auftreten:



Ausführung von *02-einf\float.c* im Konsolenfenster

Die Programme werden langsam länger. Eine Variable *zahl* mutiert schnell zur Variable *Zahl*. Diese beiden Variablen sind in C aber völlig verschiedene Bezeichner. Das kann ins Auge gehen.

Achtung:

➔ Häufiger Fehler: Die Groß- und Kleinschreibung wird nicht durchgängig beachtet.

Aus diesem Grund wurden früher meist alle Variablen und Funktionsnamen klein geschrieben. Dies

diente nicht der Übersichtlichkeit. Daher wurden meist noch Unterstriche verwendet. In den letzten Jahren haben sich insbesondere in Java und C# neue Schreibkonventionen durchgesetzt. Leider sind diese nicht hundertprozentig auf C anwendbar. Ich schlage daher folgenden Kompromiss für diese Vorlesung vor:

Konvention

- **Lokale Variablen** und **Parameter** einer Funktion werden immer klein geschrieben.
- In **globalen Variablen** wird der erste Buchstabe groß geschrieben
- **Funktionsnamen** werden klein geschrieben, Teilnamen beginnen allerdings mit Großbuchstaben
- **Konstanten** werden komplett groß geschrieben
- **Funktionen** ohne Rückgabewert (void-Funktionen) beginnen mit einem Verb

Wir haben uns bisher bereits an diese Konventionen gehalten. Wir wissen noch nicht, was globale Variablen, Konstanten und Parameter sind, doch dies wird bald nachgeholt. Natürlich halten wir uns an weitere Konventionen, etwa dass wir alle Funktionen zu Beginn deklarieren.

Zuletzt wollen wir noch weitere Standarddatentypen kennen lernen. Die fast vollzählige Liste lautet:

```
int i, j;           // meist 32 Bit Ganzzahl
short k;            // 16 Bit Ganzzahl
long l;             // >= 32 Bit Ganzzahl
long long ll;       // >= 64 Bit Ganzzahl, nur in C99/C++ verfügbar
                    // "short <= int <= long <= long long"
float x, y;         // einfache Gleitpunktzahl
double z;           // lange Gleitpunktzahl (8 Byte)
long double w;      // sehr lange Gleitpunktzahl
char ch;            // Zeichen: 1 Byte Zahl
bool b;             // Boolesche Variable, nur in C++ verfügbar
_Bool b2;           // Boolesche Variable, nur in C99 verfügbar
```

Ganzzahlen können auch als vorzeichenlose Zahlen definiert werden. Dazu ist der Bezeichner ***unsigned*** voranzustellen.

```
short i;                Bereich zwischen -32768 und +32767
unsigned short i;       Bereich zwischen 0 und +65535
```

Den Datentyp *bool* gibt es nur in C++. Eine boolesche Variable kann nur die beiden Wahrheitswerte *true* und *false* annehmen, etwa

```
bool b = true;
```

2.7 Funktionen mit Parametern

Wir wollen nun die Möglichkeiten von Funktionen erweitern. Unter Funktionen stellen wir uns meist mathematische Funktionen vor, etwa *sin(x)*. Diese Sinus-Funktion besitzt einen Parameter. In Programmiersprachen können Funktionen keinen, einen oder mehrere Parameter besitzen. Wiederholen wir zur Erinnerung die Definition einer Funktion:

Ergebnistyp Funktionsname (Funktionsparameter) { Anweisungen }

Wir legen nun unser Augenmerk auf die Funktionsparameter. Durch die Angabe von Funktionsparametern können wir einer Funktion auch Parameter übergeben, wie etwa den Wert *x* in obiger Sinus-Funk-

tion. Die Definition der **Funktionsparameter** lautet:

Datentyp Parametername [, …]

Wir geben also einen Datentyp, z.B. int, an, gefolgt von einem Namen, dem Parameter. Die eckigen Klammern haben in der Syntax immer die Bedeutung, dass der Inhalt wahlfrei ist, also nicht angegeben werden muss. Die drei Punkte wiederum geben an, dass das Vorhergehende wiederholt wird. Dies bedeutet hier also, dass weitere Parameter angegeben werden können, jeweils durch Komma voneinander getrennt. Betrachten wir nacheinander drei einfache Beispiele:

```
double sqr(double x)
{
   return x*x;   /* Berechnen des Quadrats und Rueckgabe */
}
```

Funktion *sqr* in 02-einf\funktion.c

Diese Funktion *sqr* berechnet das Quadrat einer Zahl. Die Anweisung *return* gibt den folgenden Ausdruck als Funktionsergebnis zurück. Dieser Ausdruck entspricht dem Ausdruck auf der rechten Seite eines Zuweisungsoperators. Der Aufruf dieser Funktion erfolgt wie folgt:

```
printf("Bitte geben Sie eine Zahl ein: ");
scanf("%lf", &x);
erg = sqr(x);              /* Aufruf von sqr */
printf("Das Quadrat der Zahl %f ist %f\n\n", x, erg);
```

Programmausschnitt von 02-einf\funktion.c

Das ablaufende Programm wird beim Aufruf der Funktion *sqr(x)* in die Funktion *sqr* wechseln und die dortigen Programmanweisungen ausführen. Die Return-Anweisung liefert ein Ergebnis zurück, das dann im Ausdruck rechts vom Zuweisungszeichen anstelle des Funktionsaufrufs *sqr(x)* verwendet wird.

Wir haben hier double-Werte verwendet, da diese sehr genau sind (mindestens 12 Stellen!). Somit kann die Funktion sehr universell verwendet werden. C wandelt int- und float-Werte gegebenenfalls automatisch in double-Werte um. Es ist allerdings eine kleine Falle zu beachten:

Achtung:

➔ Eine **double**-Variable wird mit dem Platzhalter **%lf** eingelesen, aber mit %f ausgegeben.

Diese vielen Spezialitäten und Ausnahmen sind mit ein Grund, warum wir bei der Ein- und Ausgabe bald die Ein- und Ausgabeströme von C++ bevorzugen werden.

Zwei weitere Funktionen wollen wir in unserem Programm noch vorstellen:

```
double power(double mantisse, int exponent)
{
   return exp(exponent*log(mantisse));
}
double abstand(double zahl1, double zahl2)
{
   double erg;
   erg = sqrt(sqr(zahl1) + sqr(zahl2));
   return erg;
}
```

Funktionen *power* und *abstand* in 02-einf\funktion.c

Die Funktion *power(x,n)* entspricht der Funktion x^n. Damit allerdings C die Exponentialfunktion *exp* und den natürlichen Logarithmus *log* kennt, muss die Bibliothek *math.h* hinzugefügt werden! Das Glei-

che gilt für die Quadratwurzelfunktion *sqrt*. Die beiden Funktionen *power* und *abstand* verwenden zwei Parameter, zum einen *mantisse* und *exponent*, zum anderen *zahl1* und *zahl2*. Wieder erkennen wir, dass wir bereits definierte Funktionen gewinnbringend einsetzen können. Dies ist hier die Funktion *sqr*. Die Funktion *abstand(x,y)* berechnet übrigens in einer Ebene den Abstand des Punktes *P(x,y)* vom Ursprung. Der Aufruf dieser drei Funktionen erfolgt in einem Programm wie folgt:

```
erg1 = sqr(x) + sqr(y);          /* Aufruf der selbst definierten Funktion sqr */
erg2 = power(x, zahl);           /* Aufruf der selbst definierten Funktion power */
erg3 = abstand(x2-x1,y2-y1));    /* Abstand der Punkte P(x2,y2) und P(x1,y1) */
```

Diese selbstdefinierten und vordefinierten Funktionen können innerhalb eines Ausdrucks verwendet werden. Aber auch die übergebenen Parameterwerte dürfen beliebige Ausdrücke sein. Der Compiler berechnet diese Parameterausdrücke und kopiert diese Ergebnisse in die Parametervariablen. Besonders gut sehen wir dies an dem zweimaligen Aufruf der Funktion *sqr*. Damit dieser Programmabschnitt korrekt ist, müssen in der Funktion, in der die Funktion *sqr* aufgerufen wird, die drei Variablen *erg1*, *x* und *y* deklariert sein, beispielsweise als Double-Variable. In der Funktion *sqr* selbst existiert der Parameter *x*, ebenfalls als Double-Wert deklariert. Wir haben also:

```
double erg1, x, y;      /* im aufrufenden Programmteil */
double x                /* Parameter der Funktion sqr */
```

Dabei taucht der Name ***x*** zweimal auf. So wie zwei Personen gleichzeitig Franz heißen können, so können in einem Programm auch mehrmals die gleichen Variablennamen auftauchen. Der Compiler wird diese gleichen Namen aber sauber trennen. Betrachten wir den zweifachen Aufruf der Funktion *sqr* etwas genauer. Gehen wir davon aus, dass in der Variable *x* der Wert 3 und in der Variable *y* der Wert 4 gespeichert ist:

Schritt 1: Zerlegen der Zuweisung *erg1 = sqr(x) + sqr(y)*;
Schritt 2: Zerlegen des Ausdrucks *sqr(x) + sqr(y)*
Schritt 3: Bearbeiten des linken Summanden *sqr(x)*
Schritt 4: Summand ist eine Funktion, daher Funktionsaufruf.
Schritt 5: Funktionsaufruf *sqr(3)*
Schritt 6: Abarbeiten der Funktion *sqr*, wobei der Wert 3 in den **Parameter x kopiert** wird
Schritt 7: Funktion liefert als Rückgabewert den Wert 9
Schritt 8: Der linke Summand *sqr(x)* wird durch den Rückgabewert 9 ersetzt
Schritt 9: Bearbeiten des rechten Summanden *sqr(y)*
Schritt 10: Summand ist eine Funktion, daher Funktionsaufruf.
Schritt 11: Funktionsaufruf *sqr(4)*
Schritt 12: Abarbeiten der Funktion *sqr*, wobei der Wert 4 in den **Parameter x kopiert** wird
Schritt 13: Funktion liefert als Rückgabewert den Wert 16
Schritt 14: Der rechte Summand *sqr(y)* wird durch den Rückgabewert 16 ersetzt
Schritt 15: Beide Summanden werden addiert zur Summe 25
Schritt 16: Der Ausdruck hat den Wert 25 und wird der Variablen *erg1* zugewiesen

Wir sehen, dass unser Programm das Gewünschte ausführt. Dies ist deshalb möglich, da der Funktionsaufruf und die Funktionsdefinition strikt voneinander getrennt sind. Der Parametername *x* und die Variablennamen *x* und *y* haben nichts miteinander zu tun, außer dass beim Funktionsaufruf die Inhalte der Variablen *x* und *y* jeweils in den Parameter *x* kopiert werden. Damit kann eine Funktion unabhängig vom Programm definiert werden. Wir hätten jederzeit dem Parameter *x* einen anderen Namen geben können. Das Ersetzen des Parameternamens *x* durch den Namen *zahl* hätte sich nicht auf das restliche Programm ausgewirkt.

Diese Arbeitsweise hat zur Folge, dass wir auch ganze Ausdrücke, oder einfach nur konstante Werte, als Parameter einer Funktion übergeben dürfen, so wie wir dies in unserem Beispiel der Anwendung der Funktion *abstand* gemacht haben. Flexibler können Funktionen kaum angewendet werden.

Leider haben wir wieder das leidige Problem, dass Funktionen vor ihrer ersten Anwendung deklariert sein müssen. Wir fügen zu Beginn also unsere Funktionsdeklarationen hinzu. Wir werden damit belohnt,

dass es dann keine Rolle mehr spielt, wo wir unsere Funktionsdefinitionen einfügen. Wir können damit den für uns am besten geeigneten Platz für unsere Funktionsdefinitionen auswählen. Die Deklarationen lauten:

```
double sqr(double);
double power(double, int);
double abstand (double, double);
```

Wo sind die Parameternamen? Wir haben sie weggelassen, da die Namen in der Deklaration absolut unnötig sind. Die Deklaration der Funktion *sqr* sagt uns hier, dass ein Double-Wert zu übergeben ist und ein Double-Wert zurückgeliefert wird. Mehr müssen Anwender dieser Funktion nicht wissen. Wie der interne lokale Parameter heißt, spielt für die Funktionalität keinerlei Rolle.

2.8 Zeichenketten: eine kurze Einführung

In der ISO-Norm C89 gibt es ausschließlich sogenannte nullterminierte Zeichenketten. In C++ wurde eine eigene String-Klasse hinzugefügt. Letztere erlaubt sehr komfortables Arbeiten mit Zeichenketten. Die nullterminierten Zeichenketten sind sehr einfach zu handhaben. Sie sind aber bei fehlerhafter Anwendung ganz leicht die Ursache von Programmabstürzen. Die Abstürze sind meist nur sehr schwer diagnostizierbar. Dies ist schon wieder ein Grund, auf C++ umzusteigen. Doch noch verweilen wir bei C.

Definition:

Die ***nullterminierte Zeichenkette*** ist ein Feld aus Zeichen, das mit dem Zeichen ‚\0' abgeschlossen wird.

Diese Definition basiert auf der Definition eines Feldes. Die Definition eines Feldes setzt die Kenntnisse von Zeigern voraus, auf die wir erst etwas später stoßen werden. Zeichenketten sind aber mit Hilfe der Ein- und Ausgabefunktionen einfach handhabbar. Aus diesem Grund wollen wir einige Grundkenntnisse schon jetzt einführen.

In C gibt es zwei Möglichkeiten, eine Zeichenkette zu deklarieren:

```
char str1[15];      /* reserviert Speicherplatz für 15 Zeichen */
char* str2;         /* Achtung: kein Speicher, nur ein Verweis auf Zeichenkette!!! */
```

Die Variable *str1* kann eine Zeichenkette bis zu 15 Zeichen aufnehmen. Die Variable *str2* muss zunächst auf eine Zeichenkette verweisen, bevor zugegriffen werden darf. Gleich drei Fallen sind hier zu beachten.

Achtung:

➔ Eine Zeichenkette muss mit dem Zeichen ‚\0' beendet werden.

Wird eine Zeichenkette nicht mit dem Zeichen ‚\0' beendet, so wird C auch den im Speicher direkt danach folgenden Bereich mit einschließen. Erst wenn dieses spezielle Zeichen gefunden wird, ist für C die Zeichenkette zu Ende. Dies kann verheerende Folgen nach sich ziehen, insbesondere wenn man in eine Zeichenkette etwas einfügt!

Achtung:

➔ Eine Zeichenkette vom Datentyp *char** ist nur ein Verweis auf eine Zeichenkette und muss vor der ersten Verwendung auf eine existierende Zeichenkette verweisen.

Verweise werden wir noch sehr genau kennen lernen. Es verhält sich mit ihnen wie mit einem Verweis auf den Mitstudenten Klaus. Wenn es diesen nicht gibt, so nützt der gesamte Verweis nichts. Wenn unsere Variable vom Datentyp *char** also nur deklariert ist, so kann ein Zugriff auf diese Variable zu völlig unvorhergesehenen Effekten im Programm führen.

Glücklicherweise gibt es auch konstante Zeichenketten. Wir haben diese bereits kennen gelernt. Dies sind Zeichenketten, die in Gänsefüßchen eingeschlossen sind. Betrachten wir ein Beispiel:

```
char* str2;                 /* Definition eines Verweises auf Zeichenkette */
str2 = "Hello world\n";     /* str2 verweist jetzt auf die konstante Zeichenkette */
```

Die Zeichenkette *str2* enthält jetzt folgenden Inhalt:

'H'	'e'	'l'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'\n'	'\0'

Wir sehen, dass C automatisch zu einer konstanten Zeichenkette das Zeichen ‚\0' hinzufügt. Die Zeichenkette *str2* können wir natürlich auch manipulieren. Wir sollten aber sehr vorsichtig sein, das letzte Zeichen (‚\0') zu überschreiben. Wollen wir etwa das erste Zeichen ‚H' in einen Kleinbuchstaben überführen, so gelingt uns dies mit

```
str2[0] = 'h';
```

Wir verwenden eckige Klammern zum Indizieren. Der Index beginnt in C immer mit Null! Nochmals sei vor einer zu großzügigen Verwendung gewarnt. C wird ohne Murren auch die Anweisung

```
str2[1013] = 'A';           /* Achtung!!!!!!! */
```

ausführen. Was an dieser Speicherstelle, also 1013 Speicherplätze nach dem Zeichen *H* steht, wird gnadenlos überschrieben. Wir sollten jetzt auch den Unterschied zwischen Zeichenkette und Zeichen verstehen. Es gilt:

```
'A'         ist das Zeichen A
"A"         ist eine Zeichenkette, bestehend aus den beiden Zeichen A und \0
```

Leider gibt es bei den C-Standardzeichenketten ein weiteres Problem zu beachten. Der folgende Programmausschnitt ist fehlerhaft:

```
char str1[15];              /* Definition eines festen Speicherplatzes */
str1 = "Hello world\n";     /* Fehler!!! */
```

Für die Zeichenkette *str1* wurden extra 15 Speicherplätze reserviert. In der zweiten Zeile wird weiterer Speicherplatz für „*Hello world*" reserviert. Die Zeichenkette *str1* kann nicht gleichzeitig auf diesen weiteren Speicher verweisen. An den alten Speicherplatz ist die Variable *str1* aber gebunden! Im Detail werden wir bei Feldern noch auf dieses Problem eingehen.

Achtung:

➔ Eine Zeichenkette mit festem Speicherplatz (*char str[nn]*) kann nicht auf eine andere Zeichenkette verweisen.

Ansonsten kann auch mit diesen Zeichenketten mit Hilfe eines Index auf die einzelnen Elemente zugegriffen werden.

Die Zeichenketten wurden auch deshalb erwähnt, da recht mächtige Funktionen aus den Bibliotheken <stdio.h> und <string.h> die Verarbeitung von Zeichenketten unterstützen. Diese Funktionen werden im nächsten Abschnitt behandelt. Hier sei nur noch erwähnt, dass die Funktionen *printf* und *scanf* einen

eigenen Platzhalter für Zeichenketten besitzen:

%s	Zeichenketten-Platzhalter in *printf* und *scanf*

Beispielsweise können wir die Zeichenkette str1 wie folgt ausgeben:

```
printf("Der Inhalt der Zeichenkette str2 lautet: \"%s\"\n", str2);
```

Ausgegeben wird: *„Der Inhalt der Zeichenkette str2 lautet: "Hello world"“*. Es folgen noch zwei Leerzeilen, eine Leerzeile ist durch die Zeichenkette *str2* verursacht, die zweite durch das angegebene Zeichen ‚\n' in der Funktion *printf*. Wir sehen, dass wir in einer Zeichenkette auch Gänsefüßchen schreiben dürfen. Wir müssen diese allerdings durch das Zeichen ‚\' entwerten.

Mit einer letzten Falle beenden wir diesen Abschnitt:

Achtung:

➔ Eine Zeichenkette wird mit der Funktion *scanf* ohne Verwendung des Adressoperators ‚&' eingelesen.

Der dahinter liegende Grund ist relativ einfach. Zeichenkettenvariablen sind eigentlich Verweise auf Speicherplätze. Int- oder Float-Variablen hingegen entsprechen den Speicherplätzen selbst. Die Funktion *scanf* benötigt jedoch Verweise, und der ***Adressoperator &*** verwandelt Speicherplätze in Verweise auf diese Speicherplätze. Im Falle der Zeichenkettenvariablen müssen wir daher auf diesen Operator zwingend verzichten.

Betrachten wir noch ein kleines Beispiel. In einer Zeichenkette werden die ersten drei Buchstaben in Großbuchstaben umgewandelt und ausgegeben:

```
char* dreiGrossbuchstaben(char* s)
{
   s[0] = s[0] + 'A' - 'a';   /* Aendern */
   s[1] = s[1] + 'A' - 'a';
   s[2] = s[2] + 'A' - 'a';
   return s;                  /* zurueckgeben */
}
```

Funktion *dreiGrossbuchstaben* in 02-einf\string.c

Der Funktion *dreiGrossbuchstaben* wird eine Zeichenkette (*char**) übergeben. Gleichzeitig liefert diese Funktion eine Zeichenkette (*char**) als Funktionsergebnis zurück. Die Sprache C ist sehr flexibel. Als Parameter kann entweder eine Variable vom Typ *char** oder *char [n]* übergeben werden. Das Funktionsergebnis ist aber immer von Typ *char**.

Dieses Beispiel wandelt die ersten drei Buchstaben in Großbuchstaben um. Voraussetzung ist hier allerdings, dass wirklich Kleinbuchstaben vorliegen. Voraussetzung ist weiter, dass die übergebene Zeichenkette tatsächlich drei Zeichen enthält! Nicht auszudenken wäre, wenn wir das Endezeichen (‚\0') überschreiben würden! Dieses Beispiel ist so also nicht zur Nachahmung empfohlen. Andererseits erkennen wir nun, wie in einer Funktionen Zeichenketten übergeben werden können.

Wir sehen weiter, dass wir mit einzelnen Zeichen arbeiten können wie mit Zahlen. Und tatsächlich, zu einem Zeichen wird der Code des Zeichens abgelegt (meist der sogenannte ASCII-Code). Der Code dieses Zeichens ist ein Wert zwischen 0 und 127, je nach Zeichen. Es gilt beispielsweise: das Zeichen ‚A' hat den Wert 65. Alle Großbuchstaben stehen im Code hintereinander, ebenso die Kleinbuchstaben. Der Abstand zwischen einem Klein- und seinem dazugehörigen Großbuchstaben ist immer gleich, also gleich 'A' – 'a'. Diesen Wert addieren wir zu einem Kleinbuchstaben hinzu. Somit erhalten wir den Großbuchstaben. Mit diesem Trick ist es möglich, Klein- in Großbuchstaben umzuwandeln, ohne das wir den tatsächlichen Code kennen.

Der Aufruf dieser Funktion erfolgt beispielsweise mittels

```
str2 = dreiGrossbuchstaben(str1);
```

Dabei enthält *str1* die zu manipulierende Zeichenkette und *str2* die manipulierte. Leider treten Seiteneffekte auf, da wir noch nicht genügend Möglichkeiten der Zeichenbearbeitung kennen: Auch die Zeichenkette *str1* wird durch die Funktion *dreiGrossbuchstaben* verändert! Die Einzelheiten lernen wir kennen, wenn wir mit Feldern arbeiten.

2.9 Standardfunktionen

Bisher haben wir die drei Funktionen *printf*, *scanf* und *system* verwendet. In C gibt es über 100 Funktionen, in C++ kommen noch viele weitere Funktionen und Klassen hinzu. Diese Funktionen werden in mehreren Bibliotheken verwaltet. Folgende Bibliotheken sind für uns in nächster Zeit am wichtigsten:

<stdio.h>	enthält zahlreiche Ein- und Ausgabefunktionen
<string.h>	enthält Funktionen zur Verarbeitung von Zeichenketten
<ctype.h>	enthält Funktionen zum Abprüfen auf bestimmte Eigenschaften
<math.h>	enthält mathematische Funktionen
<stdlib.h>	enthält nützliche spezielle Funktionen
<stream>	unterstützt die Ein- und Ausgabe mit Strömen in C++
<fstream>	unterstützt die Dateibearbeitung mit Strömen in C++

Wir sind nun neugierig geworden. Wir geben daher eine kleine Auswahl von Funktionen zu diesen Bibliotheken an, die Ein-und Ausgabeströme in C++ behandeln wir später in einem eigenen Kapitel. Wir verwenden hier die offiziellen Deklarationen (siehe Kernighan/Rhitchie), beginnen wir mit der Ein- und Ausgabe:

```
<stdio.h>:
        int printf(formatstring, …)                 /* Ausgabe auf Konsole */
        int fprintf(datei, formatstring, …)         /* Ausgabe in eine Datei */
        FILE* fopen(dateiname, modus)               /* Öffnen einer Datei */
        int fclose(datei)                           /* Schließen einer Datei */
        int scanf(formatstring, …)                  /* Eingabe */
        int fscanf(datei, formatstring, …)          /* Eingabe von einer Datei */
        int putchar(c)                              /* Zeichenweise Ausgabe */
        int fputc(c, datei)                         /* Zeichenweise Ausgabe in Datei*/
        int getchar(void)                           /* Zeichenweise Eingabe */
        int fgetc(datei)                            /* Zeichenweise Eingabe von Datei */
        char* gets(s)                               /* Ganze Zeile einlesen */
        int puts(s)                                 /* Zeichenkette ausgeben */
```

Hier sind die Parameter wie folgt definiert: *formatstring*, *dateiname*, *modus* und *s* sind Zeichenketten; *datei* ist vom Typ *FILE**, und *c* ist ein Zeichen.

Dies sind nur die wichtigsten Funktionen für die Ein- und Ausgabe. Wir erkennen aber, dass wir neben den formatierten Ein- und Ausgaben (*scanf* und *printf*) auch zeichen- und zeilenweise ein- und ausgeben können. Gleichzeitig besitzen alle diese Funktionen Erweiterungen, so dass wir auch Ein- und Ausgaben auf Dateien durchführen können. Dateien sind in C vom Datentyp *FILE**. Der Stern deutet wieder an, dass es sich hier um einen Verweis auf eine Datei und nicht um die Datei selbst handelt. Es fällt noch auf, dass die meisten Funktionen einen Int-Wert zurückliefern. Bei den Funktionen *printf* und *scanf* haben wir dies bisher nicht verwendet. Wir erkennen daraus, dass wir in C die Funktionsergebnisse nicht auswerten müssen. Übrigens liefern unsere beiden Funktionen *printf* und *scanf* bei fehlerfreier Abarbeitung die Zahl 0 zurück, im Fehlerfall einen Fehlercode ungleich 0.

Fast genauso umfangreich sind die Zeichen- und Zeichenkettenfunktionen, wir geben nur eine kleine Auswahl an (die Parameter *s1*, *s2* und *s* seien Zeichenketten):

```
<string.h>:
        char* strcpy(s1, s2)        /* kopiert Inhalt von s2 nach s1, gibt s1 zurück */
        char* strcat(s1, s2)        /* hängt Inhalt von s2 an s1 an, gibt s1 zurück */
        int strcmp(s1, s2)          /* vergleicht s1 und s2, gibt 0 zurueck, falls gleich */
        int strlen(s)               /* gibt die Länge der Zeichenkette s zurück */
<ctype.h>:
        int tolower(c)              /* gibt c als Kleinbuchstaben zurück */
        int toupper(c)              /* gibt c als Großbuchstaben zurück */
        int isalpha(c)              /* liefert ungleich 0, falls c ein Buchstabe ist */
        int isdigit(c)              /* liefert ungleich 0, falls c eine Ziffer ist */
        int isalnum(c)              /* liefert ungleich 0, falls c Buchstabe oder Ziffer ist */
```

Die Funktion *strcpy* ist die komfortableste Art, einen Zeichenketteninhalt in eine andere Zeichenkette zu kopieren. Es sei davor gewarnt, hier den Zuweisungsoperator zu verwenden (*s1 = s2*). In diesem Fall verweist *s1* nur auf die Zeichenkette *s2*. Damit sind also *s1* und *s2* identisch. Änderungen in *s1* wirken sich auch auf *s2* aus und umgekehrt. Dies ist normalerweise nicht beabsichtigt. Wir verwenden daher besser die Funktion *strcpy*.

Die Funktion *strcat* entspricht der Addition zweier Zeichenketten (*s1 + s2*). Nur ist in C die Addition für Zeichenketten nicht definiert. Wir müssen also wohl oder übel die Funktion *strcat* verwenden.

Die Funktion *strlen* liefert die Länge eines Strings (ohne das Zeichen ‚\0'), und die Funktion *strcmp* vergleicht zwei Zeichenketten miteinander. Ist die erste Zeichenkette kleiner als die zweite, so wird ein Wert kleiner Null zurückgeliefert, bei Gleichheit der Wert Null. Ist die erste Zeichenkette größer als die zweite, so wird ein Wert größer Null zurückgegeben.

Die Funktionen der Bibliothek *ctype.h* erwarten als Parameter ein Zeichen *c*. Die Funktionen *tolower* und *toupper* liefern den entsprechenden Klein- bzw. Großbuchstaben zurück. Kann das Zeichen nicht umgewandelt werden, weil es kein Buchstabe oder bereits ein Klein- bzw. Großbuchstabe ist, so wird das Zeichen unverändert zurückgeliefert. In C wird eine Zahl ungleich 0 als Wahrheitswert *true* interpretiert, also liefern die Funktionen *isalpha*, *isdigit*, *isalnum* einen Wert ungleich 0 zurück, falls die Aussage zutrifft, ansonsten wird 0 zurückgegeben.

Sehr umfangreich sind die mathematischen Funktionen. Es folgt eine Auswahl:

```
<math.h>:
        sin(x), cos(x), tan(x)      /* Trigonometrische Funktionen */
        asin(x), acos(x), atan(x)   /* Trigonometrische Umkehrfunktionen */
        sinh(x), cosh(x), tanh(x)   /* Hyperbolische Funktionen */
        exp(x)                      /* Exponentialfunktion */
        log(x)                      /* Natürlicher Logarithmus */
        pow(x,y)                    /* entspricht der Funktion x^y */
        sqrt(x)                     /* Quadratwurzel */
        floor(x)                    /* Größte Ganzzahl kleiner gleich x */
        fabs(x)                     /* Absolutbetrag */
```

Alle diese Funktionen liefern einen double-Wert zurück, ebenso werden double-Werte als Parameter erwartet. Natürlich können auch Ganzzahl oder Float-Werte übergeben werden, da C diese dann automatisch in double-Werte umwandelt.

Als letzte Bibliothek wollen wir noch einen Blick auf die Standardbibliothek werfen. Diese enthält viele nützliche Funktionen, von denen wir einige vorstellen:

```
<stdlib.h>
        malloc(n)           /* reserviert dynamischen Speicher der Größe n */
        free(adr)           /* gibt dynamischen Speicher an der Adresse adr frei */
        system(s)           /* Aufruf von Systemfunktionen */
        abs(n)              /* berechnet den Absolutbetrag von n */
```

Die hier verwendete Variable *n* ist vom Typ *int*, die Variable *s* vom Typ *char** (Zeichenkette) und die

Variable *adr* ist eine Zeigervariable. Die Funktionen *malloc* und *free* dienen zur dynamischen Speicherverwaltung. Diese Funktionen sind nicht einfach zu handhaben, wir werden daher stattdessen die wesentlich komfortableren Operatoren *new* und *delete* aus C++ verwenden. Die Funktion *system* kennen wir bereits, und die Funktion *abs* liefert den Absolutbetrag der Zahl *n* als Ganzzahl zurück.

Betrachten wir ein Programm, in dem wir einige dieser Funktionen begegnen. Beginnen wir mit der Funktion *bearbeiteZeichenkette*:

```
void bearbeiteZeichenkette()
{
   char c;               /* einzelnes Zeichen */
   char s[10000];        /* 10000 Speicherplaetze */
   printf("Bitte beliebig lange Zeile eingeben:\n");
   gets(s);              /* weniger als 10000 Zeichen! */
   printf("Bitte geben Sie einen Grossbuchstaben ein: ");
   c = getchar();        /* liest Zeichen ein */
   printf("Die eingelesene Zeile lautet:\n%s\n", s);
   printf("Die Laenge dieser Zeile ist: %d\n", strlen(s));
   printf("%c als Kleinbuchstabe ist: %c\n", c, tolower(c));
}
```

Funktion *bearbeiteZeichenkette* in 02-einf\bibliothek.c

In diesem Programmbeispiel wird eine Zeile eingelesen. Die Funktion *gets* liest so lange ein, bis ein Returnzeichen eingegeben wird. Dieses Returnzeichen selbst wird nicht gespeichert. Stattdessen wird der Zeichenkette das wichtige Zeichen ‚\0' angehängt. Das Programm verursacht einen Fehler, wenn mehr als 10000 Zeichen eingelesen werden. Einfacher ist die Funktion *getchar*. Sie gibt ein Zeichen zurück. Mit der Funktion *tolower* wandeln wir dieses Zeichen in einen Kleinbuchstaben um, falls das Zeichen ein Großbuchstabe war. Die Funktion *strlen* liefert uns die Anzahl der Zeichen einer Zeichenkette, hier die Länge der eingelesenen Zeile.

Die folgende Funktion *manipuliereZahlen* liest einen Winkel ein und liefert den dazugehörigen Sinuswert. Natürlich arbeiten die trigonometrischen Funktionen mit dem Bogenmaß, wir wandeln daher den Winkel in den entsprechenden Bogenwert um. Weiter demonstrieren wir noch die Verwendung der Funktion *abs*, die den Absolutbetrag einer Ganzzahl zurückgibt.

```
void manipuliereZahlen()
{
   int zahl;
   float x, y;
   float PI = 3.14159;
   printf("Bitte geben Sie einen Winkel im Gradmass ein: ");
   scanf("%f", &x);
   y = sin(x / 180 * PI);    /* Umrechnung ins Bogenmass */
   printf("Der Sinus von %f Grad ist %f\n", x, y);
   printf("Ganzzahl, auch negativ, eingeben: ");
   scanf("%d", &zahl);
   printf("Der Betrag von %d ist %d\n", zahl, abs(zahl));
}
```

Funktion *manipuliereZahlen* in 02-einf\bibliothek.c

3 Kontrollstrukturen und Operatoren

Selten läuft ein Programm Zeile für Zeile ab. In der Regel gibt es Verzweigungen und Wiederholungen. Nehmen wir ein kleines Beispiel. Wir lesen eine Zeile ein, wandeln alle Kleinbuchstaben in Großbuchstaben um und geben diese Zeile wieder aus. Grob schaut das Programm wie folgt aus:

Lies eine Zeile in eine Zeichenkette ein
Gehe zum ersten Zeichen
Solange das Ende der Zeichenkette nicht erreicht ist, mache Folgendes:
    Wenn das Zeichen ein Kleinbuchstabe ist, dann in Großbuchstaben ändern
    Gehe zum nächsten Zeichen
Gib die (manipulierte) Zeichenkette aus

Wir nennen eine solche Programmbeschreibung auch Pseudocode. Es handelt sich dabei um keine Programmiersprache, sondern um eine verbale Beschreibung. Es handelt sich hierbei um eine Vorschrift, ein Kochrezept, das angibt, wie wir unser Problem lösen können. Solche Vorschriften heißen ***Algorithmen*** (siehe auch unter Wikipedia).

Achtung:

➔ Es wird dringend empfohlen, vor der eigentlichen Programmierung einen Algorithmus auszuarbeiten.

Obiger Algorithmus zur Umwandlung von Klein- in Großbuchstaben ist korrekt, zumindest in C mit nullterminierenden Zeichenketten! In der Praxis lauern viele Fallen, insbesondere gibt es fast immer Spezialfälle, die natürlich mit berücksichtigt werden müssen. In unserem Beispiel könnte der Anwender beispielsweise nichts einlesen, also gleich die Returntaste drücken. In unserem Algorithmus wird sofort am Anfang überprüft, ob das Ende der Zeichenkette erreicht ist. In unserem Spezialfall wird also nichts ausgeführt, sondern die leere Zeichenkette wird direkt ausgegeben.

Das Wort ***Solange*** bezeichnet eine Schleife. Alle folgenden eingerückten Zeilen werden solange ausgeführt, bis die angegebene Bedingung, auch ***Abbruchbedingung*** genannt, erfüllt ist. Ist die Bedingung nicht mehr erfüllt, so fährt das Programm mit der nachfolgenden nicht eingerückten Zeile fort. Wieder droht eine kleine Programmfalle: Sind wir beim letzten Zeichen angelangt, so werden die folgenden Zeilen ein letztes Mal ausgeführt. Wir gehen also zum nächsten Zeichen, obwohl wir bereits beim letzten Zeichen sind. Wir wissen allerdings, dass eine nullterminierte Zeichenkette nach dem „letzten“ Zeichen noch das Zeichen ‚\0’ besitzt. Es ist also legitim, noch zu diesem Zeichen zu gehen, gleichzeitig ahnen wir schon, wie wir in C auf das Ende einer Zeichenkette abprüfen.

Das Wort ***Wenn*** leitet eine Verzweigung ein. Nur wenn die angegebene ***Bedingung*** erfüllt ist, wird die folgende Anweisung (in Großbuchstaben umwandeln) ausgeführt.

Bevor wir jetzt Schleifen und Verzweigungen noch nicht genau kennen, interessiert uns natürlich das C-Programm zu unserem obigen Algorithmus:

```
void macheGross(char* str)
{
   int i = 0;           /* Index durchlaeuft Zeichenkette */
   while (str[i] != '\0')    /* solange nicht am Ende */
   {
      str[i] = toupper(str[i]);
      i++;                   /* zum naechsten Zeichen */
   }
}
```

Funktion *macheGross* in 03-kontroll\grossbuchstabe.c

Die Ein- und Ausgabe überlassen wir dem Hauptprogramm. *Solange* heißt auf Englisch *while*. Mit diesem Wort wird daher eine Schleife eingeleitet. Direkt danach steht in Klammern eine Bedingung. Die Schleife wird solange ausgeführt, bis die Bedingung nicht mehr wahr ist. Wir indizieren unsere Zeichenkette mit einer Variable *i*. Mit dem Setzen dieser Variable auf den Wert 0 gehen wir quasi an den Anfang der Zeichenkette.

Standardmäßig führt eine Schleife nur die folgende Anweisung mehrfach aus. Sollen mehrere Anweisungen in der Schleife verarbeitet werden, so müssen diese zu einem sogenannten ***Anweisungsblock*** zu-

sammen gefasst werden.

Definition:

Ein ***Anweisungsblock*** fasst mehrere Anweisungen zusammen und besitzt folgenden Aufbau:

```
{ Anweisung; [ … ] }
```

Diese Syntax ist einfach zu interpretieren: Werden eine oder mehrere Anweisungen mit geschweiften Klammern zusammengefasst, so ist dies ein Anweisungsblock. Ein Programm wird mit der Zeit sehr umfangreich und viele Anweisungsblöcke enthalten. Es wird empfohlen, von Anfang an auf einen sauberen Programmierstil zu achten. Ansonsten wird schnell der Überblick verloren.

Achtung:

➔ Es wird dringend empfohlen, bei **Anweisungsblöcken** einzurücken und die öffnende und schließende Klammer untereinander zu schreiben.

In unserem Programm haben wir einen Anweisungsblock verwendet. Auf eine Wenn-Verzweigung konnten wir verzichten, da diese intern in der Funktion *toupper* automatisch ausgeführt wird. Wir erhöhen dann noch den Index *i* um 1. Dazu verwenden wir den praktischen Plus-Plus-Operator, auf den wir noch oft stoßen werden.

Diese Funktion sieht harmlos aus. Doch wir können damit jederzeit auch riesige Zeichenketten verarbeiten. Ob die Zeichenkette *str* nur 10 oder eine Million Zeichen lang ist, spielt für die Funktion keinerlei Rolle. Wir erahnen langsam die Mächtigkeit der Programmierung.

3.1 Die WHILE-Schleife

Wir haben die While-Schleife bereits am Anfang des Kapitels kennen gelernt. Die Syntax der While-Schleife lautet:

```
while ( Ausdruck )
   Anweisung
```

Hier wird die Anweisung so lange ausgeführt, bis der Ausdruck nicht mehr wahr ist. Ist der Ausdruck von Anfang an nicht wahr, so wird die Anweisung nicht ausgeführt. Das Programm fährt nach der Abarbeitung der Schleife in der Folgezeile fort.

Es versteht sich von selbst, dass die Anweisung etwas manipulieren muss, damit sich der Wahrheitswert des Ausdrucks irgendwann ändert. Andernfalls hätten wir ein nie endendes Programm erzeugt. Das geht übrigens viel schneller als wir glauben:

Achtung:

➔ Ein **Semikolon** in der **While**-Schleife direkt nach der schließenden Klammer erzeugt eine Endlosschleife.

Um dies zu verstehen, müssen wir wissen, dass ein Semikolon für sich allein in C eine leere Anweisung ist. Es gibt tatsächlich sinnvolle Anwendungen hierzu. Ein Semikolon direkt nach der schließenden Klammer bedeutet, dass die While-Schleife nur aus der leeren Anweisung besteht. Diese leere Anweisung ändert aber ganz sicher nichts, so dass die Schleife endlos läuft.

Betrachten wir ein kleines Beispiel. Wir wollen alle Zahlen von 1 bis 100 zusammenzählen. Dies leistet folgender Programmausschnitt

```
int summe = 0;              /* speichert die Summe der Zahlen */
int i = 1;                  /* Index */
while (i <= 100)
{
    summe = summe + i;      /* fuegt i zur Summe hinzu */
    i++;
}
```

Nun enthält die Variable *summe* den Summenwert der Zahlen von 1 bis 100, also 5050. Es ist zu beachten, dass sowohl die Variable *summe* als auch der Index *i* mit einem passenden Wert initiiert werden. Kommen wir noch zu einem anschaulicheren Beispiel. Wir haben bereits im letzten Kapitel kleine Sternenbäume ausgegeben. Wäre es nicht schön, wenn wir einen beliebig großen Sternenbaum ausgeben könnten, etwa durch die Aufrufe:

```
zeichneSternenbaum(5);                  /* Sternenbaum aus 5 Zeilen */
zeichneSternenbaum(30);                 /* Sternenbaum aus 30 Zeilen */
```

Und schon würden wir einen Baum mit 5 bzw. 30 Zeilen erhalten. Dieses so einfach wirkende Problem erweist sich als sehr knifflig. Vorbei ist es mit dem sofortigen Eintippen am Computer, denn wir wissen gar nicht, wie wir überhaupt anfangen sollen. Es sei nicht verschwiegen, dass mit Hilfe formatierter Ausgabe die Aufgabe einfacher lösbar wird, aber erstens haben wir dieses Hilfsmittel noch nicht behandelt und zweitens wollen wir uns einmal so richtig anstrengen. Schon sind wir beim einzig sinnvollen Weg angelangt. Wir erarbeiten uns einen Lösungsweg, formulieren diesen Weg aus und programmieren erst dann. Betrachten wir also zunächst einen ganz kleinen Baum, bestehend aus nur drei Zeilen:

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

Wir müssen drei Zeilen schreiben. Die Idee ist, in einem Schleifendurchlauf eine Zeile zu schreiben. Wir haben hier also drei Schleifendurchläufe vor uns. In der ersten Zeile schreiben wir zunächst zwei Leerzeichen, dann einen Stern, in der zweiten Zeile ein Leerzeichen und drei Sterne und in der dritten Zeile fünf Sterne. Es kommen also pro Zeile zwei Sterne hinzu und ein Leerzeichen weg!

Verallgemeinern wir dies: wir haben *anzahl* Zeilen. In der ersten Zeile schreiben wir *anzahl-1* Leerzeichen, gefolgt von einem Stern. In der zweiten Zeilen schreiben wir *anzahl-2* Leerzeichen, gefolgt von drei Sternen. In der i-ten Zeile schreiben wir *anzahl-i* Leerzeichen, gefolgt von *2*i-1* Sternen! Dabei nimmt i alle Werte von *1* bis *anzahl* an.

Damit hätten wir den Algorithmus bereits erfasst. Die Leerzeichen und die Sterne geben wir wieder in einer Schleife aus. Unser Pseudecode lautet damit:

```
Setze den Zeilenzähler i auf 1

Solange i kleiner oder gleich anzahl ist, mache Folgendes:
    Anzahl der Leerstellen ist anzahl-i
    Anzahl der Sterne ist 2*i-1
    Setze Laufindex auf 1

    Solange Laufindex kleiner oder gleich Anzahl der Leerstellen    (Leerzeichen ausgeben)
        Gib Leerzeichen aus
        Erhöhe Laufindex um 1

    Setze Laufindex zurück auf 1

    Solange Laufindex kleiner oder gleich Anzahl der Sterne         (Sterne ausgeben)
        Gib Stern aus
        Erhöhe Laufindex um 1

    Gib einen Zeilenvorschub aus
    Erhöhe i um 1
```

Unser Programm besteht aus einer Hauptschleife über alle Zeilen. Diese Hauptschleife enthält zwei Unterschleifen, die erste gibt die Leerzeichen aus, die zweite die Sterne einer Zeile. Am Ende der Zeile erfolgt noch ein Zeilenvorschub und mit dem Erhöhen des Zeilenzählers wird in die nächste Zeile gewechselt. Haben wir diesen Pseudocode verstanden, so können wir unser Problem nun sofort in C formulieren:

```
void zeichneSternenbaum(int anzahl)
{
   int i = 1;    /* Zeilenindex */
   while (i <= anzahl)
   {
      int leer = anzahl - i;
      int sterne = 2*i – 1;
      int j = 1;           /* Index */
      while (j <= leer)    /* Anfangsleerzeichen ausgeben */
      {
         putchar(' ');     /* Leerzeichen ausgeben */
         j++;
      }
      j = 1;               /* zuruecksetzen */
      while (j <= sterne)     /* Sternchen ausgeben */
      {
         putchar('*');
         j++;
      }
      putchar('\n');       /* neue Zeile */
      i++;
   }
}
```

Funktion *zeichneSternenbaum* in 03-kontroll\sternenbaum.c

Gegenüber dem Pseudocode haben wir hier die Variablen *leer* und *sterne* anstelle von *Anzahl der Leerstellen* und *Anzahl der Sterne* verwendet. Einzelne Zeichen sollten immer mit der Funktion *putchar* ausgegeben werden, da die mächtige Funktion *printf* wegen der Formatierung einen erheblichen Overhead verursacht. Zu beachten ist ferner, dass in C zu Beginn jedes Anweisungsblocks Variablen deklariert werden dürfen. Diese Variablen sind aber dann nur bis zum Ende des Anweisungsblock gültig. Wir benötigten drei Anweisungsblöcke, für jede Schleife einen, da innerhalb jeder Schleife mehr als eine Anweisung auszuführen ist. Wir benötigen jetzt nur noch ein kurzes Hauptprogramm, das diese Funktion aufruft, und schon können wir größere oder kleinere Sternenbäume ausgeben.

3.2 Die IF-Verzweigung

Zu einem Programm gehören neben Schleifen vor allem Verzweigungen. Diese sehr wichtige Kontrollstruktur wollen wir in diesem Absatz kennen lernen. Beginnen wir gleich mit einem Beispiel. Wir schreiben eine Funktion *max*, die das Maximum zweier Zahlen zurückliefert. Der Aufruf von

```
erg = max(10, 5);
```

soll also die Zahl 10 in der Variable *erg* ablegen. Der Pseudocode dieser einfachen Funktion ist:

```
Funktion max(a, b):
   wenn a größer gleich b ist, dann
      gib a als Funktionsergebnis zurück
   sonst
      gib b als Funktionsergebnis zurück
```

Wir erkennen an diesem Beispiel den allgemeinen Aufbau einer Verzweigung. Ist die Bedingung erfüllt, so wird der dann-Zweig aufgerufen, sonst der sonst-Zweig. Betrachten wir gleich die Implementierung:

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

Funktion *max* in 03-kontroll\max.c

Wir sehen wieder die englische Übersetzung unseres deutschen Pseudocodes. Allerdings vermissen wir das Wort *then*. Dieses Wort haben die Erfinder der Sprache C einfach weggelassen, da nach dem Schließen der runden Klammer automatisch der Dann-Zweig beginnt. Betrachten wir die Syntax der If-Verzweigung:

if (Ausdruck)
    Anweisung
[**else**
    Anweisung]

Die eckigen Klammern geben wieder an, dass dieser Teil der Verzweigung nicht angegeben werden muss. Die Funktionsweise dieser Verzweigung kennen wir eigentlich schon: Ist der Ausdruck wahr, so wird die direkt folgende Anweisung ausgeführt, ansonsten die Anweisung nach dem Bezeichner *else*. Fehlt die else-Verzweigung, so wird in diesem Fall eben nichts ausgeführt. Anschließend fährt das Programm mit der Folgezeile fort. Soll in einem Zweig mehr als eine Anweisung ausgeführt werden, dann wissen wir schon: Wir verwenden einen Anweisungsblock!

Nach so viel Vorarbeit wollen wir uns an eine etwas größere Aufgabe wagen, der Bestimmung der Nullstellen der **quadratischen Gleichung** $ax^2+bx+c=0$. Wieder hat es wenig Sinn, gleich am Rechner Code einzutippen, wie soll dieser denn auch aussehen. Wir beginnen stattdessen wieder zunächst mit einer Analyse des Problems, erstellen dann einen Pseudocode und kommen schließlich zum fertigen Programm. Beginnen wir also mit der Analyse, wobei wir auch alle Sonderfälle, etwa a=0, mit einbeziehen wollen.

Analyse:

a=0:	bx+c=0	$x= -\frac{c}{b}$ für $b \neq 0$
		alle x sind Lösungen für c=0 und b=0
		es gibt keine Lösung für $c \neq 0$ und b=0
$a \neq 0$:	$D=b^2-4ac$	(Diskriminante)
	D<0	keine reelle Lösung
	D=0	$x = \frac{-b}{2a}$ ist doppelte Lösung
	D>0	$x_{1/2} = \frac{-b \pm \sqrt{D}}{2a}$

Wir sehen an der Analyse, dass es zahlreiche Fälle gibt, die wir mit der If-Verzweigung behandeln. Der Pseudocode lautet:

Wenn a gleich 0 ist, dann
    Wenn b gleich 0 ist, dann
        Wenn c gleich 0 ist, dann
            Alle x sind Lösungen
        Sonst
            Keine Lösung
    Sonst
        Lösung x=-c/b
Sonst
    D=b*b-4*a*c
    Wenn D kleiner als 0 ist, dann
        Keine reelle Lösung
    Sonst
        Wenn D gleich 0 ist, dann
            Doppelte Lösung x=-b/(2a)
        Sonst
            x_1=(-b+sqrt(D))/(2*a), x_2=(-b-sqrt(D)/(2*a)

Mit bis zu drei Schachtelungen von If-Verzweigungen hätten wir also die Aufgabe gelöst. Wir müssen im C-Programm allerdings sehr genau aufpassen, um die richtige Zuordnung der Else-Zweige sicherzustellen. Da bei den meisten Verzweigungen nur eine Anweisung ausgeführt wird, kommen wir sogar mit nur einem einzigen Anweisungsblock aus.

```
void ausgebenQuadratGleichung (double a, double b, double c)
{
   double diskr;             /* Diskriminante */
   if (a==0)
      if (b==0)
         if (c==0)
            puts("Alle x sind Loesungen");
         else
            puts("Es gibt keine Loesung");
      else
         printf("Die einzige Loesung ist: %f\n", -c/b);
   else  /* a != 0 */
   {
      diskr = b*b - 4*a*c;        /* Diskriminante */
      if (diskr < 0)
         puts("Es gibt keine reelle Loesung");
      else
         if (diskr == 0)
            printf("Doppelte Loesung: %f\n", -b/(2*a));
         else      /* Diskr > 0 */
            printf("2 Loesungen: %f und %f\n",
                                 (-b+sqrt(diskr))/(2*a),
                                 (-b-sqrt(diskr))/(2*a) );
   }
}
```

Funktion *ausgebenQuadratGleichung* in 03-kontroll\quadGleichung.c

Der Pseudocode wurde in diesem Programm eins zu eins umgesetzt. Für die einfache Ausgabe von Informationen wurde die Funktion *puts*, für die Ausgabe von Variablenwerten die Funktion *printf* verwendet. Ganz neu in diesem Programm sind die Vergleiche. Es gibt in C sechs Vergleichsoperatoren, nämlich

== (gleich), **!=** (ungleich),
<= (kleiner gleich), **<** (kleiner), **>=** (größer gleich), **>** (größer)

Leider ist in der Sprache C das Zeichen ‚=' bereits für die Zuweisung vergeben. Aus diesem Grunde wird für den Vergleich das doppelte Gleichheitszeichen verwendet. Für den Anfänger ist dies eine gefährliche Falle.

Achtung:

➔ Der **Vergleich** auf Gleichheit erfolgt in C mit dem Operator **==.** Der Vergleich darf nicht mit dem Zuweisungsoperator = verwechselt werden!

3.3 Die For-Schleife

Nach der Verzweigung kommen wir zu den Schleifen zurück. Es gibt in C eine Schleife, die häufig wesentlich handlicher ist, die sogenannte For-Schleife. Eine For-Schleife beginnt in der Regel mit einem Startwert, zählt diesen dann automatisch je Schritt um den Wert eins hoch und endet in der Regel mit einem Endwert. In C ist die For-Schleife aber so mächtig, dass jede While-Schleife auch als For-Schleife formuliert werden kann. Es gilt auch das Umgekehrte. Welche dieser Schleifen der Programmierer verwendet, liegt in seinen Händen und hat auch etwas mit der Übersichtlichkeit der Programmierung zu tun. Schauen wir uns gleich die Syntax der For-Schleife an:

for ($Ausdruck_1$; $Ausdruck_2$; $Ausdruck_3$)
  Anweisung

Zum Unterschied zur While-Schleife stehen in einer For-Schleife drei Ausdrücke statt einem Ausdruck innerhalb der Klammer, die durch Semikolon voneinander getrennt sind. Diese For-Schleife ist äquivalent zu:

$Ausdruck_1$;
while ($Ausdruck_2$)
{ Anweisung
  $Ausdruck_3$;
}

Dies ist sehr formal. In den allermeisten Fällen schreibt man in den $Ausdruck_1$ eine Startzuweisung, in den $Ausdruck_2$ das Abbruchkriterium und in den $Ausdruck_3$ das Hochzählen des Schleifenzählers. Liegt eine Schleife mit vorher bekannten Start- und Endwerten vor, so sollte immer die For-Schleife verwendet werden. Sie ist in diesem Fall deutlich besser handhabbar als die While-Schleife. Um dies zu beweisen, schreiben wir unsere Funktion *zeichneSternenbaum* nochmals, diesmal mit For-Schleifen:

```
void zeichneSternenbaum(int anzahl)
{
   int i, j;          /* Indexe deklarieren */
   for (i=1; i <= anzahl; i++)
   {
      int leer = anzahl - i;
      int sterne = 2*i - 1;
      for (j=1; j <= leer; j++)   /* Anfangsleerzeichen */
         putchar(' ');
      for (j=1; j <= sterne; j++)      /* Sternchen */
         putchar('*');
      putchar('\n');          /* neue Zeile */
   }
}
```

Funktion *zeichneSternenbaum* in 03-kontroll\sternenbaum2.c

Diese neue Funktionen *zeichneSternenbaum* ist wesentlich kompakter als die alte. Sie gewinnt auch erheblich an Übersichtlichkeit. Der Hauptgrund ist der, dass die beiden internen Schleifen bisher zwei Anweisungen ausführten, eine Ausgabe und das Weiterzählen des Zählers. Letzteres wird jetzt im Kopf der For-Schleife erledigt, so dass nur noch eine Anweisung vorliegt und wir damit gleich noch den An-

weisungsblock streichen konnten. Doch auch schon durch seinen Aufbau trägt die For-Schleife zur Verbesserung der Übersicht bei.

Die For-Schleife beginnt mit einem Startwert. Dank der For-Schleife werden wird diesen Startwert in der Regel nicht mehr vergessen, was in einer While-Schleife durchaus mal passieren kann. Nach dem ersten Semikolon schreiben wir das Abbruchkriterium, und nach dem zweiten Semikolon folgt das Fortschalten des Zählers. Auch dieses Fortschalten werden wir dank For-Schleifen nicht mehr vergessen.

3.4 Operatoren

Wir haben bereits eine Reihe von Operatoren kennen gelernt. Fassen wir diese kurz zusammen:

*, /, +, -	Arithmetische Operatoren
=	Zuweisungsoperator
<, <=, >, >=, ==, !=	Vergleichsoperatoren
++	Inkrementoperator

Dies sind sehr wichtige Operatoren, aber bei Weitem nicht alle. Betrachten wir daher alle C-Operatoren systematisch:

Arithmetische Operatoren: + - * / %

Vier dieser fünf Operatoren kennen wir, ebenso die Regel „Punkt vor Strich“. Der fünfte Operator (‚%’) ist der Modulooperator. Dieser ist nur auf Ganzzahlen anwendbar. Eine Eigenheit weist in C die Division auf: Sind beide Operanden Ganzzahlen (int, short, long, long long oder char), so ist das Ergebnis wieder eine Ganzzahl, wobei gegebenenfalls zur nächstkleineren Zahl gerundet wird. Beispiele:

17 / 3	ergibt	5	
17.0 / 3	ergibt	5.666667	
17 % 3	ergibt	2	(17/3 ist 5 Rest **2**)

Achtung:

➔ Sind beide Operanden Ganzzahlen, so ist das Ergebnis der **Division** wieder eine Ganzzahl.

Vergleichsoperatoren: < <= > >= == !=

Das Ergebnis von Vergleichen ist entweder wahr oder falsch. Abfragen finden wir in Programmen meist in Verzweigungen und als Abbruchkriterium in Schleifen. Abfragen dürfen in allen passenden Ausdrücken verwendet werden. Da C Wahrheitswerte gegebenenfalls automatisch in Ganzzahlen umwandelt, können Vergleichsoperatoren vielfältig eingesetzt werden. Hier wird aber dringend Zurückhaltung empfohlen! Betrachten wir folgendes Beispiel:

```
int i = -5;
int j = i>0;
```

Nach Ausführung dieser beiden Zeilen wird die Variable *j* den Wert 0 enthalten. Da der Vergleich den Wahrheitswert *false* liefert, wird dies automatisch in die Zahl 0 umgewandelt. Wäre hingegen *i* auf den Wert 7 gesetzt worden, so würde der Vergleich den Wert *true* liefern, und damit würde in *j* automatisch die Zahl 1 gespeichert werden. Wie wir in diesem Abschnitt noch sehen werden, sind hier Klammern nicht erforderlich.

Boolesche (logische) Verknüpfungsoperatoren: && || !

Häufig soll eine Anweisung nur ausgeführt werden, wenn mehrere Bedingungen gleichzeitig zutreffen. Denken wir zum Beispiel an die Ausgabe unseres Sternenbaums. Es macht keinen Sinn, wenn die Anzahl der Zeilen negativ ist, ebenso wenn die Zeilenzahl zu groß wird, da dann ungewollte Zeilenumbrüche die Ausgaben verfälschen. Wir wollen also einen Sternenbaum nur dann ausgeben, wenn die Zeilenzahl größer Null und kleiner gleich 40 ist. Hier benötigen wir die logische **Und**-Verknüpfung **&&**. Unsere Abfrage hat dann folgendes Aussehen:

```
if ( anzahl > 0  &&  anzahl <= 40 )     …
```

Wieder brauchen in diesem Ausdruck keine Klammern gesetzt werden. Eine Und-Verknüpfung ist genau dann wahr, wenn sowohl der linke als auch der rechte Operand wahr sind. Wichtig ist in der Praxis auch die **Oder**-Verknüpfung ‚||'. Der Ergebnis der Oder-Verknüpfung ist genau dann wahr, wenn mindestens einer der beiden Operanden wahr ist.

Der dritte logische Operator ist der Nicht- oder **Negations**-Operator ‚**!**'. Der Negationsoperator ist ein unärer Operator, hat also nur einen Operanden. Hat der Operand den Wahrheitswert *true*, so liefert dieser Operator den Wert *false* zurück und umgekehrt. Dies gilt zumindest für C++. Da C Wahrheitswerte als Funktionsergebnis nicht kennt, wird statt *false* die Zahl 0 und statt *true* die Zahl 1 zurückgegeben. In C und C++ kann dieser Operator auch auf Ganzzahlen angewendet werden. Es gilt:

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

Natürlich führen auch Umwege zum Ziel. Obige If-Abfrage hätten wir auch mit Negations- und Oder-Operator formulieren können. Beachten Sie hier aber die Klammer:

```
if ( ! (anzahl <= 0  ||  anzahl > 40 ) )   …
```

Inkrement und Dekrement: ++ --

Sehr häufig muss eine Variable um den Wert 1 erhöht oder erniedrigt werden. Aus diesem Grund wurden dafür in C eigene Operatoren eingeführt: der **Inkrementoperator ++** und der **Dekrementoperator --** . 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
```

Diese beiden Operatoren dürfen, wie andere Operatoren auch, innerhalb beliebig komplexer Ausdrücke verwendet werden. In solchen Fällen muss genau darauf geachtet werden, ob der Operator als Präfixoperator (++i) oder Postfixoperator (i++) verwendet wird, denn es gilt:

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

Gleiches gilt analog für den Dekrementoperator --. Es wird dringend empfohlen, diese beiden Operatoren nur sehr vorsichtig innerhalb von Ausdrücken zu verwenden. Es gilt in C, wie in allen Programmiersprachen, dass Resultate, die von der Reihenfolge der Abarbeitung abhängen, und diese Reihenfolge nicht

explizit vorgegeben ist, undefiniert sind. Ein Beispiel für eine undefinierte Anweisung ist:

```
i = 2; k = (i++) + i;                    k=4 oder k=5 ?
```

Bitoperatoren: << >> & | ^ ~

Diese Bitoperatoren sind gerade in der Systemprogrammierung sehr nützlich. Damit können Ganzzahlvariablen bitweise manipuliert werden. Für diese sechs Operatoren gilt:

```
<< : 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
```

Achtung:

➔ Die **Bitoperatoren** &, | und ~ dürfen nicht mit den logischen Operatoren &&, || und ! verwechselt werden.

Betrachten wir ein kleines Beispielprogramm (03-kontroll\bit.c), das die Wirkungsweise dieser Operatoren demonstrieren soll. Geben wir die Zahlen i=13 und k=11 ein, so erhalten wir:

Ausführung von 03-kontroll\bit.c im Konsolenfenster

Um die Arbeitsweise dieser Operatoren zu verstehen, müssen wir die Binärdarstellung betrachten. Es gilt: $i = 13 = 1101_2$ und $k = 11 = 1011_2$. Dann führt ein Shift um 2 Bits nach links (i<<2) zu $110100_2 = 52$. Umgekehrt ergibt ein Shift um 1 Bit nach rechts (i>>1) zu $110_2 = 6$. Wir erkennen, dass beim Shift nach Links mit binären Nullen aufgefüllt wird, und beim Shift nach rechts einfach Bits gestrichen werden.

Die Operatoren &, | und ~ arbeiten ebenfalls bitweise. Jede einzelne Stelle der beiden Zahlen i und k wird mit dem booleschen Operator Und (&) bzw. Oder (|) bzw. Exklusives Oder (^) verknüpft. Wir erhalten bei der Und-Verknüpfung den Wert $1001_2 = 9$, bei der Oder-Verknüpfung $1111_2 = 15$ und bei der Exklusiven-Oder-Verknüpfung $0110_2 = 6$. Beachten Sie im Gegensatz dazu, dass die logischen Operatoren && und || immer nur als Ergebnis 1 oder 0 haben können.

Die Und- und Oder-Operatoren sollten bekannt sein, der Exklusive-Oder-Operator, auch kurz XOR genannt, sei kurz erklärt. Der boolesche Operator XOR liefert *true* zurück, wenn beide Operanden verschieden sind (einer *true*, der andere *false*), und *false*, wenn beide Operanden gleich sind (entweder beide *true* oder beide *false*). Wenn wir jetzt beachten, dass in C die Zahl 1 für *true* und die Zahl 0 für *false* steht, sollte das Ergebnis (13^11=6) nachvollzogen werden können.

Der Negationsoperator betrachtet alle Stellen einer Zahl, füllt also in der Binärdarstellung links mit Nullen auf und wandelt dann alle Nullen in Einsen um und umgekehrt. Ohne hier näher auf die Zahlencodierung einzugehen, sei nur erwähnt, dass das ganz links stehende Bit das Vorzeichenbit ist. Dieser wird mit dem Operator ~ ebenfalls manipuliert, eine negative Zahl ist das Ergebnis.

Betrachten wir noch eine kleine wichtige Anwendung der Bitoperatoren: Wir wollen in einem Pro-

gramm abfragen, ob eine Zahl ungerade ist. Es gibt dazu zwei gute Möglichkeiten:

```
zahl % 2 == 1
zahl & 1 == 1
```

Bitte vollziehen Sie selbst nach, dass beide Ausdrücke genau dann wahr sind, wenn die Zahl ungerade ist. Aus Sicht der Performance ist der Und-Operator deutlich schneller als der Modulo-Operator.

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

Neben dem bereits bekannten Operator ‚=' gibt es noch zehn weitere. Alle arithmetischen und binären Bitoperatoren können mit dem Zeichen ‚=' kombiniert werden. Dabei gilt:

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

wobei ‘op’ ein arithmetischer Operator oder binärer Bitoperator ist. Beispielsweise entspricht

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

Wir sehen, dass es sich hier nur um eine abkürzende Schreibweise handelt. Wichtig ist allerdings noch, dass die Zuweisung ein ganz normaler Operator ist, der einen Ergebniswert zurückliefert. Aus diesem Grund ist es legitim, folgende Anweisung zu schreiben:

```
i = ( j = ( k = 1 ) );              // oder kurz:  i = j = k = 1;
```

Hier werden von rechts beginnend mehrere Zuweisungen ausgeführt. Alle drei Variablen erhalten den Wert 1. Mit dem Ausdruck *k=1* erhält *k* den Wert 1 und das Ergebnis ist genau der in *k* gespeicherte Wert, also ebenfalls 1. Dieser Wert 1 wird jetzt an *j* zugewiesen usw. Auch wenn damit klar wird, dass die Zuweisung in allen Ausdrücken verwendet werden können, wird von zu intensivem Gebrauch abgeraten. Wie beim Inkrementoperator können auch hier ungewollte Nebeneffekte auftreten.

An dieser Stelle sei noch auf den wichtigen Unterschied zwischen **Ausdruck** und **Anweisung** hingewiesen:

Ein **Ausdruck** liefert ein Ergebnis zurück, entweder eine Ganzzahl, eine Gleitpunktzahl, einen Verweis auf eine Zeichenkette oder einen booleschen Wert. Weitere Datentypen werden wir noch kennenlernen, die dann ebenfalls das Ergebnis von Ausdrücken sein können.

Eine **Anweisung** ist ein selbstständiger Befehl innerhalb einer Funktion.

In C kann ein Ausdruck immer in eine Anweisung verwandelt werden, denn es gilt:

Ein Ausdruck, gefolgt von einem Semikolon, ist eine Anweisung!

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 immer 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 in 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 in 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 des ASCII-Wertes des Buchstaben C. Wir können schreiben:

```
char c = 'C';
printf("Der ASCII-Wert des Buchstaben %c ist %d.\n", c, (int)c);
```

Eine dringende Empfehlung:

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

Beispiel:

```
int i;  float x;
x = 3.14159;
i = (int) x;          // hier wird klar, dass der Programmierer die Umwandlung wollte
```

Eigenschaften des Datentyps char:

In C ist der Datentyp **char** eine **1 Byte Ganzzahl**! In einer Variable vom Typ *char* können Zeichen, aber auch kleine Zahlen gespeichert werden. Beim Einlesen mit *scanf* (mit dem Formatzeichen %c) oder *getchar* wird in einer Ganzzahl (int, long, short oder char) immer der ASCII-Wert des gelesenen Zeichens abgelegt. Entsprechend geben *putchar* und *printf* (mit dem Formatzeichen %c) *(ch)* immer das Zeichen aus, das durch den auszugebenen ASCII-Wert repräsentiert wird.

Ganz wichtige Besonderheiten der Ein- und Ausgabefunktionen der Bibliothek *stdio.h* sind:

getchar liefert einen Int-Wert als Funktionswert zurück, speichert darin aber natürlich den ASCII-Wert des eingelesenen Zeichens. Bisher haben wir nur in Char-Werten eingelesen, was auch problemlos funktionierte. In der Praxis ist dies allerdings meist ein schwerer Fehler, da wir bei Dateien auf das Dateiende überprüfen müssen. Hierzu wird aber ein Int-Wert zwingend benötigt!

scanf besitzt eine kleine Falle: Schreiben Sie vor dem Formatzeichen *%c* ein Leerzeichen, so können keine Leerzeichen eingelesen werden! Es werden alle eingelesenen Leerzeichen überlesen, und das dann

nächste Zeichen ungleich Leerzeichen wird entgegengenommen.

Weitere Operatoren:

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

- * und &: Zeigeroperatoren (→ Kapitel 5)
- -> und . (Punkt): Strukturoperatoren bei *struct* und *union* (wird später behandelt)
- , (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 ;
}
```

- 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 in C mit Hierarchie

	Operator	Ass.
Spezielle Op.	( ) , [ ] , -> , . *(Punkt)*	LR
Unäre Operatoren	! , ~ , ++ , -- , + , - *(Vorzeichen)* , * , & *(Zeiger, Referenz)* , (cast) sizeof	RL
Arithm. Op.	* , / , %	LR
Arithm. Op.	+ , -	LR
Shift-Op.	<< , >>	LR
Vergleichs-Op.	< , <= , > , >=	LR
Vergleichs-Op.	== , !=	LR
Bit-Op.	&	LR
Bit-Op.	^	LR
Bit-Op.	|	LR
Logischer Op.	&&	LR
Logischer Op.	||	LR
If-Op.	? :	RL
Zuweisungs-Op.	= , += , -= , *= , /= , %= , &= , ^= , |= , <<= , >>=	RL
Komma-Op.	, (*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 der umfangreichen Erweiterungen von C++ existieren fast alle Operatoren auch schon in C. Nur die Operatoren *::*, *new*, *delete*, ->* und .* kommen in C++ hinzu. Wir haben diese in obiger Tabelle bewusst nicht aufgeführt. In diesem Semester werden wir nur die Operatoren *new* und *delete* kennen lernen, am Rande auch den ::-Operator.

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, da  c += (a++) b;  kein erlaubter Ausdruck ist */
```

Denn C fasst die beiden Pluszeichen zum Inkrementoperator zusammen.

3.5 Die Do-While-Schleife

Die While-Schleife überprüft gleich zu Beginn, ob eine bestimmte Bedingung erfüllt ist. Es kann also vorkommen, dass die Schleife gar nicht durchlaufen wird. Gelegentlich kommt es vor, dass eine Schleife mindestens einmal durchlaufen wird und das Abbruchkriterium erst beim ersten Schleifendurchlauf ermittelt wird. Dann müssen wir die Schleifenanweisung bereits vor der Schleife einmal ausführen, oder wir verwenden eine Do-While-Schleife. Die Syntax dieser Do-While-Schleife lautet:

```
do
   Anweisung
while ( Ausdruck ) ;
```

Zu beachten ist hier das Semikolon am Ende dieser Kontrollstruktur. Weiter werden in der Schleife meist mehrere Anweisungen benötigt, wir verwenden daher praktisch immer einen Anweisungsblock. Ansonsten ist die Schleife leicht erklärt, indem wir die englischen Bezeichner ins deutsche übersetzen: ***Mache*** *etwas* ***solange*** *eine Bedingung gilt.*

Wir fragen uns jetzt vielleicht, ob es tatsächlich geeignete Anwendungsfälle gibt. Um dies zu belegen, betrachten wir gleich ein passendes Beispiel. Wir lesen Zahlen ein und brechen das Einlesen erst bei Eingabe einer nicht positiven Zahl ab. Anschließend geben wir die Summe dieser eingelesenen Zahlen und den Mittelwert aus. Wir merken uns dazu in der Schleife die Summe und die Anzahl der bisher eingelesenen Zahlen. Zu beachten ist, dass wir diese beiden Variablen am Anfang mit der Zahl 0 vorbelegen.

```
void berechneMittelwert()
{
   float summe = 0,        /* nimmt den Summenwert auf */
         zahl;             /* aktuell gelesene Zahl */
   int   anzahl = 0;       /* Anzahl der gelesenen Zahlen */
   puts("Bitte positive Zahlen eingeben, sonst Abbruch\n");
   do
   {
      scanf("%f", &zahl);
      summe += zahl;        /* Summe erhoehen */
      anzahl++;             /* Anzahl erhoehen */
   } while (zahl > 0);
   if (anzahl == 1)
      puts("Eine Zahl waere doch drin gewesen!\n");
   else
      printf("Summe: %f, Mittelwert: %f\n\n",
                  summe-zahl, (summe-zahl)/(anzahl-1));
}
```

Funktion *berechneMittelwert* in 03-kontroll\mittelwert.c

Es ist zu beachten, dass wir unter Verwendung einer While-Schleife die Scanf-Funktion zusätzlich vor

der Schleife hätten ausführen müssen. Die zuletzt eingelesene nicht positive Zahl verfälscht das Ergebnis. Diese wird daher von der Summe am Schluss wieder abgezogen, ebenso wird die Variable *anzahl* um 1 reduziert. Jetzt droht aber die Division durch 0, wenn gleich zu Anfang eine negative Zahl eingegeben wurde. Dies wird mit einer If-Verzweigung abgefangen.

3.6 Die Mehrfachverzweigung Switch

Eine Erweiterung der If-Verzweigung ist die Mehrfachverzweigung *Switch*. Während die If-Verzweigung immer nur 2 Fälle unterscheiden kann (wahr oder nicht wahr), können in einer Switch-Verzweigung beliebig viele Fäller untersucht werden. Betrachten wir die Syntax:

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

In dieser Verzweigung wird der Switch-Ausdruck untersucht. Dieser Ganzzahlenausdruck wird mit den folgenden Konstanten verglichen. Die direkt danach aufgeführten Anweisungen bis zum Bezeichner *break* werden ausgeführt. Wichtig ist noch, dass in einer Switch-Verzweigung die geschweiften Klammern immer notwendig sind.

Achtung:

➔ Der **Switch-Ausdruck** und die Konstanten müssen **Ganzzahlen** sein, also vom Datentyp short, int, long, long long oder char.

Betrachten wir gleich ein Beispiel. Wir zählen alle Ziffern und Whitespaces in einer beliebig langen Zeichenkette.

```
void zaehleZiffern(char* str)
{
   int i = 0;             /* Startwert in Zeichenkette */
   int zahl, white, sonst;
   zahl = white = sonst = 0;         /* init */
   while ( str[i] != '\0' )  /* Zeichenkette durchlaufen */
   {
      switch (str[i])
      {  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':    white++; break;
         default:                           sonst++;
      }
      i++;
   }
   printf("Zahlen = %d\nFreiraeume = %d\nSonstige = %d\n",
                  zahl, white, sonst);
}
```

Funktion *zaehleZiffern* in 03-kontroll\switch.c

Zu beachten ist folgendes:

– In einer Switch-Verzweigung wird zum ersten Auftreten der in Klammern stehenden Zahl gesprungen. Kommt die Zahl nicht vor, so wird zum Bezeichner ***Default*** gesprungen. Ist kein Bezeichner

Default vorhanden, so wird die Switch-Verzweigung beendet.

- Wurde ein Sprungziel gefunden, so wird der Switch-Block bis zum nächsten Bezeichner *Break* durchlaufen. Folgt kein Bezeichner *Break*, so werden alle Anweisungen bis zum Ende des Switch-Blocks abgearbeitet.
- Der Bezeichner ***Break*** bricht innerhalb eines Switch-Blocks diesen Block ab. Es wird mit der nächsten Anweisung nach der Switch-Verzweigung fortgefahren. Der Bezeichner *Break* ist auch innerhalb eines Schleifen-Blocks (*for*, *while*, *do-while*) erlaubt und bricht dann genau diese Schleife ab.
- Der Bezeichner ***continue*** ist nur innerhalb von Schleifen erlaubt. Im Gegensatz zum Bezeichner *Break* wird damit nicht die Schleife abgebrochen, sondern nur der aktuelle Schleifendurchlauf. Es wird sofort mit dem Überprüfen des nächsten Schleifenausdrucks fortgesetzt.

Im Programm *switch.c* wird in der Funktion *liesZeichenkette* vor dem Aufruf der Funktion *zaehle-Ziffern* eine Zeichenkette eingelesen. Dieses Einlesen ist mit Blick auf spätere Dateibearbeitung interessant und wird daher zusätzlich vorgestellt:

```
void liesZeichenkette(char* str)
{
   int c;               /* eingelesenes Zeichen */
   int anzahl = 0;      /* Zaehler */
   puts("Daten eingeben, mit STRG-Z beenden:");
   c = getchar();        /* Zeichen einlesen */
   while (c != EOF)
   {
      str[anzahl] = c;      /* in Zeichenkette speichern */
      anzahl++;
      c = getchar();        /* Zeichen einlesen */
   }
   str[anzahl] = '\0';      /* Ende der Zeichenkette!!! */
}
```

Funktion *liesZeichenkette* in 03-kontroll\switch.c

Beim Lesen von Konsole kann das Ende einer Datei simuliert werden, in Windows durch Drücken der Tastenkombination STRG-Z, in Unix meist durch STRG-D. Wir können damit Dateibearbeitung nachbilden. Wenn wir von einer Textdatei alles einlesen wollen, so lesen wir bis Dateiende. Dateiende heißt auf englisch *end of file*, abgekürzt ***EOF***. Diese Konstante ist in der Bibliothek *stdio.h* als Int-Zahl vordefiniert. Um damit auf das Dateiende abprüfen zu können, müssen wir mittels der Funktion *getchar* ein Zeichen einlesen, in einer Int-Variablen speichern und auf *EOF* überprüfen.

Dies haben wir auch in unserer Funktion durchgeführt. Nach dem Überprüfen wird das eingelesene Zeichen in einer Zeichenkette abgelegt, und das nächste Zeichen wird eingelesen. Dies erfolgt so lange, bis das Dateiende erreicht ist. Anschließend dürfen wir nicht vergessen, dass die Zeichenkette sauber mit dem Zeichen ‚\0' abgeschlossen wird. Natürlich muss in der aufrufenden Funktion sichergestellt sein, dass auch genügend Speicher für die Zeichenkette hinterlegt ist. Wir haben eine Zeichenkette mit 10000 Zeichen im Speicher reserviert. Dies sollte beim Lesen von der Konsole reichen.

4 Nützliche Erweiterungen in C++

Es gibt viele Dinge, die in C hervorragend gelöst sind. Es gibt aber auch einiges zu kritisieren. Und es gibt einiges, das in C++ einfach eleganter ist. Praktisch jeder C-Compiler übersetzt auch C++. Warum sollten wir die wirklich nützlichen Werkzeuge in C++ nicht nutzen?

Ab sofort werden wir generell einige Möglichkeiten von C++ verwenden. Somit enden unsere Programme nicht mehr mit der Endung *.c*, sondern mit der Endung *.cpp*. Damit wird dem Compiler mitge-

teilt, dass es sich um C++-Programme handelt.

Ab sofort werden wir intensiv die Kommentarklammer // verwenden. Damit entfällt das oft lästige Beenden des Kommentars. Mit der nächsten Zeile ist dieser Kommentar ja automatisch beendet.

4.1 Konstanten in C und C++

Wir hatten in mehreren Programmen bereits die Kreiszahl *pi* verwendet. Dies ist eine Konstante, die sich innerhalb des Programms nicht verändert. Bisher hatten wir geschrieben:

```
float PI = 3.14159;
```

Dies befriedigt nicht, da wir versehentlich diesen Wert im Programm ändern könnten, was zu unvorhergesehenen Effekten führen kann. Wir gehen lieber auf Nummer Sicher und teilen dem Compiler mit, dass dieser Wert PI nicht verändert werden darf. Dies geht in C mit der folgenden Präprozessoranweisung:

```
#define PI 3.14159
```

Ab dieser Zeile wird jeder Bezeichner *PI* vom Präprozessor durch den in der Zeile folgenden Wert ersetzt. Erst dann wird der Compiler das Programm übersetzen, der Wert 3.14159 ist also quasi eingebrannt. In der Praxis befriedigt diese Lösung aber nicht voll. Eine saubere Typüberprüfung ist so nicht möglich. Erst C++ erlaubt hier eine absolut zufriedenstellende Lösung:

```
const float PI = 3.14159;              // nur in C++
```

Jetzt weiß der Compiler, dass es sich hier um eine Float-Konstante mit einem bestimmten Wert handelt. Es ist ein Fehler, wenn versucht wird, diese Konstante zu ändern.

Es ist ein guter Programmierstil, sich selbst so weit wie möglich einzuschränken. Im Fehlerfall können dann viele Fehlermöglichkeiten von vornherein ausgeschlossen werden. Vom Bezeichner ***const*** sollte daher reger Gebrauch gemacht werden, insbesondere auch bei der Parameterübergabe. Als zusätzlicher Parameterbezeichner ist der Bezeichner *const* bereits in C erlaubt. Im Programm *switch.c* wäre es daher besser, die Funktionsdefinition von *zaehleZiffern* wie folgt zu erweitern:

```
void zaehleZiffern(const char* str)
```

Dies ist weiterhin C-Code. Der Inhalt der Zeichenkette kann nun innerhalb der Funktion nicht mehr geändert werden. Zu beachten ist, dass natürlich auch die Deklaration mit angepasst wird. Ab sofort werden wir von dieser Möglichkeit Gebrauch machen.

4.2 Boolesche Variablen

Indirekt verwendet C boolesche Variablen, es gibt auch die booleschen Operatoren &&, || und !. Doch das Ergebnis ist in C immer eine Ganzzahl, 1 bei *true* und 0 bei *false*. C++ geht hier einen deutlichen Schritt weiter. C++ führt einen eigenen booleschen Datentyp namens ***bool*** ein. Auch in C99 wurde ein entsprechender Typ eingeführt, aus Kompatibilitätsgründen heißt dieser Datentyp allerdings *_Bool*. Vom letzteren sehr gewöhnungsbedürftigen Typ werden wir aber keinen Gebrauch machen. Natürlich wendet C++ die booleschen Daten konsequent an. Das Ergebnis boolescher Operationen ist vom Typ *bool* und kein Ganzzahlwert. Aber natürlich beherrscht C++ die automatische Umwandlung des Datentypen *bool* in eine Ganzzahl. Auch hier wird *true* zu 1 und *false* zu 0. Umgekehrt wird die Zahl 0 zu *false* und alle ande-

ren Zahlen zu *true*.

Boolesche Variablen können oft die Logik eines Programms verbessern. Wir werden daher ab sofort davon Gebrauch machen.

4.3 Ein- und Ausgabeströme

Die Ein- und Ausgabe in C ist enorm leistungsfähig. Wir haben bisher von den Möglichkeiten dieser Ein- und Ausgabe kaum Gebrauch gemacht. Wir kennen nur die Funktionen *printf*, *scanf*, *getchar*, *gets*, *putchar* und *puts*. Die Funktionen *printf* und *scanf* erlauben viele Formatierungsmöglichkeiten, die kaum Wünsche offen lassen. Leider sind diese Funktionen zum Teil etwas antiquiert. Insbesondere gibt es viele Fallen, von denen bereits einige vorgestellt wurden.

Bei der Erstellung der Sprache C++ wurde festgestellt, dass diese Ein- und Ausgabefunktionen nicht auf das objektorientierte Konzept von C++ adaptiert werden können. Aus diesem Grund wurden völlig neue Wege gegangen. Es handelt sich bei der Ein- und Ausgabe um den Fluss von Datenströmen. In C++ sprechen wir daher von Ein- und Ausgabeströmen. Es gibt in C++ folgende drei Bibliotheken:

```
iostream                // Ein- und Ausgabe
fstream                 // Dateibearbeitung
iomanip                 // erweiterte Formatierung und Manipulation
```

Es versteht sich fast von selbst, dass die Formatierungs- und weiteren Möglichkeiten noch umfangreicher sind als in den C-Funktionen. Ein weiterer großer Vorteil ist die Typsicherheit bei der Ein- und Ausgabe. Die Ströme erkennen automatisch den Typ der einzulesenden oder auszugebenden Variable, auch ohne Formatierungszeichen. Bei der Eingabe ist auch nicht die Adresse einer Variable anzugeben.

Dies klingt alles sehr verlockend. Doch die Entscheidung, ab sofort Ströme zu verwenden, fiel nicht leicht, denn die Ströme bauen auf ein objektorientiertes Konzept auf. Genauer bestehen alle Ströme zusammen aus etwa 15 Klassen. Meist können wir diesen Klassenaufbau verbergen, an einigen Stellen blinzeln sie jedoch hervor. Wir müssen dies im Augenblick akzeptieren und für ein wirklich tiefes Verständnis der Ströme bis zum Arbeiten mit Objekten warten. Betrachten wir als erstes Beispiel die Konvertierung der Funktion *bearbeiteZeichenkette* im Programm *bibliothek.c* aus Kapitel 2:

```
void bearbeiteZeichenkette()
{
   char c;               // einzelnes Zeichen
   char s[10000];        // 10000 Speicherplaetze
   cout << "Bitte eine Zeile eingeben:" << endl;
   cin.getline(s,10000);         // liest Zeile ein
   cout << "Bitte geben Sie einen Grossbuchstaben ein: ";
   cin.get(c);                   // liest Zeichen ein
   cout << "Die Zeile lautet:" << endl << s << endl;
   cout << "Die Zeilenlaenge ist: " << strlen(s) << endl;
   cout << c << " als Kleinbuchstabe: "
                 << (char) tolower(c) << endl;
}
```

Funktion *bearbeiteZeichenkette* in 04-cplusplus\stream.cpp

In der Strombibliothek gibt es zwei wichtige Stromklassen:

```
ostream:        Ausgabestrom
istream:        Eingabestrom
```

Die obigen Ströme sind in der Bibliothek *iostream* definiert. Standardmäßig sind folgende Klassenobjekte 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

In der Regel sind die Standardausgabe und –fehlerausgabe der Bildschirm und die Standardeingabe die Tastatur. Wollen wir etwas auf Bildschirm ausgeben, so verwenden wir in der Regel den Ausgabestrom *cout* in Kombination mit dem Operator ‚<<' oder gefolgt von einem Punkt und einem Funktionsnamen. Analoges gilt für den Eingabestrom *cin*, nur dass hier der Operator ‚>>' verwendet wird.

Mit dem Ausgabestrom *cout* zusammen mit dem Operator ‚<<' können wir alle Variablentypen ausgeben: etwa Ganzzahlen, Gleitpunktzahlen oder Zeichenketten. Wir können nach der Ausgabe einer Variable oder Konstante durch Anfügen des Operators ‚<<' direkt die Ausgabe eines weiteren Wertes durchführen usw. Das Gleiche gilt sinngemäß auch für die Eingabe. Der Compiler überprüft automatisch den Datentyp und gibt genau gemäß dieses Datentyps aus, bzw. liest entsprechend ein.

Der im Ausgabestrom verwendete Bezeichner ***endl*** ist nichts Besonderes. Es ist eine Abkürzung für das Zeichen ‚\n'. Was wir verwenden, bleibt Geschmackssache.

Einige wichtige Funktionen in den beiden Stromklassen sind:

```
cin.get(zeichen)          // liest das nächste Zeichen ein
cin.get(str, anzahl)      // liest eine Zeile ein, ohne \n, max. anzahl Zeichen
cin.getline(str, anzahl)  // liest eine Zeile ein, überliest dann \n
cout.put(zeichen)         // gibt ein Zeichen aus
cout.put(str)             // gibt eine Zeichenkette aus
```

Angenehm ist, dass alle Eingabefunktionen *get* (bzw. *getline*) heißen und die Ausgabefunktionen *put*. Ist die Eingabe nicht möglich, etwa weil das Dateiende erreicht wird, so liefern die Funktionen den booleschen Wert *false* zurück, sonst *true*. Der Unterschied zwischen *get* und *getline* ist gering, aber oft sehr wichtig: Die *Funktion get(str, anzahl)* liest maximal *anzahl* Zeichen ein, hört aber spätestens bei Zeilenende mit dem Einlesen auf. Das Zeilenendezeichen wird nicht mit eingelesen. Die Funktion *getline* arbeitet analog, nur dass das Zeilenende überlesen wird. Dies heißt: Das Zeileende wird nicht mit in die Zeichenkette geschrieben. Die nächste Eingabe wird aber nach dem Zeilenende fortgesetzt! Es ist gut zu erkennen, dass mit diesen Funktionen nicht aus Versehen zu lange Zeichenketten eingelesen werden können. Mittels des zweiten Parameters können wir sicherstellen, dass die Zeichen auch in den zur Verfügung gestellten Speicher passen.

Wichtig ist, dass bei der Eingabe mittels des Operators ‚>>' immer zunächst alle Whitespaces überlesen werden. Daher ist die Eingabe von Zeichen mit diesem Operator nur sinnvoll, wenn Leerzeichen nicht gelesen werden sollen. Weiter endet die Eingabe mit dem nächsten Whitespace. Aus diesem Grund kann mit diesem Operator keine ganze Zeile, sondern nur das nächste Wort eingelesen werden.

Die Funktion *manipuliereZahlen* möge bitte selbst studiert werden. Das Include der Bibliothek *stdio.h* entfällt, dafür kommt *iostream* hinzu. Moderne C++-Compiler ermöglichen das Hinzufügen zahlreicher Bibliotheken mit womöglich gleichen Funktionen. Um Verwechslungen zu vermeiden wird in C++ ein Namensraum hinzugefügt. Wir verwenden den Standard-Namensraum, müssen dies aber durch folgenden Befehl am Anfang des Programms kund tun:

```
using namespace std;
```

Die obige Funktion *bearbeiteZeichenkette* enthält leider eine kleine Falle, wofür nicht die Ströme, sondern die C-Zeichenverarbeitung schuld ist: Die Funktion *tolower* liefert eine Ganzzahl zurück. Bei der Ausgabe wird daher eine Zahl, hier der ASCII-Code, zurückgegeben. Damit das Zeichen ausgegeben wird, müssen wir explizit einen Cast-Operator verwenden. Umgekehrt können wir mit einem Cast nach Int erreichen, dass der Code eines Zeichens ausgegeben wird.

Häufig wollen wir unsere Daten formatiert ausgeben. Geldbeträge sollen beispielsweise auf zwei Stellen nach dem Komma gerundet werden. Oft ist auch rechtsbündige Ausgabe sinnvoll. Im kaufmänni-

schen Bereich wird oft gewünscht, dass bei positiven Zahlen auch das positive Vorzeichen ‚+' mit ausgegeben wird. Die Wunschliste ist fast endlos. C++ erfüllt fast alle diese Wünsche. Wir geben hier eine fast vollständige Liste der Formatierungsmöglichkeiten an. Aber keine Angst, nicht alle Funktionen brauchen wir gleich am Anfang:

```
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 bei Gleitpunkt-
                        // zahlen, gültig bis zur nächsten Änderung!
cout.fill (c)           // Das Zeichen c 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
```

Die Funktion *width* dient zur rechtsbündigen Ausgabe, da die folgende Ausgabe mit der entsprechenden Breite *n* rechtsbündig ausgegeben wird. In der Regel wird links mit Leerzeichen aufgefüllt. Mit der Funktion *fill* lässt sich dies aber ändern. Eine Ausgabe erfolgt immer vollständig! Ist die Breite kleiner als die Anzahl der Zeichen, die zur Darstellung der Ausgabe benötigt wird, so wird die Breite ignoriert und die Ausgabe komplett durchgeführt. Die Anzahl der Nachkommastellen können wir mit der Funktion *precision* dauerhaft ändern.

Für spezielle Ausgabemanipulationen dienen sogenannte Flags, die mit Hilfe der Funktionen *setf* und *unsetf* gesetzt werden können. Die einzelnen Flags sind in der Stromklasse *ios* als (statische) Konstanten definiert und heißen:

ios::left	Linksbündiges Ausrichten der Ausgabe
ios::right	Rechtsbündiges Ausrichten der Ausgabe (Standardvorgabe)
ios::internal	Die Füllzeichen stehen zwischen Vorzeichen und Zahl
ios::dec	Dezimaldarstellung von Ganzzahlen (Standardvorgabe)
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

Der Operator ‚::' ist neu in C++ und wird ausschließlich in Klassen verwendet. Dieser Operator besagt hier, dass die angegebenen Flags sogenannte statische Konstanten der Klasse *ios* sind. Manche dieser Flags widersprechen sich und sollten daher auf keinen Fall gleichzeitig gesetzt werden, etwa *ios::left* und *ios::right*. Um genau dies zu verhindern, sind noch drei weitere 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
```

Flags sind einzelne Bits einer Bitleiste. Werden zwei oder mehrere dieser einzelnen Bits mit dem Bitoperator *Oder* verknüpft, so sind alle diese Bits gesetzt. Diese drei Konstanten sind daher nur eine Abkürzung für zwei bzw. drei dieser Bits. Beim Setzen eines dieser acht Flags sollte daher sicherheitshalber sichergestellt sein, dass alle anderen bisher gesetzten Flags der gleichen Gruppe zurückgesetzt werden. Sicherheitshalber setzen wir alle Bits der Gruppe zurück, denn das Zurücksetzen eines nicht gesetzten Bits wird ignoriert. Wollen wir etwa linksbündige Ausrichtung setzen, so wird folgende Anweisung empfohlen:

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

Die Flags *showpos*, *uppercase*, *showpoint* und *showbase* hingegen sollten einzeln gesetzt und auch

wieder zurückgesetzt werden. Mit

```
cout.setf(ios::showpos);
cout.unsetf(ios::showpoint);
```

werden dann positive Vorzeichen mit ausgegeben, ein Anzeigen von Dezimalpunkten erfolgt nun nicht mehr bei Gleitpunktzahlen ohne Nachkommastellen.

Meist ist es lästig, zwischen all den 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', '\r', '\v'), einziger Eingabemanipulator
setw (n)	entspricht der Funktion cout.width (n)
setfill (ch)	entspricht der Funktion cout.fill (ch)
setprecision (n)	entspricht der Funktion cout.precision (n)

Die Manipulatoren *setw*, *setfill* und *setprecision* benötigen die Bibliothek *iomanip*. Als einfaches Anwendungsbeispiel betrachten wir unsere Ausgabe eines Sternenbaums. Mit Hilfe der Breite und einer Zeichenkettenverarbeitung benötigen wir nur eine einzige Schleife:

```
void zeichneSternenbaum(int anzahl)
{
   int i;                    // Indexe deklarieren
   char sterne[1000] = "*";  // nimmt Sterne auf
   for (i=1; i <= anzahl; i++)
   {
       cout.width(anzahl+i);       // Breite
       cout << sterne << endl;     // Ausgabe der Zeile
       strcat(sterne, "**");       // Zwei Sterne hinzu
   }
}
```

Funktion *zeichneSternenbaum* in 04-cplusplus\sternenbaum3.cpp

Es ist zu beachten, dass jede Zeile ein Zeichen mehr enthält, der Baum wächst ja nach rechts. Entsprechend muss auch die Breite gesetzt werden (*anzahl + i*). Mit der Funktion *strcat* werden nach jeder Zeile zwei Sterne der Zeichenkette hinzugefügt. Die Zeichenkette wurde entsprechend groß deklariert und gleich mit einem Stern vordefiniert.

4.4 Erweiterte Möglichkeiten mit Variablen in C++

Es sei auf zwei Erweiterungen in C++ hingewiesen. Zum ersten dürfen in C++ an beliebigen Stellen Variablen deklariert werden, nicht nur am Anfang eines Blocks! Zum zweiten darf auch im For-Schleifenkopf eine Variable deklariert werden. Diese Variable ist dann allerdings nach der Schleife nicht mehr zugreifbar. In der obigen Funktion *zeichneSternenbaum* hätten wir also auch schreiben dürfen:

```
for (int i=1; i <= anzahl; i++) …
```

Wir können damit die Deklaration *‚int i;'* am Anfang der Funktion einsparen. Zusätzlich wird dadurch klar, dass die Variable *i* nur in der Schleife als Schleifenzähler benötigt wird.

4.5 Call by Reference

Funktionsaufrufe in C folgen einer ganz bestimmten Philosophie. Wir übergeben Werte an eine Funktion. Die Funktion speichert diese Werte als Kopie in den Parametern ab und rechnet dann mit diesen Parametern, also mit den Kopien weiter. Diese Parameter dürfen jederzeit modifiziert werden und wirken sich nicht auf das aufrufende Programm auf. Betrachten wir dazu eine etwas komplexere Funktion, die ***Fakultät*** einer nicht negativen Zahl. Die Fakultät *n!* einer nicht negativen ganzen Zahl ist definiert durch:

$$n! = \begin{cases} 1 & \text{für} \quad n = 0 \\ \prod_{k=1}^{n} k & \text{für} \quad n \geq 1 \end{cases}$$

Das Produktzeichen $\prod$ ist als das Produkt über alle Elemente definiert, wobei *k* von 1 bis *n* läuft. Es wird also nicht wie beim Summenzeichen aufsummiert, sondern multipliziert. Es gilt:

1!=1, 2!=2, 3! = 6, 4!=24, 5!=120, … 20!=2432902008176640000 …

Diese Funktion wollen wir nun implementieren. Wir müssen in einer Schleife einfach den Schleifenzähler zu dem bisherigen Produkt hinzumultiplizieren. Es ist zu beachten, dass das Produkt nicht mit Null sondern der Zahl 1 beginnt!

```
long long faku(int n)
{
   // berechnet die Fakultaet einer Zahl: n!
   long long erg = 1;       // Startwert
   for (int i=1; i<=n; i++)
      erg *= i;
   return erg;
}
```

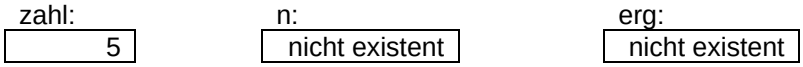
Funktion *faku* in 04-cplusplus\fakultät.cpp

Wir verwenden hier den Datentyp *long long*, da die Fakultät schnell sehr sehr große Zahlen liefert. Wir belegen die Variable *erg* mit 1 vor. Anschließend multiplizieren wir in der Schleife den Wert 1, dann 2, dann 3 usw. dazu. Wir beachten, dass diese Funktion auch für *n=0* den richtigen Wert 1 zurückliefert. Die Funktion beklagt sich auch nicht bei fehlerhafter negativer Eingabe. Hier wird stattdessen ebenfalls der Wert 1 zurückgegeben.

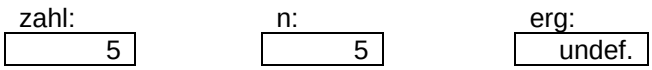
In unserem Hauptprogramm testen wir diese Fakultätsfunktion durch die drei Aufrufe:

faku(zahl) faku(7) faku(zahl+1)

Nehmen wir an, der Speicherplatz *zahl* enthält den Wert 5. Dann gilt für den Speicher vor dem ersten Aufruf der Fakultätsfunktion:



Der Parameter *n* und die in der Funktion *faku* definierte Variable *erg* existieren im Hauptprogramm noch nicht. Erst durch den Aufruf wird im Arbeitsspeicher ein Speicherplatz dynamisch zugewiesen. Mit dem Aufruf werden die Speicherplätze für *n* und *erg* angelegt. Direkt danach wird der Wert der Zahl in den Parameter *n* kopiert. Es ergibt sich direkt nach dem Aufruf der Funktion folgendes Bild:



Anschließend wird in der Funktion die Variable *erg* auf 1 gesetzt und schließlich die Fakultät berechnet. Mit dem Beenden der Funktion wird das Funktionsergebnis an den aufrufenden Ausdruck zurückgegeben, der Speicherplatz für *n* und *erg* wird wieder frei gegeben.

Genauso funktioniert der Aufruf auch für *faku(7)*. Die Zahl 7 ist zwar eine Konstante, stört aber nicht. Diese Zahl wird in den Parameter *n* kopiert. Mit dem Wert *n=7* wird dann weiter gerechnet. Auch der Aufruf *faku(zahl+1)* klappt. Der berechnete Übergabewert 6 wird einfach in den Parameter *n* kopiert.

Diese Art der Parameterübergabe wird ***Werteübergabe*** oder ***call by value*** genannt. Entscheidend ist immer, dass der übergebende Wert in den Parameter kopiert wird. Ändern wir beispielsweise den Parameterwert, so wirkt sich dies nicht auf den übergebenden Wert aus! Wir spüren dies sehr negativ, wenn wir mit Hilfe einer Funktion versuchen, die Inhalte zweier Zahlen zu vertauschen:

```
void vertausche1(int a, int b)
{
   int hilf = a;
   a = b;
   b = hilf;
}
```

Funktion *vertausche1* in 04-cplusplus\vertausche.cpp

Diese Funktion *vertausche1* vertauscht tatsächlich die beiden Parameter *a* und *b*. Aber dies geschieht natürlich nur innerhalb der Funktion *vertausche1*. Auf die beiden Werte *zahl1* und *zahl2* beim Aufruf

```
vertausche(zahl1, zahl2);
```

hat dies aber keinerlei Einfluss. Wir hatten ja mit Kopien *a* und *b* gearbeitet. Dies mag deutlich werden, wenn wir wieder die Speicherbelegung direkt nach dem Aufruf der Funktion betrachten, wobei die Zahlen *zahl1* und *zahl2* die Werte 5 und 10 belegen mögen:

zahl1:	zahl2:	a:	b:
5	10	5	10

Beim Beenden der Funktion ist tatsächlich etwas geschehen:

zahl1:	zahl2:	a:	b:
5	10	10	5

Doch wenige Mikrosekunden später wird der Speicherplatz für *a* und *b* freigegeben. Für die aufrufenden Variablen hat sich nichts geändert. Dies ist ein Beispiel einer absolut wirkungslosen Funktion. Doch wie können wir nun unsere Variableninhalte vertauschen? Dies geht in C nur mit Zeigern, in C++ hingegen wurde ein ***Referenzoperator &*** eingeführt, der uns enorm weiterhilft. Betrachten wir gleich die folgende Funktion *vertausche2*, die sich nur durch das Hinzufügen des Referenzoperators & bei den beiden Parametern unterscheidet:

```
void vertausche2(int& a, int& b)
{
   int hilf = a;
   a = b;
   b = hilf;
}
```

Funktion *vertausche2* in 04-cplusplus\vertausche.cpp

Liegt eine Referenz vor, so wird keine Kopie übergeben, sondern der Parameter identifiziert sich mit dem gleichen Speicherplatz wie die übergebene Variable. Betrachten wir daher wieder die Speicherbelegung direkt nach dem Funktionsaufruf:

Es gibt tatsächlich keinen eigenen Speicherplatz für *a* und *b*. Sowohl *a* als auch *zahl1* identifizieren den gleichen Speicher. Das Gleiche gilt für *b* und *zahl2*. Nun wird natürlich schnell klar, dass sich das Vertauschen der Inhalte von *a* und *b* auch auf den Inhalt der identischen Speicherplätze von *zahl1* und *zahl2* auswirken wird. Direkt vor dem Beenden der Funktion haben wir dann folgendes Bild:

Nach dem Beenden werden die Parameter *a* und *b* entfernt, genauer die Referenzen. Die Variablen *zahl1* und *zahl2* bleiben mit verändertem Inhalt natürlich erhalten.

Diese Parameterübergabe mittels Referenzen heißt ***Referenzübergabe*** oder ***call by reference***. Diese Referenzübergabe ist zusätzlich sehr performant, da ja kein Speicherplatz für die Variablen *a* und *b* angelegt werden muss, und da auch kein Kopieren erforderlich ist. Es genügt, eine Referenz zu setzen. Die Referenzübergabe besitzt aber auch einen kleinen Nachteil, auf den unbedingt zu achten ist:

Achtung:

➔ Beim Aufruf einer Funktion mit **Referenzübergabe** müssen **Variablen** übergeben werden. Konstanten oder Ausdrücke sind als Übergabewerte nicht möglich.

Es versteht sich fast von selbst, dass der Aufruf *vertausche2(7, 9)* Unsinn ist. Hier müssen tatsächlich immer Variablen übergeben werden. Auch ein Aufruf *vertausche2(zahl1+5, zahl2)* ist nicht möglich, da der Ausdruck *zahl1+5* keinen Speicher belegt und die Referenz *a* daher nicht gesetzt werden kann.

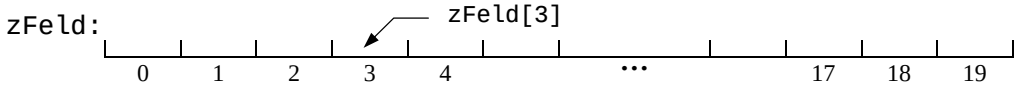
5 Felder und Zeiger

5.1 Felder

In der Datenverarbeitung sind selten nur ein paar Variablen zu bearbeiten, meist liegen riesige Datenmengen vor. Die einfachste Form mit großen Datenmengen zu arbeiten sind **Felder**. Betrachten wir gleich die Definition eines Feldes von 20 Ganzzahlen:

```
int  zFeld [ 20 ] ;
```

Mit dieser Definition wird folgender Speicher angelegt:



Wir haben damit in einem einzigen Schritt 20x4=80 Bytes an Speicher belegt. Auf diesen Speicher können wir zugreifen. Mit dem Namen *zFeld* ist das gesamte Feld angesprochen. Durch die zusätzliche Angabe eines Index können wir aber auch auf jedes einzelne Element gezielt zugreifen, etwa mittels *zFeld[3]*.

In C und C++ beginnt der Feldindex immer bei 0, das erste Feldelement ist demnach *zFeld[0]*, das letzte Feldelement wird mit *zFeld[19]* angesprochen! Ein Zugriff auf das nicht mehr existierende Element *zFeld[20]* ist daher ein Fehler! Die Syntax zur Definition eines festen Feldes ist:

```
Datentyp  Feldname  [ Ganzzahlenausdruck ]
```

Hier sind die eckigen Klammern Teil der Syntax. Es kann zu jedem beliebigen Datentyp ein Feld erzeugt werden. Die Anzahl der Feldelement darf ein arithmetischer Ausdruck sein, der in dieser Zeile berechnet wird. Das somit erzeugte statische Feld besitzt immer eine fixe unveränderliche Länge, die mit dem Operator *sizeof* auch jederzeit abgefragt werden kann. Der Inhalt eines einzelnen Feldelements kann beispielsweise wie folgt geändert oder abgefragt werden:

```
zFeld[0] = 1;
for ( int i=1 ; i<20 ; i++ )
    zFeld[i] = zFeld[i-1] + 2;                    // oder: zFeld[i] = 2*i+1;
```

Mit dieser Schleife werden den 20 Elementen des Feldes *zFeld* die ersten 20 ungeraden Zahlen zugewiesen. Bitte beachten Sie dringend, dass die Schleife beim Element Nummer 19 endet! Es bieten sich zum Beenden der Schleife zwei Möglichkeiten an:

```
for ( int i=1 ;  i=<19 ; i++ )
for ( int i=1 ;  i<20 ; i++ )              // Identifikation mit der Feldlaenge 20
```

Nur die zweite Zeile identifiziert sich mit der tatsächlichen Länge des Feldes. Es wird daher dringend empfohlen, diese Programmierweise zu verwenden! Wir bezeichnen diese Art der Programmierung auch als Verwendung ***asymmetrischer Grenzen***! Wollen wir etwa ein Feld mit *n=2500* Elementen bearbeiten, so könnte die Schleife dazu wie folgt beginnen:

```
const int N = 2500;            // gemäß Namenskonvention schreiben wir Konstanten groß!
…
for ( int i=0 ; i<N ; i++) …
```

Wer die Schreibarbeit nicht scheut, kann ein Feld auch bei der Deklaration vorbelegen. Die ersten 20 ungeraden Zahlen hätten wir in unser Feld *zFeld* auch gleich bei der Deklaration angeben können:

```
int zFeld[20] = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39 };
```

Dieses Feld können wir natürlich jetzt auch ausgeben:

```
for ( i=0 ; i<20 ; i++ )                      // nicht:  i <= 19
{   cout << setw(4) << zFeld[ i ];
    if ( i%10 == 9 )                          // Zeilenumbruch nach je 10 Ausgaben
        cout << endl;
}
```

Hier machten wir uns einige Mühe zur formatierten Ausgabe. Zunächst formatieren wir alle Ausgaben mit der Breite 4. Weiter kommt nach jeweils 10 Ausgaben ein Zeilenvorschub. Dabei setzen wir den Modulo-Operator gewinnbringend ein. Dieser Algorithmus würde auch bei Feldern mit einer Feldlänge größer als 20 funktionieren.

5.2 Sortieren mit Bubblesort

Große Felder können schnell unübersichtlich werden. Gerne sortieren wir daher solche Felder. Wir wollen dies gleich einmal selbst versuchen. Wir lesen Zahlen in beliebiger Reihenfolge in ein Feld ein und geben diese Zahlen sortiert wieder aus. Das Sortieren übernimmt eine selbstdefinierte Funktion *sort*. Das Hauptprogramm können wir schon schreiben:

```
int main()
{  const int N = 20;      // Feldgroesse
   int zaehler = 0;
   int zFeld[N];    // Zahlenfeld

   cout << "\n\nSortierung von Zahlen\n\n"
        << "Geben Sie Ganzzahlen ein. Abbruch: 0\n";
   do
   {  cout << "Zahl " << zaehler+1 << ": ";
      cin >> zFeld[zaehler];
   } while ( zFeld[zaehler] != 0 && ++zaehler < N );

   sort (zFeld, zaehler);          // sortiert das Feld

   cout << "Sortierte Ausgabe:\n";
   for (int i=0; i<zaehler; i++)
      cout << zFeld[i] << endl;
   system("PAUSE");
   return 0;
}
```

Hauptprogramm von 05-felder\sort.cpp

Das Hauptprogramm besteht im Wesentlichen aus drei Teilen, dem Einlesen, dem Sortieraufruf und dem Ausgeben. Wieder haben wir bei der Ausgabe asymmetrische Grenzen verwendet. Die Eingabe ist kurz aber trickreich. Es sollen maximal *N* Zahlen eingelesen werden. In der Variablen *zaehler* soll die Anzahl der eingelesenen Zahlen gespeichert werden. Natürlich soll die zum Abbruch führende Zahl 0 nicht berücksichtigt werden. Wir könnten also nach der Schleife die Variable *zaehler* wieder um eins reduzieren. Stattdessen wurden hier die Möglichkeiten von C voll ausgenützt: In der Do-While-Schleife wird zunächst überprüft, ob die eingelesene Zahl ungleich Null ist. Wenn ja, so wird die Schleife beendet, ohne dass das zweite Kriterium noch überprüft wird. Aber erst im zweiten Kriterium wird der Zähler erhöht! Ist die eingelesene Zahl ungleich Null, so wird das zweite Kriterium noch ausgeführt, der Zähler wird daher erhöht und erst dann wird überprüft, ob das Feld schon voll ist. Es sei als Übung dringend empfohlen, dieses Abbruchkriterium genau nachzuvollziehen.

Das Gerüst der Ein- und Ausgabe, also das Hauptprogramm, steht. Wir befassen uns jetzt mit dem Sortieren. Auf folgenden Algorithmus zum Sortieren von Zahlen kommen wir ganz schnell:

> Suche das kleinste Element und setze es an die erste Stelle
> Suche das kleinste Element aus dem verbliebenen Feld und setze es an die zweite Stelle
> Suche das kleinste Element aus dem verbliebenen Feld und setze es an die dritte Stelle
> usw.

Dieser Algorithmus ist nicht besonders effizient und wird daher in der Praxis nicht angewendet. Ein anderer Algorithmus erledigt bei der Suche des kleinsten Elements gleich noch ein paar „Zusatzarbeiten“, so dass dieser Algorithmus merklich besser abschneidet und auch heute bei kleineren Feldern mit meist nicht mehr als 10000 Elementen gerne angewendet wird. Dieser Algorithmus heißt **Bubblesort**. Betrachten wir seine Funktionsweise an einem Beispiel:

Funktionsweise:

unsortiert	1. Schritt	2. Schritt	3. Schritt	4. Schritt	5. Schritt
17	3	3	3	3	3
→ 3	17	5	5	5	5
19	→ 5	17	7	7	7
7	19	→ 7	17	12	12
12	→ 7	19	→ 12	17	13
13	12	→ 12	19	→ 13	17
→ 5	13	13	→ 13	19	19

Das Prinzip des Bubblesort wurde hier an einem sehr einfachen Beispiel dargestellt. Links sehen wir die zu sortierenden Zahlen untereinander aufgereiht. Schritt für Schritt werden diese Zahlen sortiert. Zum Verständnis des Algorithmus stellen wir uns die Zahlen als Blasen (engl.: bubble) vor, wobei kleinere Zahlen leichter sind als große. Wir beginnen bei der letzten Zahl unten (hier: 5) und stoßen diese an. Da diese recht leicht ist, wird sie so lange nach oben wandern, bis sie auf eine noch leichtere stößt (hier: 3). Die Zahl bleibt stehen, während die leichtere die Wanderung nach oben fortsetzt. Das Ergebnis dieses ersten Durchlaufs ist in der zweiten Spalte zu finden. Wir setzen dieses Verfahren so lange fort, bis alle Zahlen sortiert sind. In unserem kleinen Beispiel benötigen wir dazu 5 Schritte. Zum leichteren Verständnis sind alle Zahlen, die im jeweiligen Schritt angestoßen werden, mit einem Pfeil gekennzeichnet.

Es fällt auf, dass wir nach dem ersten Schritt das Minimum korrekt platziert haben, nach dem zweiten Schritt sind die zwei kleinsten Zahlen an der richtigen Position, bei *n* Zahlen sind nach *n-1* Schritten die ersten *n-1* Zahlen korrekt positioniert. Damit muss aber auch die *n*-te Zahl an der richtigen letzten Stelle stehen. Nach maximal *n-1* Schritten sind folglich *n* Zahlen sortiert. In unserem Beispiel reichten bei 7 Zahlen sogar 5 statt der maximal 6 Schritte aus.

Formulieren wir dieses Vorgehen für *n* Zahlen exakt:

Algorithmus:

```
Für i von 1 bis <n                                  // also n-1 Schritte
   Für j von n-1 rückwärts bis i                    // Optimieren!
      Wenn j-tes Element < j-1-tes Element, dann
         Vertausche diese beiden Elemente
```

Es sei dringend als Übung empfohlen, den Algorithmus genau nachzuvollziehen. Beispielsweise startet die äußere Schleife nicht bei *i=0*, sondern bei *i=1*. In der inneren Schleife nimmt die Zahl *j* nämlich auch den Wert *i* an. Für *j=i=0* würde aber dann kein *j-1*-tes Element existieren! Dieser Algorithmus wird nun eins zu eins in unsere Funktion *sort* umgesetzt:

```
void sort (int feld[], int n)
{
   for (int i=1 ; i<n ; i++)
      for (int j=n-1 ; j>=i ; j--)
         if (feld[j] < feld[j-1])
         {                                  // vertausche:
            int var    = feld [j-1];
            feld [j-1] = feld [j];
            feld [j]   = var;
         }
}
```

Funktion *sort* in 05-felder\sort.cpp

Zum Vertauschen wird eine Hilfsvariable *var* zum Zwischenspeichern des Inhalts einer der beiden Feldinhalte benötigt. Für uns neu ist die Übergabe eines ganzen Feldes als Parameter. Im Hauptprogramm schreiben wir dazu folgenden Funktionsaufruf:

```
sort (zFeld, zaehler);        // sortiert das Feld
```

Wie wir sehen, übergeben wir kurzerhand das gesamte Feld *zFeld* als ersten Parameter. Da wir von vornherein nicht wissen, wie viele Elemente des Feldes belegt sind, übergeben wir als zweiten Parameter auch die Variable *zaehler*, in der die Anzahl der eingelesenen Zahlen gespeichert ist. Diese zweite Variable ist vom Datentyp *int*. Ganzzahlen hatten wir bereits vorher als Parameter übergeben. Von welchem Datentyp ist aber das Feld *zFeld*?

Wir haben im Hauptprogramm definiert:

```
int zFeld [ N ] ;
```

Wir könnten daher folgenden Funktionskopf verwenden:

```
void sort (int feld [ N ], int n)
```

mit den beiden Parameternamen *feld* und *n*. Wir wir noch sehen werden, sind in C Feldnamen aber nichts weiter als Verweise auf Speicherbereiche. Dies hat für uns zwei Vorteile:

(1) Die **Größe des Feldes** spielt bei der Übergabe eines Feldes an eine Funktion keine Rolle und wird von C grundsätzlich ignoriert. Wir lassen daher die Feldgrößenangabe, hier *N*, einfach weg.

(2) Da Feldnamen nur **Verweise** sind, können die Inhalte jederzeit geändert werden. Der Referenzoperator ist daher bei der Übergabe von Feldern an Funktionen nicht notwendig.

Der in der Funktion *sort* verwendete Funktionskopf sollte damit erklärt sein, doch wie sieht es mit der Deklaration dieser Funktion aus? Auch dies fällt nicht schwer. Wir wissen bereits, dass bei der Deklaration Parameternamen nicht angegeben werden müssen. Dies setzen wir nun konsequent um und erhalten als Deklaration:

```
void sort (int [ ], int);
```

5.3 Felder mit Zeigern bearbeiten

In vielen Programmiersprachen wären damit Felder hinreichend behandelt. Dem ist nicht so in C. Die Sprache C erlaubt einen Zugang zu Feldern mittels Zeiger, ja Felder selbst sind nichts anderes als Zeiger. Damit steigen die Möglichkeiten des Zugriffs auf Felder enorm. Dies erfordert allerdings auch fundierte Kenntnisse zu Zeigern.

Definition:

Ein ***Zeiger*** ist ein Verweis auf eine Variable (genauer: auf einen Speicherplatz) mit einem fest vorgegebenen Datentyp.

Wir sprechen hier von ***indirekter Adressierung***. Wir greifen nicht direkt auf eine Variable zu, sondern indirekt über die bekannte Adresse. Beginnen wir gleich mit einem Beispiel, in dem wir den Unterschied zwischen einer Variable und dem Zeiger (engl.: pointer) auf eine Variable herausarbeiten:

```
int i;                                   i:
int *p_i;          p_i = &i;             p_i:

i = 4;             // in i wird der Wert 4 gespeichert
*p_i = 5;          // in i wird der Wert 5 gespeichert !!!
```

Wie bekannt belegt die Variable *i* einen Speicherplatz von vermutlich 4 Byte Größe. Die Variable *p_i* (pointer to *i*) ist ein Zeiger und nimmt eine Adresse auf, vermutlich ebenfalls 4 Byte. Mit der folgenden Zuweisung (*p_i = &i;*) wird die Adresse von *i (&i*) in der Variablen *p_i* abgelegt. Wir sprechen davon, dass diese Zeigervariable auf die Variable *i* zeigt (siehe obige Skizze).

Definition:

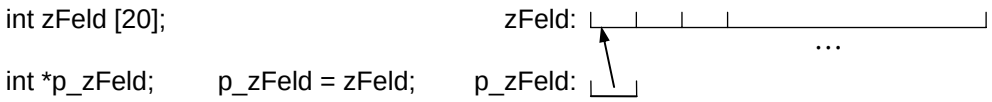
Der Operator & heißt ***Adressoperator*** und liefert die Speicheradresse zu einer Variablen.

Wir können nun direkt einen Wert in der Variable i ablegen (*i=4*) oder indirekt über die Adresse (**p_i=5*). Ist *p_i* ein Zeiger, und zeigt *p_i* auf eine definierte Adresse, so markiert **p_i* den Inhalt, auf den *p_i* zeigt. Mit dieser letzten Zuweisung wird demnach in die Variable, auf die *p_i* zeigt, der Wert 5 eingetragen. Dies ist hier die Variable *i*!

Definition:

Der Operator * heißt ***Dereferenzierungsoperator*** und liefert zu einer Speicheradresse den dazugehörigen Inhalt.

Erweitern wir dieses Wissen auf Felder:



Zunächst gilt für Felder das Gleiche wie für Zahlen. Ein Zeiger kann auch auf ein Feld verweisen. Wichtig ist nur, dass die grundlegenden Datentypen (hier: *int*) übereinstimmen. Allerdings sind Felder praktisch nichts anderes als Zeiger, so auch die Feldvariable *zFeld*. Wir schreiben daher direkt: *p_zFeld = zFeld*, ohne Verwendung des Adressoperators &.

Für die Speicherbelegung gilt (in 32Bit-Systemen):

```
sizeof zFeld:     80        // Speicher für das Feld
sizeof i:          4        // Speicher für die int-Variable
sizeof p_i:        4        // Speicher für die Adresse
sizeof p_zFeld:    4        // nur Speicher für die Adresse
```

Begeben wir uns jetzt in die Tiefen von C:

Um das gesamte Zahlenfeld zu kennen, muss C eigentlich nur die Adresse des Beginns des Feldes und die Größe des Feldes wissen. Entsprechend ist in C die Variable *zFeld* auch als konstante unveränderliche Adresse des Feldes implementiert, die zusätzlich die Feldgröße merkt. Da nun *zFeld* eine Adresse ist, so ist sie mit *&i* oder *p_i* oder *p_zFeld* zu vergleichen. Folglich müsste **zFeld* den Inhalt des ersten Elements des Feldes repräsentieren. Es ist daher nur konsequent, dass in C gilt:

```
*zFeld  ≡  zFeld[0]          // beide Ausdrücke sind äquivalent
```

Definition:

Der deklarierte ***Name eines Feldes*** ist eine nicht veränderbare Adresse und wird als ***Feldzeiger*** bezeichnet. Mit der Deklaration eines Feldes wird ein entsprechend großer zusammenhängender Speicherbereich reserviert. Mit dem sizeof-Operator kann die Länge des Feldes abgefragt werden.

Zeiger auf Felder (hier: *p_zFeld*) und Feldzeiger (hier: *zFeld*) haben gemeinsam, dass sie Zeiger sind. Damit können wir mit Hilfe der Dereferenzierung (hier: **p_zFeld* und **zFeld*) auf das erste Element des Feldes zugreifen. Beide unterscheiden sich aber auch: der Feldzeiger repräsentiert einen ganz bestimmten, fest reservierten Speicherbereich. Der Feldzeiger ist daher unveränderlich. Der allgemeine Zeiger hingegen kann jederzeit geändert werden. Erst mit dem Verweis auf eine existierende Adresse darf bei einem allgemeinen Zeiger dereferenziert werden.

Wenn jetzt Zeiger und Feldzeiger so ähnlich sind, fragen wir uns, ob Zeiger auch Eigenschaften von Feldzeigern besitzen. Können wir also auch mit echten Zeigern auf weitere Inhalte zugreifen? Im obigen Beispiel zeigt der Zeiger *p_zFeld* auf das erste Element des Feldes *zFeld*. Tatsächlich verhält sich damit *p_zFeld* wie das Feld *zFeld* selbst, insbesondere gilt für jeden allgemeinen Zeiger *p*:

```
* p   ≡   p [ 0 ]                    // beide Ausdrücke sind äquivalent
```

Zeigt der Zeiger *p* auf ein Feld, so kann auch beliebig indiziert werden, etwa mittels *p[17]* oder *p[i]*. Dieses Indizieren wird in C sogar noch kräftig erweitert: C besitzt eine umfangreiche Zeigerarithmetik. Zu einem Zeiger darf eine Ganzzahl addiert oder subtrahiert werden, ebenso dürfen zwei Zeiger voneinander subtrahiert oder miteinander verglichen werden. Es gilt:

Zeigerarithmetik:

Addition (Subtraktion) eines Zeigers p mit einer Ganzzahl i
Das Ergebnis ist ein Zeiger: die Adresse von p, addiert (subtrahiert) um *i* Feldelemente, also die Adresse des Elements p[i].

Subtraktion zweier Zeiger
Das Ergebnis ist die Anzahl der Feldelemente zwischen beiden Adressen. Beide Zeiger müssen vom gleichen Datentyp sein.

Vergleich von Zeigern
Das Ergebnis ist *true* oder *false* bzw. 1 oder 0. Alle Vergleichsoperatoren sind erlaubt. Ein Zeiger besitzt einen um so größeren Wert, je höher die Adresse ist.

Zu beachten ist, dass die Zeigerarithmetik stark vom Datentyp abhängt. Ein Zeiger ist nicht nur eine Adresse, sondern auch fest mit einem bestimmten Datentyp verbunden. In unserem Beispiel wird daher *zfeld+1* eine Adresse sein, die um 4 höher ist als *zfeld*! Die Adresse *zfeld+1* verweist ja auf das nächste Feldelement, und der Speicherbedarf für eine Int-Variable beträgt 4 Byte.

Mit dieser Arithmetik und dem obigen Feldzeiger *zFeld* erhalten folgende Zuweisungen einen Sinn:

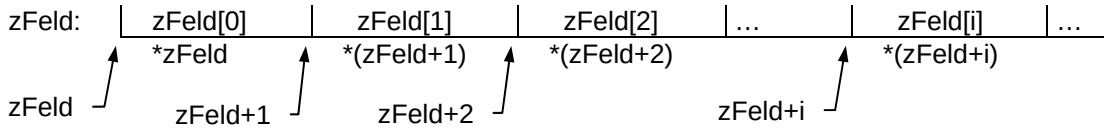
```
int i = 2, *p;          // Int-Variable i und Zeiger p auf int-Werte

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

Es ist von fundamentaler Bedeutung, dass folgende beiden Ausdrücke äquivalent sind:

Wichtig:

➔ zFeld [i] ≡ * (zFeld + i)
➔ & zFeld [i] ≡ zFeld + i



Folgerung:

Die Indexschreibweise kann vollständig durch die Zeigerschreibweise ersetzt werden und umgekehrt.

Betrachten wir zur Demonstration dieser Folgerung ein Beispiel:

```
// Summe von Zahlen mit Feldindizierung:
int summe1 ( int feld [] )
{  int sum = 0;
   for ( int i=0; i<10; i++ )
      sum += feld[i];         // Addition der 10 Zahlen
   return sum;
}

// Summe von Zahlen mit Zeigerschreibweise:
int summe2 ( int *feld )
{  int sum = 0;
   for ( int* end = feld + 10; feld < end; feld++ )
      sum += *feld;
   return sum;
}
```

Funktionen *summe1* und *summe2* in 05-felder\summe.cpp

Beide Funktionen *summe1* und *summe2* sind gleichwertig. Sie werden im Hauptprogramm aufgerufen mit:

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

Die Funktion *summe2* macht es ganz deutlich: Der Parameter ist ein Zeiger, die Angabe der Anzahl der Feldelemente ist daher in der Funktion *summe1* ohne Bedeutung. Die beiden Funktionen werden im Programm wie folgt deklariert:

```
int summe1 ( int [ ] );
int summe2 ( int * );
```

Hierbei handelt es sich wohlgemerkt nicht um zwei verschiedene Parameter. Beide Schreibweisen für die Parameter sind äquivalent und können beliebig gegeneinander ausgetauscht werden.

In der Funktion *summe2* ist zu beachten, dass der Zeiger *feld* eine Kopie des Feldzeigers *zFeld* ist, und somit verändert werden darf. Vorher wird ein Hilfszeiger (*end*) 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.

Da *zFeld* bereits ein Zeiger (Feldzeiger) ist, darf nicht '*&zfeld*' 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 [ ] ;                  nur als Parameter sinnvoll
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 = e ;                   // Zeiger zeigt auf das Feld e, das aus 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 in der 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 und besitzt einen Speicherplatz. Die Adresse dieses Speicherplatzes darf verwendet werden (*&p*). Die Variable *p* darf 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 zugänglich. Daher sind Zuweisungen (fp=…) nicht erlaubt, und die Adresse von *fp* (*&fp*) ist nicht existent.

Gemeinsamkeiten
fp und *p* sind Zeiger auf einen Datentyp (z.B. auf *int* oder *float* oder *char*). Zeigerarithmetik ist erlaubt, insbesondere 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 *e* den Wert 12 und bei *c* und *f* den Wert 4 (Speicher für die Zeiger). Als einziger der obigen 5 Zeiger besitzt die Variable *c* keinen definierten Inhalt. Sie muss demnach erst eine Adresse erhalten, bevor zugegriffen werden darf.

Betrachten wir zur Übung noch einige Beispiele zu den Variablen *a* bis *f*:

```
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 Feldzeiger!
```

Wir haben jetzt Zeiger verstanden und unterziehen uns einem kleinen Abschlusstest. Sind in beiden folgenden Beispielen alle Ausdrücke wirklich äquivalent (mit *i* als int-Variable und *A* als Int-Feld)?

```
A [ i ]   ≡  * ( A + i )   ≡  * ( i + A )   ≡   i [ A ]
A [ 5 ]   ≡  * ( A + 5 )   ≡  * ( 5 + A )   ≡   5 [ A ]
```

Frage: Wo ist der Fehler?

Antwort: in der Vorlesung!

5.4 Call by Reference in C

In Kapitel 4 haben wir kennen gelernt, wie wir in C++ statt Werteparameter Referenzen an Funktionen übergeben können. Mit Hilfe der Zeiger in C können wir dies nun auch mit C-Mitteln erreichen. Die Idee ist ganz einfach:

Wir übergeben statt Variablen die Adressen dieser Variablen an die Funktionen. Die Funktionen legen zu den Adressen Kopien an. Aber auf die Inhalte dieser Adressen können wir in den Funktionen zugreifen, und diese Inhalte können wir damit auch verändern.

Betrachten wir wieder unser Beispiel mit dem Vertauschen der beiden Zahlen *zahl1* und *zahl2*. Im Hauptprogramm übergeben wir jetzt an eine Funktion *vertausche3* die Adressen dieser beiden Variablen:

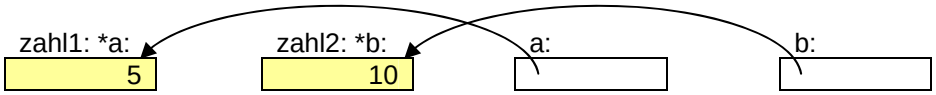
```
vertausche3 ( &zahl1 , &zahl2 ) ;
```

Dies müssen wir natürlich in der Funktion berücksichtigen. Als Parameter verwenden wir also Zeiger. Um auf die Inhalte zuzugreifen, müssen wir dereferenzieren.

```
void vertausche3(int* a, int* b)
{
   int hilf = *a;
   *a = *b;
   *b = hilf;
}
```

Funktion *vertausche3* in 05-felder\vertausche3.cpp

Mit dem Funktionsaufruf wird in den Parameter *a* die Adresse der Zahl *zahl1* kopiert. Damit zeigt die Variable *a* auf die Variable *zahl1*, folglich ist der Ausdruck **a* identisch mit der Variable *zahl1* selbst. Das Gleiche gilt für den Paramter *b* und die Zahl *zahl2*. Betrachten wir die Speicherbelegung direkt nach dem Funktionsaufruf:



Jetzt wird sofort klar, dass ein Ändern des Wertes **a* auch ein Ändern der Variable *zahl1* des Hauptprogramms nach sich zieht.

Wichtig

Der Ausdruck ***a** ist ein sogenannter „Left-Value“. Dies bedeutet, dass er auch links vom Zuweisungszeichen stehen darf. Gemeint ist damit, dass diesem Ausdruck ein Wert zugewiesen werden darf.

Der Operator **&** ist in C++ doppelt belegt, als **Adressoperator** und als **Referenzoperator**. Als Referenz steht dieser Operator immer nach einem Datentyp. Als Adressoperator hingegen bezieht er sich immer auf eine Variable. Eine Verwechslung ist daher nicht möglich.

Wir verstehen jetzt auch, warum Felder automatisch als Call-by-reference-Werte in Funktionen übergeben werden, schließlich übergeben wir ja nur die Zeiger und nicht das Feld selbst. Wir können daher in diesen Funktionen die Inhalte der Felder jederzeit ändern. Oft ist dies aber gar nicht beabsichtigt. In diesen Fällen wird dringend empfohlen, den Bezeichner *const* zu verwenden, zum Beispiel:

```
int summe2 ( const int *feld )
```

Hier signalisieren wir, dass in dieser Funktion der Feldinhalt nicht verändert wird. Der Compiler wird dies auch überprüfen.

5.5 Zeichenketten

Definition:

Die ***nullterminierte Zeichenkette*** ist ein Feld aus Zeichen, das mit dem Zeichen ‚\0’ abgeschlossen wird.

Wir haben diese Definition bereits in der Einführung zu Zeichenketten kennen gelernt, ebenso zahlreiche Zeichenketten-Funktionen. Eine Zeichenkette ist also ein Feld, und ein Feld hat viel mit Zeigern zu tun, wie wir inzwischen wissen. Wie bei Feldern stehen 6 Möglichkeiten zur Auswahl:

```
char a [ 20 ] ;          // Konstantes Feld mit 20 char-Elementen
// char b [ ] ;          nur als Parameter sinnvoll
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 Elementen 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*. Der Feldzeiger *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).

Beim Vergleich zu allgemeinen Feldern fällt die angenehme Vorbelegung bei Zeichenketten auf. Beide folgenden Deklarationen sind gleichwertig, die Verwendung der konstanten Zeichenketten ist aber doch merklich angenehmer und lesbarer:

```
char g [ ] = "hallo\n" ;
char g [ ] = {'h','a','l','l','o','\n','\0'} ;
```

Genau genommen ist die erste Schreibweise mit den Gänsefüßchen nur eine Abkürzung für die Vorbelegung mit den geschweiften Klammern.

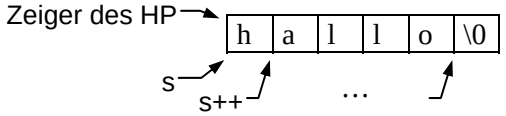
Dank mehrerer Standardfunktionen wie *strcat*, *strcpy*, *strcmp* und *strlen* aus der Bibliothek *string.h* und mehreren Ein- und Ausgabefunktionen für Zeichenketten wird die Verarbeitung von Zeichenketten optimal unterstützt.

Es ist zu beachten, dass die Funktion *strlen(g)* den Wert 6 zurückliefert. Das Endezeichen ‚\0' wird nicht mitgezählt. Um die Funktionsweise der Funktion *strlen* noch besser zu verstehen, implementieren wir diese Funktion doch selbst:

```
int strlen (char *s)
{
   int n;
   for (n=0; *s != '\0'; s++)
      n++;
   return n;
}
```

Funktion *strlen* in 05-felder\strlen.cpp

Betrachten wir das Verhalten dieser Funktion *strlen* am Beispiel der Übergabe der Zeichenkette "hallo".



Der übergebene Zeiger *s* wird fortgeschaltet, solange *s* nicht auf die binäre Null (‚\0') zeigt. Der mitlaufende Zähler *n* wird am Anfang auf den Wert 0 gesetzt, pro Schritt erhöht und am Schluss zurückgegeben. Wichtig ist hier, dass die Variable *n* bereits vor der Schleife deklariert wird. Ansonsten wäre diese Variable nach der Schleife nicht mehr verfügbar und könnte nicht zurückgegeben werden.

Im folgenden Beispiel wird eine int-Zahl *n* in eine Zeichenkette *s* umgewandelt. Die Funktion *itos* erhält die Zahl *n* und gibt diese Zahl als Zeichenkette *s* zurück. Die in der Funktion *itos* verwendete Hilfsfunktion *reverse* dreht den Inhalt einer Zeichenkette um. In den beiden Funktionen wird die Index-Schreibweise verwendet, um nochmals die Gleichwertigkeit beider Schreibweisen zu verdeutlichen. Der Programmierer hat grundsätzlich die freie Wahl zwischen Zeiger- oder Indexschreibweise. In diesem Programm ist zu beachten, dass die Funktion *itos* erwartet, dass das aufrufende Programm hinreichend Speicher für die Zeichenkette *s* zur Verfügung stellt. Andernfalls werden daran anschließende Speicher-

bereiche gnadenlos überschrieben.

```
void itos (char s[], int n)
{  int i = 0;
   int vorzeichen = n;        // merkt das Vorzeichen
   if ( vorzeichen < 0 )
      n = -n;                 // Absolutbetrag
   do                         // erzeugt umgekehrte Zeichenk.
   {  s[i++] = n % 10 + '0';
      n /= 10;
   } while ( n > 0 );          // Abbruch

   if (vorzeichen < 0)
      s[i++] = '-';
   s[i] = '\0';               // explizites Zeichenkettenende
   reverse (s);               // dreht String um
}
```

Funktion *itos* in 05-felder\itos.cpp (Funktion *itos*)

Die Schleife ist recht trickreich programmiert, wobei drei Zuweisungen ausgeführt werden. Es wird nacheinander eine Ziffer als Zeichen abgespeichert, *i* erhöht und *n* durch 10 dividiert. Zum Umwandeln in ein Zeichen spalten wir mit Hilfe des Modulooperators zunächst die letzte Ziffer einer Zahl ab und generieren zu dieser Ziffer das entsprechende Zeichen. Wir verwenden dazu eine Beziehung, die wir am Beispiel der Zahl 3 erklären und analog für alle anderen neun Ziffern zwischen 0 und 9 gilt:

```
3 + '0'  =  '3'        // zur Zahl 3 wird das Zeichen '0' addiert
```

Der Algorithmus nutzt aus, dass wir die letzte Ziffer einer Zahl mittels des Modulooperators ‚%' leicht ermitteln können. Durch die ganzzahlige Division ‚/' kann diese letzte Stelle dann abgespalten werden. Damit bearbeiten wir eine Zahl aber von rechts nach links, so dass wir diese gewonnene Zeichenkette am Schluss umdrehen müssen. Betrachten wir die Schleife am Beispiel der Zahl 4579, um den Algorithmus zu verstehen, und zwar jeweils nach dem Ausführen der ersten Zuweisung (*s[i++]=…*):

	Variable *i*	Variable *n*	*n%10*	*s* (‚\0' fehlt!)
1. Durchlauf	1	4579	9	"9"
2. Durchlauf	2	457	7	"97"
3. Durchlauf	3	45	5	"975"
4. Durchlauf	4	4	4	"9754"

Am Ende des vierten Durchlaufs wird die Variable *n* wieder durch 10 dividiert. Die Division ergibt die Zahl 0, die Schleifenbedingung ist daher nicht mehr erfüllt. Der Zeichenkette werden dann noch ein eventuelles Vorzeichen und das Endezeichen angehängt. Betrachten wir jetzt noch die in der Funktion *itos* verwendete Funktion *reverse*, die die Zeichenkette umdreht:

```
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;
}
```

Funktion *reverse in* 05-felder\itos.cpp

Diese Funktion vertauscht mit Hilfe der beiden Indizes *i* und *j* von außen nach innen die einzelnen Zeichenkettenelemente. Wir haben es hier mit zwei Laufindizes zu tun und haben dies auch in dieser Funktion so ausgedrückt. Beide Variablen werden zu Beginn der Schleife auf den Anfang bzw. das Ende der Zeichenkette vorbelegt. Beide Variablen werden pro Durchlauf erhöht bzw. reduziert. Um dies im

Schleifenkopf zu implementieren, wurde der Komma-Operator verwendet.

Die Indexdarstellung von Feldern haben wir mehrfach geübt, wir wollen uns jetzt auch den Zeigern intensiver widmen. Am Beispiel der Implementierung der Funktion *strcpy* können wir dies besonders gut illustrieren. Stellen wir die beiden Implementierungen (Index bzw. Zeiger) doch direkt gegenüber:

```
void strcpy1 (char s1[], char s2[])
{
   int i;
   for (i=0; s2[i] != '\0'; i++)
      s1[i] = s2[i];
   s1[i] = '\0';         // Zeichenketten-Ende
}

void strcpy2 (char *s1, char *s2)
{
   while ( (*s1 = *s2) != '\0' )
   {  s1++; s2++;
   }
}
```

Funktionen *strcpy1* und *strcpy2* in 05-felder\strcpy.cpp

Die Funktion *strcpy1* bedarf keiner weiteren Erklärung. Wir definieren eine Index-Variable und vergessen das Setzen des Zeichenkettenendes nicht. In der Funktion *strcpy2* gehen wir ganz neue Wege. Die Hauptarbeit leistet der Schleifenkopf, nämlich das Kopieren und das Abprüfen auf das Ende der Zeichenkette. Im Rumpf der Schleife werden nur noch die beiden Zeiger fortgeschaltet. Wir benötigen keinen Index, auch das Zeichenkettenende wird kopiert und muss daher nicht mehr explizit gesetzt werden. Beide Funktionen sind gleichwertig. Die Funktion *strcpy2* wirkt aber kompakter.

Aber C wäre nicht C, wenn es nicht noch knapper ginge. In der Funktion *strcpy3* wird das Fortschalten der Zeiger gleich mit in die Zuweisung gepackt. Wir nutzen dabei aus, dass unäre Operatoren von rechts nach links assoziativ sind. Der Ausdruck **s1++* bedeutet demnach **(s1++)*. Der Inkrementoperator wirkt also auf den Zeiger, nicht auf den Inhalt. Genau dies wollen wir ja auch. Da wir den Postfix-Operator verwenden, erfolgt erst die Zuweisung der Inhalte und dann das Fortschalten der Zeiger. Der Ausdruck ist auch wohldefiniert, da sich diese beiden Zeiger nicht gegenseitig beeinflussen.

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

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

Funktionen *strcpy3* und *strcpy4* in 05-felder\strcpy.cpp

Obwohl die Funktion *strcpy3* bereits sehr kompakt wirkt, geht es mit der Funktion *strcpy4* noch kürzer. Dazu müssen wir wissen, dass die binäre Null ‚\0' das Ascii-Zeichen mit dem Wert 0 ist, also dem Zahlenwert 0 entspricht. Solange die Schleifenbedingung einen Wert ungleich 0 liefert, wird sie ausgeführt, da dies C als wahr interpretiert. Die Schleifenbedingung wird erst unwahr, wenn die Zahl 0 zugewiesen wird. Dies entspricht dem Zeichenkettenendezeichen. Damit wird dann die Schleife beendet. Der Ausdruck in der Funktion *strcpy4* bedeutet folglich:

```
*s1 = *s2;          // Kopieren der Inhalte
s1++; s2++;         // Inkrement der Zeiger: Fortschalten zum nächsten Zeichen
s1 != 0 (='\0')     // Überprüfen auf Zeichenkettenende
```

Auch wenn die Funktion *strcpy4* vollkommen korrekt ist, wird dennoch vor der Nachahmung ein-

dringlich gewarnt. Es lauern zu viele Fehlerquellen! An diesen Beispielen erkennen wir aber schnell, warum bei den Profis die Zeiger so beliebt sind: Mittels Zeiger programmieren wir kurz und performant.

Nach dem Motto „Übung macht den Meister“ erstellen wir zwei weitere Funktionen. Natürlich verwenden wir wieder Zeiger. Die erste Funktion *rTrim* entfernt alle Leerzeichen am Ende einer Zeichenkette, und die zweite Funktion *lTrim* löscht alle Leerzeichen am Anfang einer Zeichenkette. Beginnen wir mit dem Löschen am Ende. Hier ist der Algorithmus schon die halbe Miete:

Gehe zum Ende (!!) des Zeichenkette
Gehe rückwärts zum nächsten Zeichen ungleich Leerzeichen, maximal bis Dateianfang (!)
Schreibe ‘\0’ in das folgende Zeichen

Die Funktion *rTrim* (right trim) ergibt sich jetzt zu:

```
void rtrim (char *s)
{
   char *p;
   p = s + strlen(s) - 1;       // Ende der Zeichenkette
   while (p >= s && *p == ' ')
       p--;
   *(++p) = '\0';   // '\0' hinter letztem Zeichen einfuegen
}
```

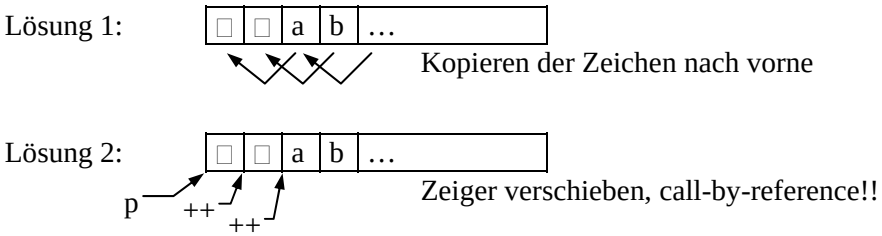
Funktion *rtrim* in 05-felder\trim.cpp

Beim Programmieren lauern viele Fallstricke. Wir dürfen die fettgedruckte Schleife nicht abgekürzen:

```
while (*p = ' ')            // Fehler!!!
   p--;
```

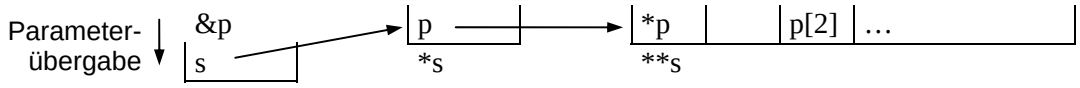
Besteht eine Zeichenkette beispielsweise nur aus Leerzeichen, so würde die Funktion über den linken Rand der Zeichenkette hinauslaufen und dort auch zugreifen. C garantiert aber nur, dass links und rechts von einem Feld noch ein Zeiger definiert ist. Ein Zugriff außerhalb eines Feldes ist aber grundsätzlich verboten. In unserer Funktion überprüfen wir daher auf Leerzeichen und auf den Anfang der Zeichenkette, so wie wir dies bereits im Pseudocode vermerkt hatten.

Für die zweite Funktion *lTrim* (left trim), 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 dann natürlich 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. Wie bereits bekannt, gibt es zwei Möglichkeiten, eine Variable call-by-reference zu übergeben, in C mit Zeigern und in C++ mit Referenzen. Die Verwendung von Referenzen ist leichter nachvollziehbar, vollständigkeitshalber ist auch die Zeigerübergabe aufgeführt. In diesem Fall übergeben wir 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 Zei-

chenkette, so identifiziert sich **s* mit der Zeichenkette. **s* ist als Zeichenkette also ein Zeiger auf das erste Zeichen. Schließlich entspricht ***s* dem ersten Zeichen der Zeichenkette!



Glücklicherweise besitzen wir in C++ die Referenzübergaben. Wir hätten damit drei Funktionen zur Auswahl: *lTrim1* (Kopieren), *lTrim2a* (Zeiger verschieben in C) und *lTrim2b* (Zeiger verschieben in C++):

```
void lTrim1 (char *s)       // Version 1
{  char *p;                 // Hilfszeichenkette
   for (p = s; *p == ' '; p++)
   ;
   strcpy (s, p);
}

void lTrim2a (char **s)     // Version 2a (C)
{  while (**s == ' ')
      (*s)++;
}

void lTrim2b (char* &s)     // Version 2b (C++)
{  while (*s == ' ')
      s++;
}
```

Funktionen *lTrim1/2a/2b* in 05-felder\trim.cpp ()

Wir sehen, dass wir mit der Referenzübergabe in C++ einen deutlich leichter lesbaren Code erhalten. Das Verständnis für Zeiger auf Zeiger muss schließlich erst wachsen. Die Funktion *main* testet alle drei Funktionen. Die Aufrufe lauten:

```
lTrim1 (zeile1);          // Version1
lTrim2a (&p2);            // Version2a
lTrim2b (p3);             // Version2b
```

Zu beachten ist, dass an die Funktionen keine Feldzeiger übergeben werden dürfen, sondern „echte“ Zeiger. In C erkennen wir die Referenzübergabe am verwendeten Adressoperator. Zum Schluss dieses Abschnitts sei noch vor einem häufigen Fehler gewarnt.

Achtung:

➔ Das Kopieren einer Zeichenkette *s1* in die Zeichenkette *s2* erfolgt mit dem Funktionsaufruf *strcpy (s2, s1)*

Die Zuweisung *s2 = s1* ist meist ein schwerer Fehler, da dann nur die Zeiger kopiert werden, nicht jedoch die Inhalte.

5.6 Einschub: Konstanten und selbstdefinierte Datentypen

Über Konstanten haben wir bei den Erweiterungen zu C++ bereits gesprochen. Mit dem Bezeichner *const* können wir hier in der Regel auf die C-Präprozessoranweisung *#define* verzichten. Hier wollen wir daher nur noch auf Besonderheiten von Feldern und der Referenzübergabe von Parametern eingehen. Betrachten wir gleich folgende Deklaration:

```
const char * str1 = "hallo";            // die Zeichenkette "hallo" ist konstant
```

Wir stellen uns unwillkürlich die Frage, was denn nun konstant und damit unveränderlich ist, der Zeiger *str1* oder die Zeichenkette selbst. Die Antwort wurde im obigen Kommentar schon vorweggenommen: die Zeichenkette ist fix. Der Zeiger selbst darf auf eine andere Zeichenkette „umgebogen“ werden. Natürlich können wir auch den Zeiger als unveränderlich deklarieren:

```
char * const STR2 = "jawohl";           // der Zeiger STR2 ist konstant
```

Hier ist der Zeiger fix, die Zeichenkette selbst darf aber manipuliert werden. Mit unseren beiden Deklarationen wäre also erlaubt:

```
str1 = "neu";       // str1 selbst ist nicht konstant.
*STR2 = 'J';        // Der Inhalt der Zeichenkette selbst darf manipuliert werden.
```

Eine wichtige Rolle spielen Konstanten bei Funktionsaufrufen. Aus dem Funktionskopf können wir damit schon ablesen, ob die Inhalte übergebener Referenzen bzw. Adressen manipuliert werden dürfen. Beachten wir nur die Funktion *strcpy*. Folgender Funktionskopf ist unserer obigen Implementierung unbedingt vorzuziehen.

```
strcpy ( char * s1, const char * s2 )
```

Wir zeigen damit nach außen, dass die Zeichenkette *s2* nicht verändert wird, während *s1* vermutlich manipuliert wird. Gleichzeitig überprüft der Compiler, ob wir als Programmierer uns auch an diese Vorgaben halten.

Die konstante Übergabe können wir auch bei der Referenzübergabe verwenden. Bei der Referenzübergabe müssen keine Kopien angelegt werden, so dass diese Übergabe meist performanter ist als die Werteübergabe. Referenzen können aber die übergebenen Variablen ändern. Soll dies nicht erlaubt sein, so verwenden wir konstante Referenzen, etwa:

```
rufMichAuf (const float & x)            // konstante Referenz x
```

Langsam werden unsere Variablendeklarationen komplizierter. Trotzdem soll der Code lesbar bleiben. Hier bietet es sich an, komplizierten Datendeklarationen eigene Namen zu geben. Betrachten wir etwa folgenden Code:

```
const int N = 50;
typedef char * string;              // Neuer Datentyp string
typedef float fFeld [N];            // Der Datentyp fFeld wird definiert
string str;                         // str ist vom Typ char*
fFeld A, B;                         // A und B sind Float-Felder aus N Elementen
```

Mittels des Bezeichners *typedef* können wir Synonyme definieren. In unserem Beispiel ist der Name *string* gleichwertig zu *char**. Ebenso wird *fFeld* mit einem N-elementigen Float-Feld identifiziert. Die Funktionsweise von *typedef* lässt sich einfach erklären: Deklarieren wir eine Variable, so wird durch ein vorangestelltes *typedef* statt dieser Variable der entsprechende Datentyp definiert.

5.7 Mehrdimensionale Felder (Matrizen)

In der Praxis reichen unsere (eindimensionalen) Felder nicht aus. Häufig gibt es (zweidimensionale) Tabellen, manchmal versuchen wir auch dreidimensionale Ergebnisse zu präsentieren. Betrachten wir nur die Matrizenrechnung aus der Mathematik. Natürlich unterstützt uns auch hier C sehr gut. Betrachten wir

doch gleich Matrizen, die wir einlesen, miteinander multiplizieren und als Ergebnis wieder ausgeben. Wir müssen natürlich Besonderheiten von C beachten, denn auch mehrdimensionale Felder sind nichts anderes als Zeiger auf Speicherbereiche, und wir beginnen bei Feldern immer mit dem Index 0. In Schleifen empfiehlt sich folglich dringend die Verwendung von asymmetrischen Grenzen!

In unserem Beispiel wollen wir gleich den Bezeichner *typedef* aus dem letzten Abschnitt gewinnbringend einsetzen. Wir definieren einen Datentyp *matrixtyp*, der im gesamten Programm bekannt sein soll. Wir beginnen unser Programm daher mit folgendem Code:

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

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

void liesMatrix (matrixtyp, int);
...
int main () ...
```

Programmbeginn von 05-felder\matrix.cpp

Ein eindimensionales Feld *vektor* aus DIM Gleitpunktzahlen definieren wir mit

```
float vektor [DIM];
```

Analog definieren wir eine Matrix *A* aus DIM x DIM Elementen, indem wir ein weiteres Paar eckiger Klammern verwenden:

```
float A [DIM] [DIM];
```

Durch die obige Definition des Datentyps *matrixtyp* im Programm *matrix.cpp* noch vor der Definition aller Funktionen ist dieser Datentyp im gesamten Programm bekannt. Diese Deklaration vereinfacht sich daher zu

```
matrixtyp A;
```

In beiden Fällen wird eine Matrix *A* mit 10x10 Float-Elementen erzeugt. Wir erkennen auch die Vorteile der Konstante DIM. Bei Bedarf brauchen wir an nur einer Stelle im Programm den Wert der Konstanten ändern. Wie wir an der Deklaration der Funktion *liesMatrix* sehen, ist dank des Datentyps *matrixtyp* auch die Parameterübergabe sehr übersichtlich. Die Funktion besitzt noch einen weiteren Parameter, denn nicht immer wollen wir 100 Elemente einlesen. Der zweite Parameter gibt die tatsächliche Größe der Matrix an. Wird die Zahl 2 übergeben, so arbeiten wir nur mit den Elementen einer 2x2-Matrix! Betrachten wir gleich die Implementierung dieser Funktion:

```
void liesMatrix (matrixtyp A, int dim)
{
   for (int i=0; i<dim; i++)
   {
      cout << "Zeile " << i+1 << ": ";
      for (int j=0; j<dim; j++)
         cin >> A[i][j];
   }
}
```

Funktion *liesMatrix* in 05-felder\matrix.cpp

Natürlich ist auch eine zweidimensionale Matrix ein Zeiger, so dass sich Änderungen, die wir in der Matrix *A* vornehmen, auch in der aufrufenden Matrix auswirken. Diesen Effekt benötigen wir natürlich hier beim Einlesen. Wie bei einer eindimensionalen Matrix können wir auch bei einer zweidimensionalen

auf die Werte mit Hilfe der Indexschreibweise zugreifen. Für die oben definierten Variablen *vektor* und *A* gilt beispielsweise:

```
vektor [ 3 ] = 12.1;
A [ 4 ] [ 3 ] = 13.13;
```

Im ersten Fall wird dem vierten Feldelement der Variable *vektor* der Wert 12.1 zugewiesen. Im zweiten Fall wird in der fünften Zeile und der vierten Spalte der Matrix *A* der Wert 13.13 gespeichert. Es ist zu beachten, dass der Index von Feldern grundsätzlich bei 0 beginnt. In obiger Funktion *liesMatrix* fordern wir Zeile für Zeile und Spalte für Spalte Matrixelemente an.

Das Schreiben einer Matrix ist nun fast schon ein Kinderspiel:

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

Funktion *schreibMatrix* in 05-felder\matrix.cpp

Wir geben die Werte zeilenweise aus, indem wir am Ende einer Zeile ein Neue-Zeile-Zeichen absetzen. Weiter haben wir vom Bezeichner *const* Gebrauch gemacht. Beim Einlesen verändern wir den Inhalt der Matrix, nicht so beim Schreiben. Dies drücken wir damit auch im Funktionskopf aus. Bei der Deklaration der Funktion *schreibMatrix* muss der Bezeichner *const* ebenfalls mit angegeben werden! Kommen wir damit zu unserer letzten Funktion, der Multiplikation von Matrizen:

```
void multMatrix(const matrixtyp A, const matrixtyp B,
                      matrixtyp C, int dim)
{
   float summe;
   for (int i=0; i<dim; i++)
      for (int j=0; j<dim; j++)
      {  summe = 0;
         for (int k=0; k<dim; k++) // Matrizenmultiplikation
            summe += A[i][k] * B[k][j];
         C[i][j] = summe;
      }
}
```

Funktion *multMatrix* in 05-felder\matrix.cpp

Wir setzen voraus, dass die Matrizenmultiplikation bekannt ist. Ganz allgemein gilt für die Multiplikation C = A*B mit den Parametern a_{ij}, b_{ij} und c_{ij} der drei Matrizen A, B und C:

$$c_{ij} = \sum_{k=1}^{DIM} a_{ik} b_{kj}$$

Diese Summe wird in der inneren For-Schleife mit dem Schleifenzähler *k* durchlaufen. Natürlich müssen diese Werte für alle *i* und *j* berechnet werden, folglich benötigen wir drei ineinander geschachtelte Schleifen. Die Funktion *multMatrix* besitzt drei Matrixparameter. Die ersten beiden werden nur lesend verwendet. Wir verwenden bei diesen beiden daher den Bezeichner *const*.

Zum besseren Verständnis der Verwendung von mehrdimensionalen Feldern seien einige Anweisungen des Hauptprogramms in der Reihenfolge ihres Auftretens aufgeführt:

```
matrixtyp A, B, C;                  // Definition von drei Matrizen
```

```
liesMatrix (A, 2);              // Einlesen der Matrix A
liesMatrix (B, 2);              // Einlesen der Matrix B
multMatrix (A, B, C, 2);        // Matrixmultiplikation, in C steht das Produkt
schreibMatrix (C, 2);           // Schreiben der Matrix C
```

Wir wollen jetzt den zweidimensionalen Feldern noch genauer auf den Zahn fühlen. Betrachten wir dazu ein kleines Feld *m*, definiert durch

```
float m [2] [3] ;
```

C garantiert, dass der Speicher aller sechs Elemente hintereinander liegt. Es gilt:

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

Die Elemente werden zeilenweise abgelegt. Dies gilt analog auch für mehr als zwei Dimensionen. Auch die Initiierung dieser Felder ist kein Problem:

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

Eine Kurzform ist ebenfalls erlaubt:

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

Vor dieser Kurzform wird aber explizit gewarnt. C füllt bei einer expliziten Vorbelegung fehlende Werte mit der Zahl 0 auf. Es ist also ein großer Unterschied, ob mit { {1, 2}, {3, 4} } oder {1, 2, 3, 4 } vorbelegt wird!

Damit hätten wir mehrdimensionale Felder im Prinzip besprochen. Wegen des Zugangs zu Feldern über Zeiger steigen wir aber noch etwas tiefer ein. Wir vermuten richtig, dass die Variable *m* ein Zeiger ist. In C ist ein Zeiger durch seine Adresse **und** den Datentyp, auf den er verweist, festgelegt. Der Datentyp entscheidet über die Zeigerarithmetik, beispielsweise den Wert von *m+1*! Die naheliegende Vermutung, dass **m* das erste Element *m[0][0]* identifiziert, ist daher leider falsch!

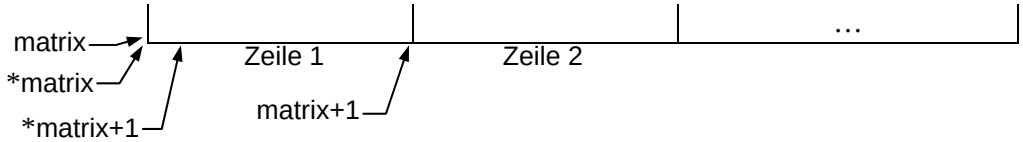
Um dies zu erklären, holen wir etwas weiter aus. Seien *M* und *N* Konstanten und sei *matrix* eine MxN-Matrix aus Float-Elementen. Wir deklarieren:

```
float matrix [M] [N];
```

Diese Matrix enthält *M* Zeilen, jede Zeile besteht aus *N* Float-Werten. Genaugenommen ist diese Matrix also ein Feld aus einem Feld von Zahlen. Um dies noch besser zu verstehen, bauen wir diese Deklaration in zwei Schritten auf:

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

Die Variable *matrix* ist also ein Feld von Zeilen, also ein Zeiger auf den Datentyp *zeile*! 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 ein Feld von Float-Elementen. Damit repräsentiert **matrix* ein Feld und ist

folglich ein Zeiger auf die erste Zeile. Genauer: Der Zeiger **matrix* zeigt auf das erste Element eines Float-Feldes! Mit der nochmaligen Dereferenzierung ***matrix* sprechen wir demnach 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. Der Zeiger *matrix+1* zeigt 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 )          // beide Ausdrücke sind äquivalent
```

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 die 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 *liesMich* ist zwar korrekt, enthält aber eine überflüssige Feldgröße:

```
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 wegen der Zeigerarithmetik vollständig festgelegt sein, die Anzahl der Spalten ist demnach zwingend 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, sind 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 überübernächsten Abschnitt darauf zurück.

5.8 Dynamische Speicherverwaltung in C++

Bisher müssen wir immer Feldzeiger zum Reservieren von Speicherplatz für Felder verwenden. Dieser Speicher wird dann im Speicherbereich des Programms angelegt. Reservieren wir diesen Speicher lokal, so wird der Speicher beim Verlassen der Funktion wieder freigegeben. In C und C++ kann Speicher aber auch dynamisch reserviert und wieder freigegeben werden. Wir können also in der einen Funktion Speicher anfordern und in einer zweiten Funktion diesen Speicher wieder freigeben. C und C++ legen dazu eigene dynamische Speicherbereiche an, den sogenannten ***Heap***. Zur dynamischen Speicherreservierung existieren in C++ die beiden Operatoren **new** und **delete**. Im folgenden Beispiel wollen wir einen Speicherplatz für 100 Ganzzahlen reservieren:

```
int *p;                    // nur ein Zeiger
p = new int [100];         // p zeigt jetzt auf 100 int-Elemente
```

Der Operator *new* erkennt automatisch den Speicherbedarf des Datentyps *int*, reserviert diesen Speicher (hier vermutlich 400 Byte) und liefert einen Zeiger (dieses Datentyps) auf diesen Speicherbereich zurück. Dieses Reservieren von Speicher ist insbesondere auch für Zeichenketten von Interesse. Ist etwa *s1* ein Zeiger auf eine definierte Zeichenkette, so kann diese Zeichenkette wie folgt auf eine Zeichenkette *s2* kopiert werden, wobei für den Zeiger *s2* nur der wirklich erforderliche Speicherplatz belegt wird:

```
char * s2 = new char [strlen(s1)+1];          // Speicher einschliesslich \0 !
strcpy (s2, s1);
```

Zu beachten ist, dass der Operator *new* den NULL-Zeiger zurückliefert, wenn der angeforderte Speicher nicht reserviert werden kann. Eine Überprüfung auf erfolgreiche Speicherreservierung könnte daher wie folgt aussehen:

```
p = new double [1000000] ;
if (p == NULL)   { cerr << "Speicherreservierung misslang\n"; return 0; }
```

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!).

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

```
delete p;          // gibt den Speicher für die 100 Ganzzahlen wieder frei
delete s2;         // gibt den Speicher für die Zeichenkette wieder frei
```

Beachten Sie, dass die Bezeichner *new* und *delete* Operatoren sind. Wir müssen also die folgenden Zeigervariablen nicht in Klammern setzen.

5.9 Dynamische Speicherverwaltung in C

Sollte kein C++-Compiler zur Verfügung stehen, so kann auch mit den Sprachmitteln von C Speicher dynamisch Speicher reserviert werden. Allerdings sind die hier verfügbaren Funktionen etwas komplexer als die Operatoren *new* und *delete* in C++. Die Funktionen zur Speicherverwaltung in C lauten:

```
malloc          calloc          realloc          free
```

Die ersten drei 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 *unsigned int*.

Die Funktion *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. Die Funktion *malloc* ist so allgemein implementiert, dass es nur die Anzahl von Bytes reserviert, unabhängig vom Datentyp. Aus diesem Grund kann die Funktion *malloc* zwar eine Adresse, aber keinen speziellen Datentyp 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 *type*, so wird Speicher für ein Element wie folgt reserviert:

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

Natürlich können wir auch mit der Funktion *malloc* Speicher für Zeichenketten reservieren. Das Beispiel des letzten Abschnitts zum Kopieren von Zeichenketten lautet hier:

```
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 die Funktion *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 dem Operator *delete* und gibt den Speicherbereich frei, auf den *p* zeigt, und der durch die Funktionen *malloc*, *calloc* oder *realloc* angefordert wurde. *p* darf der NULL-Zeiger sein. In diesem Fall geschieht nichts. Übrigens liefern alle Speicherreservierungsfunktionen den NULL-Zeiger zurück, falls der benötigte Speicher nicht reserviert werden konnte.

5.10 Zeigerfelder

Die Flexibilität bei Feldern in C ist insbesondere dadurch gegeben, dass wir bei Parametern die Feld-

größe eindimensionaler Felder nicht mit angeben mussten. Leider galt dies nicht mehr bei mehreren Dimensionen. Ferner liegt nicht immer der Idealfall von gleich langen Zeilen wie bei Matrizen vor. Hier könnten wir durch mehr Flexibilität in vielen Anwendungsfällen Speicherplatz sparen.

Wollen wir also Speicherplatz sparen, und wollen wir weder Zeilen- noch Spaltengrößen bei Funktionsparametern angeben, so sollten wir statt Matrizen sogenannte ***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. Ein besonders interessantes Beispiel sind mehrzeilige Texte. Die Zeilen sind hier in der Regel unterschiedlich lang. Definieren wir doch gleich eine Zeile als einen Zeiger:

```
typedef char * zeile;
```

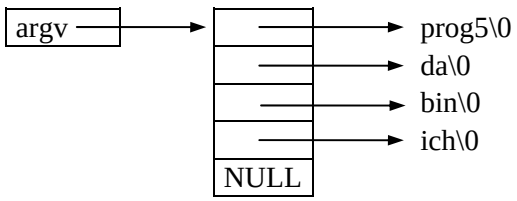
Dies ist jetzt nur ein Datentyp. Wollen wir jetzt aber 10 Zeilen definieren, wobei die i-te Zeile *N+i* Zeichen enthält, so brauchen wir Strukturen und den dazugehörigen Speicherplatz. Dies gelingt uns wie folgt:

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

Nun ist die Variable *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+i* Zeichen. In der Praxis kommen solche Konstrukte häufig vor. Wir lesen beispielsweise von Tastatur oder besser aus einer Datei eine Zeile nach der anderen ein. Nach dem Einlesen kennen wir die Größe der Zeile. Wir reservieren dann genau den benötigten Speicherplatz.

Eine sehr interessante Anwendung von Zeigerfeldern sind übrigens die Parameter der Funktion *main*. Bisher haben wir die Funktion *main* als parameterlose Funktion behandelt. Sie ist es aber nicht. Tatsächlich enthält diese Funktion zwei Parameter (nach Ansi-C), wir haben bisher nur keinen Gebrauch davon gemacht. 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 *prog5.exe*, und rufen wir es auf mit ‚*prog5 da bin ich*' von der Konsole aus auf, so enthält *argc* den Wert 4 und *argv* repräsentiert exakt folgendes Zeigerfeld:



Der Parameter *argv* zeigt demnach auf ein Feld von char*-Zeigern. Der fünfte Zeiger enthält den Wert NULL. Dieser NULL-Wert darf nicht mit der Zahl 0 verwechselt werden. Es ist ein in C und C++ definierter Wert, der anzeigt, dass hier absichtlich keine Adresse angegeben wurde. Es ist also ein Zeiger, der definiert auf keinen Speicherbereich zeigt.

Betrachten wir Zugriffe mit der Variablen *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 *p* vom Wort *prog5*. Wollen wir jetzt das zweite Zeichen des dritten Arguments ansprechen, so zeigt der Ausdruck *argv+2* auf das dritte Element des Zeigerfeldes, der Ausdruck **(argv+2)* folglich auf die dritte Zeichenkette, und **(*(argv+2)+1)* ist schließlich das gewünschte Element, das Zeichen *i* aus dem Wort *bin*.

Natürlich können wir obige Argumentationen mit Indizes wiederholen: Der Ausdruck *argv[2]* ist das dritte Element des Feldes, und damit repräsentiert der Ausdruck *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 [] )          // entweder so
int main ( int argc, char ** argv )            // oder so
```

Die zweite 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 direkt einem UNIX-Buch entnommen ist:

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

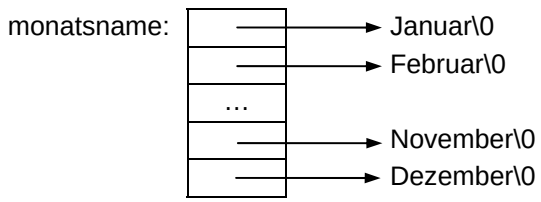
Programm 05-felder\echo.cpp

Diese Funktion bildet das Echo-Kommando von Unix nach: die Argumente werden mit Ausnahme des Programmnamens nacheinander ausgegeben. Bitte vollziehen Sie die Funktionsweise dieses Programms im einzelnen selbst 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 und die dazugehörigen Monatsnamen reserviert. Insgesamt erhalten wir:



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 müssen Funktionen nicht die Feldgrößen ihrer Parameter kennen. Dadurch können diese Funktionen sehr allgemein programmiert werden, da nicht für jede einzelne Feldgröße eine eigene Funktion benötigt wird.

6 Dateien

In der Informatikpraxis werden nur wenige Daten von Tastatur eingelesen. Vielmehr müssen meist große Datenmengen aus Dateien eingelesen, bearbeitet und wieder ausgegeben werden. Diese Daten sind entweder als Dateien oder in Datenbanken abgelegt. Es wird Zeit, sich mit dem Lesen und Schreiben von Dateien zu beschäftigen.

Daten können in Dateien entweder in Textform abgelegt oder binär codiert sein. Wir sprechen dann von ***Textdateien*** bzw. ***Binärdateien***. Daten lassen sich in Binärdateien meist kompakter abspeichern. Wir müssen dann allerdings ein bestimmtes Datenformat vorgeben und beim Auslesen auch kennen. Eine Binärdatei ist beispielsweise eine Datei, in der nacheinander nur Int-Werte abgelegt sind. Wenn jede Int-Zahl 4 Byte belegt, so wäre eine Datei mit einer Million gespeicherter Zahlen genau 4 Megabyte groß. Um diese Zahlen auslesen zu können, müssen wir natürlich die Information haben, dass hier Int-Werte gespeichert sind.

Etwas einfacher ist das Bearbeiten von Textdateien. Textdateien bestehen aus lauter Zeichen. Meist werden Leerzeichen oder Neue-Zeile-Zeichen benutzt, um die Daten voneinander zu trennen. Ein Programm kann Textdateien immer problemlos lesen.

Prinzipiell ist das Arbeiten mit Textdateien und Binärdateien sehr ähnlich. Wir werden uns daher in diesem Kapitel auf Textdateien konzentrieren. Wir werden auch sehen, dass das Arbeiten mit den Daten einer Datei dem Einlesen von Tastatur und dem Schreiben auf Bildschirm entspricht. Neu ist nur, dass wir dem Programm sagen müssen, welche Datei wir bearbeiten wollen. Die Dateibearbeitung lässt sich dabei in drei Schritte einteilen:

- **Öffnen** der gewünschten Datei
- Arbeiten mit den Daten dieser Datei
- **Schließen** der Datei

Zum Arbeiten mit Dateien benötigen wir natürlich auch Dateivariablen. Hier arbeiten C und C++ völlig unterschiedlich, da C++ objektorientiert aufgebaut ist und mit Klassen arbeitet. Wir haben ja schon die Unterschiede zwischen C und C++ bei der Ein- und Ausgabe kennen gelernt. In diesem Kapitel werden wir das Arbeiten mit Textdateien in C++ ausführlich behandeln. Das Arbeiten in C wird vollständigkeitshalber mit angegeben.

Aufgenommen wird auch die Fehlerbehandlung in C++. Gerade beim Lesen von Dateien können viele Probleme auftreten, die in einem Programm sinnvoll abgefangen werden sollten.

6.1 Dateibearbeitung in C++

Die Dateibearbeitung mit Strömen baut auf der Ein- und Ausgabe von C++ auf. Dank eines überlegten Konzepts müssen wir nur sehr wenig zur bisherigen Ein- und Ausgabe hinzulernen. Beginnen wir gleich mit den Klassen. Die Stromklassen *ostream* und *istream* kennen wir bereits. Es gesellen sich zwei weitere wichtige Klassen dazu. Somit haben wir:

- ostream: Ausgabestrom, definiert in *iostream*
- istream: Eingabestrom, definiert in *iostream*

- ofstream: Ausgabestrom in Dateien, definiert in *fstream*
- ifstream: Eingabestrom von Dateien, definiert in *fstream*

Die Dateiströme *ifstream* und *ofstream* unterscheiden sich von *istream* und *ostream* nur durch die zusätzliche Definition von Funktionen zum Arbeiten mit Dateien (z.B. Öffnen und Schließen). Ansonsten ist die Dateibearbeitung vollkommen analog zum Arbeiten mit Bildschirm und Tastatur. Zu beachten ist, dass jetzt auch die Bibliothek ***fstream*** mit einer Include-Anweisung hinzuzufügen ist. Die Bibliothek *iostream* muss meist nicht mit angegeben werden, da große Teile in der Bibliothek *fstream* enthalten sind.

Wir kennen bereits die Ströme *cout*, *cerr* und *cin*. Diese sind standardmäßig in der Bibliothek *iostream* vordefiniert. Dateiströme müssen hingegen deklariert werden, etwa durch:

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

Diese Dateiströme *eindat* und *ausdat* können jetzt mit Dateien logisch verknüpft werden. Hier hilft uns die Funktion *open* weiter, die überprüft, ob die Datei existiert, ob sie lesbar ist, ob sie das gewünschte Format besitzt und die zuletzt diese Datei mit dem Programm verbindet. Es können jetzt Lese- oder Schreibvorgänge erfolgen. Zum Schluss sollten wir Dateien auch wieder schließen. Hierzu dient die Funktion *close*. Der folgende Programmausschnitt erklärt das Prinzip. Wir lesen aus einer Datei *datei.txt* Daten in eine Variable *x* ein:

```
ifstream infile;                // Dateistrom infile
infile.open("datei.txt");       // die Datei datei.txt wird zum Lesen geoeffnet
infile >> x ;                   // Daten werden in die Variable x eingelesen
infile.close ();                // Datei wird geschlossen
```

Neu ist das Öffnen und Schließen von Dateien. Das Arbeiten mit Dateiströmen selbst ist analog zum Arbeiten mit den Strömen *cin* und *cout*. Wir müssen nur die Namen der Ströme *cin* und *cout* durch den Namen des Dateistroms, hier *infile*, ersetzen. Es stehen alle Funktionen und die Operatoren ‚<<' und ‚>>' zur Verfügung, die wir von den Stromklassen bereits kennen. Natürlich dürfen Eingabeströme nur für Eingaben und Ausgabeströme nur für Ausgaben verwendet werden.

In C++ können Dateien schon direkt mit der Deklaration einer Dateivariable geöffnet werden. Wir können also, falls gewünscht, Deklaration und Öffnen verschmelzen. Beispiele sind:

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

Wir sehen, dass wir bei der Deklaration bereits die Parameter der Funktion *open* angeben können. Die Funktion *open* und damit natürlich auch die Deklaration besitzen bis zu zwei Parameter:

1. Parameter: zu öffnender Dateiname
2. Parameter: Öffnungs-Modus

Es gibt mehrere Eröffnungsmodi. Diese sind in der Klasse *ios* als statische Bitleiste definiert. Die wichtigsten sind:

ios::app	beim Schreiben Daten an eine Datei anhängen
ios::binary	Öffnen im Binärmodus; ohne diese Angabe: Öffnen im Textmodus
ios::in	Öffnen zum Lesen (Standardvorgabe bei Eingabeströmen)
ios::out	Öffnen zum Schreiben (Standardvorgabe bei Ausgabeströmen)

Der Modus *ios::app* ermöglicht das Weiterschreiben in einer bestehenden Datei. Ohne diesen Modus wird eine existierende Datei gelöscht, bevor sie zum Schreiben geöffnet wird. Der Modus *ios::binary* kennt keine Zeilenstruktur. Es können damit sehr allgemeine Dateien bearbeitet werden. Die Modi *ios::in*

und *ios::out* werden selten benötigt, da sie standardmäßig eingestellt sind. Sollen mehrere Modi gleichzeitig gesetzt werden, so müssen diese mit dem binären Oder-Operator miteinander verknüpft werden. So ermöglicht etwa die Angabe *ios::in*|*ios::out* das Öffnen zum Lesen und gleichzeitigen Schreiben.

Gerade beim Lesen großer Dateien sind mächtige Funktionen zur Unterstützung sehr wichtig. Wie bereits erwähnt stehen alle bisher behandelten Stream-Funktionen auch für die Dateibearbeitung zur Verfügung. Zusätzlich gibt es Funktionen, die die Dateibearbeitung und insbesondere die Fehlerbehandlung unterstützen. Einige dieser Funktionen sind:

bool eof ( )	gibt den Wert *true* zurück, falls das Dateiende erreicht wurde, sonst *false*
bool fail ( )	gibt den Wert *true* zurück, wenn ein Datenfehler auftrat, sonst *false*
bool bad ( )	gibt den Wert *true* zurück, wenn ein schwerer Fehler auftrat, sonst *false*
bool good ( )	gibt den Wert *true* zurück, wenn kein Datenfehler auftrat, sonst *false*
putback (ch)	schreibt zuletzt gelesenes Zeichen *ch* in den Eingabepuffer zurück
int 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 Funktion *putback* erlaubt, ein bereits gelesenes Zeichen wieder in den Dateipuffer zurückzulegen. Beim nächsten Einlesen wird dieses Zeichen dann wieder gelesen. Dies wird in C und C++ aber nur für das Zurücklegen eines einzigen Zeichens garantiert, die Funktion *putback* darf also nicht zweimal direkt hintereinander angewendet werden. Ähnliches git für die Funktion *peek*. Sie können das nächste Zeichen im Eingabepuffer schon nachsehen, ohne es aber direkt einzulesen. Für das übernächste Zeichen funktioniert dies aber leider nicht.

Das Lesen aus einer Datei ist etwas aufwändiger als das Lesen von Tastatur. Eine Datei wird irgendwann zu Ende sein, von Tastatur können wir dagegen quasi endlos lesen. Aus diesem Grund müssen wir, wenn wir robuste Programme schreiben wollen, bei jedem Lesezugriff überprüfen, ob das Dateiende auf uns „lauert“. Ansonsten drohen schwere Programmfehler. Neben der obigen Funktion *eof* liefern alle Lesefunktionen Rückgabewerte zurück, aus denen das Dateiende ersichtlich wird. Dies sind:

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

Aber auch der Dateistrom selbst kann auf Fehler überprüft werden. Konnte etwa eine dieser Dateien nicht geöffnet werden, so liefern *eindat*, *ausdat* und *andat* den Wert *false* zurück. Auch wenn das Öffnen erfolgreich war, und der Fehler erst später auftritt, so enthält die Dateivariable den Wert *false*, natürlich auch bei Dateiende. Genauer: Liefert eine der Fehlerfunktionen *eof*, *fail* oder *bad* den Wert true zurück, so liefern die Funktion *good* und die Dateivariable den Wert *false*. Ein einfaches Beispiel für die Überprüfung der Dateivariable wäre:

```
if (!eindat) return 0;                    // bei Dateiende oder anderem Einlesefehler beenden
```

Betrachten wir ein kleines Programm, das den Inhalt einer Textdatei an eine andere anhängt. Im Programm verwenden wir eine Funktion *filecopy*, die einen Ausgabestrom auf einen Eingabestrom leitet. Wir lesen dabei Zeichen für Zeichen bis Dateiende ein und geben dieses Zeichen direkt in den Ausgabestrom aus:

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

Funktion *filecopy* in 06-datei\append.cpp

Diese Funktion *filecopy* arbeitet mit einem Ein- und einem Ausgabestrom. Wir müssen daher diese

beiden Ströme zwingend als Referenzen übergeben. Die Schleife ist extrem einfach. Wir lesen Zeichen für Zeichen mit der Funktion *get* ein und geben dieses Zeichen sofort wieder mit der Funktion *put* aus. Wird das Dateiende erreicht, so liefert die Funktion *get* den Wert false zurück. In diesem Fall beenden wir unsere Schleife und damit auch die Funktion sofort.

Unsere Funktion *filecopy* liest von normalen Ein- und Ausgabeströmen. Warum haben wir nicht die Datentypen *ifstream* und *ofstream* verwendet? Die Antwort ist ganz einfach: Wir sind so flexibler. Die Klasse *ifstream* ist von der Klasse *istream* abgeleitet. Ableiten heißt bei Klassen, dass die abgeleitete Klasse alle Eigenschaften der Originalklasse übernimmt und dann noch zusätzliche Eigenschaften hinzunimmt, hier insbesondere das Öffnen und Schließen von Dateien. Die Klasse *ifstream* ist also spezieller, die Klasse *istream* allgemeiner verwendbar. Verwenden wir daher als Parameter den Datentyp *istream*, so können wir an diese Funktion Elemente vom Typ *istream* oder *ifstream* übergeben! Entsprechendes gilt für die Klassen *ofstream* und *ostream*. Dies nutzen wir im Hauptprogramm auch aus.

Das Hauptprogramm arbeitet wieder mit den Argumenten der Funktion *main*. Dabei werden zwei Argumente erwartet, der Name der einzulesenden Datei und der Name der Ausgabedatei. Fehlt das zweite Argument, so wird die Eingabedatei auf die Standardausgabe *cout* ausgegeben. Die Standardausgabe ist vom Datentyp *ostream*. Die Variable *cout* kann daher an die Funktion *filecopy* übergeben werden, ebenso natürlich auch der Datenstrom *fout* vom Datentyp *ofstream*. Betrachten wir die Funktion *main*:

```
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;
}
```

Hauptprogramm in 06-datei\append.cpp

Wir überprüfen im Hauptprogramm die Argumente, liegt nur eines vor, wurde also nur der Programmname aufgerufen, so brechen wir ab. Sonst öffnen wir den Dateieingabestrom *fin*. Existiert die Eingabedatei nicht, oder gab es andere Probleme, so wird mit einer Fehlermeldung abgebrochen. Gab es nur zwei Argumente, so wird die Eingabedatei auf die Standardausgabe *cout*, in der Regel auf den Bildschirm, ausgegeben. Ansonsten wird in die Ausgabedatei *fout* kopiert. Dazu wird jeweils die Funktion *filecopy* aufgerufen.

Bei der Ausgabedatei *fout* fällt auf, dass wir hier keine Fehlerüberprüfung durchführen. Dies liegt daran, dass die Ausgabe meist unproblematisch ist. Existiert etwa eine Ausgabedatei nicht, so wird sie neu angelegt. Ein robustes Programm sollte natürlich trotzdem eine Überprüfung durchführen. Vielleicht versuchen wir ja gerade, in ein schreibgeschütztes Verzeichnis auszugeben.

6.2 Ausnahmebehandlung in C++

Programme müssen in der Praxis robust sein. Unvorhergesehene Ereignisse wie Division durch Null, Feldüberlauf oder Eingabefehler sollten nie zum Absturz eines Programms führen. In C gibt es eine rudimentäre Fehlerbehandlung mittels der Funktionen *setjmp* und *longjmp*. Diese beiden Funktionen genügen

jedoch den heutigen praktischen Anforderungen nicht. In C++ wurde eine Fehlerbehandlung eingeführt, die auch in Java, C# und PHP übernommen wurde. Wir sprechen hier von einer Ausnahmebehandlung, da wir nicht nur Fehler, sondern auch gewollte Spezialsituationen abfangen wollen. Diese Ausnahmebehandlung umfasst drei Teile:

den Programmteil, wo eine Ausnahme auftreten kann	(Try-Block)
den Programmteil, der eine Ausnahme behandelt	(Catch-Block)
die Anweisung, die eine Ausnahme erzeugt	(Throw-Anweisung)

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* stehen Programmanweisungen. Tritt jetzt innerhalb dieses Blocks ein Fehler auf, so wird eine Ausnahme geworfen, und die folgenden *Catch*-Blöcke werden nach diesem Ausnahmefall durchsucht. Gibt es einen *Catch*-Block, der die geworfene Ausnahme behandeln kann, so werden genau die Anweisungen dieses *Catch*-Blocks ausgeführt. Anschließend wird mit dem ersten Befehl direkt nach dem letzten *Catch*-Block fortgesetzt. Wird die aufgetretene Ausnahme in keinem *Catch*-Block abgefangen, so wird der Fehler an den übergeordneten *Try-Catch*-Block weitergereicht. Existiert kein übergeordneter *Try-Catch*-Block oder wird dieser Fehler auch in übergeordneten Blöcken 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 nur global, andere wiederum gezielt lokal abfangen. Dieses Prinzip arbeitet auch über Funktionsgrenzen hinweg.

Ausnahmen werden in der Regel nach Fehlern automatisch vom System ausgelöst, der Programmierer kann aber auch selbst gezielt Ausnahmen erzeugen; man spricht hier von einer Ausnahme werfen. Hierzu dient das Schlüsselwort ***throw***.

Oft reagiert das Programm erst sehr spät auf Fehler. Bei Überschreitung von Feldgrenzen wird C beispielsweise meist nicht sofort einen Fehler erkennen. Erst schwere Verstöße ahndet das Programm und wirft dann eine entsprechende Ausnahme, etwa bei Division durch Null. Folgende Fehler sollten daher sicherheitshalber unbedingt vom Programm überprüft werden:

Überschreitung von Feldgrenzen,
Dateibearbeitungsfehler, insbesondere beim Lesen aus Dateien

Interessanterweise führen auch Einlesefehler in der Regel nicht sofort zum Absturz eines Programms. Soll etwa eine Ganzzahl eingelesen werden, der Benutzer gibt aber stattdessen den Buchstaben *a* ein, so stürzt deshalb das Programm nicht sofort ab, reagiert aber meist recht eigenwillig. Wir betrachten daher ein Programm, das von einer Datei Ganzzahlen bis Dateiende einliest und angemessen auf fehlerhafte Eingaben reagiert. Die eingelesenen Zahlen werden in einem Feld zwischengespeichert, zuletzt sortiert und sortiert wieder ausgegeben. Dieses Programm entspricht dem Sortierprogramm aus dem letzten Kapitel, nur dass jetzt von Dateien gelesen und dort wieder ausgegeben wird.

```
int main()
{
   const int N = 10000;      // max. Feldgroesse
   int zaehler = 0, i, zFeld[N];

   try
   {
      ifstream fin ("eingabe.txt");
      if ( !fin )
         throw("sort: Datei 'eingabe.txt' nicht lesbar\n");
      do
      {
         fin >> zFeld[zaehler];
         if (fin.fail())
            throw -1;
      } while ( !fin.eof() && ++zaehler < N );
   }
   catch (int i)           // faengt int-Ausnahme ab
   {
      cout << "Einlesefehler. Sortieren nur bis hierher\n";
   }
   catch(char * s)         // faengt char*-Ausnahmen ab
   {
      cout << s; return 0; // Fehler ausgeben und beenden
   }
   catch(...)             // alle restlichen Fehler abfangen
   {
      cout << "Fehler, Programmabbruch\n\n"; return 0;
   }

   sort (zFeld, zaehler);     // sortiert das Feld

   ofstream fout("sort.txt");     // Ausgabedatei oeffnen
   for (i=0; i<zaehler; i++)
      fout << zFeld[i] << '\n';       // ausgeben

   return 0;
}
```

Hauptprogramm in 06-datei\sort2.cpp

Der Ausnahmemechanismus ist in diesem Programm klar erkennbar. Im *Try*-Block werden zwei Ausnahmen geworfen, eine *Int*- und eine *Char**-Ausnahme. Zwei *Catch*-Blöcke fangen diese beiden Ausnahmen ab. Der Wert, der hinter dem *Throw*-Bezeichner angegeben wird, wird dabei an den *Catch*-Block als Parameter übergeben. Sollen alle Ausnahmen abgefangen werden, so gibt es einen Default-*Catch*-Block, gekennzeichnet durch drei Punkte. Somit können auch unvorhergesehene Ausnahmen immer behandelt werden.

Mit dem Werfen einer Ausnahme wird der *Try*-Block sofort beendet. 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 diese Ausnahme im Catch-Block mittels der Funktion *throw* (ohne Parameter) weitergereicht werden.

Nochmals sei darauf hingewiesen, dass Einlesefehler keine Ausnahme auslösen. Wir haben daher im Programm mit der Funktion *fail* selbst auf Einlesefehler geprüft. Eine Fehlerbehandlung direkt im ablaufenden Programmteil vergrößert in der Praxis den Code enorm und macht daher ein Programm unübersichtlich. Viel besser ist es daher, Fehlerbehandlungen zentral durchzuführen, wie hier in den *Catch*-Blöcken.

6.3 Dateibearbeitung in C

Dieser Abschnitt dient nur zur Vervollständigung. Der Inhalt wird im Weiteren nicht benötigt. Manchmal gibt es aber noch Situationen, wo kein C++-Compiler zur Verfügung steht. Dann greifen wir auf die ebenfalls sehr mächtigen Möglichkeiten in C zurück. Hier steht der Datentyp *FILE* zur Verfügung, der in *stdio.h* deklariert ist. 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. Zum Öffnen einer Datei dient die 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' an, wie die Datei zunächst geöffnet wird, ob 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");
```

In der Bibliothek *stdio.h* sind 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- und 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 der

Bibliothek *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*.

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 auf diesen Rückgabewert zu überprüfen. Zu beachten ist allerdings, dass EOF kein Zeichen, sondern eine Int-Zahl ist! Zum Vergleich ist hier das Programm *append* aus dem vorletzten Abschnitt als C-Programm angegeben:

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

   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;
}
```

Hauptprogramm in 06-datei\append.c

Betrachten wir auch noch die Funktion *filecopy*:

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

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

Funktion *filecopy* in 06-datei\append.c

7 Rekursion

Wir haben bereits gesehen, dass wir innerhalb von Funktionen andere Funktionen aufrufen können. Bei etwas Nachdenken stellen wir uns unwillkürlich die Frage, ob sich eine Funktion auch selbst aufrufen kann? Bei näherer Betrachtung sehen wir schnell Probleme:

Eine Funktion *xyz* rufe sich intern, etwa als dritte Anweisung, selbst auf. Die Funktion wird daher ein zweites Mal gestartet und ruft sich als dritte Anweisung ein drittes Mal auf. Hier wird dann die Funktion ein viertes Mal gestartet usw.

Dies sieht verdächtig nach einer Endlosschleife aus. Parallelen zu den Schleifen werden sichtbar. Dort verhindern Abbruchkriterien das beliebige häufige Ausführen. Die Endlosschleife in unserer sich selbst aufrufenden Funktion *xyz* liesse sich also vermeiden, wenn weitere Funktionsaufrufe an eine Bedingung geknüpft würden, etwa mit einer *If*-Anweisung.

Rein prinzipiell wären also sich selbst aufrufende Funktionen denkbar. Wir wissen weiter, dass die meisten Funktionen auch Parameter und lokale Variablen besitzen. Hier stellt sich die Frage, ob sich diese Variablen innerhalb der einzelnen Aufrufe nicht gegenseitig in die Quere kommen. Die Antwort darauf ist ein klares **Nein**. Für einen Compiler sind alle Funktionsaufrufe gleichwertig, egal ob es sich beim Aufruf um eine andere oder die gleiche Funktion handelt. Entsprechend sind alle Parameter und Variablen der aufgerufenen Funktion lokal. Sie beeinflussen daher in keinster Weise Parameter und Variablen gleichen Namens des aufrufenden Programmteils. Dies hat für den Compiler zur Konsequenz, dass dieser pro Aufruf einer Funktion sämtliche Parameter und Variablen dieser Funktion neu anlegen muss. Sehen wir uns grob die Arbeitsweise eines Compilern an und betrachten dazu den erforderlichen Speicherbedarf eines Programms.

Laufzeit-system des Compilers	Code des Hauptpro gramms	Daten des Hauptpro gramms	Code der Funktion1	Code der Funktion2	Daten der Funktion1	Daten der Funktion2	↔ …

Nach dem Übersetzen eines Programms wird der grau hinterlegte Bereich statisch angelegt. Dies sind das Laufzeitsystem, der Code und die Daten des Hauptprogramms und der Code aller vorhandenen Funktionen. Wichtig ist, dass die lokalen Daten der Funktionen erst zur Laufzeit des Programms bei Bedarf, also bei Aufruf der entsprechenden Funktionen, dynamisch erzeugt werden. Entsprechend werden nach dem Beenden von Funktionen dessen Daten sofort wieder freigegeben. Dies ist auch der Grund, warum Daten von Funktionen nach dessen Beendigung undefiniert sind. Das Programm kann diesen Datenbereich nämlich sofort anderweitig nutzen, der Inhalt geht verloren.

Jetzt wird klar, dass sich ein Programm ohne gegenseitige Beeinflussung durch interne Variablen mehrfach selbst aufrufen kann. Bei jedem neuen Aufruf wird im Speicher einfach ein zusätzlicher Datenbereich angelegt. Beim Verlassen der zuletzt aufgerufenen Funktion wird der Speicher wieder freigegeben. Ruft sich eine Funktion etwa noch drei weitere Male auf, so sieht der Speicher des Programms wie folgt aus:

statischer Teil des Programms	Daten der Funktion (1. Aufruf)	Daten der Funktion (2. Aufruf)	Daten der Funktion (3. Aufruf)	Daten der Funktion (4. Aufruf)	⇔ …

Das Aufrufen der eigenen Funktion aus einer Funktion heraus heißt **rekursiver** Aufruf. Die Verwendung solcher Aufrufe heißt **Rekursion**. Programme, die keine rekursiven Funktionen verwenden, heißen zur Unterscheidung **iterativ**.

Wir haben nun festgestellt, dass prinzipiell rekursive Aufrufe denkbar sind und vom Compiler unterstützt werden. Wir fragen uns nun natürlich noch, ob es auch sinnvolle Anwendungen der Rekursion gibt. Dies darf auf jeden Fall uneingeschränkt bejaht werden, da viele, meist mathematische Probleme in rekursiver Form vorliegen. Betrachten wir dazu ein erstes Beispiel, die Definition der Fakultätsfunktion:

Fakultät:

a) iterativ (nur mit Schleifen):

mathematisch:

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ \prod_{i=1}^{n} i & \text{für } n \geq 1 \end{cases}$$

algorithmisch:

```
int faku(int n)
{   if (n == 0)    return 1;
    else       {   for (int i=n-1; i >= 2; i--)
                       n = n*i ;
                   return n;
               }
}
```

b) rekursiv:

mathematisch:

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n-1)! & \text{für } n \geq 1 \end{cases}$$

algorithmisch:

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

Im letzten Fall konnte die Funktion *faku* analog zur rekursiven mathematischen Definition niedergeschrieben werden. Wir wollen das Arbeiten mit Rekursionen anhand dieses Beispiels nachvollziehen. Betrachten wir dazu das Vorgehen des Programms beim Aufruf von *faku(3)*:

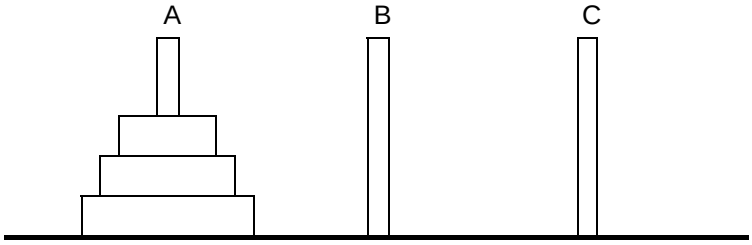
```
faku ( 3 )
     ⇓
     = 3 * faku ( 2 )
               ⇓
               = 2 * faku ( 1 )
                         ⇓
                         = 1 * faku ( 0 )
                                   ⇓
                                   = 1
                         = 1 * 1     ↵
               = 2 * 1     ↵
     = 3 * 2     ↵
= 6  ↵
```

Wir erkennen, dass in diesem Fall die Funktion *faku* insgesamt viermal aufgerufen wird. Alle bisheri-

gen Aufrufe dieser Funktion sind noch geöffnet und warten auf Antwort. Erst beim Aufruf von *faku(0)* erfolgen keine weiteren rekursiven Aufrufe mehr. Der Wert wird stattdessen direkt berechnet und damit die Rekursion abgebrochen. Das Funktionsergebnis (hier der Wert 1) wird an die aufrufende Funktion zurückgeliefert, die vierte Instanz der Funktion *faku* beendet. Schrittweise liefern auch die anderen drei aufgerufenen Funktionen ihr Funktionsergebnis zurück, die Funktionen werden beendet. Dieses Beispiel zeigt eindrucksvoll, dass eine Rekursion funktioniert. Voraussetzung ist, dass es in einer Rekursion immer ein Abbruchkriterium geben muss, hier beispielsweise *n*==0.

Kommen wir zu einem komplexen Beispiel, zu den **Türmen von Hanoi**. Hier handelt es sich um ein Problem, das iterativ schwer zu fassen ist, rekursiv nach etwas Übung aber kaum Schwierigkeiten bereiten sollte. Betrachten wir zunächst die Spielregeln dieser Türme von Hanoi:

Türme von Hanoi:



Gegeben seien drei Säulen, die wir mit *A*, *B* und *C* bezeichnen wollen. Weiter seien *n* Scheiben unterschiedlichen Durchmessers auf Säule *A* gestapelt. Diese Scheiben seien der Größe nach geordnet, beginnend unten mit der größten Scheibe. Ziel ist es, unter Beachtung bestimmter Regeln alle Scheiben von *A* nach *C* zu bringen. Es darf immer nur eine zuoberst liegende Scheibe einzeln auf eine andere Säule gelegt werden, wobei größere Scheiben nicht auf kleineren Scheiben abgelegt werden dürfen.

Schon ab n=3 wird das Spiel kompliziert. Beginnen wir daher mit 2 Scheiben, die von *A* nach *C* bewegt werden sollen:

```
n = 2 :     A nach B      (dies heißt: oberste Scheibe von A nach B bewegen)
            A nach C
            B nach C
```

Damit hätten wir das Zweischeibenproblem gelöst. Immer wenn wir also zwei Scheiben bewegen müssen, können wir uns darauf berufen. Die mittlere Säule *B* diente als Hilfssäule. Wir wollen uns mit diesem Wissen dem 3-Scheibenproblem nähern:

```
n = 3 :     2 Scheiben von A nach B   (mit C als Hilfssäule nach obigem Prinzip)
            A nach C
            2 Scheiben von B nach C   (mit A als Hilfssäule)
```

Beachten Sie, dass die Regeln zum Bewegen der Scheiben eingehalten wurden. Wir sehen, wenn wir das Zweischeibenproblem gelöst haben, können wir leicht auch das 3-Scheibenproblem lösen. Dies gilt analog für mehrere Scheiben. Allgemein können wir für *n* Scheiben schreiben:

```
bewege n-1 Scheiben von A nach B (mit C als Hilfssäule)
bewege 1 Scheibe von A nach C
bewege n-1 Scheiben von B nach C (mit A als Hilfssäule)
```

Wir haben ein Problem mit *n* Scheiben auf eines mit *n-1* Scheiben zurückgeführt. Betrachten Sie die rekursive Definition der Fakultätsfunktion. Dort wurde ebenfalls die Fakultät von *n* auf das Problem für *n-1* reduziert. Auch gab es ein Abbruchkriterium. In unserem Falle wäre es das 1-Scheibenproblem, was trivial zu lösen ist. Wir sehen, mit Hilfe der Rekursion ist das Problem der Türme von Hanoi ganz einfach nach obigem Algorithmus lösbar.

Wir benötigen eine Funktion mit vier Parametern. Der erste Parameter gibt an, von wo die Scheiben zu transportieren sind, der zweite gibt die Hilfssäule an, der dritte die Zielsäule und der vierte schließlich die Anzahl der zu bewegenden Scheiben. Die Funktion selbst enthält nichts weiter als obigen Algorithmus:

```
void loeseHanoi(char quelle, char hilf, char ziel, int n)
{
   if (n==1)                          // Ende der Rekursion
     datei << setw(n) << quelle << " -> " << ziel << '\n';
   else
   {
     loeseHanoi( quelle, ziel, hilf, n-1 );
     datei << setw(n) << quelle << " -> " << ziel << '\n';
     loeseHanoi( hilf, quelle, ziel, n-1 );
   }
}
```

Funktion *loeseHanoi* in 07-rekursion\turmHanoi.cpp

In dieser rekursiven Funktion *loeseHanoi* wird bei *n=1* abgebrochen. In diesem Fall wird die Bewegung der einzelnen Scheibe ausgegeben. Da bei mehr als fünf Scheiben viele Bewegungen und damit Ausgaben erforderlich sind, wird am besten gleich in eine Datei ausgegeben. Diese Datei wird wegen der besseren Performance global definiert und im Hauptprogramm geöffnet. Als Besonderheit wird darüber hinaus noch je nach Rekursionstiefe mit dem Manipulator *setw* eingerückt, abhängig vom Parameter *n*. Wir können damit sehr gut die Rekursionstiefe erkennen. Je weniger eingerückt ist, um so tiefer ist die Rekursion. Der Aufruf im Hauptprogramm erfolgt durch

```
loeseHanoi ( 'A', 'B', 'C', n ) ;
```

Dieses Beispiel zeigt eindrucksvoll die Stärke der Rekursion. Ein unglaublich einfaches Programm löst ein Problem, das ohne Rekursion erst nach sehr vielen Versuchen und Überlegungen, wenn überhaupt, erkannt wird.

In der Praxis tauchen oft Probleme auf, die rekursiv relativ einfach lösbar sind. Diese rekursiven Lösungen können dann direkt in rekursive Funktionen umgesetzt werden. Beispiele hierfür sind:

- Schnelle Suche in sortierten Datenbeständen; Stichwort: **Binäre Suche**
- Schnelle Sortierverfahren; Stichwort: Mergesort, **Quicksort**

Es sei aber nicht verschwiegen, dass die Rekursion einen Nachteil hat: Wird eine sehr große Rekursionstiefe erreicht, so wird wegen des dynamischen Speicherbedarfs je Funktionsaufruf sehr viel Speicher benötigt. Es muss daher sichergestellt werden, dass der verfügbare Speicher auch wirklich ausreicht, um einen Absturz des Programms zu verhindern.

8 Modulare Programmentwicklung

Langsam werden unsere Programme umfangreich. Alle Funktionen in einer einzigen Datei zu halten wird mit vielen Funktionen schnell unübersichtlich, ist aber machbar. Probleme treten aber dann auf, sobald mehrere Personen an einem gemeinsamen Projekt arbeiten. Jede Person wird natürlich seine Funktionen unabhängig von den anderen erstellen und jeweils in einer eigenen Datei speichern. Diese einzelnen Dateien dann zusammen zu mischen, eröffnet viele Fehlerquellen.

Die Programmiersprache C unterstützt erfreulicherweise ein modulares Programmierkonzept: Jeder Entwickler erstellt seine eigenen Funktionen in eigenen Dateien. Alle Dateien können dann voneinander getrennt übersetzt und zusammengebunden werden. Compiler und Linker überprüfen, ob auch alle Funk-

tionen und Funktionsaufrufe zueinander passen. Damit dies reibungslos möglich ist, muss jeder einzelne Entwickler aber ein paar Dinge beachten. Das Zusammenbinden kann weiter automatisiert werden, entweder mit einem sogenannten ***Makefile*** oder, noch einfacher, mit Projektdefinitionen in den Entwicklungswerkzeugen wie Bloodshed und vor allem Visual Studio.

Die zu beachtenden Details wollen wir in diesem Kapitel kennen lernen. Wir werden die Sichtbarkeit von Variablen und Funktionen aufzeigen, wir werden uns mit Speicherklassen beschäftigen und schließlich die getrennte Übersetzung vorstellen. Wir zeigen, dass ***Headerfiles*** eine wichtige Rolle spielen und probieren schließlich die getrennte Übersetzung an einem Beispiel aus.

8.1 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** eines Blocks (insbesondere innerhalb eines Funktionsblocks) deklariert werden. Bisher haben wir ausschließlich interne Variablen verwendet. Bei Konstanten und Datentypen arbeiteten wir meist mit globalen Bezeichnern.Es gilt generell:

- eine Funktion ist außerhalb eines Blocks zu definieren, ist also immer **extern**,
- ein **externer** Bezeichner gilt ab der Zeile der Deklaration bis zum Dateiende,
- ein **interner** Bezeichner gilt ab der Zeile der Deklaration bis zum Ende des Blocks, wo er deklariert wird,
- ein **interner** Bezeichner in C muss zwingend am Anfang eines Blocks deklariert werden,
- 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.

Die Deklaration interner Bezeichner am Anfang eines Blocks ist in C fest vorgegeben. Damit ist gemeint, dass alle internen Bezeichner vor der ersten Anweisung deklariert werden müssen. In C++ ist diese Bedingung ersatzlos entfallen.

In C++ kann eine globale, außerhalb einer Funktion definierte Variable mit Hilfe des ‚::'-Operators angesprochen werden, auch wenn eine lokale Variable gleichen Namens existiert. 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 verwendet.

In C und C++ können Funktionen an beliebiger Stelle innerhalb einer Datei platziert 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 am besten zu Programmbeginn durch Schreiben des Funktionskopfs, abgeschlossen durch ein Semikolon. Wir haben dies bereits regelmäßig praktiziert. Funktionsdeklarationen unterliegen ebenfalls 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 Platzierung einschränken können. Noch besser ist es aber, soweit wie möglich auf globale Variablen ganz zu verzichten.

8.2 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

Darüber hinaus wird der gesamte Programmcode einschließlich aller darin enthaltenen C-Funktionen ebenfalls zur Compilierzeit angelegt, dieser ist also quasi statisch.

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 kann durch Schlüsselwörter geändert werden. Diese 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 Variablen und Funktionsdeklarationen auf außerhalb deklarierte statische Variablen bzw. Funktionen

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.

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 internen Variablen werden wie externe Variablen schon beim Compilieren 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 *zaehler()* den Wert 1, der zweite den Wert 2 usw. zurück. Beim Verlassen der Funktion bleibt nämlich der Speicherplatz und damit der Inhalt von *n* erhalten. Beim nächsten Aufruf wird genau auf diesen Speicherplatz wieder zugegriffen. Static-Variablen werden dabei nur einmal – zu Programmstart – initiiert. Wird das Schlüsselwort *static* weggelassen, so würde in unserem Beispiel 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 somit 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. Für Parameter einer Funktion ist nur das Schlüsselwort *register* erlaubt.

Einige dieser Schlüsselwörter besitzen weitere Bedeutungen bei der getrennten Übersetzung. Wir kommen gleich im nächsten Abschnitt darauf zurück.

8.3 Getrennte Übersetzung

In C++ ist es möglich, ein umfangreiches Programm auf 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.cpp*, das intern die Funktion *strcpy* verwendet. Schließen wir jetzt die Datei *string.h* nicht mit ins Programm ein, so wird das Übersetzen mit einer Fehlermeldung misslingen. Diese wichtige Datei *string.h* ist eine normale Textdatei und enthält alle Funktionsdeklarationen zur Zeichenkettenbibliothek. Dateien mit der Endung *.h* heißen **Headerfiles** oder **Headerdateien** und spielen eine ganz entscheidende Rolle bei der getrennten Übersetzung. Bisher wurden diese Dateien einfachheitshalber als Bibliotheken bezeichnet. Genau genommen enthalten sie nur die Verweise auf die Bibliotheksfunktionen.

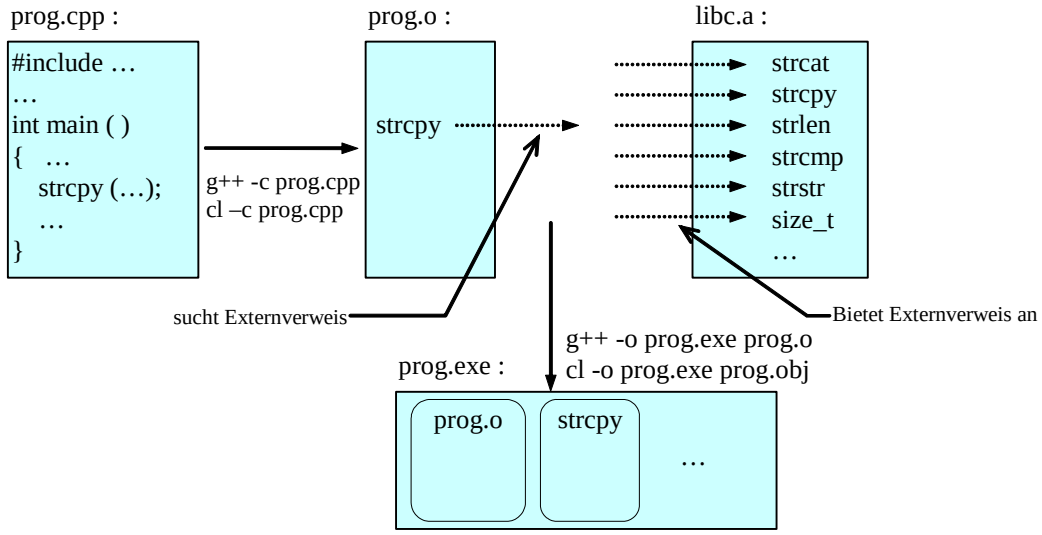
Mittels der Präprozessoranweisung *#include* wird eine Headerdatei an dieser Stelle des Programms eingefügt. In unserem Beispiel werden damit alle Funktionsdeklarationen der Zeichenkettenfunktionen am Anfang des Programms *prog.cpp* eingefügt. Jetzt kann der Compiler überprüfen, ob die Verwendung der Funktion *strcpy* mit der Syntax dieser Funktion, die ja in der Headerdatei steht, übereinstimmt.

Meist arbeiten wir heute mit Oberflächen. Oberflächen nehmen dem Benutzer so manche Arbeiten ab. Zum Verständnis wollen wir aber mit der Konsole beginnen. Hier sehen wir sehr deutlich, welche Arbeiten sowohl der Compiler als auch der Linker übernehmen müssen. Zum Arbeiten mit **Bloodshed** genügt eine normale Eingabeaufforderung. Allerdings muss mit dem Path-Befehl der Pfad auf den Compiler gesetzt werden. Standardmäßig ist dies in Windows: *C:\Programme\Dev-Cpp\Bin*. Zum Arbeiten mit **Visual Studio** ist der Visual Studio 2005 Command Prompt zu starten. Damit sind alle erforderlichen Pfade automatisch gesetzt.

Der Befehl zum Aufruf nur des Compilers ohne Linker lautet:

```
g++ -c prog.cpp          mit Bloodshed und Mingw-Compiler
cl -c prog.cpp           in Visual Studio
```

Mittels der Option ‘-c’ wird der C++-Compiler angewiesen, das Programm *prog.cpp* nur zu übersetzen. Der Compiler erzeugt nun bei einem fehlerfreien Programm die Datei *prog.o* bzw. *prog.obj*. Dieses Modul enthält den Maschinen-Code des Programms, ist für sich aber nicht ablauffähig. Es fehlt das Laufzeitsystem von C++ und die Implementierung der zahlreich verwendeten Standardfunktionen. Es ist nun die Aufgabe des Linkers, alle benötigten Teile zum Modul *prog.o* bzw. *prog.obj* dazuzubinden. Dazu legt der Compiler im Modul *prog.o/prog.obj* sogenannte Externlisten an. Diese enthalten die Namen der Bezeichner, die gesucht werden. Umgekehrt bieten die Standardbibliotheken, etwa *libc.a*, eine Menge von Externverweisen an. Der Linker sucht nun gleiche Namen, verknüpft diese und baut schließlich mittels dieser Verknüpfungen ein lauffähiges Programm zusammen. Veranschaulichen wir uns dies an der folgenden Grafik. Dabei gehen wir davon aus, dass unser Programm intern die Funktion *strcpy* verwendet:



Die Grafik zeigt auf, dass wir das Erzeugen eines fertigen Programms in zwei Schritten durchführen können. Wir erzeugen zunächst mittels des Compilers ein Modul und anschließend mittels des Linkers ein fertiges Programm. Etwas verwirrend mag für den Anfänger sein, dass beide Male der gleiche Programmaufruf verwendet wird, also *g++* bzw. *cl*. In der Praxis ist dies aber sehr praktisch. Wir müssen nur die Besonderheiten beachten:

g++ -o prog.exe prog.cpp	erzeugt das ablauffähige Programm prog.exe
g++ -c prog.cpp	erzeugt das Modul prog.o
g++ -o prog.exe prog.o	erzeugt das ablauffähige Programm prog.exe

Gleiches gilt für den Visual-Studio-Compiler *cl*. Beim Visual-Studio-Compiler wird meist zusätzlich ein Eventhandler benötigt. Es muss daher fast immer bei den drei Befehlen die Option ***/EHsc*** hinzugefügt werden! Außerdem erzeugt der Visual-Studio-Compiler das Modul *prog.obj*.

Die Option ‚*-o Dateiname*' gibt an, wie das fertige Programm heißen soll. Die Standardvorgabe ohne diese Option ist nicht gerade aussagekräftig (unter Windows: *a.exe*, unter Unix: *a.out*).

Obige drei Befehle zeigen auf, dass der erste Befehl in die Befehle zwei und drei zerlegt werden kann. Es wird also beim zweiten Befehl nur der Compiler und beim dritten nur der Linker aufgerufen. Anders ist dies beim ersten Befehl: Compiler und Linker werden nacheinander verwendet, um ein ausführbares Programm zu erhalten.

Wir verallgemeinern nun dieses Vorgehen für das Arbeiten mit mehreren Quelldateien. Betrachten wir dazu nochmals unser Matrizenprogramm. Die Funktionen *liesMatrix, schreibMatrix* und *multMatrix* wollen wir in einer eigenen Quelldatei *matrizen.cpp* verwalten. Das Hauptprogramm steht in einer eigenen Datei *haupt.cpp*. Hier werden die Matrixfunktionen nur deklariert. Tatsächlich gibt sich der Compiler allein mit der Deklaration der verwendeten Funktionen zufrieden und übersetzt das Programm zum Modul *haupt.o*. Natürlich ist dieses Modul nicht lauffähig, es fehlen ja noch die Implementierungen der Matrixfunktionen. Wir übersetzen daher auch noch die Quelldatei *matrizen.cpp* zum Modul *matrizen.o*. Dieses Modul wiederum enthält keine Funktion *main*, und damit keinen Einsprungpunkt. Aber wir wissen ja schon, dass Module für sich nicht ablauffähig sind. Der Compiler hat seine Arbeit gemacht. Der Rest ist nun Aufgabe des Linkers. Er sucht ganz analog zur obigen Grafik alle Verknüpfungen zusammen, schaut nach, ob genau eine Funktion *main* existiert, und bindet alles zusammen mit dem Laufzeitsystem zu einem lauffähigen Programm zusammen. Einzige Voraussetzung ist natürlich, dass alle Quelldateien fehlerfrei waren. Betrachten wir zum besseren Verständnis unsere beiden Quelldateien:

Datei1 (haupt.cpp):

```
#include <iostream>
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 (matrizen.cpp):

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

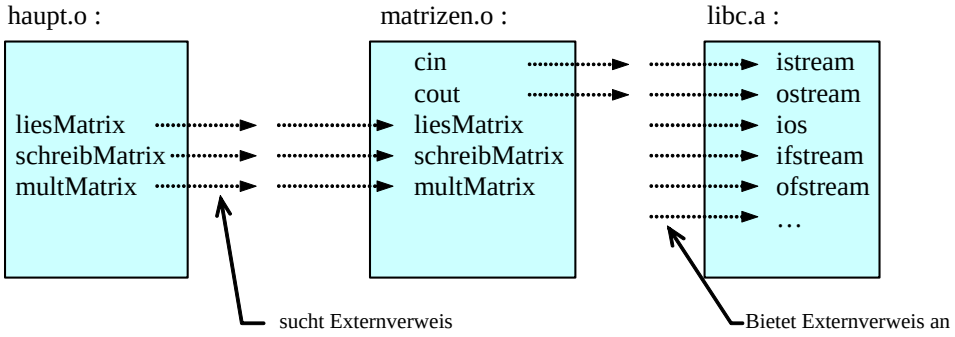
Die beiden Teilprogramme lassen sich mittels eines einzigen Befehls komplett übersetzen:

```
g++ -o matrix.exe haupt.cpp matrizen.cpp
```
erzeugt Programm *matrix.exe*

Aber natürlich können wir auch wieder Compiler und Linker getrennt arbeiten lassen:

```
g++ -c haupt.cpp
g++ -c matrizen.cpp
g++ -o matrix.exe haupt.o matrizen.o
```
erzeugt Modul *haupt.o*
erzeugt Modul *matrizen.o*
erzeugt Programm *matrix.exe*

Wir übersetzen beide Quelldateien für sich und binden erst dann mit dem Linker diese Module zusammen. Das Verlinken geschieht wieder über die Externlisten, wobei diesmal *matrizen.o* sowohl nach externen Funktionen sucht, als auch welche anbietet:



Natürlich funktioniert dies auch mit dem Visual-Studio-Compiler *cl* analog. Wieder ist zu beachten, dass dann die Module die Endung *.obj* aufweisen und die Compileroption */EHsc* zu verwenden ist.

Dieses Vorgehen veranschaulicht das Zusammenspiel zwischen Compiler und Linker sehr gut, zeigt aber auch Schwächen der getrennten Übersetzung auf. Wir zeigen dies an einem einfachen Beispiel:

Wir wollen Speicher sparen und ändern dazu im Hauptprogramm *haupt.cpp* die Konstante DIM auf den Wert 5. Vergessen wir nun, dies auch im Matrizenpaket *matrizen.cpp* nachzuvollziehen, so wird trotzdem ein ausführbares Programm erzeugt! Dieses wird jedoch garantiert nicht das Gewünschte tun. Der gleiche Effekt tritt auch auf, wenn wir uns 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 und sicheren Programmierung. Mit nur geringem Zusatzaufwand lässt sich erreichen, dass Compiler und Linker alle syntaktischen Fehler auch dann entdeckt, wenn mehrere Quelldateien vorliegen.

Betrachten wir zum besseren Verständnis die Reaktion eines C/C++-Compiler, wenn er nicht alle

Funktionen und Variablen in seiner Übersetzungseinheit vorfindet:

- 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 werden 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 sind immer Verweise auf Funktionen, die in der gleichen oder einer anderen Datei 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.
- Globale Variablendefinitionen mit dem Zusatz *extern* reservieren keinen Speicher, sondern verweisen nur auf eine Variable des gleichen Namens, 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.

8.4 Headerdateien

Das Verwenden der gleichen Deklarationen in mehreren Übersetzungseinheiten lässt sich mit Hilfe von sogenannten **Headerdateien** oder Headerfiles sicherstellen. Diese Headerdateien enthalten alle Externverweise und globalen Typ- und Konstantendefinitionen einer oder mehrerer Übersetzungseinheiten und werden mittels der Präprozessoranweisung *#include* den einzelnen Dateien hinzugefügt. Es ist in C üblich, dass Headerdateien 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 ( const matrixtyp, int );
void multmatrix ( const matrixtyp, const matrixtyp, matrixtyp, int );
```

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

Datei1 (haupt.cpp):

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

Datei2 (matrizen.cpp):

```
#include <iostream>
#include "matrizen.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 spätestens jetzt, dass wir mit einer Include-Anweisung keine Bibliothek einfügen, sondern nur eine Textdatei, die fast nur Funktions- und Datentypdeklarationen enthält. Diese Deklarationen beziehen sich aber immer auf entsprechende Funktionen, die übersetzt in Modulen und Bibliotheken gespeichert sind.

In unserem Beispiel haben wir alle Deklarationen ausgelagert. Diese Deklarationen existieren jetzt nur noch einmal. Eine Änderung wirkt sich auf beide Programmteile aus. Beim Übersetzen des Hauptprogramms *prog1.cpp* wird die Syntax der Funktionen dank der Headerdatei sauber überprüft, ebenso beim Übersetzen des Matrizenprogramms *matrizen.cpp*. Unter Beachtung der folgenden Hinweise ist daher eine getrennte Übersetzung mit Hilfe von Headerdateien genauso sicher wie ohne Aufsplittung in mehrere Dateien:

Jede Datei, die Dienste (Externverweise) anbietet, bietet diese über eine Headerdatei an. Diese Headerdatei muss auch von allen Programmteilen, die diese Dienste in Anspruch nehmen, mittels einer Include-Anweisung eingebunden werden. In eine Headerdatei gehören:

- alle global sichtbaren Funktionsdeklarationen, die in einer Datei definiert werden
- alle global verwendeten Konstanten und Datentypdefinitionen
- alle globalen Verweise auf Variablen

Umgekehrt haben folgende Definitionen nichts in einer Headerdatei zu suchen:

- Variablendeklarationen
- komplette Funktionsdefinitionen

Besteht ein Projekt aus vielen einzelnen Quelldateien, so wird davon dringend abgeraten, nur eine Headerdatei für das gesamte Projekt zu verwenden. Stattdessen wird jede einzelne Quelldatei über eine eigene Headerdatei mit dem Rest des Projekts verbunden. Meist besitzt nur die Datei, die die Funktion *main* enthält, keine Headerdatei.

8.5 Getrennte Übersetzung an einem Beispiel

Wir haben bereits einige Funktionen zum Zugriff auf Matrizen und einige Funktionen zur Zeichenkettenbearbeitung 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) Zeichenkettenmodul und dazugehörige Headerdatei:

```
#include "strpriv.h"

int strlength (char *s)
{  int n;
   for (n=0; *s != '\0'; s++)
      n++;
   return n;
}

void strcopy (char *s1, const char *s2)
{  while ( (*s1++ = *s2++) != '\0' )
   ;
}
```

Modul *strpriv* in 08-modul\strpriv.cpp

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

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

Headerdatei zu Modul *strpriv* in 08-modul\strpriv.h

b) Matrizenmodul und dazugehörige Headerdatei:

```
#include "matrizen.h"

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

Modul *matrizen* in 08-modul\matrizen.cpp

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

void addMatrix( const matrixtyp, const matrixtyp,
                      matrixtyp, int);

void multMatrix( const matrixtyp, const matrixtyp,
                       matrixtyp, int);

void liesMatrix( matrixtyp, int);

void schreibMatrix( const matrixtyp, int);
```

Headerdatei zum Modul *matrizen* in 08-modul\matrizen.h

c) Hauptprogramm:

```
#include "strpriv.h"
#include "matrizen.h"
#include <iostream>
#include <iomanip>

int main ()
{
   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
   schreibMatrix( C, 2 );      // schreibt C auf Bildschirm

   s2 = new char[strlen(s1)+1];        // Speicher anfordern
   strcopy(s2, s1);            // in strpriv.h
   cout << ">"<< s2 <<"< hat "<<strlength(s2)<<" Zeichen\n";
   return 0;
}
```

Hauptprogramm *main* in 08-modul\hauptprg.cpp

Das gesamte Programm lässt sich mit dem Befehl

```
g++  -o matrix.exe  hauptprg.cpp  strpriv.cpp  matrizen.cpp
```

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 und später je nach Wunsch zum Programm *matrix.exe* zusammen gebunden werden:

```
g++  -c hauptprg.cpp
g++  -c strpriv.cpp
g++  -c matrizen.cpp
g++  -o matrix.exe  hauptprg.o  strpriv.o  matrizen.o
```

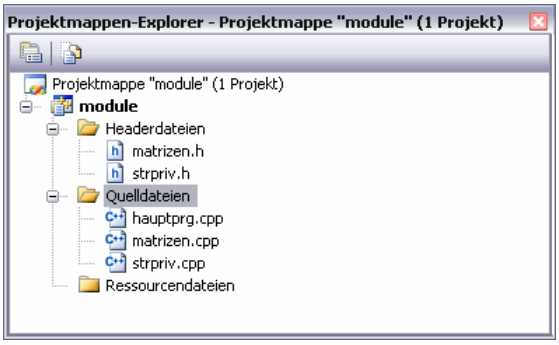
Analog funktioniert dies auch mit dem Visual-Studio-Konsolen-Compiler *cl*. Hier lautet die letzte Zeile wegen der anderen Modulnamen und der zusätzlichen Option nur geringfügig anders:

```
cl  -o matrix.exe  hauptprg.obj  strpriv.obj  matrizen.obj  /EHsc
```

Das Arbeiten mit der Konsolenanwendung birgt leider noch eine Gefahrenquelle: Das Versionsmanagement wird nicht unterstützt. Aus Versehen könnten ältere nicht mehr aktuelle Module mit neueren Modulen zusammengebunden werden. Der Linker selbst kennt nur die Externverweise, so dass nicht korrekt zusammenpassende Teile verknüpft werden könnten. Hier hilft das sehr mächtige **Makefile** weiter. Insbesondere in Unix-Systemen ist es sehr weit verbreitet. Mit dem Siegeszug der Programmier-Oberflächen werden Makefiles immer mehr in den Hintergrund gerückt. So wollen auch wir hier nicht auf Makefiles

zurückgreifen, sondern nur den interessierten Lesern im Verzeichnis *08-modul* ein Makefile ohne weiteren Kommentar zur Verfügung stellen. Stattdessen betrachten wir die Arbeitsweise von Visual Studio, das uns alle Arbeiten in Bezug auf Übersetzen, Binden und korrekte Versionsführung vollständig abnimmt.

Dazu starten wir Visual Studio, wählen eine C++-Konsolenanwendung und hier ein leeres Projekt. Der Projektmappen-Explorer zeigt uns nun an, dass tatsächlich keine Dateien im Projekt existieren. Nun fügen wir die beiden Headerdateien und die drei Quelldateien hinzu. Unser Projektmappen-Explorer sollte dann wie folgt aussehen:



Wir haben also ein Projekt namens *module* definiert mit insgesamt fünf Dateien. Dieses Projekt wird von Visual Studio selbstständig verwaltet. Jede Änderung in einer der fünf Dateien wird bei einer Neuübersetzung berücksichtigt, alle erforderlichen Dateien werden neu compiliert. Nicht betroffene Dateien bleiben unverändert.

Arbeiten nun mehrere Mitarbeiter an einem gemeinsamen Projekt, so können geänderte Programmdateien einfach hinzugefügt bzw. geändert werden. Visual Studio achtet darauf, dass die einzelnen Teile korrekt zusammen passen.

9 (Dynamische) Datenstrukturen

Mit Hilfe von Feldern können wir große und größte Datenmengen verwalten, denken wir nur etwa an ein Feld mit einer Million und mehr Elementen. In der Praxis liegen aber meist keine homogenen Daten vor. Die vorliegenden Daten setzen sich zusammen aus Zeichenketten (Name, Ort, Artikelbezeichnung usw.), Zahlen (Artikelnummer, Kosten, Telefonnummer usw.) und weiteren Datentypen. Um auch diese heterogenen Strukturen verarbeiten zu können, gibt es in C sogenannte **Strukturen**.

Diese Strukturen zeichnen sich dadurch aus, dass sie unterschiedliche Daten aufnehmen können, warum nicht auch Zeiger auf diese Strukturen? Und schon öffnet sich das weite Feld der dynamischen Datenstrukturen. In diesem Kapitel steigen wir in diese Thematik ein und stellen **Lineare Listen** vor.

9.1 Die Datenstruktur Struct

Zur Verwaltung heterogener Daten gibt es in C den Datentyp ***struct***. Der Datentyp *struct* fasst sozusagen mehrere Variablen zu einer Einheit zusammen. Betrachten wir als Beispiel ein Telefonbuch. Dieses enthält gewöhnlich pro Eintrag folgende Informationen:

Name
Vorname
Adresse
Telefonnummer

Liegen Tausende von Einträgen vor, so können wir alle vier Einzeldaten zu einer Zeichenkette zusammen fassen und das Telefonbuch als Feld von Zeichenketten anlegen. Der Nachteil ist, dass sich dieses Telefonbuch nur nach dem ersten Begriff einfach sortieren lässt.

Eine weitere Idee wäre, für jede Information wie Name oder Vorname ein eigenes Feld anzulegen. Da diese vier einzelnen Felder aber voneinander unabhängig sind, muss sichergestellt sein, dass alle Änderungen auch synchron erfolgen. Eine Änderung in nur einem Teil der vier Felder hätte fatale Folgen.

Beide Lösungen überzeugen daher nicht. Jetzt kommen Strukturen ins Spiel. Wir fassen die vier Elemente zu einer Einheit zusammen, wobei der Zugriff auf die Einzeldaten erhalten bleibt:

```
struct telefonEintrag
   {   char name [20];
       char vorname [15];
       char adresse [40];
       int nummer;
   } ;                                    // Semikolon am Ende ist wichtig!!
```

Mit dieser Definition wird in C der neue Datentyp *struct telefonEintrag* und in C++ der Datentyp *telefonEintrag* definiert. In C++ ist die Struktur also allein durch den Namen der Struktur definiert, in C muss der Bezeichner *struct* immer mit angegeben werden. Ein Datentyp definiert noch keine Variable, dies wollen wir jetzt nachholen:

```
struct telefonEintrag  hans, erna ;                    /* C-Version */
struct telefonEintrag telefonBuch [10000] ;
```

In C++ vereinfacht sich dies zu:

```
telefonEintrag hans, erna ;                            // C++ Version
telefonEintrag telefonBuch [10000] ;
```

Wir haben damit zwei Variablen *hans* und *erna* definiert und zusätzlich ein Feld *telefonBuch* mit 10000 Eintragsmöglichkeiten mit dem zugrundeliegenden Datentyp *(struct) telefonEintrag* definiert. Die Definition der Struktur *telefonEintrag* und dieser drei Variablen hätten wir auch zusammen fassen können:

```
struct telefonEintrag
   {   char name [20];
       char vorname [15];
       char adresse [40];
       int nummer;
   } hans, erna, telefonBuch [10000] ;
```

Wieder erkennen wir, dass am Ende einer Strukt-Definition ein Semikolon stehen muss, auch dann, wenn keine Variablen deklariert werden. Der Struktur-Name (hier: *telefonEintrag*) kann auch weggelassen werden, wenn diese Struktur im weiteren Verlauf des Programms nicht explizit verwendet wird.

Auf einzelne Einträge in einer Struktur greifen wir mit Hilfe des Punkt-Operators zu. Wir geben etwa die Telefonnummer und den Familiennamen von *hans* aus mittels:

```
cout << "Die Telefonnummer von Hans ist " << hans.nummer << endl;
cout << "Mit Familiennamen heißt Hans: " << hans.name << endl;
```

Wir haben diesen Operator schon bei den Ein- und Ausgabeklassen kennen gelernt, und tatsächlich sind Strukturen und Klassen sehr eng miteinander verwandt. Der wesentliche Unterschied ist der, dass Strukturen in C nur Variablen, nicht aber zusätzlich Funktionen aufnehmen können.

Zu beachten ist weiter, dass es sich bei Strukturen nicht wie bei Feldern um Zeiger handelt. Folgende

Anweisung (in C++) kopiert tatsächlich die gesamten Daten in die neue Variable *sepp*:

```
telefonEintrag sepp;
sepp = hans;                    // Kopie
```

Der von *sizeof* zurückgelieferte Wert (hier ca. 80) gibt den benötigten Speicher für die Struktur an. Wird der sizeof-Operator auf die Feldvariable *telefonBuch* angewendet, so wird das Ergebnis ungefähr den Wert 800000 ergeben. Auch hier wird also der Speicher komplett angefordert, im Speicher werden 10000 Einträge mit je ca 80 Byte angelegt!

Wir haben schon auf die einzelnen Elemente einer Struktur zugegriffen. Natürlich können wir auch auf Strukturen von Strukturen, auf Felder von Strukturen oder Strukturen von Feldern zugreifen. Auch können Funktionen Strukturen als Rückgabewert besitzen. Immer sind die Strukturen selbst aber keine Zeiger. Betrachten wir ein Beispiel, wo eine Funktion eine Struktur erzeugt, vorbelegt und als Funktionsergebnis zurückliefert. Wichtig ist hier zu wissen, dass Rückgaben wie Funktionsübergaben call-by-value erfolgen. Es werden also Kopien zurückgeliefert! Dies ist hier zwar nicht performant, aber die Struktur existiert auch noch nach dem Beenden der Funktion!

```
telefonEintrag vorbelegen(void)
{
    telefonEintrag temp;                          // belegt Speicherplatz
    temp.name[0] = temp.vorname[0] = '\0';        // leere Zeichenketten
    temp.adresse[0] = '\0';
    temp.nummer = 0;
    return temp;                                  // kopiert Ergebnis in die Rückgabe
}
```

Diese Vorbelegung können wir auch abkürzen, da wir Strukturen auch direkt initieren können:

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

Ein Aufruf dieser Funktion, etwa durch

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

reserviert und belegt Speicher für die lokale Variable *temp*, kopiert diesen beim Beenden der Funktion in die Struktur *peter* und gibt sofort danach den Speicher der Variable *temp* frei.

Betrachten wir nun unser Telefonbuch mit bis zu 10000 Einträgen. Die Variable *telefonBuch* ist eine Feldvariable, die in jedem Feldelement eine Struktur enthält. Drei der Strukturelemente sind wieder Felder. Dies scheint kompliziert zu sein, ist es aber nicht, wenn wir nur konsequent zugreifen. Beginnen wir mit ein paar Ausdrücken:

```
telefonBuch                     // identifiziert das gesamte Feld
telefonBuch[ 13 ]               // greift das 13. Element des Feldes heraus
telefonBuch[ 13 ] . nummer      // greift auf die Struktur nummer des 13. El. zu
```

Eine Überprüfung des ersten Buchstabens des Namens des 1017. Elementes auf den Großbuchstaben *M* könnte dann wie folgt aussehen:

```
if ( telefonBuch[ 1017 ] . name [ 0 ] == 'M' ) …
if ( *(telefonBuch + 1017) . (*name) == 'M' ) …
```

Im zweiten Beispiel haben wir die Indizierung von Feldern durch die Zeigerschreibweise ersetzt. Es sind hier Klammern erforderlich, da der Punkt stärker als der Stern bindet.

Das Besondere bei Strukturen ist, dass wir mit Hilfe des Punktoperators jederzeit auf die einzelnen Elemente einer Struktur zugreifen können. Andererseits können wir aber auch die gesamte Struktur verwenden. Wir haben damit sehr mächtige Zugriffsmöglichkeiten. Sind beispielsweise Sepp und Hans Brüder, die zusammen wohnen und ein gemeinsames Telefon benutzen, so kann Sepp wie folgt angelegt werden:

```
sepp = hans ;                          // vollständige Kopie der gesamten Struktur
strcpy( sepp.vorname, "Sepp" );        // Anpassung des Vornamens
```

oder durch:

```
strcpy( sepp.name, hans.name ) ;       // der Familienname wird kopiert
strcpy( sepp.vorname, "Sepp" );        // Setzen des Vornamens
strcpy( sepp.adresse, hans.adresse ) ; // die Adresse wird kopiert
sepp.nummer = hans.nummer ;            // die Telefonnummer wird kopiert
```

Ist nun Hans der 317. Eintrag im Telefonbuch, so können wir setzen:

```
telefonBuch[ 317 ] = sepp ;
```

Wie bereits erwähnt wurde, sind Strukturen keine Zeiger. Strukturen können in der Praxis sehr groß werden. Es empfiehlt sich daher bei der Parameterübergabe an Funktionen, Zeiger statt der gesamten Struktur zu übergeben. Zeiger auf Strukturen sind auch bei Linearen Listen sehr wichtig. Ein Zeiger auf eine Struktur kann wie folgt deklariert und verwendet werden:

```
telefonEintrag * pHans;            // Definition eines Zeigers
pHans = &hans;                     // Zeiger zeigt auf existierende Struktur
(*pHans).nummer = 123456;          // Telefonnummer angeben
```

Wegen der hohen Bindungsstärke des Punktes sind Klammern erforderlich. Da diese Art des Zugriffs häufig vorkommt, das Schreiben der Klammern aber lästig ist, existiert in C eine Abkürzung: der Operator ‚->' (Pfeil). Die letzte obige Zuweisung ist äquivalent zu:

```
pHans->nummer = 123456;
```

Der Pfeil besitzt die gleich hohe Bindungsstärke wie der Punkt. Im folgenden Ausdruck wird daher die Nummer und nicht der Zeiger auf die Struktur erhöht:

```
++pHans->nummer;
```

Die Strukturen werden im übernächsten Abschnitt noch intensiv verwendet. Wir wollen hier zum Schluss nur noch die Verwendung von Strukturen einschließlich des Reservierens von dynamischen Speicher angeben. Es fällt auf, wie wesentlich mächtiger und einfacher die Reservierung von dynamischen Speicher in C++ ist:

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

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

```
// nur in C++:

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

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

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

Zunächst sehen wir, dass wir in C++ das Wort *struct* weglassen dürfen, außer natürlich in der Deklaration des neuen Datentyps *list*. Dies haben wir bereits behandelt. Diese Struktur enthält einen Zeiger auf die gleiche Struktur und eine Int-Variable. Strukturen, die intern einen Zeiger auf die gleiche Struktur besitzen, sind die Basis für Lineare Listen, die wir im übernächsten Abschnitt kennenlernen werden. Diese Linearen Listen werden in der Regel dynamisch angelegt. Dabei sehen wir, dass bei der Speicherreservierung der Operator *new* wesentlich einfacher handhabbar ist als die schwerfällige *Malloc*-Funktion. Wir belegen noch den Speicher vor. Der Zeiger wird dabei auf den definierten NULL-Wert gesetzt. Weitere Details folgen im übernächsten Abschnitt.

9.2 Die Datenstruktur Union

Neben der Datenstruktur *struct* gibt es auch die Struktur *union*. Die Syntax beider Strukturen stimmt überein. Auch die feinen Unterschiede zwischen C und C++ sind identisch. Auch bei der Struktur *union* darf in C++ der Bezeichner *union* beim neu geschaffenen Datentyp einfach weggelassen werden. Der Unterschied zwischen *struct* und *union* liegt in der unterschiedlichen Speichertechnik: Alle Daten einer Struktur *struct* werden nacheinander angelegt, in der Struktur *union* aber überlappend, also übereinander.

Wir fragen uns, welchen Sinn das Überlappen von Speicher haben kann. Naheliegend ist natürlich die Minimierung des Speicherplatzes, wobei dies heute oft eine untergeordnete Rolle spielt. Eine weitere Anwendung ergibt sich aus der hardwarenahen Programmierung, wo gleiche Schnittstellen in verschiedenen Anwendungen unterschiedlich belegt sind.

Ein Beispiel zur Reduzierung des Speichers wäre das Speichern der Kenntnisse des Personals. Beim Büropersonal interessieren Erfahrungen in Office-Anwendungen, beim technischen Personal die Erfahrung in Programmiersprachen und beim Fahrpersonal die erworbenen Führerscheinklassen. Für jeden Mitarbeiter brauchen wir nur eine der drei Informationen, nicht alle zusammen. Mit folgender Definition sparen wir also erheblich an Speicher, ohne auf die entsprechenden Bezeichner verzichten zu müssen:

```
union kenntnis
{   char officeAnwendung[30];
    char programmSprache[30];
    char führerscheinKlasse[20];
} ;
```

Mittels des Operators *sizeof* werden wir feststellen, dass der neue Datentyp *kenntnis* nicht mehr als 30 Byte Speicher belegt. Wir können diesen Datentyp beispielsweise in einer Struktur einsetzen, wo Mitarbeiterdaten gespeichert werden:

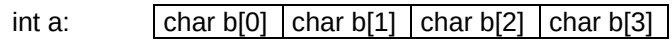
```
struct person
{   char name[20];
    char vorname[15];
    char strasse[20];
    int hausnummer;
    int postleitzahl;
    char ort[20];
    char abteilung[15];
    kenntnis erfahrung;        // in C: union kenntnis erfahrung;
    …
} mitarbeiter;
```

Ein Zugriff auf die Daten eines Mitarbeiters könnte dann wie folgt erfolgen:

```
if ( mitarbeiter.abteilung == "Verwaltung" )
    cout << "Office-Erfahrung: " << mitarbeiter.erfahrung.officeAnwendung << endl;
```

An diesem Beispiel erkennen wir, dass sich dem äußeren Betrachter Strukturen und Unions gleich präsentieren. Beim Zugriff werden also die Operatoren ‚.' (Punkt) und ‚->' (Pfeil) verwendet. Der Unterschied liegt tatsächlich nur in der Speichertechnik: bei Strukturen werden die Daten nacheinander, bei Unions übereinander angelegt.

Betrachten wir an einer Überlappung einer Int-Zahl mit vier vorzeichenlosen Zeichen die Idee der Schnittstellenprogrammierung. Wir zeigen mit Hilfe von Unions, wie Int-Variablen genau gespeichert werden. Dazu überlappen wir eine Int-Zahl mit vier vorzeichenlosen Char-Variablen:



Mit einem kleinen Programm wollen wir dies demonstrieren. Wir überlappen eine Int-Zahl mit vier Zeichen und geben diese vier Zeichen aus:

```
void uniondemo (const int zahl)
{  union
   {   int a;                  // 32 Bit Zahl
       unsigned char b[4];     // 4  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])<<' '
        <<setw(2)<<int(b[2])<<' '<<setw(2)<<int(b[3])<<endl;
}
```

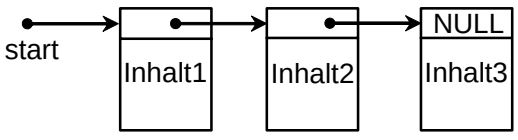
Funktion *uniondemo* in 09-struktur\union.cpp

Wir haben ins diesem Programm noch eine Besonderheit in C++ ausgenutzt: Wird in C++ in einer Union weder ein Datentyp noch eine Variable angegeben, so können wir direkt auf die Elemente der Struktur zugreifen. Damit sind in dieser Funktion *a* und *b* zwei „normale" Variablen, die allerdings den gleichen Speicherplatz belegen.

9.3 Lineare Listen

Wer hat sich nicht schon darüber geärgert, dass bei der Definition eines Feldes auch dessen Länge mit angegebenen werden muss. Immer wieder benötigen wir aber Strukturen, deren Länge variabel ist. Hier bieten sich sogenannte **dynamische Datenstrukturen** an. Die einfachsten Vertreter dieser Spezies sind die **Linearen Listen**.

Betrachten wir eine Lineare Liste, die aus drei Elementen besteht:



Mit einem Startzeiger verweisen wir auf das erste der drei Elemente. Jedes Einzelelement besteht aus einem Zeiger auf das nächste Element und einem Nutzinhalt, im Bild mit *Inhalt1*, *Inhalt2* und *Inhalt3*

bezeichnet. Weiter wird der Zeiger des letzten Elements auf den Wert NULL gesetzt. Das Besondere an einer Linearen Liste ist, dass an das Ende der Liste jederzeit weitere Elemente dynamisch hinzugefügt werden können, theoretisch so lange, bis der virtuelle Speicher voll ist.

Allgemein ist eine Lineare Liste wie folgt definiert (nach Wirth):

Eine **Lineare Liste** ist
entweder die leere Datenstruktur (Zeiger auf NULL)
oder eine Datenstruktur, die eine Lineare Liste als Nachfolger besitzt.

Diese rekursive Definition bedeutet, dass eine Lineare Liste aus beliebig vielen Einzelelementen bestehen kann, aufgelistet in einer Reihe. Es gibt also ein eindeutig definiertes erstes Element, ein zweites und so weiter bis zum Ende der Liste. Aus obiger Definition sehen wir, dass die leere Liste als Sonderfall mit enthalten ist. Dies ist natürlich klar, da eine Liste beim Erzeugen zunächst immer leer ist.

In den einzelnen Elementen einer Linearen Liste speichern wir ähnlich wie bei Feldern Daten ab. Der Vorteil bei Linearen Listen liegt darin, dass wir uns über die Feldgrenzen beim Erzeugen keine Gedanken machen müssen. Der große Nachteil ist, dass wir uns um den Aufbau dieser Listen selbst kümmern müssen.

Wir wollen nun die Struktur einer Linearen Liste in der Programmiersprache C definieren. Gehen wir davon aus, dass wir als Inhalt eine Zeichenkette speichern wollen, so sieht diese Struktur wie folgt aus:

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

Wie wir am obigen Beispiel mit den drei Listenelementen erkennen, besitzt ein Element immer einen Zeiger auf den Nachfolger und Daten, hier eine Zeichenkette. Ein einzelnes Listenelement ist daher eine Struktur. Diese Struktur belegt vermutlich 84 Byte, 4 Byte für den Zeiger und 80 Byte für die Zeichenkette. Die tatsächliche Größe kann durch den Ausdruck *sizeof (list)* ermittelt werden.

Es fällt noch auf, dass wir in der Definition der Struktur einen Zeiger auf genau diese Struktur verwenden. Dies ist in C und anderen Programmiersprachen erlaubt. Es gilt: Ein Zeiger auf einen Datentyp darf verwendet werden, auch wenn dieser Datentyp erst später definiert wird.

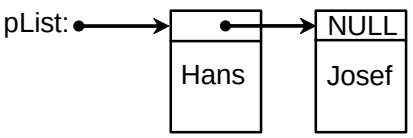
Wir wollen nun eine Liste erzeugen. Wir definieren eine Variable *pList* als Zeiger auf die Liste, genauer auf den Listenanfang. Dieser Anfangszeiger auf die Liste wird auch als **Anker** bezeichnet. Bei Programmstart ist die Liste zunächst leer. Dies wollen wir dadurch untermauern, dass wir unseren Listenzeiger auf *NULL* setzen:

```
list pList;
pList = NULL;
```

Die Lineare Liste hat somit folgendes triviales Aussehen:

```
pList:  NULL
```

Fügen wir jetzt zwei Elemente mit dem Inhalt ‚Josef' und ‚Hans' hinzu, so ergibt sich:



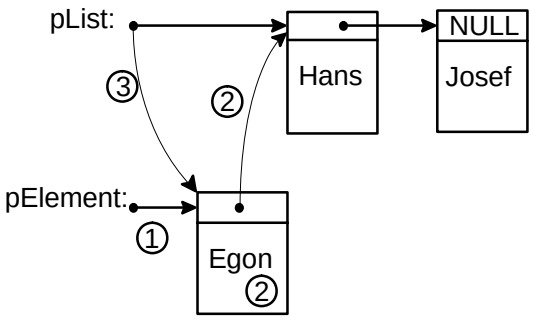
Die Funktionsweise dieses Hinzufügens wollen wir jetzt am Beispiel des Einfügens eines Elements mit dem Inhalt ‚Egon' kennenlernen. Dieses Einketten in die bestehende Lineare Liste geschieht am ein-

fachsten am Anfang dieser Liste. Wir gehen dazu in folgenden Schritten vor:

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

(2) Dieses neue Element wird mit Inhalt versehen. Wir schreiben also Text in die Zeichenkette *inhalt* und setzen den Zeiger auf den bisherigen Listenanfang

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



Betrachten wir die Realisierung dieses Einkettens:

```
void einkettenElement( const char * name, list * & p )
{
   list * pElement;

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

Funktion *einkettenElement* in 09-struct\liste.cpp

Zu beachten ist:

– Eine Zuweisung der Form ‚*pElement->inhalt = name*' wäre nur eine Zeigerzuweisung und hier zudem falsch, da das Element *inhalt* als (konstanter) Feldzeiger definiert ist. Wir müssen also die Funktion *strcpy* verwenden!

– Die Funktion *einkettenElement* funktioniert auch bei zunächst leerer Liste. Die Liste muss allerdings mit ‚*p=NULL;*' im Hauptprogramm initiiert werden!

– Der Zeiger *p* auf die Lineare Liste muss call-by-Reference übergeben werden, damit auch der Zeiger des Hauptprogramms auf den neuen Listenanfang zeigt.

Wir wollen an Hand des Hauptprogramms den Aufruf der Funktion *einkettenElement* aufzeigen:

```
int main()
{  list *pList;
   char name[N];

   pList = NULL;                 // korrekt initiiert!!
   cout << "Listenprogramm\n\n
        << "Namen eingeben. Ende ist leerer String!\n";

   cin.getline(name, N);
   while ( strlen(name) != 0 )
   {  einkettenElement( name, pList );
      cin.getline(name, N);
   }                                        // ...
```

Anfang der Funktion *main* in 09-struct\liste.cpp

Im Hauptprogramm wird der Zeiger *pList* korrekt mit dem Wert NULL vorbelegt. Dieser Zeiger wird

mittels call-by-reference an die Funktion *einkettenElement* übergeben. Nach jedem dieser Funktionsaufrufe zeigt dann der Zeiger *pList* auf das zuletzt eingefügte Element.

Wichtig ist, dass dynamische Strukturen keinen Namen besitzen. Sie sind nur über entsprechende Zeigernamen des Hauptprogramms ansprechbar. Es ist daher besonders wichtig, dass hier beim Umbiegen von Zeigern keine Fehler gemacht werden. Wollen wir den Inhalt der drei obigen Elemente auf Bildschirm ausgeben, so müssen wir den Weg über den Zeiger *pList* gehen. Das erste Element lässt sich noch recht leicht ausgeben. Beim zweiten müssen wir den Umweg über das erste Element machen, beim dritten sogar den Umweg über das erste und zweite Element:

```
cout << pList->inhalt << pList->next->inhalt << pList->next->next->inhalt ;
```

Wir wissen, dass *pList* ein Zeiger ist. Somit ist **pList* das Element, auf das der Zeiger *pList* zeigt. Dies ist hier eine Struktur, deren Einzelinstanzen mittels eines Punktes angesprochen werden. Die beiden möglichen Strukturinstanzen sind hier also *(*pList).next* und *(*plist).inhalt*, wobei wir der Pfeilschreibweise den Vorzug geben: *pList->next* und *pList->inhalt*. Mit dem letzten Ausdruck geben wir also den Inhalt der Struktur aus. Die Ausgabe des Inhalts des zweiten Elements wird schon schwieriger. Schließlich verweist vom Hauptprogramm kein Zeiger auf dieses Element.

Auf das zweite Element verweist aber ein Zeiger des ersten Elements! Und auf diesen Zeiger kann wiederum über den Anker *pList* mittels *pList->next* zugegriffen werden. Vielleicht wird nun der Begriff Anker einleuchtender. Alle Zugriffe auf die gesamte Lineare Listen geschehen über diesen einen Zeiger! Da *pList->next* auf das zweite Element verweist, wird mit *pList->next->inhalt* der Inhalt des zweiten Elements angesprochen. Diese Argumentation wird dann auch auf das dritte Element angewendet. Als Ausgabe erhalten wir:

```
Egon  Hans  Josef
```

Zur Vertiefung betrachten wir noch zwei weitere Funktionen. Die Funktion *anzahl* liefert die Anzahl der Listenelemente zurück und die Funktion *ausgebenListe* gibt den Inhalt aller Listenelemente nacheinander aus:

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

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

Funktionen *anzahl* und *ausgebenListe* in 09-struct\liste.cpp

Mit der Funktion *anzahl* wird das Durchsuchen der gesamten Liste demonstriert. Vermutlich hatten wir uns im vorherigen Beispiel gefragt, wie wir auf das 117. Element der Liste zugreifen sollen. Die Form

```
pList->next->next->next->next->next->next->…
```

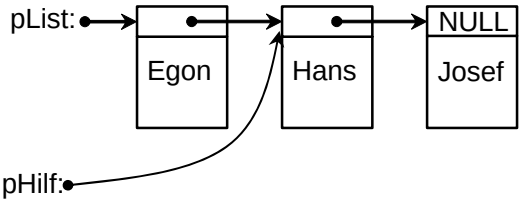
ist nicht gerade vielversprechend. Aus der obigen Funktion *anzahl* ersehen wir, dass wir stattdessen den Listenzeiger *p* in einer Schleife durch die Zuweisung

```
p = p->next;
```

einfach fortgeschaltet haben. Dies entspricht dem Weiterzählen einer Schleifenvariablen auf den nächsten Wert (z.B. mittels *i++*). Der Abbruch erfolgt, wenn der Zeiger eines Elements auf *NULL* zeigt. Wir haben unsere Liste entsprechend aufgebaut: der Zeiger des letzten Elements zeigt immer auf *NULL*. Dies ist beim Arbeiten mit Listen ganz wichtig. Noch erwähnt sei, dass sich das Ändern des Parameters *p* nicht auf das Hauptprogramm auswirkt. Der Zeiger wurde schließlich als Kopie übergeben! Ganz analog arbeitet auch die Funktion *ausgebenListe*.

Beim Ablaufen des Programms bemerken wir, dass die Elemente gegenüber der Eingabe in umgekehrter Reihenfolge ausgegeben wurden. Das Einketten und wieder Ausgeben in unserem Listenbeispiel entspricht einem **Stack**. Wir sprechen auch von einem **Stapel** oder einem **Keller**. Anschaulich kann ein Stack mit einem Stapel Papier verglichen werden. Die zuletzt auf diesen Stapel gelegten Blätter werden wir zuerst wieder entfernen.

Diese Stackverwaltung findet in Programmen häufig Anwendung. Es sei nur auf die bei Compilern erforderliche Verwaltung rekursiver Programme erinnert. Hier wird auch die zuletzt geöffnete Funktion wieder geschlossen. Zur kompletten Stackverwaltung fehlt uns noch die Funktion zum Entfernen (Ausketten) eines Elementes. Wir wollen dieses Ausketten gleich aufzeigen:



Das Prinzip des Löschens ist einfach: Wir setzen den Hilfszeiger *pHilf* auf das zweite Element. Anschließend entfernen wir das erste Element aus dem Stack und biegen den Listenzeiger *pList* auf das bisherige zweite und neue erste Element. Dies funktioniert auch, wenn nur ein Element in der Liste enthalten ist. In diesem Fall wird *pHilf* auf NULL gesetzt und damit letzlich auch *pList*. Den Fall einer bereits leeren Liste dürfen wir nicht übersehen: hier geben wir die leere Zeichenkette zurück. Die komplette Funktion *auskettenElement* lautet:

```
void auskettenElement( char *name, list * &p )
{
   if (p == NULL)                // Liste ist bereits leer
   {  name[0] = '\0';            // Leerer String
      return;                    // Abbruch
   }
   list *pHilf = p->next;        // auf 2. Element setzen
   strcpy( name, p->inhalt );    // Inhalt zurueckliefern
   delete p;                     // 1. Element loeschen
   p = pHilf;                    // neuer Listenanfang
}
```

Funktionen *auskettenElement* in 09-struct\liste2.cpp

Wieder wird der Listenzeiger call-by-reference übergeben, da dieser in der Funktion geändert wird. Wichtig ist weiter, dass der Speicher des freiwerdenden Elements auch wirklich freigegeben wird.

Dieses Ein- und Ausketten am Anfang wird auch als **LIFO** bezeichnet. Dies ist eine Abkürzung für *Last In First Out*. Schließlich wird das zuletzt eingehängte Element als erstes wieder ausgehängt. Diese LIFO-Listen sind einfach implementierbar und werden daher gerne verwendet. Betrachten wir die Wartezeiten in diesen Listen, so geht es sehr ungerecht zu. Derjenige, der am kürzesten warten musste, wird aus der Liste herausgenommen.

Aus Sicht der Wartezeiten ist das Prinzip **FIFO** (*First In First Out*) wesentlich gerechter: Das am längsten wartende Element wird als erstes ausgekettet. Man findet es häufig bei Warteschlangen.

9.4 FiFo-Listen

Mit dem Ein- und Ausketten am Anfang ist eine Stackverwaltung möglich. Um auch FIFO-Listen zu realisieren, müssen wir auch am Ende der Liste ein- oder ausketten. Es gibt nämlich zwei Möglichkeiten:

a) Einketten am Anfang und Ausketten am Ende der Liste,
b) Einketten am Ende und Ausketten am Anfang der Liste.

Wir wollen hier den Fall b) behandeln. Der Fall a) sei zusätzlich als Übung empfohlen. Das Ausketten am Anfang haben wir bereits im letzten Abschnitt kennen gelernt. Wir müssen daher nur das Einketten am Ende der Liste neu programmieren. Dies ist allerdings nicht ganz einfach. Schließlich zeigt kein Zeiger auf das Listenende. Vielmehr müssen wir uns bis zum letzten Element durchhangeln. Wir müssen dabei aufpassen, dass wir wirklich beim letzten Element stehen bleiben und nicht erst beim Next-Zeiger des letzten Elements, denn dieser hat den Wert NULL. Wir hängen an diesem letzten NULL-Zeiger das neue Element ein. Wir müssen den Spezialfall einer vorliegenden leeren Liste extra behandeln, denn dann ketten wir direkt beim Anker ein. Die Funktion *einkettenEnde* ist daher recht umfangreich und lautet:

```
void einkettenEnde( const char *name, list * &p )
{
   if (p == NULL)                 // Liste ist leer
   {  p = new list;               // Element erzeugen
      p->next = NULL;             // Ende der Liste
      strcpy( p->inhalt, name );  // Inhalt fuellen
   }
   else
   {  list* hilf = p;

      while (hilf->next != NULL)  // Wichtig!!
         hilf = hilf->next;       // fortschalten

      hilf->next = new list;      // Liste am Ende anhaengen
      hilf       = hilf->next;
      strcpy( hilf->inhalt, name );  // Inhalt fuellen
      hilf->next = NULL;          // Ende der Liste
   }
}
```

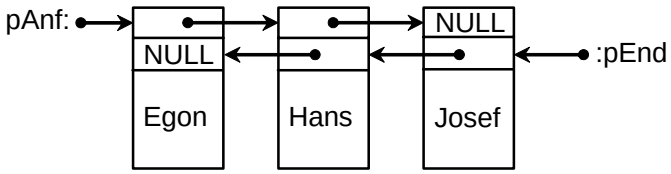
Funktionen *einkettenEnde* in 09-struct\liste2.cpp

In der While-Schleife wird überprüft, ob der Zeiger *hilf->next* ungleich NULL ist. Dies ist wichtig, damit am Ende der Schleife der Zeiger *hilf* auf das letzte Element zeigt. Nur so ist der Zeiger *next* des letzten Elements zugreifbar und damit manipulierbar. Da wir vorher bereits überprüft hatten, dass die Liste nicht leer ist, ist der Zeiger *hilf* nicht leer, wir können also mit Sicherheit auf den Zeiger *hilf->next* zugreifen.

Sowohl von der Komplexität als auch von der Laufzeit ist das Einketten am Ende aufwändig. Nicht einfacher wird es im obigen Fall a), da dort das Ausketten am Ende ähnliche Probleme bereitet. In der Praxis ist daher in FIFO-Listen das Arbeiten mit **doppelt verketteten Listen** vorzuziehen. Die Datenstruktur wird zwar ein wenig komplexer, dafür werden die Zugriffe einfacher und die Laufzeit besser. Die Datenstruktur muss entsprechend erweitert werden:

```
struct list
{   list * next, * before ;          // zwei Zeiger!
    char inhalt [N] ;
} ;
```

Unsere doppelt verkettete Liste selbst besitzt dann etwa folgendes Aussehen. Zu beachten sind die jetzt zwei Zeiger *pAnf* und *pEnd* als Zugang zur Liste.



Ist nun eine solch doppelt verkettete Liste mit zwei Zeigern auf den Anfang und das Ende der Liste gegeben, so sieht unsere Prozedur zum Einketten am Anfang wie folgt aus:

```
void einkettenElement(const char*name, list*&anf, list*&end)
{  list * pElement;

   pElement = new list;            // 1: Speicher belegen
   strcpy(pElement->inhalt,name);  // 2: Inhalt eingeben
   pElement->next = anf;           // 2: vorn verketten
   pElement->before = NULL;        // 2: rueck verketten

   if (anf == NULL)       // bisher leere Liste
      anf = end = pElement;        // beide Zeiger setzen!
   else
   {  anf->before = pElement;      // 3: neu einhaengen
      anf = pElement;              // 3: neu einhaengen
   }
}
```

Funktionen *einkettenEnde* in 09-struct\doppelliste.cpp

Als Übungsaufgabe sei das Ausketten am Ende dringend empfohlen. Der Aufwand ist nicht größer als bei obiger Einkettprozedur, da direkt über das Listenende auf das letzte Element zugegriffen werden kann. Als Hinweis sei empfohlen, die Funktion *auskettenElement* des Auskettens am Anfang als Vorlage zu verwenden. Eine zusätzliche Fallunterscheidung nach einer einelementigen Liste ist erforderlich, da in diesem Fall auch der Listenanfang auf *NULL* zu setzen ist! Quasi spiegelsymmetrisch verläuft auch das Ausketten am Anfang und Einketten am Ende. Dies sei ebenfalls zur Übung empfohlen.

9.5 Lineare geordnete Listen

Bisher betrachteten wir Lineare Listen nur im Zusammenhang mit dynamischer Zwischenspeichung und mit Warteschlangen. Sie eignen sich aber auch ausgezeichnet, um Daten geordnet abzulegen. Im Gegensatz zu statischen Feldern ist es hier ein Leichtes, Elemente nachträglich entsprechend ihrer Anordnung einzutragen. Nachfolgende Elemente müssen nicht verschoben werden. Auch das Löschen von Elementen ist einfach, Lücken bleiben nicht zurück. Wir benötigen dazu allerdings neue Algorithmen, und zwar zum Ein- und Ausketten an einer beliebigen Stelle innerhalb einer geordneten Liste. Hierzu müssen wir die Stelle des Ein- und Auskettens zunächst suchen und finden.

In geordneten Listen sind die Daten nach einem Sortierkriterium abgelegt, bei Zahlen etwa nach der Größe einer Zahl, bei Namen nach der alphabetischen Reihenfolge. Wir werden in unserem Beispiel die Namen nach dem ASCII-Codes anordnen. Gleichzeitig kehren wir zu einfach verketteten Listen zurück.

Im Sinne der strukturierten Programmierung betten wir das Erzeugen und Vorbelegen eines neuen Elements in einer eigenen Funktion *erzeuge* ein. Dieser Funktion wird ein Name übergeben. Die Funktion erzeugt ein neues Strukturelement, trägt die notwendigen Daten ein und liefert den Zeiger auf dieses Element zurück. Die Funktion lautet:

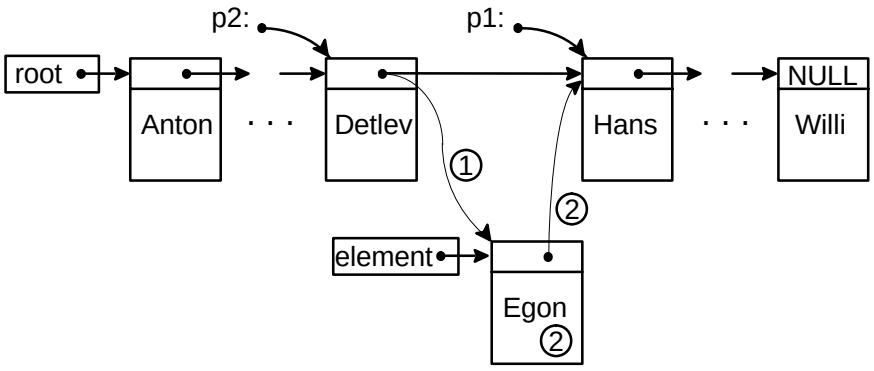
```
list * erzeuge( const char * name )
{  list * element = new list;
   strcpy( element->inhalt, name );
   element->next = NULL;
   return element;
}
```

Funktion *erzeuge* in 09-struct\suche.cpp

Wir kommen nun zur zentralen Funktion *suche*. Diese Funktion sucht den richtigen Platz für das neu einzufügende Element gemäß der vorgegebenen Sortierung und fügt dann ein. Der Funktion werden zwei Parameter übergeben, ein Zeiger auf den Listenanfang und ein Zeiger auf das mit der Funktion *erzeuge* neu generierte Element. Wir müssen hier wieder einige Fälle unterscheiden. Beginnen wir daher mit einer Analyse des Algorithmus. Zu beachten ist:

1) Die Liste kann leer sein: Es wird direkt am leeren Listenanfang eingehängt. Der Listenanfang muss daher call-by-reference übergeben werden.
2) Das neue Element kann in der nichtleeren Liste am Anfang eingekettet werden. Wieder ändert sich der Listenanfang, wir benötigen call-by-reference.
3) Wird das neue Element in der nichtleeren Liste nicht am Anfang eingekettet, so müssen wir den geeigneten Platz suchen.

Im dritten Fall haben wir bei der Suche ein Problem: Wenn wir ein Listenelement gelesen haben, wissen wir, ob das neue Element davor einzuhängen ist. Um aber vor einem Element einzuketten, benötigen wir einen Zeiger auf das Vorgängerelement, um den darin befindlichen Next-Zeiger umbiegen zu können. Wir werden daher folgenden bewährten Weg gehen: Wir arbeiten mit zwei Zeigern *p1* und *p2*, wobei der Zeiger *p1* auf das aktuelle Element zeigt und *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, an der das gewünschte Element (hier: *Egon*) einzuhängen ist.



Der Algorithmus im dritten Fall lautet daher:

Setze Zeiger p1 auf den Listenanfang
Solange p1 nicht NULL und der Inhalt des Listenelements (auf den p1 zeigt) kleiner oder gleich dem einzufügenden Inhalt ist:
    Setze p2 auf p1
    Setze p1 auf das nächste Element
Listenplatz wurde gefunden: Zwischen den beiden Zeigern p1 und p2 einketten (Grafik)

Mit diesen Überlegungen können wir unsere Funktion *suche* programmieren:

```
void suche( list * & root, list * element )
{  if (root == NULL)            // 1. Fall: Liste leer
       root = element;
   else
    if ( strcmp( root->inhalt, element->inhalt ) > 0 )
    {  element->next = root;    // 2. Fall: Einketten Anfang
       root = element;
    }
    else                        // 3. Fall: allgemein
    {  list * p1, *p2;             // Laufzeiger

       p1 = root;
       do                           // Fortschalten
       {   p2 = p1;    p1 = p1->next;
       }
       while ( p1 != NULL &&
               strcmp( p1->inhalt, element->inhalt ) <= 0 );

       p2->next = element;        //  einketten
       element->next = p1;
    }
}
```

Funktion *suche* in 09-struct\suche.cpp

Besonders zu beachten ist das Abbruchkriterium in der Do-While-Schleife. Wegen der garantierten Bewertung eines Ausdrucks von links nach rechts bei Verwendung des Und-Operators (‚&&') überprüfen wir zunächst den Zeiger *p1* auf NULL. Nur wenn der Zeiger *p1* ungleich NULL ist, wird dann auf den Inhalt zugegriffen.

Bei sehr großen Datenbeständen ist ein Durchsuchen langer Listen sehr zeitaufwändig. Hier greifen wir auf weitere verwandte dynamische Datenstrukturen zurück, auf ***Bäume***. Diese Bäume sind wiederum die Basis für B-Bäume. Hier sind wir dann schon bei der Verwaltung komplexer Daten angelangt. Dies ist ein Ausblick auf weitere Möglichkeiten. In dieser Vorlesung wollen wir das Kapitel Dynamische Datenstrukturen beenden und es beim Ausblick auf diese Möglichkeiten bewenden lassen.

10 Windows-Oberflächen

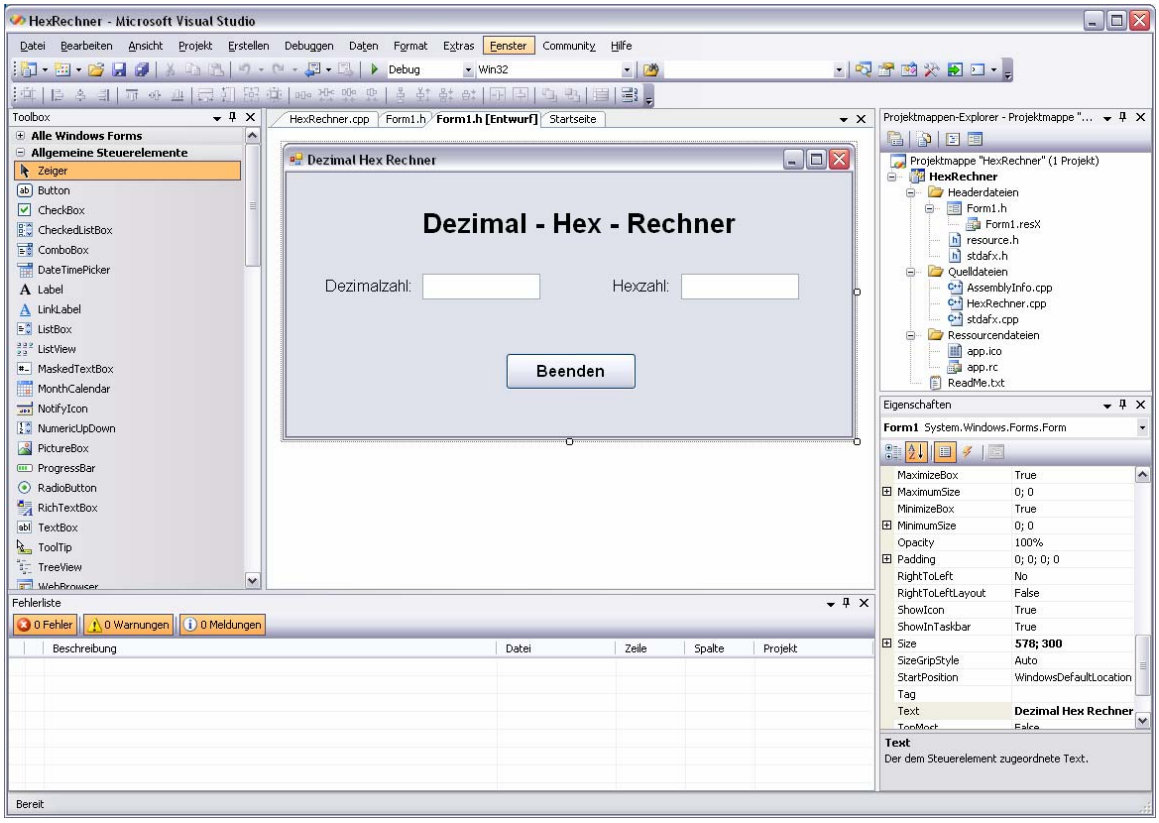
Bis etwa 1990 war das Thema Oberflächen nur für Spezialisten von Interesse. Die Rechner waren zu jener Zeit einfach zu langsam, um in Echtzeit Grafikdaten zu verarbeiten. Seit 1995 setzten sich die Oberflächen durch, die angenehme und einfache Bedienung waren der Garant für ihren Siegeszug. Eine der ersten wirklich guten Entwicklungsumgebung für Oberflächenprogrammierung waren Borland Delphi und Microsoft Visual Basic. Seit 2003 erobert Visual Studio .NET die Software-Entwicklungsabteilungen.

Alle modernen Entwicklungsumgebungen für Oberflächenprogrammierung arbeiten objektorientiert. Jedes Fenster, jede Texteingabebox, jeder Button, einfach alles sind Objekte. Es fällt daher nicht leicht, in die Oberflächenprogrammierung einzusteigen, ohne uns mit Objektorientierung beschäftigt zu haben. Trotzdem wollen wir dieses Experiment mit Visual Studio .NET wagen.

Mit Visual Studio wurde die Sprache C# ausgeliefert. Visual Studio und C# sind optimal aufeinander abgestimmt. Weiter werden aber auch Visual Basic und C++ unterstützt. Wir wollen mit unseren C++-Kenntnissen ein kleines aber interessantes Programm erstellen. Es sei aber gewarnt: Nicht alles ist sofort zu verstehen: Mit der Objektorientierung und Hunderten von vordefinierten Klassen in Visual Studio stoßen wir in absolutes Neuland. Aber wie heißt es so schön: Wer nicht wagt, der nicht gewinnt.

Wir wollen ein kleines Programm entwerfen, das automatisch Zahlen in das Hexformat überführt und umgekehrt. Dazu gibt es zwei Texteingabefelder. In das eine Feld werden Zahlen im Dezimalformat, im zweiten Zahlen im Hexformat eingegeben. Automatisch wird das jeweilige andere Feld angepasst. Der Button schließt schließlich die Anwendung.

Doch beginnen wir am Anfang. Wir öffnen Visual Studio, starten ein neues Projekt, wählen die Sprache C++ aus und hier eine Windows Forms-Anwendung. Jetzt müssen wir noch einen Namen (wie wäre es mit HexRechner?) und ein Verzeichnis eingeben. Visual Studio wird jetzt das eigentliche Arbeitsfenster anzeigen:



Natürlich ist unser Formular in der Mitte noch völlig leer. Wir haben noch ein wenig Arbeit vor uns. Auf der linken Seite sehen wir eine Toolbox, die die wichtigsten Oberflächenelemente für uns bereit hält. Wir bedienen uns hier mit zwei Textboxen, drei Labels (zwei Anzeigen und eine Überschrift) und einem Button. Wir klicken einfach auf das gewünschte Element und ziehen dieses dann im Formular in die gewünschte Größe.

Auf der rechten Seite finden wir oben den Projektmappen-Explorer, der uns einen Überblick über die verwendeten und erzeugten Dateien des Projektes gibt. Es wurde ein Hauptprogramm *HexRechner.cpp* angelegt, der auch die Funktion *main* enthält. In der Datei *Form1.h* wird unser noch zu erzeugender Code angelegt.

Unten rechts werden die zahlreichen Eigenschaften unserer Elemente angezeigt. Hier können wir Eigenschaften anzeigen lassen, aber auch ändern. Viele automatisch generierte Namen sind nicht aussagekräftig. So ändern wir etwa die Namen *button1* in *buttonEnde*, *textBox1* in *textBoxDez*, *textBox2* in *textBoxHex* usw. Auch gefällt die Standardüberschrift des Formulars nicht. Wir geben im Eigenschaftenfeld *Text* etwa ein: *Dezimal Hex Rechner*. Wir ändern die Schriftgröße und –stärke mittels der Eigenschaft *Font*. Wir ziehen die einzelnen Elemente auf die gewünschte Größe oder ändern diese Eigenschaften direkt im Eigenschaftenbereich. Beim ersten Mal dauert dies alles ein wenig, doch schließlich werden wir das obige Fenster nachgebildet haben.

Dies alles hat zunächst nichts mit Programmieren zu tun. Bevor wir aber mit dem eigentlichen Programm anfangen, sei auf wichtige Unterschiede zwischen einer Konsolen- und einer Windowsanwendung hingewiesen:

Eine **Konsolenanwendung** beginnt immer in der ersten Zeile der Funktion *main* und arbeitet das Programm sozusagen von oben nach unten durch.

Eine **Windowsanwendung** wartet in einer Schleife auf Ereignisse, also Eingaben durch den Benutzer. Dies können Tastatureingaben oder Mausklicks oder auch nur Mausbewegungen sein. Eine Windowsanwendung ist komplett ereignisgesteuert.

In Windowsprogrammen reagieren wir also ausschließlich auf Ereignisse. Ein Ereignis ist etwa ein Mausklick auf einen Button. Wird auf unseren Button *buttonEnde* geklickt, so soll das Programm enden. Dies müssen wir aber unserem Programm erst beibringen. Dazu führen wir einen Doppelklick auf den Button aus. Es wird ein Programmfenster angezeigt. Gleichzeitig wird eine neue Funktion *buttonEnde_Click* generiert. Der Cursor wird auf die erste Zeile dieser noch leeren Funktion gesetzt. Wir geben ein:

```
Close( ) ;
```

Damit wird die gleichnamige Funktion *Close* der Klasse *Form1* aufgerufen. Diese Funktion beendet ein Fenster. Ist dies das Hauptfenster, so wird das Programm beendet. Nach dem Abarbeiten eines Ereignisses bzw. der dazugehörigen Funktion kehrt ein Windowsprogramm immer automatisch in den Wartezustand zurück (wenn es nicht vorher beendet wurde).

Dies ging ja wie von Geisterhand. Aber was steckt dahinter? Um dies zu erfahren, kehren wir durch Klick auf den entsprechenden Reiter (*Form1.h [Entwurf]*) in der Mitte der Visual Studio Oberfläche zum Formular zurück. Wir klicken vorsichtig einmal auf den Button und konzentrieren uns auf den Eigenschaftenabschnitt rechts unten. Dort existiert ein Icon mit einem Blitz. Wir klicken darauf und schon werden alle denkbaren Ereignisse angezeigt. Neben dem Ereignis *Click* finden wir unsere Funktion *buttonEnde_Click*. Diese Zuordnung wird auch im Programm intern gemerkt. Somit weiß Visual Studio immer, welche Funktion bei welcher Aktion aufzurufen ist.

Es gibt etwa 50 verschiedene Ereignisse. Wir können uns diese gerne ansehen. Einige Ereignisse beginnen mit dem Wort *Mouse* für Mausereignisse, andere mit dem Wort *Key* für Tastaturereignisse. Wir wollen uns in diesem Programm auf vier dieser Ereignisse konzentrieren:

```
Click                   // Aufruf durch Mausklick
KeyDown                 // Aufruf beim Drücken einer Taste
KeyPress                // Aufruf beim Drücken einer Taste
KeyUp                   // Aufruf beim Loslassen einer Taste
```

Den Klick auf den Button haben wir bereits behandelt. Es fehlen noch die Eingaben in die beiden Textboxen. Damit unser Programm sicher gegenüber Falscheingaben ist, wollen wir mit dem Ereignis *KeyPress* fehlerhafte Eingaben abfangen und mit dem Ereignis *KeyUp* die eigentliche Berechnung durchführen. Wir klicken dazu auf die Textbox *textBoxDez* und klicken dann doppelt auf das weiße Feld neben dem Ereignis *KeyPress*. Es wird dann automatisch eine Funktion *textBoxDez_KeyPress* generiert. Wir ergänzen diese Funktion durch folgenden Code:

```
private: System::Void textBoxDez_KeyPress(
           System::Object^  sender,
           System::Windows::Forms::KeyPressEventArgs^  e) {
 if (!(e->KeyChar>='0' && e->KeyChar<='9' || e->KeyChar==8))
    e->Handled = true;
   // 8 = Backspace, Handled = true: bricht die Eingabe ab
}
```

Funktion *textBoxDez_KeyPress* in 10-windows\HexRechner\form1.h

Wir müssen den Funktionskopf etwas genauer betrachten. Die Funktion besitzt zwei Parameter mit den Namen *sender* und *e*. Praktisch jede Ereignisfunktion besitzt den Parameter *sender*. Mit Hilfe dieses Wertes lässt sich ermitteln, welcher Button oder welche Textbox dieses Ereignis auslöste. In unserem Fall ist dies jedoch klar. Der zweite Parameter *e* ist ein Argument, das uns Auskunft über die gedrückte Taste

gibt. Der Parameter *e* ist ein Zeiger auf ein Objekt. Dieses Objekt besitzt unter anderem die Elemente *KeyChar* und *Handled*. Im Element *KeyChar* ist die soeben gedrückte Taste abgelegt. Wir greifen darauf mittels *e->KeyChar* zu. Wurde keine Zifferntaste gedrückt und auch nicht die Backspace-Taste ‚←', so wollen wir die Eingaben nicht annehmen. Hier kommt das Element *Handled* ins Spiel. Setzen wir diese Variable auf den Wert *true*, so wird die soeben vorgenommene Eingabe verworfen. Wir garantieren damit, dass nur Ziffern eingelesen werden können.

In der Textbox *textBoxHex* müssen wir auch die Buchstaben von *a* bis *f* zulassen. Die Funktion *textBoxHex_KeyPress* sieht daher ein wenig anders aus:

```
private: System::Void textBoxHex_KeyPress(
           System::Object^  sender,
           System::Windows::Forms::KeyPressEventArgs^  e) {

 if (!(e->KeyChar>='0' && e->KeyChar<='9' ||
       e->KeyChar>='A' && e->KeyChar<='F' ||
       e->KeyChar>='a' && e->KeyChar<='f' || e->KeyChar==8))
    e->Handled = true;
}
```

Funktion *textBoxHex_KeyPress* in 10-windows\HexRechner\form1.h

Natürlich können wir das Programm schon testen. Wir sehen, dass nur erlaubte Zeichen angenommen werden. Die Funktionalität selbst fehlt aber noch. Unser Ein- und Ausgabemedium ist nicht mehr die Konsole. In diesem Programm sind es die beiden Textboxen. Wir benötigen daher keine Klassen *istream* und *ostream* sondern die Zeichenketten *textBoxDez->Text* und *textBoxHex->Text*.

Leider arbeitet die Oberfläche nicht mit nullterminierten Zeichenketten, sondern mit einer Klasse **String**^, die von C# übernommen wurde. Microsoft spricht hier von „Managed C++". In dieser Klasse gibt es unter anderem das Element *Length*, das die Länge der Zeichenkette zurückgibt. Weiter gibt es zu den Standardvariablen eine Methode **ToString**, die eine Variable in eine Zeichenkette verwandelt, also unsere Funktion *itos* aus Kapitel 5 nachbildet. Wird dieser Methode der Parameter *"X"* mit übergeben, so wird die Variable in einen Hexstring verwandelt, entsprechend unserer Funktion *itob* mit der Basis 16.

Zuletzt benötigen wir noch eine Funktion, die den Eingabetext von einer Zeichenkette in eine Zahl umwandelt. Hier bietet Visual Studio die Klasse *Convert* mit der statischen Methode *Convert::ToInt64* für 64Bit-Zahlen an. Mit diesem Wissen können wir unsere Funktion *textBoxDez_KeyUp* schreiben:

```
private: System::Void textBoxDez_KeyUp(
           System::Object^  sender,
           System::Windows::Forms::KeyEventArgs^  e) {

   long long zahl;
   String^ s;

   if (textBoxDez->Text->Length == 0)
      textBoxDez->Text = "0";

   zahl = Convert::ToInt64(textBoxDez->Text);   // in Zahl
   s = zahl.ToString("X");                      // Zahl in Hex
   textBoxHex->Text = s;           // Ergebnis ausgeben
}
```

Funktion *textBoxDez_KeyUp* in 10-windows\HexRechner\form1.h

Im Wesentlichen wandeln wir die Texteingabe aus der Textbox *textBoxDez* in eine Zahl um, führen Sie dann in eine Hexzahl über und geben diese in die Textbox *textBoxHex* aus. Das Umwandeln des Textes in eine Zahl misslingt, wenn die Textbox leer ist. Aus diesem Grund wird dies am Anfang überprüft und gegebenenfalls korrigiert.

Leider ist das Umwandeln einer Hex- in eine Dezimalzahl aufwändiger, da eine automatische Umwandelfunktion fehlt. Wir müssen dies vielmehr zu Fuß programmieren. Im Wesentlichen müssen wir eine umgekehrte Funktion *itob* verwenden. In der Funktion *itob* hatten wir Ziffern nacheinander mittels

Modulofunktion und Division abgesplittet. Hier multiplizieren wir die einzelnen Ziffer in einer Schleife hinzu. Wir verwenden dabei die Basis 16. Die Hexzahlen enthalten neben den Ziffern 0 bis 9 noch die Buchstaben A bis F. Diese werden in die Zahlen 10 bis 15 umgewandelt. Das grundsätzliche Verfahren ist ähnlich zur Funktion *textBoxDez_KeyUp*. Wir lesen den Text aus der Textbox *textBoxHex* aus, berechnen aus den einzelnen Ziffern die Dezimalzahl, wandeln diese wieder in einen Text um und geben ihn in der Textbox *textBoxDez* aus:

```
private: System::Void textBoxHex_KeyUp(
           System::Object^  sender,
           System::Windows::Forms::KeyEventArgs^  e) {

   const int BASIS = 16;
   long long zahl;
   String^ s;

   s = textBoxHex->Text;              // Einlesen in String
   zahl = 0;             // Startwert

   for (int i=0; i<s->Length; i++)   // Hex- in Dezimalzahl
   {  zahl = BASIS * zahl;
      if (s[i] >= '0' && s[i] <= '9')
         zahl = zahl + s[i] - '0';
      else if (s[i] >= 'A' && s[i] <= 'F')
         zahl = zahl + s[i] - 'A' + 10;
   }
   textBoxDez->Text = zahl.ToString();   // ausgeben
}
```

Funktion *textBoxHex_KeyUp* in 10-windows\HexRechner\form1.h

Wir haben noch eine Kleinigkeit vergessen. Wir haben nicht auf die Kleinbuchstaben *a* bis *f* geprüft. Dies konnten wir uns durch einen kleinen Trick ersparen. Textboxen besitzen die Eigenschaft *Character-Casing*. Diese Eigenschaft setzen wir in der Textbox *textBoxHex* auf den Wert *Upper*. Damit werden alle Eingaben von Kleinbuchstaben automatisch in Großbuchstaben umgewandelt.

Dieses kleine Beispiel zeigt die Mächtigkeit von Visual Studio auf, zeigt aber auch, dass es tausende von Funktionen und Variablen und genauso viele Tricks gibt. Hier führt wirklich erst die Übung zum Meister, und natürlich Erfahrung in objektorientierter Programmierung.