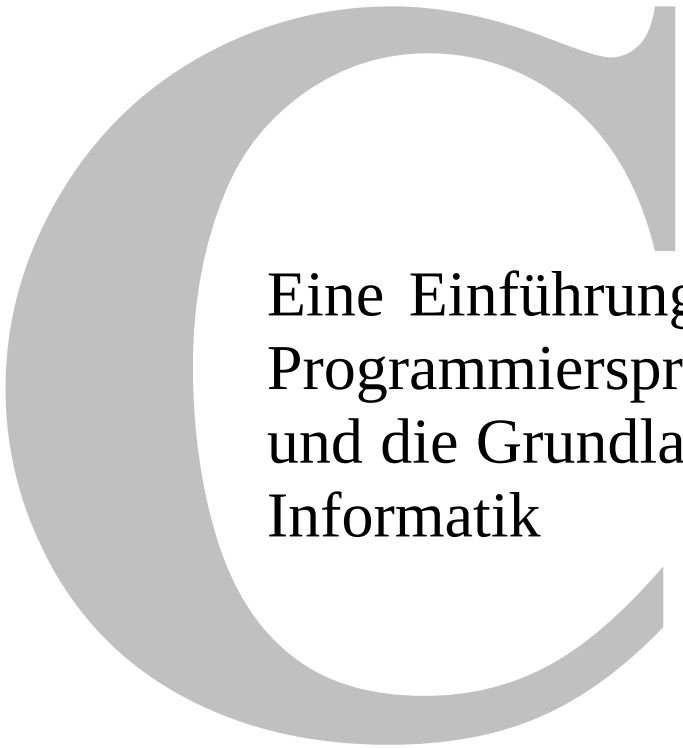
Informatik



Eine Einführung in die Programmiersprache **C** und die Grundlagen der Informatik

Dieses Skript dient als Grundlage für die Informatikvorlesung im ersten und zweiten Semester, es kann und will keine Bücher ersetzen, es ist aber durchaus auch zum Selbststudium einsetzbar.

Ivo Oesch

April 2001
Februar 2003
Oktober 2003
September 2005

Inhaltsverzeichnis

1 Einführung

1.1 Vorwort

In diesem Skript und während der Vorlesung können nur die theoretischen Grundlagen der Programmiersprache C und der strukturierten Programmentwicklung vermittelt werden, das Programmieren selbst kann nur durch selbständiges Anwenden und Üben wirklich erlernt werden. Deshalb werden viele Übungsaufgaben zum Stoff abgegeben und es ist Zeit für praktisches Arbeiten vorgesehen. Nutzen sie diese Zeit, selbständiges Üben wird für ein erfolgreiches Abschliessen in diesem Fach vorausgesetzt!

Für viele Übungen sind auf dem Netzwerk oder beim Dozenten Musterlösungen erhältlich. Denken Sie daran, im Gegensatz zu anderen Fächern gibt es in der Informatik nicht nur eine richtige Lösung für ein Problem, sondern beliebig viele. (Natürlich gibt es aber noch mehr falsche Lösungen...). Die Musterlösungen sind deshalb nur als eine von vielen möglichen Lösungen zu betrachten und sollen Hinweise auf mögliche Lösungsstrategien geben. Vergleichen Sie Ihre Lösungen auch mit Ihren Kollegen, sie werden dabei viele unterschiedliche Lösungsansätze kennenlernen.

1.2 Hinweise

Um 1999 ist ein neuer *ANSI-Standard* für die Programmiersprache C verabschiedet worden, bekannt unter dem Namen *ANSI C99*. Dabei wurde die Sprache um einige Elemente erweitert.

Diese Erweiterungen werden in diesem Dokument ebenfalls aufgeführt, aber mit dem Vorsatz *[C99]* kenntlich gemacht. Diese Erweiterungen sind noch nicht auf allen *Compilern* implementiert, und speziell bei älteren oder Mikrocontroller-Compilern ist noch damit zu rechnen, dass diese Features nicht unterstützt werden. Wenn auf höchste Kompatibilität Wert gelegt wird, sollte auf sie verzichtet werden. (Die neuen *Schlüsselwörter* hingegen sollten nicht mehr als Bezeichner verwendet werden)

1.3 Zu diesem Skript

Dieses Skript wird laufend überarbeitet und den Bedürfnissen des Unterrichts angepasst. Kritiken, Anregungen und Verbesserungsvorschläge zu diesem Skript werden vom Autor jederzeit gerne entgegengenommen. (An ivo.oesch@bfh.ch oder ivo@oesch.org)

Das Kapitel über die Bibliotheksfunktionen ist ein Auszug der WEB-Seite von Axel Stutz und Peter Klingebiel, FH Fulda, DVZ, unter *'http://www.fh-fulda.de/~klingebiel/c-stdlib/index.html'* erreichbar.

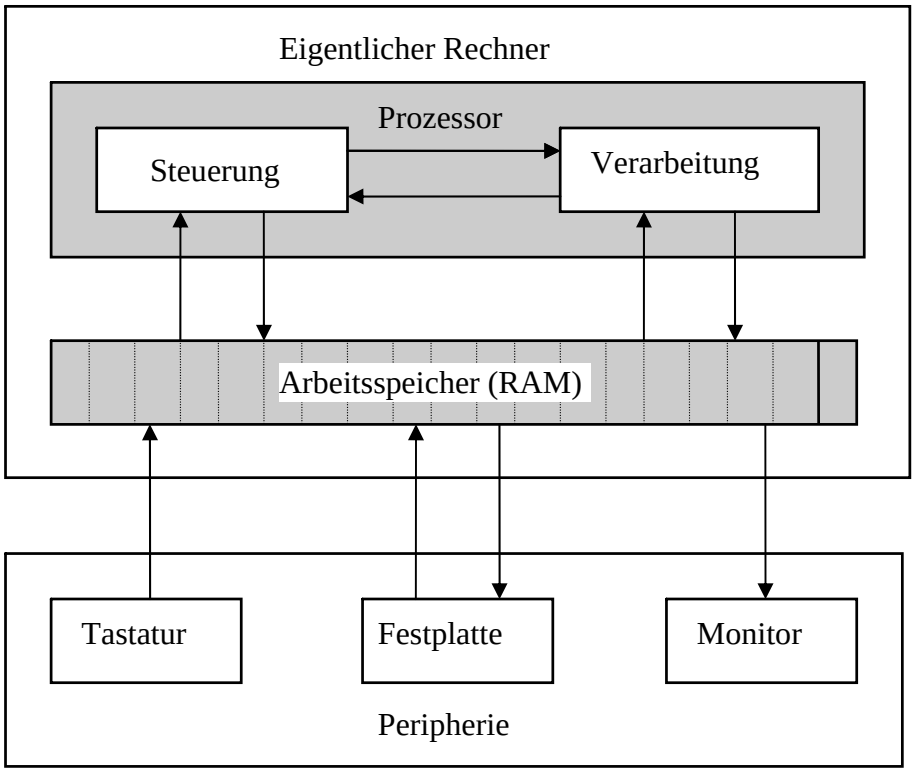
Versionen

2.0 02.2003 Erste veröffentlichte Version.
2.1 10.2003 Leicht überarbeitet und Fehler korrigiert, mit Kapitel zu Extreme Programming ergänzt.
2.2 09.2005 Leicht überarbeitet und Fehler korrigiert, mit Kapitel zu systematischem Testen ergänzt.

Copyright

1.4 Der Computer

Was ist ein Computer? Diese Frage werden wir hier nicht ausführlich beantworten, für diesen Kurs genügt es zu wissen, was im unteren Diagramm dargestellt wird.



Ein Computer besteht grundsätzlich aus einem *Prozessor*, *Arbeitsspeicher (RAM)* und *Peripherie*. Im Arbeitsspeicher werden auszuführende Programme und gerade benötigte Daten abgelegt. Der Inhalt des Arbeitsspeichers hat die unangenehme Eigenschaft *flüchtig* zu sein, d.h. nach einem Abschalten des Rechners geht alles verloren, was nicht vorher auf ein anderes Medium kopiert wurde.

Ein Programm besteht aus einer Folge von einfachen Prozessor *Anweisungen* und *Datenstrukturen*, die im Arbeitsspeicher abgelegt sind. Der Prozessor holt sich der Reihe nach die Anweisungen aus dem Arbeitsspeicher, verarbeitet sie und legt die Ergebnisse wieder im Arbeitsspeicher ab. Der Arbeitsspeicher ist so organisiert, dass die einzelnen *Speicherzellen* von 0 an durchnumeriert werden. Der Zugriff auf die Speicherzellen und die darin abgelegten Daten und Befehle erfolgt über diese Nummer (*Adresse*). Die Darstellung des Arbeitsspeichers soll darauf hinweisen, dass die Daten *sequentiell* (linear, d.h. ein Datum nach dem anderen) abgelegt sind.

Weiterhin gibt es noch die Peripherie (Ein-/Ausgabegeräte):

die *Tastatur*: um Daten einzugeben

den *Monitor*: um zu sehen, was der Rechner zu sagen hat

Speichermedien (Harddisk, Floppy, ZIP): um Daten auszutauschen oder längerfristig aufzubewahren

Weitere Peripherie wie Drucker, Netzwerkanschluss, CD-Laufwerke...

1.5 Hochsprachen

Da es sehr aufwendig und fehleranfällig ist, Programme direkt in Prozessoranweisungen (*Assembler*) zu Schreiben, verwendet man im allgemeinen eine *Hochsprache* wie zum Beispiel C zum Programmieren. Diese Hochsprachen werden aber vom Prozessor nicht direkt verstanden und müssen deshalb mit weiteren Programmen wie *Compiler* oder *Interpreter* in Prozessorbefehle übersetzt werden. Aus einem Befehl der Hochsprache wird dabei eine Sequenz von mehreren Prozessorbe-

fehlen (*Maschinencode*) erzeugt, und aus Variablennamen werden wieder Speicheradressen. Erst diese Prozessorbefehle werden, sobald in den Arbeitsspeicher geladen, vom Prozessor verstanden und ausgeführt. Interpreter wandeln die Befehle zur Laufzeit des Programmes jedes mal wieder einzeln um, Compiler wandeln das ganze Programm auf einmal in eine ausführbare Datei um, welche anschliessend selbständig lauffähig ist. Compilierte Programme laufen üblicherweise deutlich schneller (Faktoren) als interpretierte. Bei einigen Sprachen wird die Hochsprache in einen *Zwischencode* übersetzt, welcher dann seinerseits interpretiert wird, die dabei erreichte Laufgeschwindigkeit ist besser als bei einem reinem Interpreter.

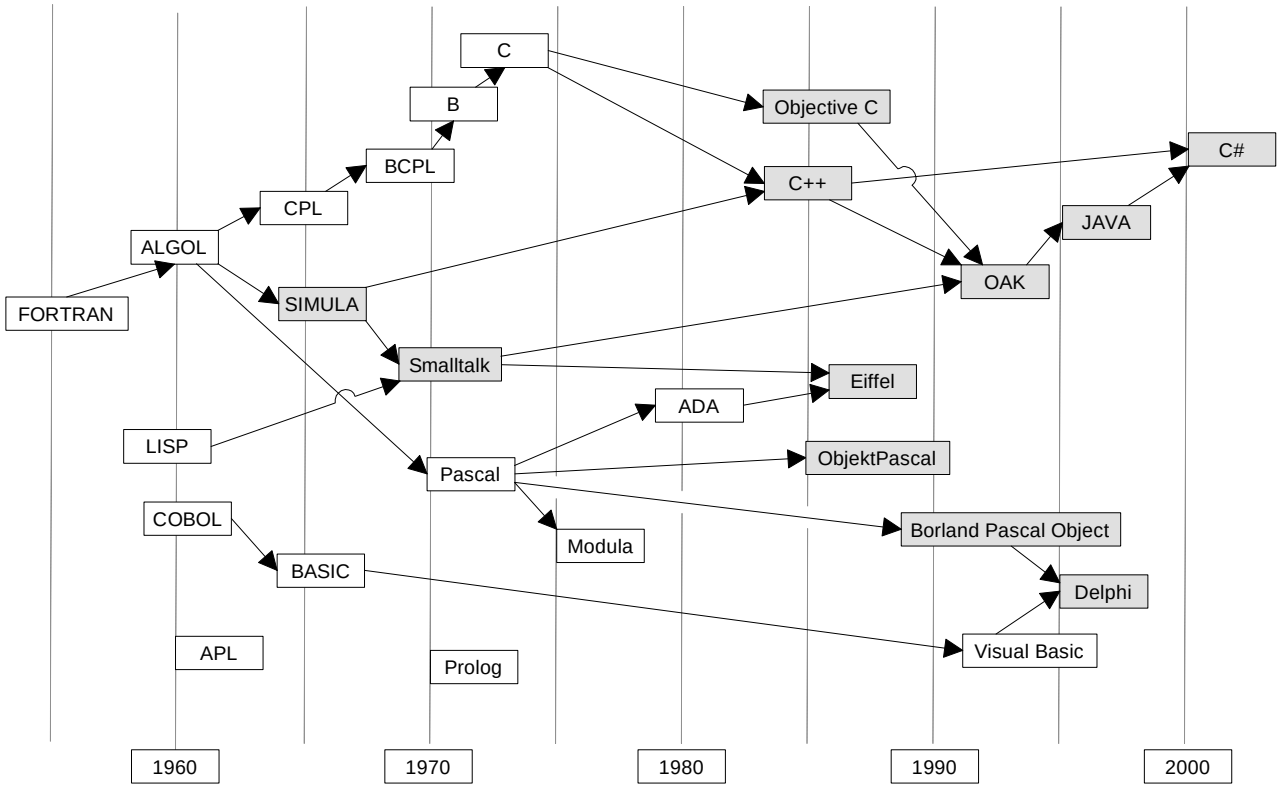
Nachfolgend einige typische Vertreter von compilierten und interpretierten Sprachen:

	BASIC	*Java*	*Pascal*	*C*	*C++*	*Smalltalk*	*Lisp*
Compiler	(x)		X	X	X		X
Zwischencode		X	(x)				
Interpreter	X					X	X

Für einige ursprünglich interpretierte Sprachen existieren mittlerweile auch Compiler.

1.6 Die Entwicklung der Programmiersprache C und ihr Umfeld

Im folgenden Bild ist die Entwicklung der gängigsten Programmiersprachen dargestellt, grau hinterlegt sind *Objektorientierte Sprachen*. (OO-Sprachen). Die Vorderkanten der Felder entsprechen dem jeweiligen Erscheinungsjahr der Sprache. Die meisten der aufgeführten Sprachen werden noch heute eingesetzt. Die Programmiersprache C wurde um 1970 von Dennis Ritchie entwickelt, 1978 wurde das berühmte Buch 'The C Programming Language' von Dennis Ritchie und Brian Kernighan geschrieben, welches lange Zeit den Sprachstandard definierte, und noch heute als Referenz gilt (Auch wenn die neueren Sprachelemente von C99 darin noch nicht berücksichtigt sind).



2 Grundlagen

2.1 Algorithmen und Struktogramme

Die allgemeine Vorgehensweise beim Programmieren lautet:

Problemanalyse

Entwurf eines Algorithmus (einer Strategie) zur Lösung des Problems

Erstellung des Programms (inklusive alle Testphasen)

Das mag einem etwas banal vorkommen, aber beobachten Sie sich selbst mal beim Arbeiten: Wie viel Zeit geht allein dadurch verloren, dass Sie sich nicht richtig klar gemacht haben, wie das Problem eigentlich aussieht?

Ein *Algorithmus* ist ein genau festgelegtes Ablaufschema für oft wiederkehrende Vorgänge, das nach einer endlichen Anzahl von Arbeitsschritten zu einem eindeutigen Ergebnis führt. Jeder Algorithmus zeichnet sich dadurch aus, dass er absolut *reproduzierbare* Ergebnisse liefert. Das bedeutet, unter immer gleichen Voraussetzungen bzw. Anfangsbedingungen muss ein bestimmter Algorithmus stets dasselbe Endergebnis liefern.

Jeder Algorithmus kann in eine Abfolge von drei Typen von Handlungsvorschriften zerlegt werden: *Sequenz*, *Alternation* und *Iteration*. Für die hier zu behandelnden Algorithmen gibt es ein Verfahren, das die Arbeit im Umgang mit Algorithmen sehr erleichtern kann: *Struktogramme*. Ein Struktogramm ist eine grafische Veranschaulichung eines Algorithmus. Für die drei Typen von Handlungsvorschriften gibt es jeweils eine bestimmte graphische Darstellung:

2.1.1 Sequenz - die Aneinanderreihung, Aufeinanderfolge von Anweisungen

Anweisung 1
Anweisung 2
...
Anweisung n

Eine Sequenz ist schlicht eine Abfolge einzelner Anweisungen. Eine Sequenz gilt selbst wieder als Anweisung.

2.1.2 Alternation - Auswahl unter 2 möglichen Teilalgorithmen

Bedingung: Wahr	Bedingung: Falsch
Anweisung1	Anweisung2

Wenn die Bedingung zutrifft, wird Anweisung 1 ausgeführt, sonst die Anweisung 2. Die Anweisungen können selbstverständlich auch Sequenzen sein. Die Alternation selbst gilt auch als eine Anweisung

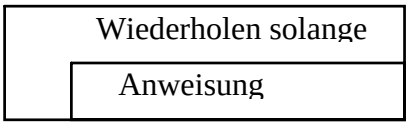
2.1.3 Fallauswahl - Auswahl unter mehreren Teilalgorithmen

Falls a: = 1	= 7	= 13	Sonst
Anw. 1	Anw. 2	Anw. 3	Anw. n

Je nach Fall (Hier der Wert der Variable a) wird eine der Anweisungen ausgeführt. Wenn keiner der Fälle zutrifft, wird die Anweisung von 'Sonst' ausgeführt. Die Auswahl selbst gilt auch als eine Anweisung

2.1.4 Iteration - Wiederholung eines Teilalgorithmus

Wie oft dieser Teilalgorithmus wiederholt wird, kann, muss aber nicht, von vornherein feststehen. Es gibt eine Unterscheidung: Wiederholung mit vorausgehender Bedingungsprüfung (s. Bild) und Wiederholung mit nachfolgender Bedingungsprüfung.



Die Anweisung wird solange wiederholt, wie die Bedingung zutrifft. Wenn die Bedingung von Anfang an falsch ist, wird die Anweisung überhaupt nicht ausgeführt. Die Prüfung findet jeweils vor dem Durchführen der Anweisung statt. Die Wiederholung selbst gilt auch als eine Anweisung

Wie würden Sie das Diagramm für die Iteration mit nachfolgender Bedingungsprüfung zeichnen?

Die Anweisung wird solange wiederholt, wie die Bedingung zutrifft. Die Anweisung wird dabei in jedem Fall mindestens einmal ausgeführt.

Die Prüfung findet jeweils nach dem Durchführen der Anweisung statt. Die Wiederholung selbst gilt auch als eine Anweisung

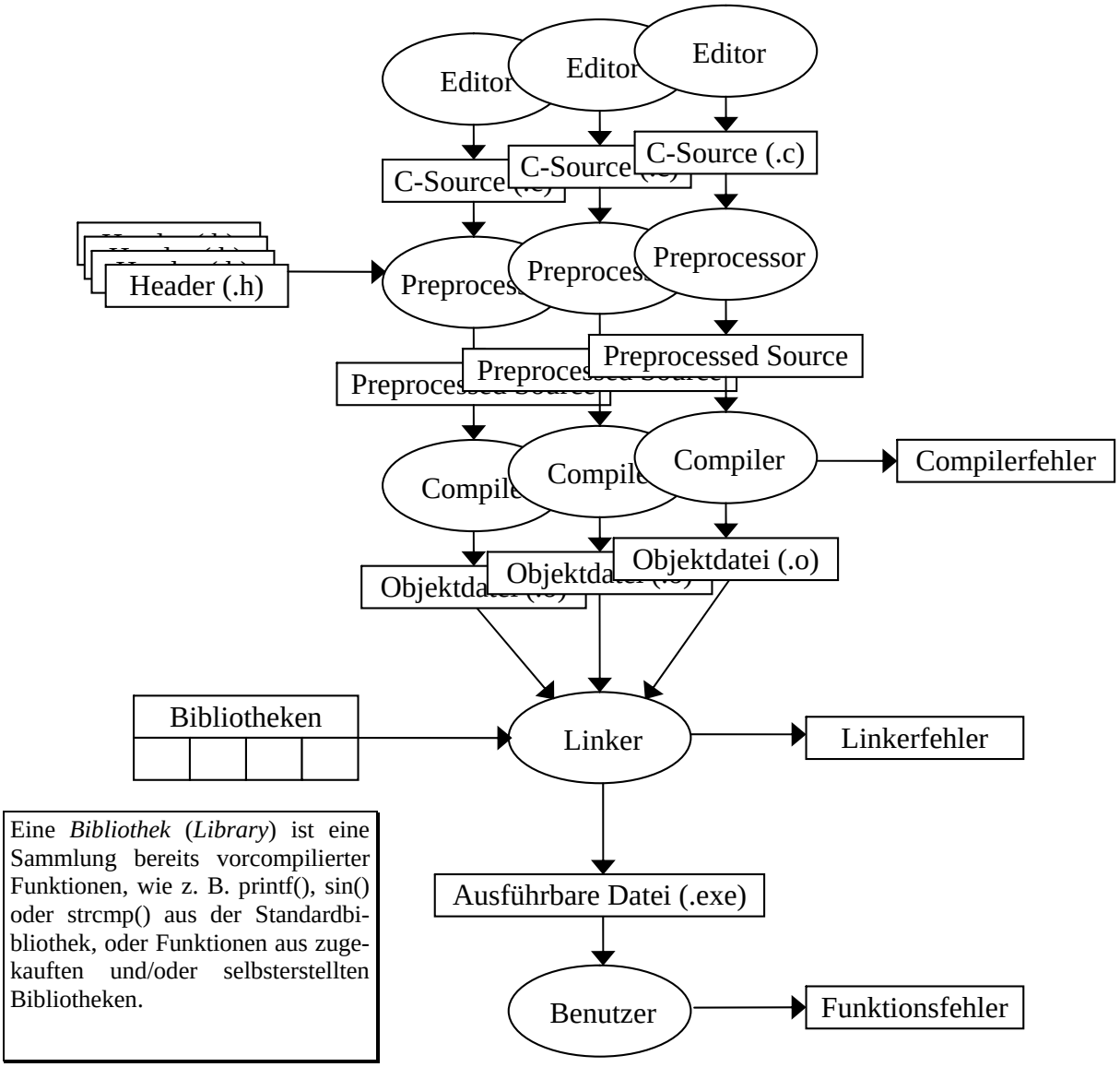
Struktogramme bieten die Möglichkeit, ein Problem systematisch anzugehen, indem man es zuerst nur grob strukturiert, und anschliessend durch allmähliche Verfeinerung bis zum fertig ausgearbeiteten Programmcode gelangt.

2.1.5 Aufgabe 2.1:

Versuchen Sie mit einem Struktogramm den Ablauf eines Telefongesprächs zu beschreiben. Denken Sie an die verschiedenen Möglichkeiten: Nummer vergessen, Besetzt, Falsche Verbindung, Apparat defekt, Druckfehler im Telefonbuch.

2.2 Aufbau eines C-Projektes

Ein Umfangreicheres Programm besteht im Normalfall aus mehreren Quelltextdateien. Diese werden einzeln compiliert und erst danach vom *Linker* zum vollständigen Programm zusammengesetzt.



Eine *Bibliothek* (*Library*) ist eine Sammlung bereits vorcompilierter Funktionen, wie z. B. printf(), sin() oder strcmp() aus der Standardbibliothek, oder Funktionen aus zugekauften und/oder selbsterstellten Bibliotheken.

Das obige Diagramm zeigt die wichtigsten Phasen, welche bei der Programmerstellung durchlaufen werden. Mit dem Editor wird das C-Programm geschrieben und verändert. Der Quellcode des C-Programms wird vor der Compilierung durch den *Präprozessor* bearbeitet. Dazu gehört:

das Entfernen von Kommentaren

das Ersetzen von Makros (#define)

das Einfügen gewisser Dateien (#include)

Der Compiler erzeugt aus der vom Präprozessor erzeugten Datei eine Objektcode-Datei, die aus Maschinensprache und *Linkeranweisungen* besteht . Der *Linker* wiederum verknüpft die vom Compiler erzeugte(n) Datei(en) mit den Funktionen aus Standardbibliotheken oder anderen, vom Programmierer generierten Objekt-Code-Dateien. Wenn an einer Stelle dieses Prozesses ein Fehler auftritt, muss die fehlerverursachende Sourcedatei korrigiert werden, und der Prozess beginnt von vorne. (Compiliert werden müssen aber nur die veränderten Dateien). Die Verwaltung von mehreren Dateien in einem Projekt, und die richtige Steuerung von Linker und Compiler wird bei allen modernen *Compilersystemen* von einer *IDE* (Integrated Developpment Environment) übernommen, der Programmierer muss nur definieren, welche Dateien zum Projekt gehören, den Rest erledigt die IDE.

Für manche Programmierumgebungen gibt es sogenannte *Debugger*-Programme, mit denen man sich auf die Fehlersuche begeben kann. Man kann damit z.B. das Programm schrittweise durchlaufen und sich die Inhalte von Variablen und Speicher ansehen und auch modifizieren. In den Übungen zu diesem Kurs werden Sie eine solche Umgebung kennen lernen.

Die Programmierfehler können auch mit der **`printf()-`**Anweisung lokalisiert werden, indem man innerhalb des Programms ständig den Inhalt von interessanten Variablen, und den aktuellen Ort im Code ausgibt, so kann der Programmablauf auch verfolgt, resp. rekonstruiert werden. Die Funktion **`printf()`** wird später noch vorgestellt. Den Kreislauf - Editieren - Kompilieren - Linken - Ausführen - Fehlersuche müssen Sie solange durchlaufen, bis das Programm fehlerfrei die gestellte Aufgabe erfüllt. Mit einer guten Analyse und einem guten Entwurf kann der Kreislauf beträchtlich verkürzt werden, da viele Denkfehler so bereits früh eliminiert werden.

2.3 Übersicht über die Sprache C

Ein C-Programm besteht aus einzelnen "Bausteinen", den *Funktionen*, die sich gegenseitig aufrufen. Jede Funktion löst eine bestimmte Aufgabe. Eine Funktion ist entweder selbsterstellt oder eine fertige Routine aus der Standardbibliothek oder einer fremden Bibliothek. Die Funktion **`main()`** hat eine besondere Rolle: Sie bildet das steuernde Hauptprogramm. Jede andere Funktion ist direkt oder indirekt ein Unterprogramm von **`main()`**. Die Form von **`main()`** (Argumente, Rückgabewert) ist vorgeschrieben.

Die Anweisungen, aus denen die Funktion(en) besteht(en), bilden zusammen mit den notwendigen Deklarationen und Präprozessor-Direktiven den *Quelltext* eines C-Programms. Dieser wird bei kleineren Programmen in eine *Quelldatei*, bei grösseren in mehrere *Quelldateien* geschrieben. *Quelldateien* haben die Endung .c, *Headerdateien* (Schnittstellenbeschreibung der Module) die Endung .h.

2.3.1 Beispiele für einfache C-Programme

Hier das wohl berühmteste Programm der Welt (*Hello World*) in einer Variante für C:

```
/*****************************************************************************/
/*  Module     : HelloWorld                                      Version 1.0  */
/*****************************************************************************/
/*                                                                           */
/*  Function   : Einfaches Einfuehrungsbeispiel in die Programmiersprache C  */
/*               Gibt den Text 'Hello World' auf den Bildschirm aus          */
/*                                                                           */
/*  Procedures : main                                                        */
/*                                                                           */
/*  Author     : I. Oesch (IO)                                               */
/*                                                                           */
/*  History    : 13.03.2000  IO Created                                      */
/*                                                                           */
/*  File       : Hello.c                                                     */
/*                                                                           */
/*****************************************************************************/

#include<stdio.h>     /* Anweisung an den Praeprozessor die Datei mit den
                         Infos ueber Funktionen zur Ein-/Ausgabe einzubinden */

int main(int argc, char *argv[])   /* Die Funktion main (das Hauptprogramm)  */
                                   /* beginnt hier.                          */
{
      /* main ruft die Bibliotheksfunktion printf auf, um die
                                              Zeichenfolge zu drucken */
   printf("Hello, world\n");
   return 0;                        /* Fehlerfreie Ausfuehrung anzeigen */
}    /*Hier endet das Programm (die Funktion main() ) */
```

Und hier noch ein etwas anspruchsvolleres Programm:

```
/* Programm zur Berechnung einer Rechteckflaeche      */
/* 10.10.2002 by Ivo Oesch, Version 1                 */

#include <stdio.h>                              /* Ein-/Ausgabe Funktionen */

int main(int argc, char* argv[])
{
   int Laenge = 0;                                 /* Variablen definieren */
   int Breite = 0;
   int Flaeche;

   /* Dimensionen vom Benutzer eingeben lasse */
   printf("Bitte geben Sie die Laenge ein:");           /* Text Ausgeben */
   scanf("%d", &Laenge);                                /* Wert einlesen */
   printf("Bitte geben Sie die Breite ein:");           /* Text Ausgeben */
   scanf("%d", &Breite);                                /* Wert einlesen */

   /* Flaeche aus Lange und Breite Berechnen... */
   Flaeche = Laenge * Breite;

   /* ...und ausgeben */
   printf("Die Fläche ist %d\n", Flaeche);
   return 0;
}
```

2.3.2 Der Aufbau eines C-Programmes

Ein C-Programm besteht aus Zeichen, aus diesen wird der Quelltext gebildet. Der Quelltext besteht aus Kommentaren, Präprozessorbefehlen und Anweisungen. Anweisungen setzen sich aus Schlüsselwörtern, Operatoren, Literalen (Konstanten) und Bezeichnern zusammen. Die einzelnen Elemente werden im Folgenden erläutert.

2.3.2.1 Kommentare

Kommentare sollen das Verstehen und Nachvollziehen eines Programmes erleichtern, sie sind für den/die Programmierer gedacht, und werden vom Compiler ignoriert. Unkommentierte Programme sind schlecht wartbar, und nach kurzer Zeit selbst vom Autor meist nicht mehr einfach nachvollziehbar.

In C werden Kommentare zwischen /* und */ eingeschlossen, in C99 wurde neu auch der *Zeilenkommentar* // (Ein Kommentar, der vom // bis zum Ende der Zeile geht) eingeführt.

```
    /* Das ist ein Kommentar */

   /* Blockkommentare koennen sich auch ueber mehrere
       Zeilen erstrecken, das ist kein Problem */

   /* Aber verschachtelte /* Kommentare */ sind nicht erlaubt, auch wenn
                                        manche Compiler Sie akzeptieren*/

[C99]    // Dieser Kommentar geht bis zum Ende der Zeile

   a = 3.14 *  /* Kommentare koennen irgendwo stehen */ R * R;
[C99]   a = 3.14 * r * r; // Kreisflaeche berechnen
```

2.3.2.2 Präprozessorbefehle

Die Zeilen mit dem Doppelkreuz # bilden die *Präprozessor-Direktiven*. Das sind die Befehle, die der Präprozessor vor dem Kompilieren ausführen soll. In unserem Beispiel:

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

bewirkt, dass der Compiler Information über die Standard-Ein/Ausgabe-Bibliothek einfügt. Die Standard-Ein/Ausgabe-Bibliothek wird im Kapitel Bibliotheksfunktionen beschrieben.

2.3.2.3 main

```
int main(int argc, char* argv[])
{
   /* Code von main */
   return 0;
}
```

Jedes C-Programm muss mindestens die Funktion **`main()`** enthalten. Diese ist das eigentliche *Hauptprogramm* und wird automatisch vom System aufgerufen wenn das Programm gestartet wird. In unserem Beispiel ist **`main()`** die einzige selbstgeschriebene Funktion. Die Form von **`main()`** (Rückgabewert und Argumente) ist vorgeschrieben. In den Argumenten **`argc`** und **`argv`** werden dem Programm Informationen vom Betriebssystem übergeben (die Befehlszeilenargumente). Diese Argumente können vom Programm ignoriert werden. **`main()`** muss dem Betriebssystem am Ende einen Fehlercode zurückliefern, normalerweise 0 bei fehlerfreiem Programmablauf. Wenn das Programm weitere Funktionen enthält, werden diese direkt oder indirekt von **`main()`** aufgerufen.

2.3.2.4 Anweisungen

Die geschweiften Klammern `{}` umgeben die *Anweisungen*, aus denen die Funktion besteht. Die Funktion **`main()`** von HelloWorld enthält nur eine Anweisung:

```
printf("Hello, world");
```

Die Funktion **`printf`** wird mit dem Argument `"Hello, world\n"` aufgerufen. **`printf()`** ist eine Bibliotheks-Funktion, die Ausgaben erzeugt; in diesem Fall die Zeichenkette (engl. *string*) zwischen den doppelten Anführungszeichen ('Gänsefüsschen').

Die Zeile

```
   Flaeche = Laenge * Breite;
```

ist ebenfalls eine Anweisung, es soll nämlich das Produkt der Variablen Laenge und Breite gebildet und in der Variablen Flaeche abgespeichert werden.

2.3.2.5 Zeichensatz

Jede Sprache besteht aus Wörtern, die selbst wieder aus Zeichen aufgebaut sind. Die Wörter der Sprache C können nur aus folgenden Zeichen bestehen:

den 26 Grossbuchstaben:

A B C D E F G H I J K L M N 0 P Q R S T U V W X Y Z

den 26 Kleinbuchstaben:

a b c d e f g h i j k 1 m n o p q r s t u v w x y z

den 10 Ziffern:

1 2 3 4 5 6 7 8 9 0

den 29 Sonderzeichen:

! „ # : . { } ^ % & ‘ ; [ ] ~ < = > ? | \ _ ( ) * + , - /

den Zwischenraumzeichen:

Leerzeichen, horizontaler und vertikaler Tabulator, neue Zeile und neue Seite

Der Ausführungs-Zeichensatz enthält darüber hinaus noch folgende Zeichen:

Das Null-Zeichen '`\0`', um das Ende von Strings zu markieren.

Die Steuerzeichen, die durch einfache *Escape-Sequenzen* repräsentiert werden, um Ausgabegeräte wie Bildschirme oder Drucker zu steuern (Siehe Kap. 4.1).

2.3.2.6 Schlüsselwörter von C/*C++*

Wie in jeder Sprache gibt es auch in C vordefinierte Wörter, die bestimmte Bedeutungen haben. Diese *Schlüsselwörter* sind für die jeweilige Programmiersprache reserviert und dürfen vom Programmierer nicht als *Bezeichner* (Namen) verwendet werden. Auf die Verwendung von C++ Schlüsselworten als Bezeichner in C sollte ebenfalls verzichtet werden, obwohl dies möglich wäre (Sonst kann das Programm nicht mit einem C++-Compiler übersetzt werden).

In der nachfolgenden Tabelle sind alle Schlüsselwörter von C und C++ aufgelistet, *ANSI-C 89* Schlüsselworte sind **Fett** gedruckt, *ANSI-C 99* Schlüsselworte sind **Fett** mit dem Vorsatz *[C99]* gedruckt, zusätzliche C++ Schlüsselworte sind *schwach kursiv* gedruckt

*[c99]*__Bool	**default**	*mutable*	**struct**
*[c99]*__Complex	*delete*	*namespace*	**switch**
*[c99]*__Imaginary	**do**	*new*	*template*
*[c99]*__Pragma	**double**	*not*	*this*
and	*dynamic_cast*	*not_eq*	*throw*
and_eq	**else**	*operator*	*true*
asm	**enum**	*or*	*try*
auto	*explicit*	*or_eq*	**typedef**
bitand	*export*	*private*	*typeid*
bitor	**extern**	*protected*	*typename*
bool	*false*	*public*	**union**
break	**float**	**register**	**unsigned**
case	**for**	*reinterpret_cast*	*using*
catch	*friend*	*[c99]***restrict**	*virtual*
char	**goto**	**return**	**void**
class	**if**	**short**	**volatile**
compl	*inline*	**signed**	*wchar_t*
const	*[c99]***inline**	**sizeof**	**while**
const_cast	**int**	**static**	*xor*
continue	**long**	*static_cast*	*xor_eq*

2.3.2.7 Bezeichner

Bezeichner (engl. *identifier*) sind Namen von Variablen, Funktionen, Makros, Datentypen usw. Für die Bildung von Bezeichnern gelten die folgenden Regeln:

- Ein Bezeichner besteht aus einer Folge von Buchstaben (`A` bis `Z`, `a` bis `z`), Ziffern (`0` bis `9`) und Unterstrich (_).
- Das erste Zeichen darf keine Ziffer sein.
- Gross- und Kleinbuchstaben werden unterschieden.
- Ein Bezeichner kann beliebig lang sein. Signifikant sind in der Regel nur die ersten 31 Zeichen. (in C99 die ersten 63)
- Der *Linker* (Globale Bezeichner, **`extern`**) muss nur 6 Zeichen beachten und darf Gross/Klein ignorieren, (in C99 31 Zeichen und muss Gross/Klein berücksichtigen).

Bezeichner werden vom Programmierer vergeben, oder sind von Bibliotheken (Z. B. der Standardbibliothek) vorgegeben. Schlüsselwörter sind reserviert und dürfen nicht als Bezeichner verwendet werden.

Achtung, Bezeichner die mit einem *Unterstrich* beginnen sind für den Compilerhersteller reserviert, und sollten nicht verwendet werden.

2.3.2.8 Namensklassen und Geltungsbereiche

Jeder Bezeichner gehört zu genau einer der vier *Namenklassen*, nämlich:

- Namen von Marken (engl. *labels*)
- Namen von Strukturen, Unions und Aufzählungen (*tags*). Das sind Namen, die einem der Schlüsselwörter **`struc`**, **`union`** oder **`enum`** folgen.
- Namen von Struktur- oder Union-*Komponenten* wobei jede Struktur oder Union eine eigene Namensklasse bildet.
- Alle anderen Bezeichner.

Innerhalb einer Klasse darf ein Bezeichner nur einmal definiert werden, der gleiche Bezeichner darf aber in verschiedenen Klassen definiert werden (Auch wenn das, abgesehen bei Struktur und Union-Komponenten, nicht unbedingt zu empfehlen ist.)

Im folgenden Beispiel sind Laenge, Breite und Flaeche Bezeichner.

```
int Laenge = 0;
int Breite = 0;
int Flaeche;
```

Aufgabe 2.2:

Überlegen Sie sich selbst einige Beispiele für Bezeichner. Achten Sie auf aussagekräftige Namen.

2.4 Aufgabe 2.3:

Tippen Sie das "Hello world" Programm ab und versuchen Sie, es zu übersetzen und zum Laufen zu bringen. Wenn Ihr Programm läuft, haben Sie den ersten grossen Schritt in C gemacht!

3 Datentypen

3.1 Basisdatentypen

C ist eine stark *typisierte* Programmiersprache, das heisst dass stark zwischen verschiedenen Arten von Daten unterschieden wird. Man unterscheidet grundsätzlich zwischen *einfachen Datentypen* und *zusammengesetzten Datentypen*. In diesem Kapitel werden die einfachen Datentypen vorgestellt. Alle einfachen Datentypen in C sind für skalare Werte definiert.

Bei den einfachen Datentypen wird zunächst zwischen *Integer* (*Ganze Zahlen*) und *Fliesskomma* Datentypen unterschieden. *Zeichenketten* (*Strings*) sind ein Spezialfall von Feldern (Arrays), welche in einem späteren Kapitel behandelt werden. Es gibt zwar Stringliterale (Konstanten), aber keinen eigentlichen String-Datentypen.

Ganzzahlige Datentypen sind **`int`** und **`char`**, Fliesskommatypen sind **`float`** und **`double`**. Der Datentyp **`char`** ist eigentlich zur Aufnahme von einzelnen Buchstaben (ASCII-Codes) vorgesehen, kann aber auch für kleine Zahlen (8-Bit-Werte) benutzt werden. Die ganzzahligen Datentypen können zusätzlich mit den Qualifizierern **`signed`** und **`unsigned`** explizit als *vorzeichenbehaftet* (positiv und negativ), oder *vorzeichenlos* (nur positiv) festgelegt werden. Die Grösse des Datentyps **`int`** kann zudem mit den qualifizierern **`short`** und **`long`** modifiziert werden. Alle **`int`** Typen sind per Default vorzeichenbehaftet. Beim Datentyp **`char`** ist nicht festgelegt (!), ob er mit oder ohne Vorzeichen implementiert ist.

Datentyp	Bits *2)	Wertebereich	Literal (Konstante)
*[C99]*_Bool, bool	>=8	0 und 1	true, false, 0, 1
char	8	-128 ... +127 oder 0 ... 255 *1)	'a' '\n' 'B'
unsigned char	8	0 ... 255	'a' '\n' 'B'
signed char	8	-128 ... 127	'a' '\n' 'B'
short int	16	-32768 ... 32767	---
unsigned short int	16	0 ... 65335	---
signed short int	16	-32768 ... 32767	---
int	16-32	$-2^{31}...2^{31}-1$	123 0x123 077 -44
unsigned int	16-32	$0 ... 2^{32}-1$	123u 0x123U 077u
signed int	16-32	$-2^{31}...2^{31}-1$	123 0x123 077
long int	32	$-2^{31}...2^{31}-1$	123L 0x123L 077 -44L
long unsigned int	32	$0 ... 2^{32}-1$	123uL 0x123uL 077UL
long signed int	32	$-2^{31}...2^{31}-1$	123L 0x123L 077L
[C99] long long int	>= 64	$-2^{63}...2^{63}-1$	123LL 0x123LL 077LL
[C99] unsigned long long int	>= 64	$0 ... 2^{64}-1$	12uLL 0x123ULL 07ULL
[C99] signed long long int	>= 64	$-2^{63}...2^{63}-1$	123LL 0x123LL 077LL
float	32	$-3.4*10^{38}...3.4*10^{38}$	1.23f 3.14f 1e-10f 0.0f
double	64	$-1.7*10^{308}..1.7*10^{308}$	1.23 3.14 1e-10 0.0
long double	64...80	$+/-1.18*10^{4932}$	1.23L 3.14L 1e-10L 0.0L
char * (String)	n*8	-	"Hallo" "Welt \n"

2^{31} = 2'147'483'648, **2^{32}** = 4'294'967'296, **2^{63}** = $9.22*10^{18}$, **2^{64}** = $18.4*10^{18}$

*1) Der C-Standard lässt es dem Compilerhersteller frei, ob **char signed** oder **unsigned** implementiert wird (Im Zweifelsfall explizit angeben oder Nachschlagen) !!!

*2) Der *Wertebereich* der Datentypen ist vom Standard nicht festgelegt, es ist einzig vorgeschrieben das **char** <= **short int** <= **int** <= **long** [*[C99]* <= ***long long*** *]* und **float** <= **double** <= **long double**. Angegeben wurden heute übliche Werte.

Bei **unsigned**, **signed**, **short**, **long** und [*[C99]***long long**] kann das Schlüsselwort **int** weggelassen werden

```
short a;            /* Anstelle von   short int a           */
long b;             /* Anstelle von   long int b            */
unsigned c;         /* Anstelle von   unsigned int c        */
unsigned short d;   /* Anstelle von   unsigned short int d  */
```

3.2 Qualifizierer

Alle Datentypen (Einfache und Zusammengesetzte) können zusätzlich noch mit den *Qualifizierern* **const**, **volatile**, **[C99] restrict** (nur für Pointer) versehen werden, auch dabei ist **int** jeweils redundant.

Mit **const** bezeichnet man Variablen. die nur gelesen werden sollen (Also Konstanten), wobei Schreibzugriffe darauf trotzdem möglich sind, aber das Ergebnis ist undefiniert. (**const** erlaubt dem Compiler, gewisse Optimierungen vorzunehmen)

```
const float Pi = 3.1416f;   /* Konstante definieren                    */
int Flaeche;
Flaeche = r * r * Pi;       /* OK So ists gedacht                      */
Pi = 17;                    /* Nicht verboten, aber Ergebnis undefiniert */
```

Mit **volatile** bezeichnet man Variablen. die 'flüchtig' sind. Das sind Variablen die ändern können ohne dass das dem Compiler ersichtlich ist. Mit **volatile** zwingt man den Compiler, den Wert dieser Variablen bei jeder Benutzung erneut aus dem Speicher zu lesen, und mehrfaches Lesen nicht wegzuoptimieren. Das ist wichtig bei Speicheradressen und Variablen, die Zustände von Hardwarekomponenten anzeigen, oder von der *Hardware* oder *Interruptroutinen* verändert werden.

```
volatile int Tastenzustand;    /* Wird von Interrupt gesetzt */

Tastenzustand = 0;
while (Tastenzustand == 0) {   /* Ohne volatile koennte der Compiler  */
   /* Warten auf Taste */      /* daraus eine Endlosschlaufe erzeugen */
}                              /* da er nicht wissen kann das der     */
                               /* Zustand Tastenzustand waehrend der  */
                               /* Schleife aendern kann               */
```

[C99] Mit **restrict** können Zeiger qualifiziert werden, bei denen der Programmierer garantiert, dass innerhalb ihrer Lebensdauer nie andere Zeiger auf die selben Werte zeigen. Das eröffnet dem Compiler weitere Optimierungsmöglichkeiten. Ein Compiler muss das Schlüsselwort akzeptieren, darf es aber ignorieren.

3.3 Definition von benutzerdefinierten Typen

Mit dem Schlüsselwort **`typedef`** können bestehenden Datentypen neue Namen gegeben werden, sowie eigene *Datentypen* definiert werden. Eine Typendefinition sieht genau gleich wie eine Variablendeklaration aus, nur dass das Schlüsselwort **`typedef`** vorangestellt wird. Ein benutzerdefinierter Typ kann nach der Definition wie ein eingebauter Typ verwendet werden.

```
int a;                      /* deklariert und definiert die Variable a   */
                            /* vom Typ int                               */

typedef int I32;            /* Definiert einen neuen Typ I32, der        */
                            /* einem int entspricht                      */

typedef short int I16;      /* Definiert einen neuen Typ I16, der        */
                            /* einem short int entspricht                */

typedef unsigned short int U16;  /* Definiert einen neuen Typ U16, der  */
                                 /* einem unsigned short int entspricht */

U16 b;                      /* deklariert und definiert die Variable b   */
                            /* vom Typ U16, also unsigned short int      */
```

Mit **`typedef`** können einerseits häufig verwendete, komplizierte Datentypen mit einem einfacheren Namen benannt werden, die Benutzung von **`typedef`** erlaubt aber auch eine einfachere *Portierung* von Programmen. Wenn z. B. wichtig ist, dass bestimmte Variablen immer 32Bit gross sind, definiert man einen Typ I32 (Oder ähnlicher Name), benutzt diesen Typ für all diese Variablen, und beim **`typedef`** setzt man den Typ I32 auf **`int`** bei Sytemen in welchen **`int`** 32 Bit ist, und auf **`long int`** bei Systemen in welchen **`long`** 32 und **`int`** 16 Bit ist.

```
typedef int I32;            /* Bei Systemen mit sizeof(int) == 4  */

typedef long int I32;       /* Bei Systemen mit sizeof(int) != 4, */
                            /* sizeof(long) == 4                  */
```

Typedefs können auch zur Erzeugung von *komplizierten Typen* eingesetzt werden:

Frage: Wie deklariere ich ein Array a von N Zeigern auf Funktionen, welche Zeiger auf Funktionen welche ihrerseits Zeiger auf **`char`** zurückliefern, zurückliefern?

Antwort 1: Wie Bitte???

Antwort 2:

```
char *(*(*a[N])(void))(void);   /* Alles klar ??? */
```

Antwort 3, mit Typedefs schrittweise...

```
typedef char  *pc;          /* Zeiger auf char                                 */
typedef pc     fpc(void);   /* Funktion welche Zeiger auf char zurueckliefert */
typedef fpc   *pfpc;        /* Zeiger auf obiges (Zeiger auf fpc)              */
typedef pfpc   fpfpc(void); /* Funktion welche obiges zurueckliefert           */
typedef fpfpc *pfpfpc;      /* Zeiger auf obiges (Zeiger auf fpfpc)            */
pfpfpc a[N];                /* Array von obigen Elementen (Feld von pfpfpc)    */
```

4 Konstanten

Bei *Konstanten* (*Literalen*) wird der Datentyp durch die Schreibweise festgelegt. Unter C99 wurden neu auch zusammengesetzte Literale für Arrays und Strukturen eingeführt, diese werden in den entsprechenden Kapiteln näher beschrieben.

4.1 Buchstabenkonstanten (char, Zeichen, Character,)

Werden in einfache Hochkommas eingeschlossen. 'A' ist eine *Zeichenkonstante* mit dem Wert 65 (ASCII-Code von A). Für nichtdruckbare Zeichen gibt es spezielle *Escapesequenzen*:

'\n'	Newline, Zeilenvorschub
'\r'	Carriage Return , Wagenrücklauf
'\t'	Tabulator (Horizontal, üblich)
'\f'	Formfeed, Seitenvorschub
'\v'	Tabulator (Vertikal)
'\b'	Backspace (Rückwärts löschen, Rückwärtsschritt)
'\a'	Alarmton (Pieps)
'\''	für das ' (Einfaches Hochkomma)
'\"'	für das " (Doppeltes Hochkomma, 'Gänsefüsschen')
'\?'	für das Fragezeichen, wird selten benötigt, da ? meist direkt eingegeben werden kann
'\\'	für den \ (Backslash)
'\nnn'	für einen beliebigen ASCII-Code in Oktaler Schreibweise (Bsp. '\0' '\12' '\123' '\377') nnn können 1 bis 3 oktale Ziffern sein (Ziffern von 0 bis 7)
'\xnn'	für einen beliebigen ASCII-Code in Hexadezimaler Schreibweise(Bsp. '\x2F' '\x1B') nn sind 2 Hexadezimale Ziffern (0-9, A, B, C, D, E, F, a, b, c, d, e, f), Gross-/Kleinschreibung spielt keine Rolle.

Achtung, auch eine *Buchstabenkonstante* ist nur ein numerischer Wert, nämlich einfach der *ASCII-Code* des Zeichens. Mit Buchstabenkonstanten kann deshalb gerechnet werden wie mit jeder anderen Zahl auch:

```
'A' + 2     /* Ergibt 67 oder 'C'                                */
'x' / 2     /* Macht weniger Sinn, aber ergibt 60 oder '<'   */
'Z' - 'A'   /* Ergibt die Anzahl Buchstaben zwischen A und Z */
```

4.2 Stringkonstanten (char *)

[Eigentlich ein Array, wird bei Arrays näher Beschrieben]

Strings sind *Texte*, also einfach Aneinanderreihungen von Zeichen (Buchstaben), die aber an letzter Stelle eine 0 als Kennzeichen für das Textende aufweisen, dafür muss die Länge des Textes nicht mit abgespeichert werden. *Stringkonstanten* werden in doppelten Hochkommas eingeschlossen. "Hallo" ist eigentlich eine Aneinanderreihung (Array) der Zeichen 'H' 'a' 'l' 'l' 'o' '\0'. Einer Stringkonstante wird vom Compiler automatisch eine abschliessende 0 ('\0') angehängt.

Innerhalb eines Strings können ebenfalls die Character-Escape-Sequenzen verwendet werden:

```
"Er sagte: \"Ach so\" \nund ging weiter"
```

definiert den Text

```
Er sagte: "Ach so"
und ging weiter
```

4.3 Integerkonstanten:

Der Typ der Konstanten wird durch ein angehängtes *Suffix* kenntlich gemacht, es spielt keine Rolle, ob der Suffix mit Gross- oder Kleinbuchstaben geschrieben wird, bei L/l empfiehlt sich aber aus Verwechslungsgründen dringend das grosse 'L':

Ganzzahlige Konstanten können auch in hexadezimaler oder oktaler Schreibweise angegeben werden. Hexadezimale Konstanten werden durch ein führendes 0x gekennzeichnet, oktale mit einer führenden 0. **Achtung** 033 ist somit nicht 33 Dezimal, sondern 33 Oktal, also 27 Dezimal.

Dezimal	**Hex**	**Oktal**	**Typ**
7964	0x1F1C	017434	**int**
7964u 7964U	0x1F1Cu 0x1F1CU	017434u 017434U	**unsigned int**
7964l 7964L	0x1F1Cl 0x1F1CL	017434l 017434L	**long**
7964ul 7964UL	0x1F1Cul 0x1F1CUL	017434ul 017434UL	**unsigned long**
[C99] 7964ll 7964LL	*[C99]* 0x1F1Cll 0x1F1CLL	*[C99]* 017434ll 017434LL	**long long**
[C99] 7964ull 7964ULL	*[C99]* 0x1F1Cull 0x1F1CULL	*[C99]* 017434ull 017434ULL	**unsigned long long**

(Für **short** gibt es keinen Suffix)

4.4 Fliesskommakonstanten:

Fliesskommakonstanten werden durch einen Dezimalpunkt und/oder die Exponentialschreibweise als solche kenntlich gemacht. **Float** und **long double** werden wiederum durch *Suffixes* gekennzeichnet, Fliesskommakonstanten ohne Suffix sind vom Typ **double**:

Konstante	**Typ**
0.0 3.14 1e5	**double**
1.0f	**float**
1e-20L	**long double**

5 Variablen

Um einen Computer sinnvoll nutzen zu können, müssen die anfallenden Werte irgendwo gespeichert werden können. Der Speicher eines Computers ist eigentlich nichts anderes als eine grosse Ansammlung von durchnumerierten Speicherplätzen. Ein Computer mit 128Mbyte Speicher hat dementsprechend 134217728 Speicherplätze, die von 0 bis 134217727 durchnumeriert sind. Es wäre aber sehr umständlich, die Speicherplätze selbst zu verwalten, und jedesmal über ihre Nummer (*Adresse*) anzusprechen. Deshalb werden für die Speicherplätze Namen vergeben, diese benannten Speicherplätze werden als *Variablen* bezeichnet. Der Compiler vergibt automatisch den nächsten freien Platz, wenn eine neue Variable definiert wird, um die Adresse (Position im Speicher) der Variable braucht sich der Programmierer meist nicht zu kümmern (Ausnahme: Pointer).

5.1 Definition von Variablen

Eine Variable wird *definiert* durch die Angabe das Datentyps und ihres Namens:

Datentyp Name;

Beispiele:

```
int Laenge;
double Zins;
long int Sekunden;
```

Es ist auch möglich, mehrere Variablen desselben Typs auf einmal zu definieren, hinter dem Datentyp folgen die einzelnen Namen durch Kommas getrennt:

```
int Laenge, Breite, Hoehe;
```

Anhand des Datentyps weiss der Compiler, wieviel Speicherplatz eine Variable benötigt, und kann auch die passenden arithmetischen Operationen bei Berechnungen auswählen.

Ein Variablenname (Bezeichner) kann aus Buchstaben (Keine Umlaute und Sonderzeichen, nur a-z und A-Z), Ziffern und Unterstrich (_) bestehen, wobei an erster Stelle keine Ziffer stehen darf. Die Länge des Namens darf bis zu 31 ([C99] 63) Buchstaben bestehen, vom Linker werden 6 ([C99] 31) berücksichtigt. Bezeichner dürfen auch länger sein, aber der Compiler betrachtet nur die ersten n Zeichen als *signifikant*, Variablen, die sich in den ersten n Zeichen nicht unterscheiden gelten als gleich.

Der Compiler unterscheidet zwischen Gross- und Kleinschreibung, der Linker muss nach Standard nicht zwischen Gross und Kleinschreibung unterscheiden (Die meisten Linker achten aber darauf, und in C99 muss ein Linker auch die Gross/Kleinschreibung beachten).

5.2 Initialisierung von Variablen

Variablen von einfachen Datentypen können folgendermassen *initialisiert* werden.

```
int Laenge = 15;
double Zins = 5.25;
long int Sekunden = 2562435L;
```

Wenn mehrere Variablen auf einer Zeile definiert und initialisiert werden, müssen sie einzeln initialisiert werden:

```
int x=17, y=17, z=25;
```

Bei der *Initialisierung* wird der Variable gleich bei der Erzeugung ein bestimmter Wert zugewiesen. Nicht initialisierte globale Variablen werden automatisch auf den Wert 0 gesetzt, lokale Variablen hingegen haben zufälligen Inhalt, wenn sie nicht initialisiert werden!!!!

Auf den Unterschied zwischen globalen und lokalen Variablen wird im Kapitel Funktionen noch näher eingegangen.

5.3 Deklaration von Variablen

Bei der *Deklaration* einer Variablen wird dem Compiler nur bekanntgemacht, dass es eine Variable eines bestimmten Typs gibt, es wird aber kein Speicherplatz dafür reserviert. Die Variable muss deshalb woanders (meist in einem anderen Modul) definiert werden. Um eine Variable zu deklarieren wird das Schlüsselwort **`extern`** vorangestellt. Nur globale Variablen können **`extern`** deklariert werden.

```
extern int x;          /* Deklaration, es wird kein Speicher belegt */
extern long double UniverselleKonstante;
```

5.4 Speicherklassen:

Alle Variablen in einem Programm werden verschiedenen *Speicherklassen* zugeordnet. Es gibt die Klasse *Static*, die Klasse *Automatic* und die Klasse *Register*.

Die Zuordnung zu den Speicherklassen erfolgt mit den Schlüsselworten **`register`**, **`auto`**, **`static`** und **`extern`** sowie dem Ort der Definition (Innerhalb oder ausserhalb von Blöcken).

Globale Variablen (Ausserhalb von Blöcken definierte) und **`static`** Variablen gehören zur Klasse Static. Sie werden zu Beginn des Programmlaufes einmal initialisiert und sind bis zum Ende des Programmlaufes ständig verfügbar, sie existieren während des gesamten Programmlaufs.

Nur Variablen innerhalb von Blöcken (Mit {} eingeschlossene Anweisungen) können zu den Klassen Automatic oder Register gehören. Variablen der Speicherklasse Automatic werden automatisch erzeugt, wenn der Programmlauf in ihren Block eintritt, und wieder gelöscht, wenn der Block verlassen wird. Alle Variablen, die innerhalb von Blöcken definiert werden, sind per Default automatische Variablen, deshalb wird das Schlüsselwort **`auto`** eigentlich kaum verwendet.

Variablen der Speicherklasse Register werden in Prozessorregister abgelegt, wenn dies möglich ist. Das ist sinnvoll für Variablen die besonders oft verwendet werden. Auf Variablen dieser Klasse kann jedoch der Adressoperator nicht angewendet werden, da Register keine Speicheradresse besitzen. Bei modernen Compiler ist das Schlüsselwort **`register`** eigentlich auch überflüssig, da sie sehr gute Optimierungsalgorithmen besitzen und Codeanalysen durchführen, um selbst oft gebrauchte Variablen und Werte zu identifizieren, und diese selbständig in Register ablegen. Mit dem Schlüsselwort **`register`** kann man diese Optimierungen sogar stören oder zunichte machen.

5.5 Lebensdauer

Die *Lebensdauer* einer Variable hängt vom Ort ihrer Definition und ihrer Speicherklasse ab. Globale Variablen und static-Variablen Leben solange wie das Programm läuft. Automatische und Registervariablen beginnen zu existieren, sobald der Programmlauf bei ihrer Definition ankommt, und hören auf zu existieren, sobald der Programmlauf den Block verlässt, in dem die Variable definiert wurde.

```
int i;              /* Globale Variable, lebt waehrend ganzem Programmlauf   */
static int i;       /* Lokale Variable, lebt waehrend ganzem Programmlauf    */

void Demo(int a)    /* Parameter, lebt bis zum Ende der Funktion             */
{
   int b;           /* Automatische Variable, lebt bis zum Ende der Funktion */
   while (a) {
      static int c; /* Statische Variable, lebt waehrend ganzem Programmlauf */
      int d;        /* Automatische Variable, lebt bis zum Ende der Funktion */
     /* mehr Code */
   }
     /* mehr Code */
}
```

5.6 Sichtbarkeit

Unter *Sichtbarkeit* versteht man, von wo aus auf eine Variable zugegriffen werden kann. Auf Variablen, die innerhalb von Blöcken definiert werden, kann nur ab Definition innerhalb dieses Blokkes (und darin eingebettete Blöcke) zugegriffen werden. Wenn ausserhalb des Blockes eine Variable gleichen Namens existiert, wird sie durch die Blockvariable verdeckt und ist nicht sichtbar. Auf globale Variablen (ohne **`static`** ausserhalb jedes Blockes definiert) kann von überall her zugegriffen werden, auch von anderen, zum Projekt gehörenden Modulen (In den anderen Modulen muss diese Variable aber als **`extern`** deklariert werden). Auf Modullokale (Ausserhalb jedes Blockes **`static`** definiert) kann nur von Funktionen innerhalb dieses Moduls (Datei) zugegriffen werden.

```
int i;              /* Von ueberall her sichtbar (Auch anderen Dateien)      */
static int i;       /* Nur in dieser Datei sichtbar                          */

void Demo(int a)    /* Nur in dieser Funktion (Demo) sichtbar                */
{
   int b;           /* Nur in dieser Funktion (Demo) sichtbar                */
   while (a) {
      static int c; /* Nur innerhalb des while()-Blocks sichtbar             */
      int d;        /* Nur innerhalb des while()-Blocks sichtbar             */
   }
}
```

5.7 Vordefinierte Variablen

[C99] In C99 ist die lokale statische Variable **`__func__`** für jede Funktion automatisch *vordefiniert*, und zwar wie wenn der Programmierer

```
static const char __func__[] = "Funktionsname";
```

geschrieben hätte. Diese Variable enthält somit einfach für jede Funktion den Namen der aktuellen Funktion. Damit kann in Fehlermeldungen oder Logdaten auf einfache Art der Name der fehlerverursachenden Funktion mit ausgegeben werden. (Ohne dass der Name der Funktion nochmals eingetippt werden müsste)

```
if (a < 0) {
   printf("Fehler in Funktion <%s>!!\n", __func__);
}
```

6 Ein-/Ausgabe einfacher Datentypen

Das *Einlesen* und *Ausgeben* von einfachen Datentypen erfolgt durch die Funktionen **`scanf()`** und **`printf().`** Die Funktionen sind sehr vielseitig, können aber bei falscher Anwendung zu Problemen und Abstürzen führen. Hier werden nur die wichtigsten Optionen dieser Funktionen erklärt, weitergehende Informationen finden sich im Kapitel 19.5.

6.1 Ausgabe mit printf():

Die Funktion **`printf()`** dient grundsätzlich zur Ausgabe von Texten, es können aber *Platzhalter* für Werte angegeben werden:

```
printf("Hallo Welt");
```

gibt den Text `'Hallo Welt'` auf der Konsole aus (Genau genommen erfolgt die Ausgabe auf den *Standard-Ausgang* [Standard Output, **`stdout`**]).

Platzhalter für Werte beginnen mit einem %, die darauf folgen Zeichen geben an, von welchem Typ der Wert ist, und wie die Ausgabe formatiert werden soll (Anzahl Stellen, rechts/linksbündig, führende Nullen...).

Wenn auf Formatierungen verzichtet wird, ergeben sich die folgenden Platzhalter:

- **`%c`** für Daten vom Typ **`char`**, gibt den Buchstaben aus, der dem ASCII-Code des Argumentes entspricht.
- **`%d`** für Daten vom Typ **`int`** (geht auch für char)
- **`%hd`** für Daten vom Typ **`short`**
- **`%ld`** für Daten vom Typ **`long`**

Für unsigned Typen ist das u anstelle von d einzusetzen

- **`%u`** für Daten vom Typ **`unsigned int`**, (geht auch für unsigned char)
- **`%hu`** für Daten vom Typ **`unsigned short`**
- **`%lu`** für Daten vom Typ **`unsigned long`**

Wenn die Ausgabe in hexadezimaler Notation erfolgen soll, muss x anstelle von d verwendet werden. (Die Zahl wird als unsigned betrachtet)

- **`%x`** für Daten vom Typ **`int`** und **`unsigned int`**, (geht auch für unsigned char)
- **`%hx`** für Daten vom Typ **`short`** und **`unsigned short`**
- **`%lx`** für Daten vom Typ **`long`** und **`unsigned long`**

Für Fliesskommazahlen gelten folgende Notationen:

- **`%f`** für Daten vom Typ **`double`**, Ausgabe in der Form nnnn.nnnn
- **`%Lf`** für Daten vom Typ **`Long double`**, Ausgabe in der Form nnnn.nnnn
- **`%e`** für Daten vom Typ **`double`**, Ausgabe in der Form n.nnnne+/-xx
- **`%E`** für Daten vom Typ **`double`**, Ausgabe in der Form n.nnnnE+/-xx
- **`%Le`** für Daten vom Typ **`Long double`**, Ausgabe in der Form n.nnnne+/-xx

  Bei g/G anstelle von f wird je nach Grösse der Zahl e/E oder f genommen

Für die Ausgabe von Strings muss **`%s`** benutzt werden.

Wenn das %-Zeichen selbst ausgegeben werden soll, muss %% geschrieben werden:

```
int Teil = 5, Ganzes = 20;
printf("%d von %d ist %d %%\n", Teil, Ganzes, 100*Teil/Ganzes);
```

Ausgabe: `5 von 20 ist 25 %`

6.2 Einlesen mit scanf():

Die Funktion **`scanf()`** dient grundsätzlich zum (formatierten) Einlesen von Werten von der Tastatur (Genaugenommen liest die Funktion vom *Standard-Eingabegerät* [Standard Input, **`stdin`**]). Die Funktion muss wissen, wo im Speicher sie den eingelesenen Wert abspeichern soll, deshalb muss ihr die Adresse der Zielvariablen übergeben werden. Die Adresse der Variablen erhält man mit dem Operator &. Nur bei Strings wird direkt die Variable angegeben, da in einer Stringvariable bereits die Adresse des ersten Zeichens des Strings abgespeichert wird.

Achtung, die Funktion liest nur so viele Zeichen wie sie benötigt, und lässt den Rest im *Eingabepuffer*. Sobald die Funktion auf ein Zeichen trifft welches sie nicht verarbeiten kann, bricht sie ab und belässt das Zeichen im Eingabepuffer. Deshalb sollte man den Eingabepuffer nach dem Aufruf von **`scanf()`** immer leeren. Die Funktion **`scanf()`**liefert die Anzahl der erfolgreich eingelesenen Werte zurück. Variablen, die nicht eingelesen werden konnten behalten ihren vorherigen Wert.

Welche und wieviele Werte eingelesen werden sollen, wird der Funktion mit einem ähnlichen Formatstring wie bei **`printf()`** angegeben. Die Einleseanweisungen beginnen mit einem %, die darauf folgenden Zeichen geben an, von welchem Typ der einzulesende Wert ist, und welches Format die Eingabe hat.

Wenn auf Formatierungen verzichtet wird, ergeben sich die folgenden Anweisungen:

`%c`	für Daten vom Typ **`char`**,	liest einen Buchstaben ein, speichert dessen ASCII-Code ab
`%d`	für Daten vom Typ **`int`**,	liest eine ganze Zahl ein und speichert diese ab
`%hd`	für Daten vom Typ **`short`**,	liest eine ganze Zahl ein und speichert diese ab
`%ld`	für Daten vom Typ **`long`**,	liest eine ganze Zahl ein und speichert diese ab

Für unsigned Typen ist das u anstelle von d einzusetzen

`%u`	für Daten vom Typ **`unsigned int`**,	liest eine ganze Zahl ein und speichert diese ab
`%hu`	für Daten vom Typ **`unsigned short`**,	liest eine ganze Zahl ein und speichert diese ab
`%lu`	für Daten vom Typ **`unsigned long`**,	liest eine ganze Zahl ein und speichert diese ab

Wenn die Eingabe in hexadezimaler Notation erfolgen soll, muss x anstelle von d verwendet werden. (Die Zahl wird als unsigned betrachtet)

`%x`	für Daten vom Typ **`int`** und **`unsigned int`**
`%hx`	für Daten vom Typ **`short`** und **`unsigned short`**
`%lx`	für Daten vom Typ **`long`** und **`unsigned long`**

Für Fliesskommazahlen gelten folgende Notationen:

`%f`	für Daten vom Typ **`float`** (**ACHTUNG**, anders als bei **`printf()`** !!!)
`%lf`	für Daten vom Typ **`double`** (**ACHTUNG**, anders als bei **`printf()`** !!!)
`%Lf`	für Daten vom Typ **`Long double`**

Anstelle von f kann auch e oder g verwendet werden, das hat keinen Einfluss auf die Art des Einlesens.

Zum Einlesen von Strings muss **`%s`** benutzt werden, **Achtung**, bei Stringvariablen darf der &-Operator nicht benutzt werden. Strings werden zudem nur bis zum ersten Whitespace (Leerzeichen, Tabulator, Zeilenvorschub...) gelesen. Mit %s können somit nur einzelne Worte eingelesen werden.

Da **`scanf()`** wie erwähnt beim Einlesen unbenutzte Zeichen im Tastaturpuffer zurücklässt, sollte der Puffer vor dem nächsten Einlesen geleert werden (Oder die ganze Zeile mit **`gets()`** auf einmal eingelesen werden und erst anschliessend mit **`sscanf()`** konvertiert werden):

```
scanf("%d", &i);               /* Einlesen      */
while (getchar() != '\n') {};  /* Puffer leeren (Lesen bis Zeilenvorschub)*/
```

6.3 Formatbezeichner für scanf (Einlesen)

`scanf("%c", &cc);`	**char** einlesen	`char cc;`
`scanf("%s", cp); Kein &!`	**char** * (String) einlesen	`char cp[20];`
`scanf("%d", &ii);`	**int** einlesen	`int ii;`
`scanf("%u", &uu);`	**unsigned int** einlesen	`unsigend int uu;`
`scanf("%hd", &ss);`	**short int** einlesen	`short int ss;`
`scanf("%hu", &su);`	**unsigned short int** einlesen	`unsigned short int su;`
`scanf("%ld", &ll);`	**long int** einlesen	`long int ll;`
`scanf("%lu", &lu);`	**unsigned long int** einlesen	`unsigned long int lu;`
`scanf("%f", &ff);`	**float** einlesen (!!!)	`float ff;`
`scanf("%lf", &dd);`	**double** einlesen (!!!)	`double dd;`
`scanf("%Lf", &ld);`	**long double** einlesen (!!!)	`long double ld;`

6.4 Formatbezeichner für printf (Ausgeben)

`printf("%c", cc);`	**char** ausgeben	`char cc;`
`printf("%s", cp);`	**char** * (String) ausgeben	`char *cp = "Hallo";`
`printf("%d", ii);`	**int** ausgeben	`int ii;`
`printf("%u", uu);`	**unsigned int** ausgeben	`unsigend int uu;`
`printf("%hd", ss);`	**short int** ausgeben	`short int ss;`
`printf("%hu", su);`	**unsigned short int** ausgeben	`unsigned short int su;`
`printf("%ld", ll);`	**long int** ausgeben	`long int ll;`
`printf("%lu", lu);`	**unsigned long int** ausgeben	`unsigned long int lu;`
`printf("%f", ff);`	**float** ausgeben(!!!)	`float ff;`
`printf("%f", dd);`	**double** ausgeben(!!!)	`double dd;`
`printf("%Lf", ld);`	**long double** ausgeben(!!!)	`long double ld;`

Beispiel:

```
double l, b;                                    /* Variablen definieren */

printf("Bitte Laenge eingeben:");               /* Text Ausgeben        */
scanf("%lf", &l);                               /* Laenge einlesen      */

printf("Bitte Breite eingeben:");               /* Text Ausgeben        */
scanf("%lf", &b);                               /* Breite einlesen      */

                                    /* Lanege, Breite und berechnete  */
                                    /* Flaeche ausgeben               */
printf("Fläche eines Rechtecks von %f * %f ist %f\n", l, b, l*b);
```

7 Operatoren

Mit *Operatoren* können ein oder zwei (beim ?-Operator sogar drei) Werte miteinander verknüpft werden. Jeder Operator liefert ein der Verknüpfung entsprechendes Ergebnis zurück. Der Typ des Ergebnisses hängt vom Operator und den beteiligten Operanden ab. Bei den arithmetischen Operatoren ist der Ergebnistyp derselbe wie jener der Operanden, wenn alle beteiligten Operanden von selben Typ sind. Ansonsten werden die Operanden in den Typ des genauesten (Grössten) beteiligten Operanden umgewandelt, und erst anschliessend die Berechnung durchgeführt. Das Ergebnis ist ebenfalls von diesem Typ.

7.1 Arithmetische Operatoren

Alle beteiligten Operanden werden in den genausten beteiligten Typ umgewandelt, das Ergebnis ist ebenfalls von diesem Typ.

Operator	Beschreibung	Beispiel
+ - * /	Standard Arithmetik. Bei / wird bei ganzzahligen Operanden nur das ganzzahlige Ergebnis geliefert (Abgeschnitten).	`a+b*c` `d / y - 4.0`
+ -	Unäre Negation	`-a`
%	Modulo Operator (Nur für ganzzahlige Typen), liefert den Rest der Division (15%4 ergibt 3)	`a % b`

++ --	Inkrement und Dekrementoperatoren, diese gibt es in postfix und prefix Form. (**a--** entspricht **a = a - 1**)
	Bei Postfix wird der Wert nach Gebrauch verändert
	Bei Prefix wird der Wert vor Gebrauch verändert
		`int x = 3; z = 7;`
		`int y = 0;`
		`y = x++; /*y ist 3*/`
		`y = ++z; /*y ist 8*/`

7.2 Bitweise Operatoren

Diese Operatoren können nur auf ganze Zahlen angewendet werden. Bei den Operatoren **& |** und **^** werden jeweils die in der binären Schreibweise einander entsprechenden Bits der Operanden miteinander verknüpft.

Operator	Beschreibung	Beispiel
&	Bitweise AND Verknüpfung (6 & 3 ergibt 2).	`a & b`
|	Bitweise ODER Verknüpfung (6 | 3 ergibt 7).	`a | b`
^	Bitweise EXOR Verknüpfung (6 ^ 3 ergibt 5).	`a ^ b`
~	Einerkomplement, Bitweise Negation.	`~a`
<<	Bitweises nach links schieben, es werden Nullen nachgeschoben (6 << 3 ergibt 48). a << b entspricht a * 2^b	`a << b`
>>	Bitweises nach rechts schieben (9 >> 3 ergibt 1). a >> b entspricht a / 2^b **ACHTUNG**, bei **signed** Operanden wird üblicherweise das Vorzeichenbit dupliziert (Prozessor und Compilerabhängig), bei **unsigned** wird immer eine 0 nachgeschoben.	`a >> b`

Beispiele:

```
3   0011        3   0011        3   0011
&               |               ^
6   0110        6   0110        6   0110
2   0010        7   0111        5   0101
```

7.3 Relationale Operatoren (Vergleichsoperatoren)

Mit den relationalen Operatoren werden die beteiligten Operanden miteinander verglichen. Wenn die Vergleichsbedingung zutrifft, wird 1 zurückgeliefert, wenn sie nicht zutrifft wird 0 zurückgeliefert. In C gilt der Wert 0 als falsch, und jeder von 0 verschiedene Wert als wahr.

Operator	Beschreibung	Beispiel
==	Gleich, liefert wahr wenn a gleich wie b ist	**a == b**
!=	Ungleich, liefert wahr wenn a nicht gleich wie b ist	**a != b**
>	Grösser als, liefert wahr wenn a grösser als b ist	**a > b**
<	Kleiner als, liefert wahr wenn a kleiner als b ist	**a < b**
>=	Grösser oder gleich	**a >= b**
<=	Kleiner oder gleich	**a <= b**

7.4 Zuweisungs Operatoren (Assignement)

Mit den Zuweisungsoperatoren wird einer Variablen ein Wert zugewiesen. Als Besonderheit in C hat der Zuweisungsoperator ebenfalls ein Ergebnis, nämlich den soeben zugewiesenen Wert (Also den Wert, den die Variable links vom Zuweisungsoperator nach der Zuweisung enthält). Zuweisungen können somit auch innerhalb von Ausdrücken auftauchen, davon sollte aber abgesehen werden, weil der Code sonst schwer verständlich wird. Einzig bei Kettenzuweisungen könnte diese Eigenschaft sinnvoll eingesetzt werden:

```
a = b = c = d = 3;  /* Alle Variablen werden auf 3 gesetzt */
```

Eine Zuweisung kann mit den meisten Operatoren zu einer *kombinierten Zuweisung* zusammengefasst werden:
Anstelle von **a = a + b** kann **a += b** geschrieben werden.

Operator	Beschreibung	Beispiel
=	Einfache Zuweisung, weist einer Variablen einen Wert zu	**a = b** **c = 17**
+=	Kombinierte Zuweisung, addiert den Wert b zu der Variablen a. a += b ist die Kurzform von a = a + b	**a += b**
-= **/= *= %=** **&= |= ^=** **<<= >>=**	Weitere kombinierte Zuweisungen, siehe Beschreibung bei **+=**. **a /= b** entspricht **a = a / b**, **a <<= b** entspricht **a = a << b**	**a %=b** **a &= b** **a *= b**

Beispiele:

```
int a = 3, b = 4, c = 1;

a += b;           /* Gleich wie a = a + b  */
c <<= 3;          /* Gleich wie c = c << 3 */
b = a + b;
a %= c;           /* Gleich wie a = a % c */
b &= c;           /* Gleich wie b = b & c */
a = b == c;       /* Weist a 0 zu wenn b ≠ c und 1 wenn b gleich wie c ist */
```

7.5 Logische Operatoren

Mit den logischen Operatoren werden *Wahrheitswerte* miteinander verknüpft. Wenn das Ergebnis der Verknüpfung *wahr* ist, wird 1 zurückgeliefert, sonst 0. In C gilt der Wert 0 als *falsch*, und jeder von 0 verschiedene Wert als *wahr*.

Operator	Beschreibung	Beispiel
&&	Logische *AND* Verknüpfung, liefert wahr wenn beide Operanden wahr sind. Der zweite Operand wird nur ausgewertet, wenn die Auswertung des ersten Operanden wahr ergibt. (Wenn der erste Operand falsch ist, ist bereits klar das auch das Ergebnis falsch ist)	`a && b` `(a > b) && (a < c)`
||	Logische *ODER* Verknüpfung, liefert wahr wenn mindestens einer der beiden Operanden wahr ist. Der zweite Operand wird nur ausgewertet, wenn die Auswertung des ersten Operanden falsch ergibt. (Wenn der erste Operand wahr ist, ist bereits klar dass auch das Ergebnis wahr ist)	`a || b` `(a > b) || (a < c)`
!	Logisches *NOT*, liefert wahr wenn der Operand falsch ist, und umgekehrt	`!a` `!(a > b)`

7.6 Spezielle Operatoren

Ausser '**Cast**', '**sizeof**', '**,**' und '**?:**' werden diese Operatoren erst später im Skript benötigt und deshalb hier noch nicht weiter erklärt.

Operator	Beschreibung	Beispiel
&	*Adressoperator*, liefert die Speicheradresse einer Variablen oder Funktion. Kann nur auf Variablen oder Funktionen angewendet werden.	`&a`
*****	*Dereferenzierungsoperator*, liefert den Wert, der an der angegebenen Adresse steht. Ist das Gegenstück zum &. Dieser Operator kann nur auf Adressen (Pointer) angewendet werden.	`*p`
sizeof **sizeof()**	Liefert den Speicherplatzbedarf seines Operanden	`sizeof(int)` `sizeof(a)`
(cast)	*Castoperator*, *Typenumwandlung*, wandelt seinen Operanden in den angegebenen Typ um, falls möglich, ansonsten erfolgt eine Compilerfehlermeldung.	`(float)a`
.	Elementauswahloperator, wird bei Strukturzugriff benötigt	`a.b`
->	Elementauswahloperator, wird bei Strukturzugriffen via Adresse (Pointer) benötigt	`p->b`
,	*Kommaoperator*, wertet seine beiden Operanden aus, gibt den Wert des rechten Operanden zurück. `a = (b++, c--, d+1);` Inkrementiert b, dekrementiert c und weist a den Wert von d+1 zu.	`a, b`
?:	*Bedingter Ausdruck*, wenn a wahr ist, wird b zurückgegeben, sonst c. `a>b?a:b` liefert das Maximum von a und b.	`a ? b : c`
()	Funktionsaufrufoperator	`printf("Hallo")`
[]	Indexoperator, wählt ein Feldelement aus (Arrayzugriff)	`x[i]`

7.7 Typumwandlungen

Eine *Typumwandlung* bewirkt, dass der Wert eines Ausdrucks einen neuen Typ erhält. Dies ist nur für skalare Typen möglich, also für arithmetische Typen und Zeiger. Die Typumwandlung wird stets so durchgeführt, dass der Wert erhalten bleibt, sofern der Wert mit dem neuen Typ darstellbar ist.

Eine Typumwandlung kann implizit sein, d.h. sie wird vom Compiler automatisch vorgenommen, oder explizit, d.h., sie wird durch die Anwendung des Cast-Operators erzwungen.

7.7.1 Implizite Typumwandlung

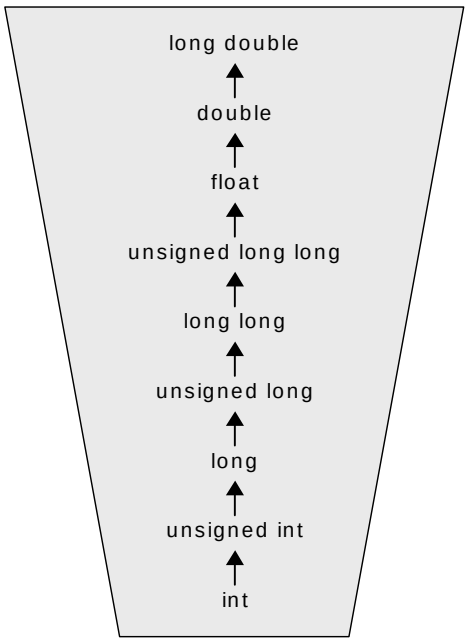
7.7.1.1 Ganzzahlige Erweiterung

Operanden vom Typ **`_Bool`**, **`char`**, **`unsigned char`**, **`short`**, **`unsigned short`** oder Bitfelder können in einem Ausdruck überall dort benutzt werden, wo Operanden vom Typ **`int`** oder **`unsigned int`** zulässig sind. Für diese Operanden wird stets die ganzzahlige Erweiterung durchgeführt, d.h. sie werden zu **`int`** bzw. **`unsigned int`** erweitert.

Ist etwa `c` eine **`char`**-Variable, wird im Ausdruck: **`c + '0'`** der Wert von c (ASCII-Code von 'c') vor der Addition zu **`int`** erweitert.

7.7.1.2 Übliche arithmetische Typumwandlungen

Die Operanden eines *binären* (zweistelligen) *Operators* dürfen unterschiedliche skalare Typen besitzen. Durch Anwendung der *Üblichen arithmetischen Typumwandlungen* wird implizit ein gemeinsamer Typ gebildet.



Dazu wird die ganzzahlige Erweiterung ausgeführt, treten danach noch Operanden mit verschieden Typen auf, wird in den Typ desjenigen Operators umgewandelt, der in der nebenstehenden Hierarchie am weitesten oben steht. Das Ergebnis ist ebenfalls von diesem Typ.

Bei der Zuweisung wird der rechte Operand immer in den Typ des linken Operanden umgewandelt. Dabei werden überzählige (Nachkomma-) Stellen oder höherwertige Bits die nicht mehr Platz haben, einfach abgeschnitten, es wird nicht gerundet. Wenn der Originalwert im neuen Typ nicht mehr darstellbar ist, findet ein Überlauf statt, das Ergebnis hat mit dem Originalwert meist nicht mehr viel gemein. (Keine Sättigung).

Beispiel:

```
int a = 5; long l = 4L; float f = 3.14f; double d = 2.0;

/* Wo wird mit welchem Typ gerechnet? */

a =   a   *   l   +   a   *   f   *   d;
```

7.7.2 Explizite Typumwandlung (cast-operator)

Die implizite Typumwandlung kann zu Fehlern, oder zumindest zu unerwünschten Ergebnissen führen. Dies kann durch Ändern der Variablentypen oder durch explizite Typumwandlungen mit dem Castoperator (gegen die Compilerregeln sozusagen) umgangen werden. Mit dem Castoperator kann eine Typumwandlung erzwungen werden. Der Castoperator besteht aus einem in Klammern eingeschlossenen gültigen Datentyp. Der Castoperator hat eine hohe Priorität und bindet sein rechtes Argument, im Zweifelsfall empfiehlt es sich dennoch, Klammern zu setzen.

Achtung, mit dem Castoperator können auch grössere Datentypen in Kleinere gezwängt werden. Der Programmierer ist aber selbst dafür verantwortlich, dass dabei keine Daten verlorengehen oder unerwünschte Ergebnisse entstehen:

Umwandlung	Bemerkungen
Kleiner Integer in grossen Integer	**Keine Probleme**
Grosser Integer in kleinen Integer	Höherwertige Bits werden abgeschnitten, falsches Ergebnis wenn Zahl ausserhalb Wertebereich
Unsigned in Signed	Falsches Ergebnis wenn Zahl ausserhalb Wertebereich
Signed in Unsigned	Falsches Ergebnis bei negativen Zahlen
Float nach Integer	Auf maximalen Wert gesättigt falls Zahl ausserhalb Wertebereich, Nachkommastellen werden abgeschnitten.
Integer nach Float	**Kaum Probleme**, ev. gehen Stellen verloren
Kleiner Float nach grossem Float	**Keine Probleme.**
Grosser Float nach kleinem Float	Auf maximalen Wert gesättigt falls Zahl ausserhalb Wertebereich, es gehen Stellen verloren.
Pointer nach Float	**Verboten**
Float nach Pointer	**Verboten**
Pointer auf A nach Pointer auf B	Zeiger behalten Adresse, Speicherinhalt wird anders interpretiert
Pointer nach Integer	Integer enthält Adresse als Wert, sofern sie Platz hat, sonst nur Teil von Adresse
Integer nach Pointer	Wert wird in Adresse umgewandelt
In gleichen Typ mit anderen Qualifizierern	**Keine Probleme** (Wenn man weiss was man tut) (z.B. **`volatile int`** in **`int`**, oder **`int`** in **`const int`**)

Beispiele:

```
int a = 5, b = 2;
float f = 0.0;
int c = 257;

f = a / b;           /* f ist 2 !!!! nicht etwa 2.5 weshalb?    */
f = (float) a / b;   /* jetzt ist f 2.5                         */
f = (float) (a / b); /* jetzt ist f wieder 2                    */

b = (char)c;         /* b ist nun 1,                weshalb??   */
f = a / 3.0;         /* f ist jetzt 1.6666666       weshalb??   */
c = a / 3.0;         /* c ist jetzt 1               weshalb??   */
c = a / 3.0 + 0.5;   /* c ist jetzt 2               weshalb??   */
c = a / 3   + 0.5;   /* c ist jetzt 1               weshalb??   */
```

7.8 Präzedenzen

Wie in der gewöhnlichen Mathematik gibt es auch in C eine *Operatorrangfolge*, nach deren Regeln die Rechenreihenfolge der Berechnungen definiert ist (Analog zu der Regel 'Punkt vor Strich'). Diese Reihenfolge kann durch das setzten von *Klammern* ausser Kraft gesetzt werden, Klammern haben die höchste Priorität. Es empfiehlt sich, auch Klammern zu setzen wo es nicht nötig ist, Klammern schaffen Klarheit.

In der nachfolgenden Tabelle stehen die Operatoren nach Priorität geordnet, die mit der höchsten Priorität stehen zuoberst. Die Operatoren innerhalb eines Feldes haben dieselbe Priorität.

Die *Assoziativität* definiert, ob bei gleichrangigen Operatoren von rechts nach links, oder von links nach rechts gerechnet wird.

Beispiel für Assoziativität: **`a*b/c*d`** wird in der Reihenfolge **`(((a*b)/c)*d)`** ausgewertet.
`a=b=c=d` wird in der Reihenfolge **`(a=(b=(c=d)))`** ausgewertet.

Operator	Bemerkungen	Assoziativität
`[] () . ->` `++ --`	Postfix Version (a++ b--)	Von links nach **rechts**

`& *`
`+ -`
`! ~`
`++ --`
`sizeof()`
`(cast)`
	Adress- und Dereferenz-Operator.
	Unäres Plus oder minus (-b)
	Logisches Not; Einerkomplement
	Prefix Version (++a --b)
	z.B. (int) (unsigned char) (float)
		Von **rechts** nach links

`* / %`		Von links nach **rechts**
`- +`		Von links nach **rechts**
`>> <<`	Schieben	Von links nach **rechts**
`> < >= <=`	Vergleich	Von links nach **rechts**
`== !=`	Vergleich	Von links nach **rechts**
`&`	Binäres And	Von links nach **rechts**
`^`	Binäres Exor	Von links nach **rechts**
`|`	Binäres Or	Von links nach **rechts**
`&&`	Logisches And	Von links nach **rechts**
`||`	Logisches Or	Von links nach **rechts**
`?:`		Von **rechts** nach links
`= *= /= %= += -=` `^= |= &= <<= >>=`	Zuweisungen	Von **rechts** nach links
`,`	Kommaoperator, Sequenzoperator	Von links nach **rechts**

Beispiel:

Setzen Sie Klammern nach den Präzedenzregeln, und bestimmen Sie den Wert des Ausdrucks:

3 & 1 == 3 > 5 * 6 * 2 + 3 >> 4 % 3 && 4 - 5

8 Anweisungen/Statements

8.1 Ausdrücke/Expressions

Ausdrücke (*Expressions*) sind im wesentlichen nichts anderes als Berechnungen. Ein Ausdruck ist eine erweiterte Rechenvorschrift, welche ein Ergebnis/Resultat liefert.

Beispiele für Ausdrücke:

```
(12+Durchschnitt)/15
Radius*2.0*3.1416
sin(2.0*3.1416*f)+0.5*sin(4.0+3.1416*f)
```

Ausdrücke können eine beliebige Anzahl von Operatoren, Operanden und Kombinationen derselben enthalten, es können auch Klammern gesetzt werden und Funktionsaufrufe benutzt werden. Die Funktionen müssen aber einen Wert zurückliefern.

8.2 Einfache Anweisung

Eine *Anweisung* (*Statement*) ist ein Befehl an den Computer, etwas zu tun. Jede Anweisung wird mit einem Semikolon (;) abgeschlossen. Ein alleinstehendes Semikolon gilt auch als Anweisung (Eine *leere Anweisung* die nichts tut). Mit Semikolon abgeschlossene Expressions sind auch Statements.

Beispiele für Anweisungen:

```
a = 15;
Flaeche = Radius*Radius*3.1416;
printf("Hallo");
;                   /* Leere Anweisung, macht einfach nichts    */
printf;             /* Ebenfals korrekt (!!), hat aber keinen Effekt */
a + 4 * b;          /* Ebenfals korrekt (!!), hat aber keinen Effekt */
```

8.3 Blöcke

Mehrere Anweisungen können mit geschweiften Klammern zu einem *Block* und so zu einer einzigen Anweisung (*compound statement*) zusammengefasst werden:

```
{
   int Radius = 3;                              /* Deklarationen */
   static long Flaeche;

   ++Radius;                                   /* Anweisungen    */
   Flaeche = Radius*Radius*3.1416;
   printf("Hallo");
   if(Flaeche >= 34)
   { ... }                                      /* noch ein Block */
}
```

Das Einrücken des Blockinhaltes ist nicht erforderlich, erhöht aber die Übersicht im Code und erleichtert das Verstehen des Codes. Jeder gute Programmierer wird seinen Code so übersichtlich wie möglich gestalten.

Am Anfang eines neuen Blocks dürfen Variablen definiert werden. Variablen, die innerhalb eines Blocks definiert werden, gelten nur innerhalb des Blocks, und *verdecken Variablen* des gleichen Namens von ausserhalb des Blockes. Sobald der Block verlassen wird, sind innerhalb dieses Blocks definierte Variablen nicht mehr gültig und verlieren ihren Inhalt (Ausser statische Variablen).

[C99] In C99 dürfen Variablen an beliebigen Stellen innerhalb eines Blockes definiert werden, können aber erst nach ihrer Definition benutzt werden.

9 Kontrollstrukturen

Da eine einfache Abfolge von Befehlen für viele Programme nicht ausreicht, gibt es Anweisungen zur Steuerung des *Programmflusses*. Dazu stehen dem Programmierer folgende Anweisungen zur Verfügung:

Verzweigungen mit **`if else`** oder **`switch`**,

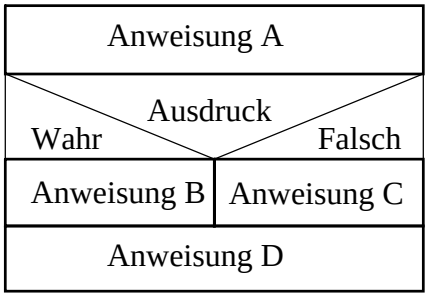
Schleifen mit **`while`**, **`do while`** oder **`for`**,

Unbedingte Sprünge mit **`goto`**, **`continue`**, **`break`** oder **`return`**.

9.1 Verzweigungen

Mit Verzweigungen können abhängig von Bedingungen bestimmte Codeteile ausgeführt oder ignoriert werden. Damit kann auf verschiedene Daten unterschiedlich reagiert werden.

9.1.1 if, else



Wenn der Ausdruck im nebenstehenden Struktogramm wahr ist, wird nach der Anweisung A die Anweisung B ausgeführt und anschliessend Anweisung D. Wenn der Ausdruck falsch ist, wird die Anweisung C ausgeführt, und dann geht's weiter mit der Anweisung D. In C wird dies wie folgt geschrieben:

```
Anweisung A
if(Ausdruck)
   Anweisung B
else
   Anweisung C
Anweisung D
```

Ausdruck muss einen *skalaren* Typ haben. Zuerst wird der **`if`**-Ausdruck ausgewertet. Ist das Ergebnis ungleich 0, d.h. wahr, wird die Anweisung B ausgeführt. Andernfalls wird bei vorhandenem **`else`**-Zweig die Anweisung C ausgeführt.

Es gibt häufig auch den Fall, dass der **`else`**-Zweig mit der Anweisung C entfällt:

```
Anweisung A
if(Ausdruck)
   Anweisung B
Anweisung D
```

Wenn der Ausdruck falsch ist, wird nach A also sofort D abgearbeitet.

Beispiel:

```
/* Read a number and test if zero or non-zero */
#include <stdio.h>                  /* standard stream I/0 library */

int main(int argc, char *argv[])
{
   int i;
   printf("Ganze Zahl eingeben !\n");
   scanf("%d", &i);                                  /* read number */
   if (i == 0) {
      printf("value was zero (%d)\n", i);
   } else {
      printf("value was positive or negative (%d)\n", i);
   }
   return 0;
}
```

Wenn die der **`if`**-Abfrage folgende Anweisung aus mehreren Anweisungen besteht, müssen diese in einem Block zusammengefasst werden (In geschweiften Klammern eingeschlossen werden). Entsprechendes gilt für die auf **`else`** folgenden Anweisung(en). **Grundsätzlich empfiehlt es sich, auch einzelne Anweisungen in Klammern zu setzen.**

Fehlermöglichkeiten:

Das Zusammenfassen mehrerer Instruktionen zu einem Block wird vergessen, oder direkt hinter dem **`if()`** ein Semikolon gesetzt (= leere Anweisung), und statt "**`if(i == 0)`**" wird vor allem von Anfängern oft "**`if(i = 0)`**" geschrieben (Dies ist eine gültige Syntax und wird vom Compiler höchstens mit einer Warnung geahndet, es wird der Variable i der Wert 0 zugewiesen, und der Wert des Ausdrucks ist 0, und somit falsch).

Beispiel:

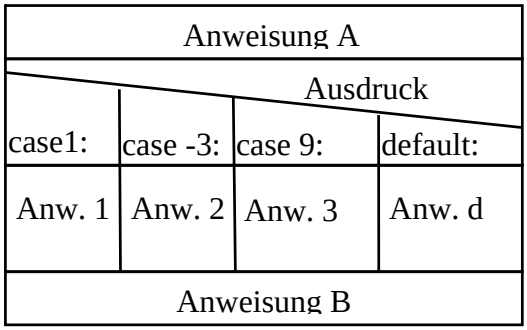
Bei der Eingabe von Werten durch den Benutzer muss mit Fehleingaben gerechnet werden. Zum Beispiel mit ungültigen Zeichen bei der Eingabe von Zahlen. Solche Fehler können (und sollten) bereits bei der Eingabe abgefangen werden. Die Funktion **`scanf()`** gibt einen Integerwert zurück, der die Anzahl der erfolgreichen Umwandlungen angibt: Wenn das Einlesen eines oder mehrerer Werte nicht erfolgreich war, ist die Anzahl der erfolgreichen Umwandlungen kleiner als die Anzahl der einzulesenden Argumente. Programmseitig kann man diese Eigenschaft wie folgt nutzen:

```
if (scanf("%d", &i) == 1) {     /* Es sollte 1 Argument eingelesen werden */
      printf("1 korrekter Einlesevorgang. Wert: %d\n", i);
} else {
      printf("Fehler beim Einlesen");
}
```

Aufgabe 9.1:

`if`-Anweisungen können beliebig verschachtelt sein. Schreiben Sie ein Programm, in welchem Sie eine ganze Zahl einlesen und ausgeben, ob die eingegebene Zahl grösser, gleich oder kleiner Null war.

9.1.2 switch



Mit der **switch**-Anweisung kann eine Mehrfachverzweigung realisiert werden. Der Wert des **switch**-Ausdrucks wird mit den Konstanten bei den **case**-Marken verglichen, anschliessend wird zur entsprechenden Marke verzweigt und die entsprechende Anweisung ausgeführt. Wenn keine passende Marke existiert, wird bei der Marke **default** weitergefahren, falls vorhanden, und sonst nach der **switch**-Anweisung

```
Anweisung A
switch ( Ausdruck ) {    /* Ausdruck wird ausgewertet */
   case 1:  Anweisung 1  /* Falls Ausdruck 1 ergibt, wird hier weitergefahren
*/
            break;
   case -3: Anweisung 2  /* Falls Ausdruck -3 ergibt, wird hier weitergefahren
*/
            break;
   case  9: Anweisung 3  /* Falls Ausdruck 9 ergibt, wird hier weitergefahren
*/
            break;
   default: Anweisung d  /* In allen anderen Faellen wird hier weitergefahren */
            break;
}
Anweisung B
```

Ausdruck muss ein ganzzahliges Ergebnis haben. Alle **case**-Konstanten müssen verschieden sein und es ist nicht möglich, bei einer Marke einen Bereich anzugeben. Um einen Bereich abzudecken müssen alle enthaltenen Werte explizit mit Marken angegeben werden. **Achtung**, bei durch Kommas (Kommaoperator!) getrennten Konstanten gilt einfach die letzte Konstante! (Frage: wieso?).

Einsatz:

Eine mögliche Anwendung für die **switch**-Anweisung ist das Überprüfen einer Eingabe, die in verschiedene Fälle verzweigt. Zum Beispiel bei der Behandlung einer Menuabfrage. Dabei wird zuerst eine Bildschirmmaske mit den verschiedenen Eingabemöglichkeiten angezeigt, die Auswahl des Benutzers eingelesen und anschliessend in eine der vier Möglichkeiten verzweigt.

```
printf("Bitte waehlen Sie eine der folgenden Aufgaben \n");
printf("K - Kontostand abfragen \n");
printf("E - Dauerauftrag einrichten \n");
printf("U - Ueberweisung taetigen \n");
printf("A - Aufhoeren \n");

scanf("%c", &eingabe);
switch(eingabe) {

   case 'K': kontostand();            /*Rufe kontostand() Funktion auf*/
             break;
   case 'E': dauerauftrag();        /*Rufe dauerauftrag() Funktion auf*/
             break;
   case 'U': ueberweisung();        /*Rufe ueberweisung() Funktion auf*/
             break;
   case 'A': aufhoeren();             /*Rufe aufhoeren() Funktion auf*/
             break;
   default : printf("Bitte vertippen Sie sich nicht andauernd \n");
             break;
}
```

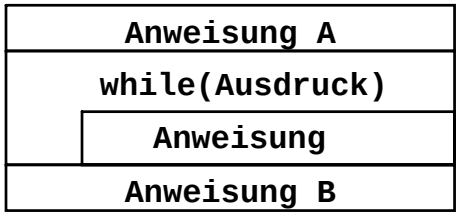
Ausgabe des Programms:
Bitte waehlen Sie eine der folgenden Aufgaben
K - Kontostand abfragen
E - Dauerauftrag einrichten
U - Ueberweisung taetigen
A - Aufhoeren

Nach der Verzweigung zu einer Marke wird das Programm sequentiell fortgesetzt, wobei die Marken keine Bedeutung mehr haben. Mit der **break**-Anweisung kann ein **switch** jederzeit beendet werden. Ein **break** ist also notwendig, wenn die Anweisungen der nächsten **case**-Konstanten nicht ausgeführt werden sollen. Es ist keine Reihenfolge für **case** und **default** vorgeschrieben.

9.2 Schleifen

In einer *Schleife* wird eine Gruppe von Anweisungen, der sogenannte Schleifen-Rumpf, mehrfach ausgeführt. Zur Bildung einer Schleife stehen in C drei Anweisungen zur Verfügung: **while**, **do-while** und **for**.

9.2.1 While



Bei der **while**-Anweisung wird die zugehörige Anweisung solange wiederholt, wie der Ausdruck wahr ist. Sobald der Ausdruck falsch ist, wird mit der auf die **while**-Schleife folgenden Anweisung B weitergefahren. Wenn der **while** Ausdruck von Anfang an falsch ist, wird die Schleifen-Anweisung gar nie Ausgeführt.

```
Anweisung A
while (Ausdruck)
   Anweisung
Anweisung B
```

Nehmen wir an, Sie wollen herausfinden, wie lange Sie sparen müssen um Millionär zu werden, wenn Sie jeden Monat hundert Taler sparen und zur Bank bringen, wo das Kapital noch verzinst wird. In C könnte das Problem folgendermassen gelöst werden:

```
mein_guthaben = 0.0;
monat = 0;
while(mein_guthaben < 1000000.0) {          /* Wiederholen bis Millionaer */
   mein_guthaben = mein_guthaben * 1.003;    /*      monatl. Zinssatz 0,3% */
   mein_guthaben += 100.0;                                 /* Monatsrate */
   monat++;                                /* Monatszaehler inkrementieren */
}
printf("Nach %d Monaten bin ich endlich Millionaer \n", monat);
```

9.2.1.1 Fehlerquelle

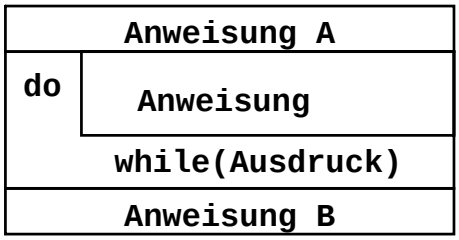
Wenn am Ende der Whileschlaufe ein Semikolon gesetzt wird, gilt dieses als leere Anweisung, welche solange wiederholt wird, wie die **while**-Bedingung wahr ist.

```
/* Fehler, Block wird hoechstens einmal durchlaufen, While moeglichger     */
/* weise nie beendet                                                       */

while(mein_guthaben < 1000000.0);
{
   mein_guthaben = mein_guthaben * 1.003 + 100.0;  /* monatl. Zins u. Rate */
}
```

Diesen Fehler kann man vermeiden, wenn die öffnende geschweifte Klammer gleich an **while** angefügt wird (Zumindest wird so der Fehler offensichtlich):

```
while(mein_guthaben < 1000000.0) {
   mein_guthaben = mein_guthaben * 1.003 + 100.0;  /* monatl. Zins u. Rate */
}
```

9.2.2 do while



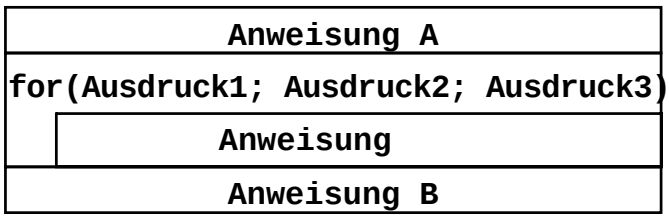
Auch mit der **do while**-Anweisung wird ein Programmstück solange wiederholt, wie ein Ausdruck wahr ist. Diese Iteration unterscheidet sich von der vorigen dadurch, dass die Iterations-Bedingung am Ende abgefragt wird; diese Schleife wird deshalb in jedem Fall mindestens einmal durchlaufen.

```
Anweisung A
do
   Anweisung
while (Ausdruck);   /* Nach dem while muss ein Semikolon stehen !!!*/
Anweisung B
```

Das nachfolgende Beispiel bestimmt das ε (Die Auflösung) von **float**

```
/* Bestimmt das epsilon (Aufloesung) vom Typ float, also die kleinste */
/* Zahl epsilon, so dass 1.0 + epsilon != 1.0                         */

#include <stdio.h>          /* standard stream I/0 library */
#include <float.h>          /* float types header */

int main(int argc, char *argv[])
{
   float epsilon = 1.0f;
   printf("float epsilon aus <float.h> = %g\n", FLT_EPSILON);

   do {
      epsilon = epsilon / 2.0f;
   } while((1.0f + epsilon) != 1.0f);

   printf("float epsilon berechnet = %g", epsilon * 2.0f);

   while(1);                /* Endlosschlaufe, die nichts tut, Programm       */
   return 0;                                        /* bleibt hier haengen */
}
```

9.2.3 for

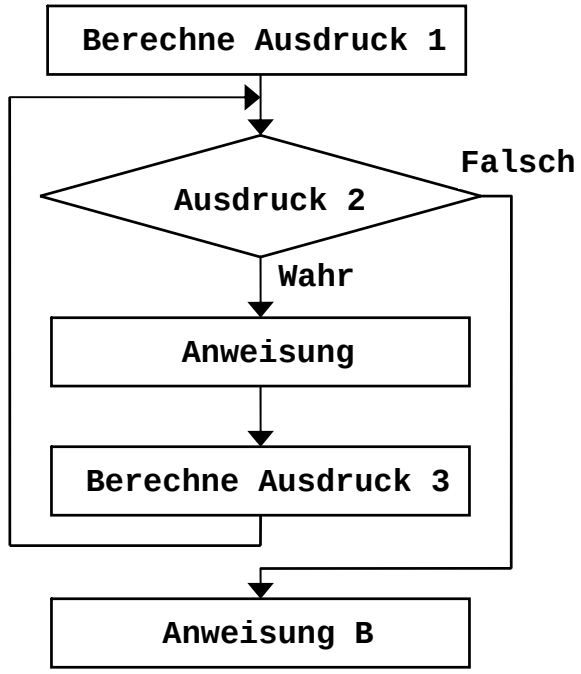


Die **for**-Schleife wird verwendet, wenn die Anzahl der Iterationen (Schleifendurchläufe) bereits im Voraus bekannt ist. Im Struktogramm sieht die **for**-Schlaufe gleich wie die **while**-Schlaufe aus.

Die Schleife wird solange wiederholt wie Ausdruck2 wahr ist. Ausdruck1 wird nur einmal vor dem ersten Schlaufendurchlauf ausgewertet, Ausdruck3 am Ende jeder Iteration.

```
Anweisung A
for(ausdruckl; ausdruck2; ausdruck3)
      Anweisung
Anweisung B
```

Der obige allgemeine Ausdruck für die **for**-Schlaufe wird im folgenden Flussdiagramm illustriert.



Die einzelnen Ereignisse laufen dabei wie folgt ab:

Ausdruck1 wird ausgewertet
Ausdruck2 wird ausgewertet
Wenn **Ausdruck2** wahr ist, wird **Anweisung** ausgeführt und **Ausdruck3** ausgewertet und in die vorige Zeile zurückgesprungen, sonst wird die Schleife beendet und mit **Anweisung B** weitergefahren

`Ausdruck1` dient zur Initialisierung der Schlaufe, **`Ausdruck2`** ist die Abbruchbedingung und **`Ausdruck3`** die Iterationsanweisung.

In Ausdruck1, 2, 3 kann der Kommaoperator verwendet werden, um mehrere Variablen zu Initialisieren oder aktualisieren.

Der oben beschriebene Sachverhalt kann auch mit einer while-Schlaufe erzielt werden. Er ist nämlich äquivalent zu:

```
ausdruckl;
while(Ausdruck2) {
      Anweisung
      ausdruck3;
}
Anweisung B
```

Da beide Formen äquivalent sind, kann man sich aussuchen, für welche man sich entscheidet. Allgemein gilt die Regel: **`for(ausdruck1; ausdruck2; ausdruck3)`** sollte in eine Zeile passen, ansonsten sollte man **`while(ausdruck2)`** wählen, was sicher kürzer ist als der entsprechende for-Ausdruck, aber theoretisch natürlich ebenfalls länger als eine Zeile werden kann.

Hier ein typisches Beispiel für den Einsatz einer **`for`**-Schlaufe:

```
float x = 2.0f;                              /* was passiert hier wohl? */
for(i=0; i<10; i++) {
      printf("%d\n", i);
      x = x*x;
}
```

Die obige Iteration beginnt bei i=0 und endet mit i=9. Andere Initialisierungen für den ersten Wert von i und andere Schrittweiten (z.B. **`i += 13`** anstelle von **`i++`**) sind möglich. Die **`for`**-Schlaufe ist aber noch viel allgemeiner und flexibler einsetzbar.

Mit der leeren Anweisung **`for(;;)`** kann eine *Endlosschlaufe* erzeugt werden:

```
for(;;) {
   /* Diese Schlaufe hoert nie auf, wenn kein break enthalten ist */
}
```

Betrachten Sie nun das folgende Programm genau, welches das Kapitalwachstum einer Einheit (1000) über einen bestimmten Zeitraum bei einem bestimmten Zinssatz tabellarisch darstellt:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   int years, rate, input;
   printf("\nZinssatz in %% und Laufzeit in Jahren eingeben (integer):");
   /* Zinssatz und Zeitraum einlesen */
   input = scanf("%d %d", &rate, &years);
   while(input != EOF) {                          /* Bei Abbruch beenden */
      /* Bei Einlesefehler das Fehlerverursachende Zeichen ausgeben       */
      if (input != 2) {
         printf("\a Ungueltiges Zeichen '%c', neuer Versuch...", getchar());
      } else {
         /* Alles OK, Tabelle berechenen und ausgeben                     */
         float capital = 1000;                          /* Startkapital */
         int year;

         /* Kapital ueber alle Jahre verzinsen, Tabelle ausgeben          */
         for (year = 1; year <= years; year++) {
            float interest = capital * rate / 100.0f;   /* Zins berechnen */
            capital = capital + interest;

            /* Tabellenzeile formatiert ausgeben */
            printf("Jahr = %d, Zins = %6.2f%%, Kapital = %7.2f\n",
                                              year, interest, capital);
         }
         printf("Nach %d Jahren bei %d%%: Kapital = $%.2f, Wachstum =
              %.2f%%", years, rate, capital, 0.1f * (capital - 1000.0f));
      }
      printf("\nZinssatz in %% und Laufzeit in Jahren eingeben (integer):");
      /* Zinssatz und Zeitraum einlesen */
      input = scanf("%d%d", &rate, &years);
   }
   return 0;
}
```

[C99] In C99 können innerhalb des **`for`**-Statements im Initialisierungsteil auch gleich Variablen definiert werden. Diese gelten für den ganzen Schleifenkörper.

```
/* [C99], Definition von Variablen in Initialisierung */
for (int r = 17; r > 0; r -= 2) {
   a = a * r;
}

/* ist äquvalent zu */
{
   int r;   /* Variablendefinition, nur innerhalb des Blocks gueltig */
   for (r = 17; r > 0; r -= 2) {
      a = a * r;
   }
}
```

Weiteres Beispiel (Codefragment mit Kommaoperator und flexiblerem Einsatz):

```
for (x = Wert, Stellen = 0, Einer = 0; x > 0; x /= 2, Stellen++)
{
   Einer += x % 2;   /* Einer inkrementieren falls x ungerade */
}
printf("\n%d hat %d Binaerstellen, wovon %d Einer\n", Wert, Stellen, Einer);
```

9.3 Weitere Steuerungsanweisungen: `continue, break, goto`

Die **`continue`**-Anweisung wird innerhalb von **`while`**-, **`do`**- und **`for`**-Anweisungen eingesetzt. Man kann damit den Rest der Schleife überspringen und mit dem nächsten Schleifendurchlauf beginnen (**`continue`** wirkt wie ein Sprung an das Schleifenende, somit wird die Iterationsanweisung der **`for`**-Schlaufe bei **`continue`** auch ausgeführt):

```
while( i < 10)
{
      ...
      if(x == 0) {
         continue;
      }
      ...
}
```

Wenn möglich sollte **`continue`** vermieden werden:

```
while(i < 10 )
{
      ...
      if(x != 0) {
            ...
      }
}
```

Die **`break`**-Anweisung wird innerhalb von **`while`**-, **`do`**-, **`for`**- und **`switch`**-Anweisungen eingesetzt. Mit **`break`** wird die aktuelle Schleife abgebrochen und die weitere Abarbeitung des Programms mit der nächsten auf diese Schleife folgenden Anweisung fortgesetzt. Die häufigste Anwendung von **`break`** finden Sie in der weiter oben schon beschriebenen **`switch`**-Anweisung. Wie **`continue`** sollte auch **`break`** (Ausser in **`switch`**-Anweisungen) nur mit grosser Vorsicht benutzt werden.

Die **`goto`**-Anweisung ist in guten Programmiererkreisen so verpönt, dass schon ihre blosse Erwähnung anstössig wirkt. Ein Programm mit **`goto`**-Anweisungen ist meist so unverständlich und verwickelt ("Spaghetti-Code"), dass nur der Entwickler selbst (wenn überhaupt!) noch einen Überblick darüber besitzt, was in seinem Programm vorgeht. **`goto`** wird deshalb hier nur der Vollständigkeit halber erwähnt. Die **`goto`**-Anweisung hat folgende allgemeine Form:

```
goto bezeichnung;
...
bezeichnung: Anweisung;
...
goto bezeichnung;
```

Mit einem **`goto`** kann nur innerhalb einer Funktion herumgesprungen werden, das Springen in andere Funktionen ist nicht möglich.

[C99] In C99 dürfen **`goto`**'s nicht in den Gültigkeitsbereich eines *VLAs* springen, das Verlassen des Gültigkeitsbereichs hingegen ist gestattet.

10 Funktionen

Im Gegensatz zu anderen Programmiersprachen wird in C nicht zwischen *Funktionen* und *Prozeduren* unterscheiden. In C gibt es nur Funktionen, wobei eine Funktion aber nicht zwingend einen *Rückgabewert* haben muss.

Eine Funktion ist ein *Unterprogramm*, eigentlich ein Stück Code, welches einen Namen hat, dem Werte übergeben werden können und das ein Ergebnis zurückliefern kann. Jede Funktion kann von anderen Funktionen aus über ihren Namen aufgerufen werden.

Ein C-Programm besteht grundsätzlich aus einer Menge von Funktionen, und mindestens die Funktion **`main()`** muss vorhanden sein, denn diese wird beim Start eines Programmes automatisch aufgerufen, und wenn **`main()`** an sein Ende gelangt wird auch das Programm beendet.

10.1 Definition einer Funktion

Eine *Funktionsdefinition* besteht aus einem Ergebnistyp (Dem Typ des Rückgabewertes), einem Funktionsnamen, einer *Parameterliste* und dem eigentlichen *Funktionsrumpf* (Code) in geschweiften Klammern.

```
float Parallelschaltung (float Ra, float Rb)
{
   float Rparallel = 0.0;

   Rparallel = 1/(1/Ra + 1/Rb);   /* Berechnung der Parallelschaltung */
   return Rparallel;
}
```

10.2 Aufruf einer Funktion

Eine Funktion wird aufgerufen durch Angabe ihres Namens, gefolgt von einer in einem Klammerpaar eingeschlossenen *Argumentenliste* (Die Argumentenliste kann auch leer sein):

```
float R1 = 3.3, R2 = 5.6, Rtot = 0.0;
Rtot = Paralleleschaltung (4.7, 2.2);
Rtot = Paralleleschaltung (Rtot, R1);
Rtot = Paralleleschaltung (Rtot, R2);
```

10.3 Übergabe von Argumenten

Einer Funktion muss immer die korrekte Anzahl an Argumenten übergeben werden, zudem müssen auch die Datentypen der Argumente mit den entsprechenden Parametern der Funktion kompatibel (ineinander umwandelbar) sein. Die Argumente werden dabei gemäss ihrer Reihenfolge an die Funktion übergeben, die Namen der Variablen spielen keine Rolle.

In C werden alle Argumente *by Value* übergeben, das heisst dass die Werte kopiert werden, und in der Funktion mit Lokalkopien gearbeitet wird. Somit kann eine Funktion die Originalvariablen nicht verändern. (Zumindest nicht direkt, siehe 'Call by Reference' im nächsten Abschnitt)

Die einzige Ausnahme sind Arrays (Felder), diese werden nicht kopiert, sondern es wird eine Referenz (Pointer) auf das Array übergeben. Die Funktion hat somit direkten Zugriff auf das Originalfeld, und kann dieses auch verändern.

Wenn eine Funktion auf Variablen der aufrufenden Funktion zugreifen soll, müssen ihr die Adressen der betroffenen Variablen übergeben werden (Kapitel Pointer). Diese Übergabe wird auch als '*by Reference*' (Im Gegensatz zu 'by Value') bezeichnet.

10.4 Lokale Variablen

Auf lokale Variablen, (Argumente und innerhalb des Funktionskörpers definierte Variablen) kann von ausserhalb der Funktion nicht zugegriffen werden. Die Variablen werden zudem ungültig und verlieren ihren Inhalt, sobald die Funktion beendet wird. (Ausnahme: statische (**`static`**) Variablen behalten ihren Inhalt.)

Innerhalb von Funktionen können keine weiteren Funktionen definiert werden, Funktionslokale Funktionen sind somit nicht möglich.

10.5 Rückgabewert einer Funktion ('Ergebnis')

Mit dem Befehl **`return`** wird die Funktion beendet, und der angegebene Wert als Ergebnis des Funktionsaufrufes zurückgeliefert. Eine Funktion kann mehrere **`return`** Statements enthalten, eine Funktion ohne Rückgabewerte darf nur **`return`**-Anweisungen ohne Wert aufweisen, oder kann auch gar kein **`return`** Statement aufweisen. Sobald der Programmlauf auf ein **`return`** stösst, wird die Funktion beendet. Eine Funktion mit Rückgabewert muss mindestens ein Returnstatement aufweisen, und jedes **`return`** muss einen Rückgabewert aufweisen.

```
int Maximum1(int a, int b)
{
   int Resultat = 0;

   if (a > b) {
      Resultat = a;
   } else {
      Resultat = b;
   }
   return Resultat;
}
```

```
int Maximum2(int a, int b)
{
   if (a > b) {
      return a;
   } else {
      return b;
   }
}
```

Auch möglich, aber weniger verständlich:

```
int Maximum3(int a, int b) { return (a > b) ? a : b; }
```

10.6 Spezielle Funktionen

Eine Funktion die nichts zurückliefert (in anderen Sprachen als Prozedur oder Unterprogramm bezeichnet) wird als Funktion die **`void`** zurückliefert definiert:

```
void PrintStar(int n)    /* Zeichnet eine Linie von Sternen */
{
   while (n-- > 0) {
      putchar('*');
   }
}
```

Eine Funktion, die keine Argumente hat, besitzt entsprechend eine leere Parameterliste:

```
double CalculatePi(void)    /* Berechnet Pi */
{
   return 22.0/7.0;    /* Algorithmus kann noch verbessert werden */
}

Flaeche = Radius * Radius * CalculatePi();   /* Aufruf der Funktion */
```

Analog sind auch Funktionen möglich, die weder Argumente, noch Rückgabewerte besitzen.

Funktionen sind in C *reentrant*, d.h. sie können rekursiv verwendet werden, also sich selbst aufrufen (Rekursionen werden später noch behandelt). Nachfolgend die Berechnung der Fakultät als Beispiel für eine rekursive Funktion:

```
int Fakultaet(int n)
{
   if (n > 0) {
      return n * Fakultaet(n-1);
   } else {
      return 1;
   }
}
```

Definition der Fakultät:
n! = n * (n-1)! *und* ***0! = 1***
oder weniger formal:
*n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1*

(Die *Fakultät* kann mit Schleifen natürlich viel effizienter berechnet werden)

10.7 Deklaration einer Funktion (Funktions-Prototyp)

Bevor eine Funktion verwendet werden kann, muss sie definiert, oder zumindest deklariert werden. Eine *Funktionsdeklaration* wird als *Prototyp* einer Funktion bezeichnet. Ein *Funktionsprototyp* besteht einfach aus dem Funktionskopf, gefolgt von einem Semikolon. Damit wird dem Compiler die Funktion bekannt gemacht, und sie kann verwendet werden. (In C kann eine Funktion auch ohne vorangehende Definition/Deklaration verwendet werden, der Compiler nimmt dann an, dass sie **`int`** zurückliefert und die gerade verwendeten Argumente erwartet, was zu unerwünschten und überraschenden Ergebnissen führen kann. [C99] In C99 muss eine Funktion immer bekannt sein, bevor sie verwendet werden darf.).

Beispiele für Prototypen:

```
int Fakultaet(int n);        /* Prototyp fuer Fakultaet   */
double CalculatePi(void);    /* Prototyp fuer CalculatePi */
```

Prototypen stehen für globale Funktionen normalerweise in *Headerdateien* (.h -Files), für lokale (static) Funktionen zu Beginn der entsprechenden Datei.

10.8 Lebensdauer lokaler Variablen

Lokale Variablen von Funktionen verlieren ihren Inhalt und werden ungültig, sobald der Block verlassen wird, in dem sie definiert wurden. Wenn man eine lokale Variable als **`static`** definiert, ist sie zwar nach wie vor nur innerhalb ihres Blocks zugreifbar, aber sie behält ihren Wert bis zum nächsten Funktionsaufruf. Die Initialisierung von static-Variablen wird nur einmal zu Beginn des Programms ausgeführt.

Beispiel:

```
int Counter(void)
{
   static int Count = 0;    /* Statische lokale Variable */

   Count++;
   printf("Die Funktion wurde %d mal aufgerufen\n", Count);
   return Count;
}
```

10.9 [C99] Inline Funktionen

[C99] In C99 können Funktionen auch **`inline`** definiert werden. Bei Inlinefunktionen wird beim Aufruf der Funktion nicht die Funktion aufgerufen, sondern vom Compiler der Code der Funktion an dieser Stelle eingefügt. Dadurch wird der Code schneller, da der Funktionsaufruf wegfällt, dafür wird er länger, da bei jedem Aufruf der Funktionscode eingefügt wird. **`inline`** sollte nur bei kurzen, einfachen Funktionen verwendet werden. **`inline`** ist nur eine Anfrage, der Compiler muss die Funktion nicht inline compilieren, er darf das Schlüsselwort auch einfach ignorieren. (Rekursive Funktionen können z. B. gar nicht inline sein.).

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

10.10 Funktionen mit variabler Argumentenliste

Es können auch Funktionen mit einer unbekannten Anzahl an Argumenten definiert werden, wobei aber mindestens ein Argument vorhanden sein muss. Die unbekannten Argumente werden mit einer *Ellipse* ... (drei Punkte) als letztes Argument angegeben. Auf die zusätzlichen Argumente kann mit den Makros **`va_list`**, **`va_start`**, **`va_end`**, **`va_arg`** aus stdarg.h zugegriffen werden. Auf den Umgang mit variablen Argumentlisten wird in diesem Skript nicht weiter eingegangen. Eine Funktion mit variabler Anzahl an Argumenten ist z. B. die **`printf()`** Funktion

```
int printf(char *, ...);   /* Prototyp von printf */
```

10.11 [C99] VLA's (Variable Length Arrays)

[C99] in C99 können auch *VLAs* (Variable Length Arrays, siehe Kap. 11.5) als Argumente an Funktionen übergeben werden, die Grösse des Arrays muss dabei als separates Argument übergeben werden, dieses muss dabei in der Parameterliste vor dem Array stehen:

```
int f(int n, int m, int Feld[n][m])
{
    /* Bei jeden Aufruf der Funktion f             */
    /* kann das Array eine andere Grösse           */
    /* haben, aber während einem Funktionsdurchlauf */
    /* bleibt die Grösse gleich                    */
}
```

11 Felder (Arrays)

Um mit grösseren Mengen von Daten effizient zu arbeiten, können *Felder* (*Arrays*) eingesetzt werden. Bei einem Feld wird eine Gruppe von Werten unter einem Namen zusammengefasst, die einzelnen Werte haben keinen eigenen Namen mehr, sondern nur noch eine Nummer (Den Index). Der *Index* beginnt bei 0, und endet bei Grösse - 1.

11.1 Definition eines Arrays

Bei der Definition eines Arrays muss die Grösse (Die Anzahl der Elemente) in eckigen Klammern angegeben werden:

```
int Werte[100];    /* Definiert ein Array mit 100 Werten */
```

Die Grösse eines Arrays muss zur Compile-Zeit bekannt sein, und muss deshalb ein konstanter Ausdruck sein. (Ausnahme: VLAs unter [C99], siehe Kap. 11.5)

11.2 Initialisierung eines Arrays

Ein Array kann bei der Definition bereits initialisiert werden, wobei die einzelnen *Initialisierer* konstante Werte sein müssen (Ausnahme [C99], siehe Kap. 11.3):

```
double Resultate[8] = {1.0, 2.0, 3.0, 4.0, 5,.0, 6.0, 7.0, 8.0};
```

Wenn weniger Werte angegeben werden, als das Array gross ist, werden die restlichen Arrayelemente auf 0 gesetzt. Wenn ein Array initialisiert wird, muss seine Grösse nicht angegeben werden. Ohne Grössenangabe wird die Grösse automatisch auf die Anzahl der Initialwerte gesetzt:

```
long int Zustand[] = {1, 1, 1, 1, 1, 1};   /* Array hat die Grösse 6 */
```

11.3 [C99] Nichtkonstante Initialisierer

[C99] in C99 dürfen die Initialisierer für nicht-globale und nicht-statische Arrays auch nichtkonstante Ausdrücke sein:

```
double x = 0.4, y = 3.14;
double Resultate[8] = {1.0, x, 3.0, y, 5,.0, x*y, 7.0, sin(x)};
```

11.4 Zugriff auf die Elemente einer Arrayvariable

Auf die einzelnen Werte eines Arrays wird mit den eckigen Klammern (*Indexoperator*) zugegriffen, in den eckigen Klammern steht dabei die Nummer des gewünschten Elementes:

```
Werte[4] = 17;
Werte[9] = Werte[99] - Werte[0];
```

Bei einem Array mit der Grosse N läuft der Indexbereich von 0 bis N-1, das erste Element hat den Index 0, und das letzte Element hat den Index N-1.

`int arr[10];`

0	1	2	3	4	5	6	7	8	9

Achtung, es findet keine Überprüfung auf Bereichsüberschreitung statt. Wenn man die Arraygrenzen verlässt, wird irgendwo auf den Speicher zugegriffen. Dabei können eigene Variablen oder Daten unbemerkt verändert werden, oder das Programm stürzt ab oder legt merkwürdiges Verhalten an den Tag.

11.5 [C99] Variable Length Arrays

Seit dem neuen Standard C99 gibt es auch Arrays mit variabler Länge, sogenannte *VLA* (Variable Length Array). VLAs können nur innerhalb von Blöcken definiert werden, sie dürfen weder global, noch statisch sein, auch dürfen sie nicht innerhalb von Strukturen definiert werden. VLAs können zudem als Argumente an Funktionen übergeben werden. VLAs können wie gewöhnliche Arrays beliebig viele Dimensionen beinhalten. Die Grösse des Arrays bleibt während seiner Lebensdauer konstant, sie wird beim Erzeugen des Arrays festgelegt.

Definition eines VLA:

```
int f(int n) {
   double Flexibel[n];  /* Bei jeden Aufruf der Funktion f kann das */
                        /* Array eine andere Grösse haben, aber     */
                        /* während einem Funktionsdurchlauf bleibt  */
                        /* die Grösse konstant                      */
}
```

VLAs können selbstverständlich auch mehrdimensional sein.

Wenn VLAs als Argumente an Funktionen übergeben werden, muss die Grösse des Arrays separat als weiteres Argument übergeben werden. Dabei muss die Grösse in der Parameterliste vor dem Array stehen. Bei mehrdimensionalen Arrays müssen entsprechend auch alle Grössen übergeben werden:

```
int f(int n, int m, int Feld[n][m]) {
   /* Bei jeden Aufruf der Funktion f kann das Array eine andere */
   /* Grösse haben, aber während einem Funktionsdurchlauf bleibt */
   /* die Grösse konstant                                        */
}
```

11.6 [C99] Direkte Initialisierer

In C99 können in der Initialisiererliste Arrayelemente direkt initialisiert werden, dazu muss das entsprechende Element mit einem Index angegeben werden:

```
int Vektor[10] = {1, 2, [4] = 6, 5, [9] = 1};
          /*ergibt 1 2 0 0 6 5 0 0 0 1 */

int Matrix[3][3] = {[0][0] = 1, [1][1] = 1, [2][2] = 1};
     /*ergibt 1 0 0
              0 1 0
              0 0 1 */
```

Nach einer expliziten Zuweisung wird mit den nachfolgenden Initialisiererwerten an dieser Stelle weitergefahren. Es können auch bereits initialisierte Werte wieder überschrieben werden, auch wenn das nicht besonders sinnvoll und verständlich ist:

```
int Vektor[10] = {1, 2, 3, 4, 5, 6, 7, 8, [4] = 9, 8, 7, 6, 5, 4};
/* Initialisiert den Vektor auf folgenden Inhalt: 1 2 3 4 9 8 7 6 5 4 */
```

11.7 Arrayzuweisungen

Zuweisungen von ganzen Arrays sind nicht möglich, sie müssen elementweise kopiert werden:

```
int a1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int b3[10];

b3 = a1;  /* Fehler, nicht erlaubt */

/* Loesung: Elementweises kopieren mit einer for-Schlaufe */
for (i = 0; i < 10; i++) {
   b3[i] = a1[i];
}
```

Wenn Arrays an Funktionen übergeben werden, wird nicht das ganze Feld kopiert, sondern nur ein Verweis auf das Feld übergeben. Die Funktion hat also Zugriff auf das originale Array.

11.8 [C99] Zusammengesetzte Array-Literale

In C99 können auch zusammengesetzte Literale (Konstanten) erzeugt werden, dabei wird in Klammern der Typ angegeben (Gleiche Syntax wie beim Cast-Operator), und anschliessend folgt die entsprechende *Initialisiererliste*:

```
Arraykonstante:
(int []){1, 2, 3}  /* Diese Konstante kann ueberall eingesetzt werden  */
                   /* wo sonst eine int [] Arrayvariable stehen könnte */

/* Beispiel Umwandlung von Tagnummer in Tagname mit Array */

/* Ohne zusammengesetztes Literal */
char *TagName;
int  Tag = 3;
char * Umwandlung[] = {"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};

TagName = Umwandlung[Tag]; /* Array 'Umwandlung' wird nur hier
gebraucht*/

/* Mit zusammengesetztem Literal */
char *TagName;
int  Tag = 3;
                    /* 'Umwandlung' ersetzt durch ein Literal */
TagName = ((char *[]){"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"})[Tag];
```

Die Initialisiererliste folgt dabei den Konventionen der Initialisiererliste gewöhnlicher Arrays. *Arrayliterale* können überall verwendet werden, wo auch ein gewöhnliches Array benutzt werden kann, sie sind eigentlich namenlose Variablen, welche denselben Gültigkeitsbereich wie eine an dieser Stelle definierte Variable haben.

Arrayliterale können insbesondere an Funktionen übergeben und an Pointer zugewiesen werden.

11.9 Mehrdimensionale Arrays

Arrays können mehrdimensional sein, pro Dimension ist einfach ein Klammernpaar erforderlich, wobei der hinterste Index die innerste Struktur darstellt. Bei der Initialisierung werden die Daten mit geschweiften Klammern gruppiert, unvollständige oder fehlende Werte werden auf 0 gesetzt.

```
int a1[2][3] = {{0, 1, 2},  {3, 4, 5}};
int b3[20][10][5];

b3[12][9][3] = a1[1][2];  /* Zugriff auf mehrdimensionale Arrays */
```

11.10 Arrays und Funktionen

Arrays können an Funktionen übergeben werden, aber nicht von Funktionen zurückgeliefert werden (Kein return von Arrays). Arrays werden dabei immer By Reference übergeben, d.h. es wird nie eine Lokalkopie erzeugt, sondern nur eine Referenz (Adresse, Pointer) übergeben. Innerhalb der Funktion wird also mit den Originalwerten gearbeitet.

```
int Summe(int a1[10])
{
   int i;
   int Summe = 0;

   for(i = 0; i < 10; i++) {    /* Summe ueber Array bilden */
      Summe += a1[i];
   }
   return Summe;
}

void Modify(int a1[10])
{
   a1[3] = 17;    /* Originalarray von Aufrufer wird verändert */
}
```

Die Grössenangabe ist für die erste Dimension optional, und wird auch nicht überprüft, d.h. die Grösse des übergebenen Arrays muss nicht mit der Grösse in der Funktionsdefinition übereinstimmen.

```
int b1[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int b2[3]  = {1, 2, 3};
int b3[15] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

int c;

c = Summe(b1);    /* Kein Problem */

c = Summe(b2);    /* Wird fehlerfrei Compiliert, duerfte aber Abstuerzen */

c = Summe(b3);    /* Wird fehlerfrei Compiliert, aber nur die ersten 10  */
                  /* Elemente werden Summiert                           */
```

Die obige Funktionsdefinition darf auch wie folgt geschrieben werden

```
void Modify(int a1[])   /* Arraygroesse nicht festgelegt */
{
   a1[3] = 17;          /* Originalarray von Aufrufer wird verändert */
}
```

In diesem Fall übergibt man aber besser auch gleich noch die effektive Grösse des Arrays als weiteren Parameter, und benutzt ihn innerhalb der Funktion, z.b. als Schlaufenendwert.

```
void Modify(int a1[], int Groesse);
```

Bei mehrdimensionalen Arrays ist nur die erste Dimensionsangabe optional, die restlichen Dimensionen müssen angegeben werden, und müssen auch beim Aufruf übereinstimmen. (Man kann sich mehrdimensionale Arrays einfach als ein Array von Arrays mit einer Dimension weniger vorstellen, also ein Array welches aus Arrays besteht)

```
void DoSomething(int a1[][10][5][3], int Groesse); /* Erste Dimension optional*/
```

12 Strings

12.1 Grundsätzliches

Strings oder *Zeichenketten* werden in C nicht direkt unterstützt. Zur Verarbeitung von Strings müssen Bibliotheksfunktionen aus der Standardlibrary (string.h) oder selbstgeschriebener Code herangezogen werden.
Zeichenketten werden in C als Buchstabenfelder (**`char`**-Arrays) repräsentiert. Das Ende einer Zeichenkette wird durch den ASCII-Wert 0 ('\0') kenntlich gemacht.
Die einzige direkte Unterstützung für Strings in C sind die *Stringliterale* (Beispiel: "Hans") sowie die Arrayinitialisierung mit Strings:

```
char Text[] = "Mueller";
```

Bei Stringliteralen und Arrayinitialisierungen wird automatisch das Endekennzeichen '\0' am Ende der Zeichenkette angehängt. Die Variable Text würde im Speicher also so aussehen:

'M'	'u'	'e'	'l'	'l'	'e'	'r'	'\0'

Und somit 8 Byte Speicherplatz belegen.
Stringliterale sind konstante Arrays, deren Inhalt zwar verändert werden kann, aber nicht verändert werden sollte. Wenn man den Inhalt von Zeichenkonstanten ändert, entstehen unportable Programme.
Alle Bibliotheksfunktionen, die mit Strings arbeiten, verlangen als Argumente die Anfangsadresse(n) (Zeiger, Arraynamen) der betroffenen Strings.

Der Typ **`char *`** welcher oft für Strings benutzt wird, ist eigentlich ein Zeiger auf einen Buchstaben, er enthält also nur die Adresse (Den Platz im Speicher) an welcher des erste Zeichen des Strings steht. Deshalb ist auch das Zeichen '\0' am Ende erforderlich, weil die Funktionen sonst das Ende des Textes nicht erkennen könnten.

12.2 Ein-/Ausgabe von Strings:

Mit den Funktionen **`gets()`** und **`puts()`** können ganze Zeilen eingelesen und ausgegeben werden. Mit **`scanf("%s", a)`** können nur einzelne Worte (Durch Whitespaces getrennte Zeichenfolgen) eingelesen werden.

```
char Name[] = "Mueller";
scanf("%s", Name); /* Achtung scanf liest nur bis zum ersten whitespace  */
                   /* (leerzeichen, Tabulator, Zeilenwechsel)           */
                   /* Achtung, hier kommt als grosse Ausnahme kein & vor */
                   /* Name, da der Arrayname selbst schon auf die Adresse*/
                   /* des ersten Elementes verweist                     */
printf("Der Name ist %s\n", Name);
/* ganze Zeile einlesen */
gets(Name);
/* ganze Zeile ausgeben */
puts(Name);
```

12.3 Weitere Stringfunktionen

Strings können nicht direkt miteinander verglichen werden, mit einem direkten == wie hier

```
char Text1[] = "Hans";
char Text2[] = "Fritz";

if (Text1 == Text2) {}  /* Vergleicht Adressen, nicht Inhalte !! */
```

werden nur die Speicheradressen der Strings, nicht aber deren Inhalt verglichen. Um Strings zu vergleichen muss z.B. die Funktion **`strcmp()`** verwendet werden, oder der String von Hand Zeichen für Zeichen verglichen werden. Wenn der Vergleich auf Grösser oder Kleiner lexikalisch korrekt sein soll (mit Umlauten) kann **`strcmp`** nicht verwendet werden. **`strcmp`** vergleicht nur nach ASCII-Code. **`strcmp()`** liefert 0 zurück, wenn beide Strings gleich sind, einen Wert kleiner 0 wenn der erste kleiner ist und sonst einen Wert grösser 0.

Um die Stringfunktionen verwenden zu können muss die Datei string.h eingebunden werden (#include <string.h>), sie enthält unter anderem folgende Stringfunktionen:

```
char s1[20] = "Hans";
char s2[20] = "Fritz";
```

`strlen(s1)`	liefert die Länge eines Strings zurück (Ohne der abschliessenden 0). **Achtung**, bestimmt nicht den effektiv vorhandenen Platz, sondern nur die Länge des Textes, also hier 4 (Den belegten Platz).
`strcpy(s1, s2)`	kopiert den String s2 nach s1 (Inklusive der abschliessenden 0).
`strcat(s1, s2)`	hängt den String s2 an String s1 an.
`strcmp(s1, s2)`	vergleicht zwei Strings miteinander (**Achtung**: '0' < '9' < 'A' < 'Z' < 'a' < 'z').

Bei den kopierenden Funktionen (**`strcpy`**, **`strcat`**) findet keine Prüfung auf ausreichend Platz im Ziel statt, es wird einfach geschrieben. Der Programmierer muss selbst für ausreichend Platz (= Genügend grosses Array) im Ziel sorgen, sonst ist das Verhalten des Programms unvorhersehbar (Es kann funktionieren, gelegentliche Fehlfunktionen aufweisen, hin und wieder Abstürzen oder auch immer Abstürzen)

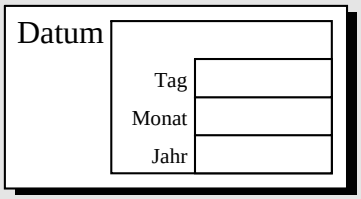
```
char Text1[100] = "Welt";
char Text2[100];
strcpy(Text2, Text1);           /* Kopiert Text1 nach Text2           */
strcpy(Text1, "Hallo");         /* Kopiert "Hallo" nach Text1         */
strcat(Text1, " ");             /* Haengt ein Leerzeichen an Text1 an */
strcat(Text1, Text2);           /* Haengt Text2 an Text1 an           */

/* Vergleicht Text1 mit "Hallo Welt" */
if (strcmp(Text1, "Hallo Welt")) {
   printf("Alles OK, Laenge von \'%s\' ist %d\n", Text1, strlen(Text1);
} else {
   printf("Error");
}
```

13 Strukturen

Mit *Strukturen* können mehrere verschiedene Variablen zu einer Einheit zusammengefasst werden. So können zusammengehörende Variablen auch zusammen behandelt werden. Unterschiedliche Strukturen gelten jeweils als neuer, eigenständiger Datentyp.
Eine *Strukturdeklaration* wird mit dem Schlüsselwort **`struct`** eingeleitet, gefolgt von einem fakultativen *Strukturnamen* und einem anschliessenden Block mit einer Liste von Variablendeklarationen. Der Strukturname kann bei späterer Benutzung anstelle der gesamten Deklaration verwendet werden.

13.1 Deklarieren einer Struktur.

```
/* Ein Datum besteht aus Tag, Monat und Jahr */

struct Datum {
   char Tag;
   int Monat;
   int Jahr;
};
```



[C99] In C99 darf des letzte Element einer Struktur ein Array mit einer nicht bekannten Grösse sein. Dies macht aber nur im Zusammenhang mit dynamischer Speicherverwaltung Sinn, wenn der Benutzer selbst für die Zuordnung von Speicher verantwortlich ist.

```
struct Telegram {       /* Struktur fuer Datenkommunikation */
   int Empfaenger;
   int Absender;
   int Laenge;
   int Typ;
   int Command;
   char Data[];         /* Flexible Datenlaenge */
};
```

13.2 Definition einer Strukturvariablen

```
struct Datum Geburtstag; /* Eine Datumsvariable definieren (erzeugen) */
```

13.3 Initialisierung von Strukturvariablen

```
struct Datum Geburtstag = {17, 4, 1976};
/*Variable wird definiert und Tag auf 17, Monat auf 4 und Jahr auf 1976
gesetzt*/
```

Die *Initialisierung* einer Struktur erfolgt in der selben Reihenfolge wie die Auflistung der Felder in der Definition der Struktur. Die Initialwerte müssen in geschweiften Klammern aufgelistet werden. Wenn weniger Werte als Felder in der Initialisiererliste stehen, werden die restlichen Felder mit 0 gefüllt.

13.4 [C99] Nichtkonstante Initialisierer

[C99] In C99 dürfen die Initialisierer für nicht globale und nicht statische Struktur-Variablen auch nichtkonstante Ausdrücke sein:

```
long did = GetDateInDays();                    /* did = DateInDays  */
struct Datum Demo = {did%365%31, did%365/31, did/365};
```

13.5 [C99] Direkte Initialisierer

In C99 können in der Initialisiererliste Strukturelemente direkt initialisiert werden, dazu muss das entsprechende Element mit seinem Namen und einem vorangestellten Punkt angegeben werden:

```
struct Datum Geburtstag = { .Jahr = 1944, .Monat = 11, .Tag = 2 };
```

Auch diese Initialisierung darf unvollständig sein, nicht explizit initialisierte Werte werden auf 0 gesetzt.

Die direkte Initialisierung kann mit der gewöhnlichen Initialisierung beliebig gemischt werden. (Allerdings kann die Übersicht darunter leiden). Nach einer expliziten Zuweisung wird mit den nachfolgenden Initialisiererwerten nach der soeben initialisierten Variablen weitergefahren. Es können auch bereits initialisierte Werte wieder überschrieben werden, auch wenn das nicht besonders sinnvoll und verständlich ist:

```
/* Etwas Komplizierter (Kombination v. Array u. Struktur initialisieren
*/
struct Eintrag {
   char Name[30];
   struct Datum Termine[10];
}

struct Eintrag Liste[10] = {
                  [2].Name = "Hans",
                  [2].Termine[3].Tag = 21,
                  [2].Termine[3].Monat = 3,
                  [2].Termine[3].Jahr = 1976,
                  [0].Name = Fritz,
                  [0].Termine[0] = {24, 5, 1990},
                  [4] = {"Urs", 17, 12, 1982, 13, 9, 1987},
                  "Sepp", 18, 2, 1999, 17, 4, 2004
};
```

13.6 Die Definition kann mit der Deklaration zusammengefasst werden

```
struct Datum {
   char Tag;
   int Monat;
   int Jahr;
} Ostern, Weihnacht = {24, 12, 0}; /* Definition an Deklaration angehaengt    */
                                   /* Definiert die 2 Variablen Ostern und    */
                                   /* Weihnacht, Weihnacht wird initialisiert */
```

13.7 Zugriff auf die Elemente einer Strukturvariablen

Ein Feld einer Struktur wird mit **.** (Punkt, = Elementauswahloperator) selektiert.

Strukturvariablenname.Feldname

```
struct Datum Geburtstag = {24, 12, 0};

...
if (Geburtstag.Jahr == 1976) {
   Geburtstag.Monat = 10;
}
```

Die Benutzung einer Struktur kann mit **`typedef`** vereinfacht werden. (Der Strukturname kann [muss aber nicht] weggelassen werden)

```
typedef struct {
   char Tag;
   int Monat;
   int Jahr;
} DatumType;


DatumType MeinGeburtstag = {9, 9, 1999};
DatumType DeinGeburtstag;
```

Strukturen können einander zugewiesen, als Argumente an Funktionen übergeben und als Rückgabewerte von Funktionen zurückgeliefert werden. Arithmetische, logische und relationale (Vergleichs) Operatoren können nicht auf Strukturen angewendet werden.

```
void PrintDate(DatumType Date)   /* Struktur als Argument */
{
   printf("%d.%d.%d", Date.Tag, Date.Monat, Date.Jahr);
}

DatumType GetDate(void)
{
   DatumType Result = {1, 1, 1911};
   return Result;            /* Struktur als Rückgabewert */
}

DeinGeburtstag = Ostern;     /* Strukturzuweisung */
PrintDate(Weihnacht);        /* Funktionsaufruf mit Struktur als Argument */
MeinGeburtstag = GetDate();  /* Funktionsaufruf mit Struktur als Rückgabewert */
```

13.8 Verschachtelung

Strukturen und Arrays können beliebig kombiniert und *verschachtelt* werden

```
struct Person {
   char Name[20];
   int Gewicht;
   struct Datum Geburtsdatum;
} Student = {"Hans", 70, {1, 2, 1975}};

/* Array mit 20 Personen */
struct Person Klasse[20];

/* Zugriff auf PersonenArray */
Klasse[12].Gewicht = 78;
Klasse[12].Geburtsdatum.Tag = 1;
Klasse[12].Geburtsdatum.Monat = 12;
Klasse[12].Geburtsdatum.Jahr = 1999;
Klasse[12].Name[0] = 'U';
Klasse[12].Name[1] = 'R';
Klasse[12].Name[2] = 'S';
Klasse[12].Name[3] = '\0';   /* Abschliessende Null, nicht vergessen !! */

printf("Gewicht von %s ist %d\n", Klasse[12].Name, Klasse[12].Gewicht);
```

13.9 [C99] Zusammengesetzte Literale

In C99 können auch *zusammengesetzte Literale* (Konstanten) erzeugt werden, dabei wird in Klammern der Typ angegeben (Gleiche Syntax wie beim Cast-Operator), und anschliessend folgt die entsprechende Initialisiererliste:

```
Strukturkonstante:
(struct Datum){17, 2, 2003} /* Diese Konstante kann ueberall eingesetzt  */
                            /* werden wo sonst eine Datumstruktur stehen */
                            /* könnte                                    */

/* Beispiel Zuweisung an Struktur ohne struct Literal (vor C99) */

Klasse[12].Gewicht = 78;
Klasse[12].Geburtsdatum.Tag = 1;
Klasse[12].Geburtsdatum.Monat = 12;
Klasse[12].Geburtsdatum.Jahr = 1999;
Klasse[12].Name[0] = 'U';
Klasse[12].Name[1] = 'R';
Klasse[12].Name[2] = 'S';
Klasse[12].Name[3] = '\0';

/* Beispiel Zuweisung an Struktur mit struct Literal (C99) */


Klasse[12] = (struct Person) {"Urs", 78, {27, 5, 1978}};
/* oder */
Klasse[12] = (struct Person) {"Urs", 78, 27, 5, 1978};
/* oder */
Klasse[12] = (struct Person) {.Name = "Urs",
                              .Gewicht = 78,
                              .Geburtsdatum = {27, 5, 1978}};
/* oder */
Klasse[12] = (struct Person) {.Name = "Urs",
                              .Gewicht = 78,
                              .Geburtsdatum.Tag = 27,
                              .Geburtsdatum.Monat = 5,
                              .Geburtsdatum.Jahr = 1978}};
```

Die Initialisiererliste folgt dabei den Konventionen der Initialisiererliste gewöhnlicher Strukturen.

Strukturliterale können überall verwendet werden, wo auch eine gewöhnliche Struktur benutzt werden kann, sie sind eigentlich namenlose Variablen, welche denselben Gültigkeitsbereich wie eine an dieser Stelle definierte Variable haben.

Strukturliterale können insbesondere an Funktionen übergeben, an Strukturvariablen zugewiesen und ihre Adressen können an Pointer zugewiesen werden.

```
/* Strukturliteral als Argument fuer eine Funktion */
AddEntry( (struct Person) {"Urs", 78, {27, 5, 1978}});

/* Adresse eines Strukturliterals einem Pointer zuweisen */
struct Person *p;
p = &((struct Person) {"Urs", 78, {27, 5, 1978}});
```

13.10 Aufgaben

Aufgabe 13.1

Schreiben Sie ein Programm, das zwei Uhrzeiten (hh, mm, ss) einliest, die Differenz (hh, mm, ss) zwischen den beiden Zeiten bestimmt und ausgibt. Die Berechnung der Differenz soll in einer Funktion erfolgen. Für die Uhrzeit soll eine Struktur verwendet werden.

Aufgabe 13.2

Schreiben Sie ein Programm, das Studenten verwaltet. Von einem Studenten sollen Name, Vorname und Durchschnittsnote abgelegt werden. Dem Programm sollen bis zu 30 Studenten eingegeben werden können. Anschliessend soll ein Notendurchschnitt eingegeben werden können, worauf die Namen aller Studenten welche über diesem Schnitt liegen ausgegeben werden. Die Ausgabe soll wiederholt getätigt werden können, ohne dass das Programm beendet werden muss. Teilen Sie das Programm in einzelne Funktionen auf (z.B. Ausgabe, Eingabe, Menu, Suchen,...). Kommentar und Struktogramme gehören ebenfalls zur Lösung der Aufgabe.

Erweiterung 1: Fügen Sie dem Programm ein kleines Menu hinzu, bei dem der Benutzer wählen kann, ob er Daten eingeben, die Daten löschen oder Daten ausgeben will.

Erweiterung 2: Erweitern Sie das Programm um die Möglichkeit, die Einträge nach Namen oder nach Notendurchschnitt zu sortieren.

Erweiterung 3: Verwenden Sie Zeiger um Einträge im Array zu suchen.

Erweiterung 4 (Später) Erweitern Sie das Programm um Funktionen zum Laden und Abspeichern von Studentendaten (Load und Save). Es sollen jeweils alle Daten in eine Datei abgespeichert werden und zu einem späteren Zeitpunkt wieder eingelesen werden können.

Erweiterung 5 (Später) Erweitern Sie das Programm um dynamische Funktionen. Die Studenten sollen nicht mehr in einem Array, sondern in einer Liste verwaltet werden. Die Listenelemente sollen nach bedarf Alloziert und wieder freigegeben werden. (Ein Modul mit Listenfunktionen wird zur Verfügung gestellt).

14 Unions

Eine *Union* ist fast dasselbe wie eine Struktur, mit dem Unterschied dass alle Datenfelder denselben Platz im Speicher belegen (Sich also am selben Ort im Speicher aufhalten). In einer Union kann somit immer nur eine der enthaltenen Variablen verwendet werden. Die Union braucht immer nur soviel Platz wie die grösste der in ihr enthaltenen Variablen. Eine Union kann verwendet werden, wenn zur selben Zeit immer nur eine der möglichen Variablen benutzt wird, damit kann Speicherplatz gespart werden.

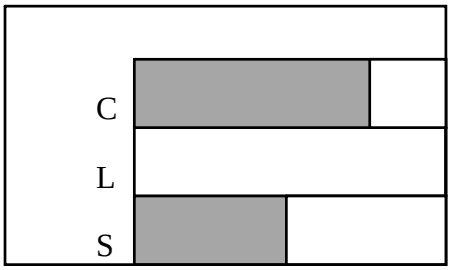
```
union Convert {
   char Bytes[4];        /* Ein Array von 4 Bytes
*/
   long Value;           /* und ein long teilen sich denselben Speicher
*/
};

union Convert x;
long l = 0x12345678;
x.Value = l;             /* Den Longwert in den gemeinsamen Speicher schreiben
*/

printf("Byte 0: %d\n", x.Bytes[0]);   /* Und Byteweise wieder auslesen... */
printf("Byte 1: %d\n", x.Bytes[1]);
printf("Byte 2: %d\n", x.Bytes[2]);
printf("Byte 3: %d\n", x.Bytes[3]);
```

Dieses Beispiel zerlegt einen Long-Wert in seine einzelnen Bytes. **Achtung**, dieser Code ist nicht portabel, da jeder Prozessor/Compiler die Bytes von Variablen in unterschiedlicher Reihenfolge im Speicher ablegt und ein **`long`** nicht bei jedem System 4 Bytes (32 Bit) belegt.

Der Unterschied zwischen Struktur und Union ist nachfolgend zu sehen:

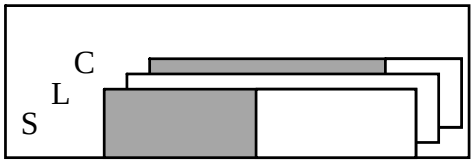
Struktur

```
struct demo {
   char C;
   long L;
   short S;
};
```



Union

```
union demo {
   char C;
   long L;
   short S;
};
```



Die Union legt die Elemente übereinander im selben Speicherbereich ab, und benötigt deshalb weniger Speicherplatz, aber es kann dafür immer nur eines der Elemente unbeeinflusst verwendet werden.

15 Bitfelder

Bei Strukturen kann für **int** Member angegeben werden, wieviele Bit's sie belegen sollen. (Aber pro Feld nicht mehr als es Bits in einem **int** hat). Die Bits werden fortlaufend von einem int vergeben, und sobald es für ein neues Feld keinen Platz im angebrochenen int mehr hat, wird das ganze Feld in einem neuen int abgelegt und von diesem int mit der Bitzuteilung weitergefahren. Bitfelder können nicht über int-Grenzen hinweggehen.

Mit Bitfeldern kann Speicherplatz gespart werden, oder es können Register von Peripheriebausteinen abgebildet werden. **Achtung**, bei Bitfeldern ist fast alles herstellerabhängig, Programme die von der tatsächlichen Bitposition im Speicher abhängen, sind mit Sicherheit **nicht portabel**!

```
struct Date {
   int Day   : 5;     /* Fuer den Tag brauchen wir nur 5 Bit (0...31)        */
   int Month : 4;     /* Fuer den Monat reichen 4 Bit (0...15)               */
   int Year  : 7;     /* Fuer 2 stellige Jahrzahlen genuegen 7 Bit (0...128) */
} Heute;

Heute.Day   = 24;
Heute.Month = 12;
Heute.Year  = 99;
```

Diese Struktur Date für Daten braucht somit nur 16 Bit oder 2 Bytes Speicherplatz.

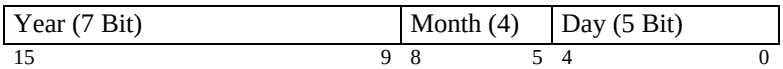
Auch Flags (Einzelne Bits) können so Platzsparend abgelegt werden:

```
struct FileInfo {
   int Readonly   : 1;
   int Hidden     : 1;
   int System     : 1;
   int Archived   : 1;
   int Link       : 1;
};
```

Diese Flags benötigen nur 5 Bits, können also problemlos in einem Byte abgelegt werden.

Man muss sich aber im klaren sein, dass man so zwar Speicherplatz spart, aber der Rechenaufwand grösser wird. Schliesslich muss der Compiler auf die einzelnen Bits mit Schieben, And- und Or-Verknüpfungen zugreifen.

Beispiel Datestruktur:




```
/* Die Anweisung */
Heute.Month = a;
/* wird vom Compiler übersetzt zu */
Heute = (Heute & 0xFE1F) | ((a & 0x000F) << 5);  /* 1111111000011111 = 0xFE1F */

/* Und die Anweisung */
b = Heute.Month;
/* wird vom Compiler übersetzt zu */
b = (Heute >> 5) & 0x000f;

/* Und falls das Vorzeichen erhalten bleiben soll, zu */
b = (Heute << 7) >> 11;
```

16 Enums

Mit dem *Aufzählungstyp* (*Enum*) können *Konstanten* definiert werden. Ein **enum** ist ein **int**, der vordefinierte Konstanten aufnehmen kann. Obwohl ein **enum** eigentlich ein eigener Datentyp ist, können Enums und **int** gemischt werden.

Wenn *Enumkonstanten* bei der Definiton kein Wert zugewiesen wird, werden sie fortlaufend numeriert, beginnend bei 0 oder dem Wert des Vorgängers plus 1.

Beispiel, Enum für die Wochentage:

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

enum Tag Heute, Morgen;
Heute = Mi;
if (Heute == Do) {
   Morgen = Fr;
}

/* Leider auch moeglich */
int Zahl;
enum Tag Gestern;
Zahl = Mo;           /* Enumkonstante an integer zuweisen */
Gestern = 77;        /* Integer an Enum zuweisen          */
Gestern = Zahl / 5;  /* Integer an Enum zuweisen          */
Gestern = Gestern + Sa / Mo % 17;  /* Kompleter Mix       */
```

Beispiel, Enum für Fehlercodes:

```
enum Fehlercode {
   NoError = 0,                /* Hat den Wert 0   */
   SyntaxError,                /* Hat den Wert 1   */
   RuntimeError,               /* Hat den Wert 2   */
   FileError = 10,             /* Hat den Wert 10  */
   ReadError,                  /* Hat den Wert 11  */
   WriteError = FileError + 5; /* Hat den Wert 15  */
   LastError  = 999;           /* Hat den Wert 999 */
};
```

Die Namen der Enumkonstanten gelten über alle Enums. Wenn ein Name bereits in einem Enum vergeben wurde, kann er in einem anderen Enum nicht nochmals verwendet werden.

```
     /* Fehler, Do und Mi bereits in Enum Tag verwendet!!! */
enum Tonleiter {Do, Re, Mi, Fa, So, La, Ti};
```

Enumkonstanten können die Lesbarkeit eines Programmes stark erhöhen und auch helfen, Fehler zu vermeiden.

Bei der Ausgabe von Enums mit printf() wird jedoch der numerische Wert, und nicht der Name ausgegeben. Wenn man den Namen ausgeben möchte, muss man eine eigene Ausgabefunktion schreiben. (Mit einem Array oder einem switch).

Beispiel:

```
enum Tag {Mo, Di, Mi, Do, Fr, Sa, So};
char *Translate[] = {"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};

enum Tag Heute = Di;
printf("Heute ist %s !", Translate[Heute]); /* Annahme: keine Ungültigen Werte*/
```

17 Zeiger (Pointer)

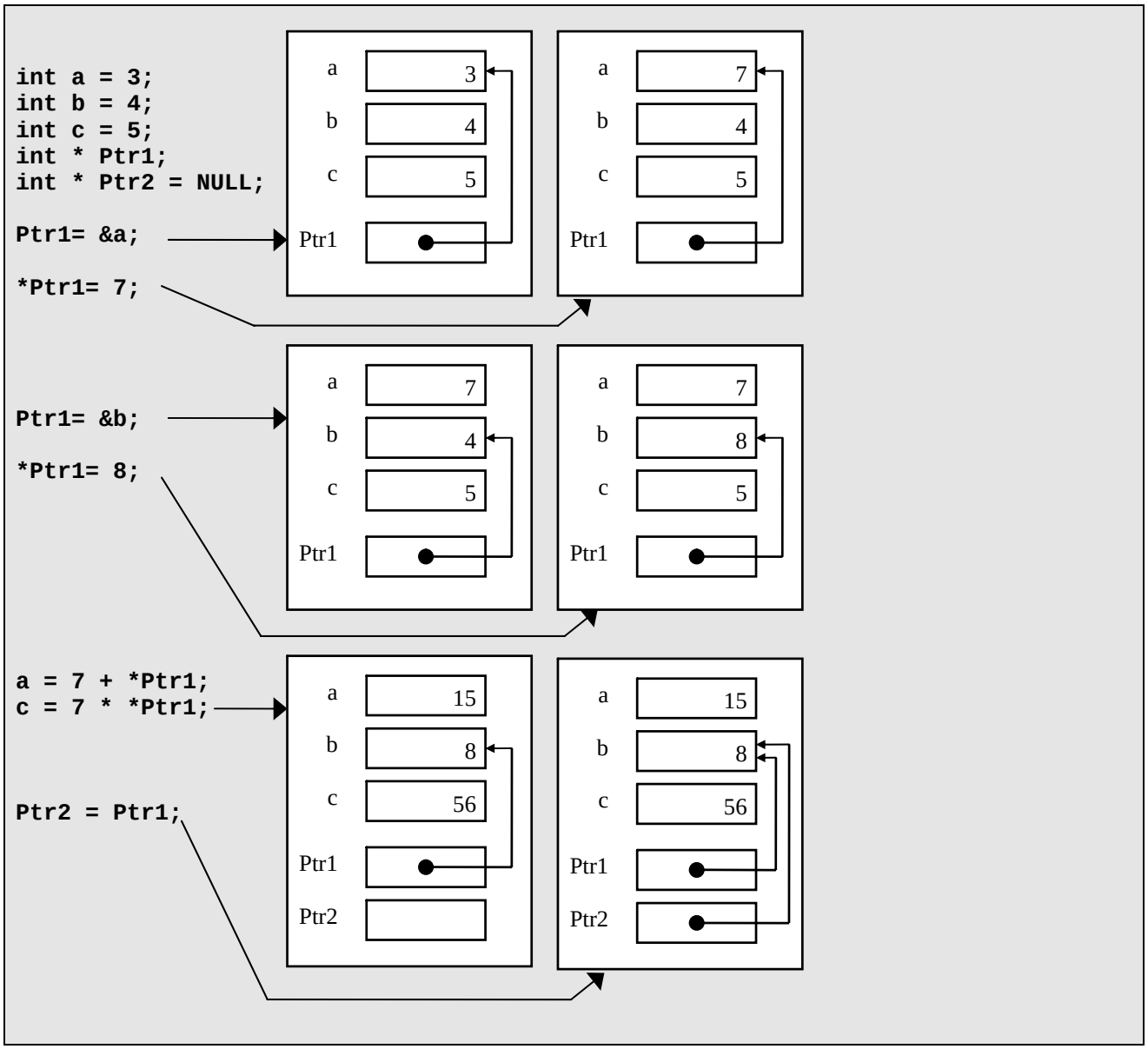
Zeiger sind in C ein sehr wichtiges Sprachmittel, es gibt kaum ein Programm das ohne Zeiger auskommt. Um richtig C Programmieren zu können, muss man Zeiger verstehen.

Im Speicher eines jeden Computers sind die Speicherplätze (Bytes) durchnumeriert. Jede Variable hat somit eine eindeutige *Adresse* (Platz im Speicher). Als Adresse von mehrbytigen Variablen wird üblicherweise die Adresse des ersten von ihr belegten Speicherplatzes verwendet.

Zeiger sind nun spezielle Variablen, welche die Adresse (Position) einer anderen Variablen aufnehmen können. Über die *Zeigervariable* kann nun auch auf die Originalvariable zugegriffen werden. Bei der Deklaration einer Zeigervariablen muss man angeben, auf welchen Variablentyp sie zeigt. (Es sind auch Zeiger auf Zeiger usw. möglich).

Mit dem * Operator wird einerseits eine Zeigervariable deklariert und definiert, andererseits kann damit auf die von einem Zeiger adressierte Variable zugegriffen werden (Man sagt dazu: 'der Zeiger wird *dereferenziert*').

Mit dem & Operator kann die Adresse einer beliebigen Variable ermittelt werden.



Achtung, wenn nicht initialisierte Zeiger verwendet werden, verhält sich ein Programm unvorhersehbar, im günstigsten Fall stürzt es ab, sonst werden irgendwo zufällig Variablen verändert. Bevor ein Zeiger benutzt wird, muss ihm immer eine gültige Adresse zugewiesen werden. Zeiger die nirgendwohin zeigen, sollte man den Wert **`NULL`** zuweisen (**`NULL`** ist eine in stdio.h vordefinierte Konstante). Vor der Benutzung eines Zeigers sollte man ihn entsprechend sicherheitshalber auf den Wert **`NULL`** testen, und im Falle von **`NULL`** nicht verwenden und ev. eine Fehlermeldung ausgeben.

Achtung, auf einfache Literale kann der Adressoperator nicht angewendet werden.

```
int* p1 = &5; /* Nicht erlaubt, 5 hat keine Adresse */
```

Achtung, bei der Deklaration/Definiton von Pointer gehört der * zum Pointer, nicht zum Datentyp!

```
int* p1, p2, p3;  /* Definiert nicht 3 Pointer, sondern 1 Pointer p1, sowie
                     2 int Variablen p2 und p3 */
int *p1, *p2, *p3; /* Definiert 3 Pointer */
```

Achtung, *Pointervariablen* können einander nur zugewiesen werden, wenn sie vom selben Typ sind, also auf denselben Typ zeigen. Ausgenommen davon sind **`void`**-Pointer, diese können an alle anderen Zeiger zugewiesen, und von allen Zeigern zugewiesen werden).

```
int i = 3;
float f = 3.14;
int *ip1, *ip2 = &i;
float *fp1, *fp2 = &f;
void *Universalpointer;

ip1 = ip2;  /* OK */
fp1 = fp2;  /* OK */
fp1 = ip2;  /* Error, benoetigt cast : fp1 = (float *)ip2; */

/* Via void-pointer geht's auch: */
Universalpointer = ip1;   /* int-Pointer an void Pointer, kein Problem   */
fp1 = Universalpointer;   /* void Pointer an float-Pointer, kein Problem */
```

17.1 Call By Reference

Mit Pointer ist nun auch *Call by Reference* möglich, d.h. Funktionen können die übergebenen Argumente verändern, sie erhalten nicht nur Kopien. Dazu müssen einfach die Adressen der Argumente anstelle der Argumente selbst übergeben werden. (Und weil Arraynamen eigentlich auch Zeiger sind, werden Arrays grundsätzlich immer by Reference übergeben)

```
void Modify(int *p, int c)
{
   c += 2;
   *p = *p + c;     /* Veraendert Wert, auf den p zeigt */
}

int main(int agrc, char *argv[])
{
   int x = 3;
   int y = 7;
   printf(" x=%d, y=%d\n", x, y);
   Modify(&x, y);                       /* x wird veraendert, y nicht */
   printf(" x=%d, y=%d\n", x, y);
   Modify(&y, 5);                       /* y wird veraendert */
   printf(" x=%d, y=%d\n", x, y);
   return 0;
}
```

17.2 Pointerarithmetik

Mit Zeiger kann auch gerechnet werden. Zeiger können inkrementiert und dekrementiert werden, dabei werden sie jeweils um die Grösse des Datentyps auf den sie zeigen, verändert. Damit gelangt man zur nächsten Variablen desselben Typs, vorausgesetzt dass die Variablen genau hintereinander im Speicher liegen. Dies ist zum Beispiel bei einem Array der Fall. Pointerarithmetik macht deshalb eigentlich nur bei Arrays Sinn.

Beispiel:

```
char Text[] = "Hallo Welt";

char *Ptr = &(Text[6]);    /* Laesst Zeiger auf 'W' zeigen */

putchar(*Ptr);             /* Gibt 'W' aus */
Ptr++;                     /* Setzt Zeiger um eine Position vorwaerts ('e') */
putchar(*Ptr++);           /* Gibt 'e' aus und inkrementiert Zeiger danach  */
putchar(*Ptr++);           /* Gibt 'l' aus und inkrementiert Zeiger danach  */
putchar(*Ptr++);           /* Gibt 't' aus und inkrementiert Zeiger danach  */
```

Zu Pointer können Integer-Werte addiert oder subtrahiert werden. Dabei wird der Zeiger um die entsprechende Anzahl von Variablen vor- oder zurückgestellt.

Was wird in diesem Beispiel ausgegeben?

```
char Text[] = "0H1a2l3l4o5";
char *Ptr = &(Text[1]);
int Offset = 2;

putchar(*Ptr);
Ptr += 2;
putchar(*Ptr);
Ptr += Offset;
putchar(*Ptr);
Ptr += 2;
putchar(*Ptr);
Ptr += 2;
putchar(*Ptr);
```

Pointerarithmetik ist auch möglich, ohne den Pointer zu verändern:

```
char Text[] = "Hallo Welt";
char *Ptr = &(Text[4]);       /* Zeiger zeigt auf 'o' */
int Offset = 4

putchar( *(Ptr + 2) );        /* gibt 'W' aus, Zeiger zeigt weiterhin auf 'o' */
putchar( *(Ptr - 3) );        /* gibt 'a' aus, Zeiger zeigt weiterhin auf 'o' */
putchar( *(Ptr + Offset ) );  /* gibt 'l' aus, Zeiger zeigt weiterhin auf 'o' */
putchar( *Ptr );              /* gibt 'o' aus, Zeiger zeigt weiterhin auf 'o' */
```

Die Differenz zwischen zwei Pointer gleichen Typs, die auf Elemente innerhalb des gleichen Arrays zeigen ergibt die Anzahl von Elementen zwischen den beiden Zeigern. (Dass beide Zeiger dabei in dasselbe Array zeigen ist dazu Voraussetzung, ansonsten macht das Ergebnis keinen grossen Sinn). Es ist nicht möglich, Pointer verschiedenen Typs voneinander zu subtrahieren.

17.3 Pointer und eindimensionale Arrays

Eindimensionale Arrays und Pointer sind in C sehr nahe verwandt. Ein Arrayname ist in C nichts anderes als ein konstanter Zeiger auf den Anfang des Arrays. Der *Indexoperator* **[]** kann deshalb auch auf Pointer und der ***** auf Arraynamen angewendet werden:

Die Operation ***(Ptr + n)** ist äquivalent zu **Ptr[n]**. Zwischen den beiden Varianten kann frei gewählt werden. Für Arrays ist es aber üblich ausschliesslich **[]** zu verwenden, bei Zeigern wird hingegen oft auch **[]** anstelle von ***** verwendet, insbesondere wenn der Zeiger auf ein Array zeigt.

```
double Tabelle[] =
         {1, 1.2, 1.5, 1.8, 2.2, 2.7, 3.3, 3.9, 4.7, 5.6, 6.8, 8.1, 0};

double *Ptr = Tabelle + 3;      /* Zeigt nun auf 1.8 */

printf("%f", *(Tabelle + 4) );  /* Gibt 2.2 aus */

printf("%f", Ptr[2] );          /* Gibt 2.7 aus */
printf("%f", Ptr[-2] );         /* Gibt 1.2 aus (Negative Indices erlaubt !) */
printf("%f", *Ptr++ );          /* Gibt 1.8 aus, zeigt danach auf 2.2 */

/* Gibt den Inhalt des gesamten Arrays aus */
for (Ptr = Tabelle; *Ptr != 0.0; Ptr++) {
   printf("%f\n", *Ptr);
}

/* Gibt ebenfalls den Inhalt des gesamten Arrays aus (jeweils 2 mal) */
Ptr = Tabelle;
for (i = 0; i < 12; i++) {
   printf("%f\n", Tabelle[i]);
   printf("%f\n", Ptr[i]);
}
```

17.4 Pointer und mehrdimensionale Arrays

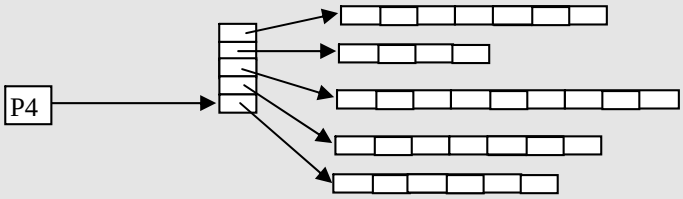
Bei mehrdimensionalen Arrays wird die Sache schon etwas komplizierter, int **p und int a[][3] sind nicht äquivalent zueinander!

Sondern der Pointer int (*p)[3] ist kompatibel zum Array int a[][3].

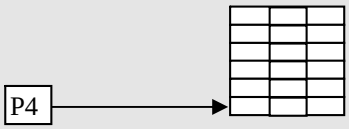
```
int a[7][3];

int **p1;        /* Achtung, Pointer auf Pointer auf int       */
int (*p2)[3];    /* Korrekt, Pointer auf Array mit 3 Elementen */
int (*p3)[7];    /* Korrekt, Pointer auf Array mit 7 Elementen */
int *p4[3];      /* Achtung, Array von drei Pointern, nicht Pointer auf Array */

p1 = a;          /* Nicht Korrekt */
p2 = a;          /* Korrekt       */
p3 = a;          /* Nicht Korrekt, Array hat nur 3 Spalten */
p4 = a;          /* Nicht Korrekt */

p2[5][1] = 5;    /* Zugriff auf Array via Pointer */

/* Auf alle oben definierten Pointer kann aber mit einer 2-Dimensionalen      */
/* Indizierung zugegriffen werden, aber die darunterliegenden Datenstrukturen */
/* sind Grundverschieden                                                      */
p1[3][2] = 7;    /* Korrekter Zugriff, aber nicht auf Array */
p2[3][2] = 7;    /* Korrekter Zugriff auf n*3 Array, passt zu a */
p3[3][2] = 7;    /* Korrekter Zugriff auf n*7 Array */
p4[2][3] = 7;    /* Korrekter Zugriff, aber nicht auf Array */
```

Die zugrunde liegenden Datenstrukturen sehen wie folgt aus:

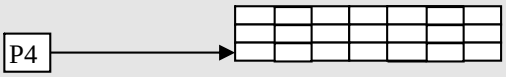
```
p1[3][2] = 7;    /* Korrekter Zugriff, aber nicht auf 2-Dimensionales Array */
```



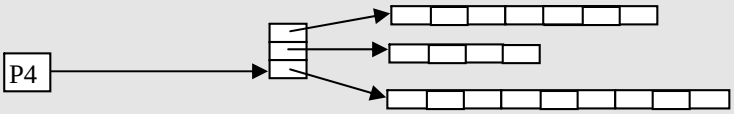

```
p2[3][2] = 7;    /* Korrekter Zugriff auf n*3 Array, passt zu a */
```




```
p3[3][2] = 7;    /* Korrekter Zugriff auf n*7 Array */
```




```
p4[2][5] = 7;    /* Korrekter Zugriff, aber nicht auf 2-Dimensionales Array */
```



17.5 Pointer und Strukturen

Zeiger können auch auf Strukturen zeigen, und Strukturen können ihrerseits auch Zeiger enthalten:

```
typedef struct Eintrag {
   int Wert;
   char Text[20];
   struct Eintrag *Next;  /* Hier muss der Strukturname verwendet werden, da */
} EintragType;            /*               der Typedef noch nicht fertig ist */


EintragType *Ptr;         /* Ab hier kann der Typedef verwendet werden       */

EintragType Demo = {42, "Hitchhiker", NULL};

Ptr = &Demo;
```

Der Zugriff auf ein Strukturelement via Pointer sieht eigentlich so aus:

```
(*Ptr).Wert = 77; /* Zuerst Pointer dereferenzieren und dann Element waehlen */
```

Weil das umständlich ist, gibt es eine intuitivere Kurzform:

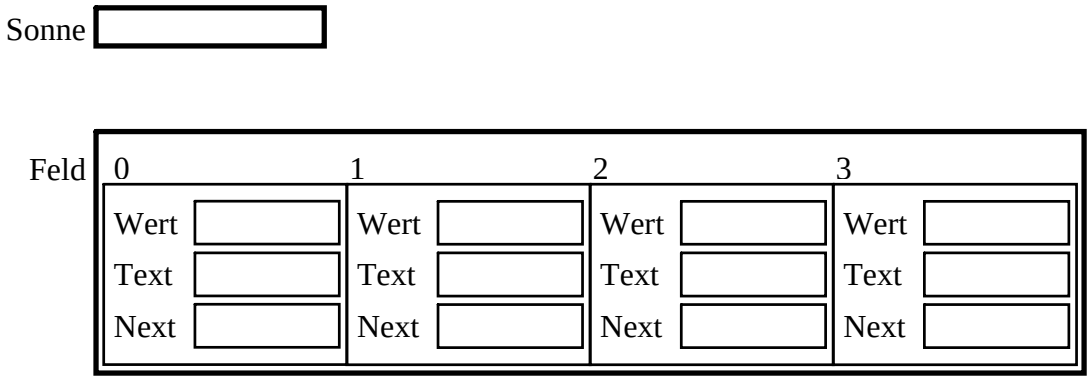
```
Ptr->Wert = 77;
```

Dies ist auch die normalerweise verwendete Form, die beiden Varianten sind aber gleichwertig..

Und nun ein etwas komplizierteres Beispiel:

```
typedef struct Eintrag {
   int Wert;
   char Text[20];
   struct Eintrag *Next;
} EintragType;

EintragType Feld[4] = {
                        {11, "Venus",   NULL},
                        {22, "Erde",    NULL},
                        {33, "Mars",    NULL},
                        {44, "Jupiter", NULL}
                      };

EintragType *Sonne;

Sonne = &(Feld[1]);

Feld[0].Next = &(Feld[1]);
Feld[1].Next = &(Feld[3]);
Feld[2].Next = &(Feld[0]);
Feld[3].Next = &(Feld[2]);
```

Zeichnen Sie die sich daraus ergebenden Daten und Zeiger in folgendem Diagramm ein:



Tragen Sie jetzt mit einer anderen Farbe ein, wie das Diagramm nach der Ausführung der folgenden Zeilen aussieht:

```
for (i = 0; i < 2; i++) {
   Sonne->Wert = i;
   Sonne = Sonne->Next;
}
```

17.6 Pointer und Funktionen

Zeiger können auch auf Funktionen zeigen, es ist also möglich, die Adresse einer Funktion in einer Zeigervariablen abzuspeichern. Dies erlaubt es, zum Beispiel einer Funktion selbst eine Funktion als Argument zu übergeben (*Callback* Funktion).

Ein Zeiger auf eine Funktion wird wie folgt definiert:

```
int (*Fptr)(int a, float b); /* Zeiger auf Fkt der Form int f(int x, float y) */
```

Achtung, die Klammern um ***** und den Funktionsnamen sind nötig, sonst Deklariert man schlicht eine Funktion die einen Zeiger auf **int** zurückliefert:

```
int *f (int x);     /* Funktion, die einen Zeiger auf int zurückliefert */
int (*f) (int x);   /* Ein Zeiger auf eine Funktion, die int zurückliefert
                                                                        */
int *(*f) (int x);  /* Ein Zeiger auf eine Funktion, die einen Zeiger
                                                auf int zurückliefert */
```

Eine Funktion wird wie folgt über einen Zeiger aufgerufen:

```
r = (*Fptr)(x, y); /*Mit expliziter Pointerdereferenzierung */
r = Fptr(x, y);    /*Vereinfachte, uebliche Variante (Syntaktisch
inkonsequent)*/
```

Nachfolgend ein Beispiel für den Einsatz eines Funktionszeigers:

```
int (*FPtr)(int a);

int f1(int y)
{
   return y - 2;
}

int f2(int y)
{
   return y + 2;
}

FPtr = f1;     /* oder FPtr = &f1;  Adresse der Funktion abspeichern */

a = FPtr(17);  /* Aufruf der Funktion, auf die der Pointer zeigt */

FPtr = f2;     /* Adresse der Funktion abspeichern */

a = FPtr(a);   /* Aufruf der Funktion, auf die der Pointer zeigt */
```

Achtung, die Funktion muss zum Zeigertyp passen, einem Funktionszeiger kann nur die Adresse einer Funktion übergeben werden, die eine passende Argumentenliste und einen passenden Rückgabetyp besitzt.

```
int (*FPtr)(int a);

int f2(int y, int z)
{
   return y + 2;
}
FPtr = f2; /* Error, Funktion passt nicht zu Pointer (f2 hat 2 Argumente) */
```

Ein etwas grösseres Beispiel zum Einsatz von Funktionspointern:

```
typedef void (*FunPtr)(int *x); /* Typedef, ergibt einfachere Syntax */

/* Anwenden ruft fuer alle Daten die Funktion Fkt auf */
void Anwenden(int *Daten, FunPtr Fkt, int Len)
{
   while (Len > 0) {     /* Solange nicht Datenende erreicht     */
      Fkt(Daten);        /* Funktion auf aktuellen Wert anwenden */
      Daten ++;          /* Zum naechsten Wert gehen             */
      Len--;
   }
}

/* Funktion ersetzt Wert durch seinen Betrag, passt zu FunPtr */
void Betrag(int *Wert)
{
   if (*Wert < 0) {
      *Wert = -*Wert;
   }
}

/* Funktion ersetzt Wert durch sein Quadrat, passt zu FunPtr */
void Quadriere(int *Wert)
{
   *Wert = *Wert * *Wert;
}

/* Funktion schreibt Wert auf den Bildschirm, passt zu FunPtr */
void Print(int *Wert)
{
   printf("%3d ", *Wert);
}



int main (int argc, char *argv[])
{
   int Daten[] = {-3, -2, -1, 0, 1, 2, 3};

   Anwenden(Daten, Print, 7);       /* Gibt Alle Daten aus            */
   Anwenden(Daten, Betrag, 7);      /* Berechnet Betrag von Daten     */

   Anwenden(Daten, Print, sizeof(Daten)/sizeof(Daten[0])); /* Ausgabe */
                                    /* Mit dem sizeof() Ausdruck wird */
                                    /* die Arraygroesse ausgerechnet  */

   Anwenden(Daten, Quadrieren, 7);  /* Berechnet Quadrat von Daten    */
   Anwenden(Daten, Print, 7);       /* Gibt Alle Daten aus            */
   return 0;
}
```

Damit kann die Funktion Anwenden() sehr universell eingesetzt werden, sie kann auch Funktionen verwenden, die beim Programmieren der Funktion Anwenden noch gar nicht bekannt waren.

Durch den Einsatz der Funktion Anwenden() ist es auch sehr einfach, die Datenstruktur zu ändern. Wenn die Daten nicht mehr in einem Feld, sondern in einer Liste abgelegt werden sollen, muss nur die Funktion Anwenden() angepasst werden. Wenn anstelle der Funktion Anwenden() überall eine eigene Schlaufe verwendet worden wäre, müssten bei Änderungen alle Schlaufen angepasst werden.

17.7 Pointer auf irgendetwas

Es ist auch möglich, Zeiger zu definieren, die auf nichts bestimmtes zeigen, also nur eine Adresse enthalten. Dabei ist aber nicht bekannt, was für ein Datentyp an dieser Adresse steht. Diese Zeiger werden als Zeiger auf nichts (**`void`**) definiert:

```
void *Pointer;
```

Ein *void-Pointer* kann nur zum Aufbewahren einer Adresse und sonst für nichts verwendet werden. Er kann weder dereferenziert werden (*Pointer), noch ist Pointerarithmetik (++, --, Pointer + 4, ...) möglich. Einem **`void`** Pointer kann jeder beliebige andere Zeiger oder jede beliebige Adresse zugewiesen werden, und ein **`void`**-Pointer kann jedem anderen Pointertyp zugewiesen werden.

```
int a;
int *p = &a;
float *fp;
struct Datum* sptr;
void *Pointer = p;
void *Pointer2;

int *p2 = Pointer;  /* Kein Problem, void* zu int*    */
fp = Pointer;       /* Kein Problem, void* zu float*  */
sptr = Pointer;     /* Kein Problem, void* zu struct* */
Pointer2 = sptr;    /* Kein Problem, struct* zu void* */

/* aber */
fp = sptr;          /* Error, nicht erlaubt           */
sptr = fp;          /* Error, nicht erlaubt           */
```

17.8 Pointer direkt auf bestimmte Adressen zeigen lassen.

Pointer können auch direkt auf eine bestimmte Adresse gesetzt werden. So kann direkt auf den Speicher oder auf Hardware zugegriffen werden. Dies wird normalerweise nur für HW-Nahe Programmierung benötigt, wie bei Mikrokontrollern oder Gerätetreibern.

```
int *p2;
p2 = (int *) 0x1020; /* Pointer direkt auf Adresse 1020 Hex setzen */
                     /* Mit dem Cast wird der Wert in einen Zeiger umgewandelt
*/
*P2 = 77;            /* Den Wert 77 an diese Adresse schreiben */
```

Kann auch kürzer geschrieben werden :

```
*((int *) 0x1020) = 77;
```

Und mit **`#define`** kann das ganze lesbarer gestaltet werden:

```
/* Steht irgenwo zu beginn, ev.in einer Headerdatei */
#define AnalogAusgang *((int *) 0x1020)

/* Steht im Code bei jeden Zugriff auf den AnalogAusgang */
AnalogAusgang = 77;
```

Achtung, solche *direkten Speicherzugriffe* sollten nur durchgeführt werden, wenn man genau weiss was man tut, ansonsten ist der Programmabsturz oder Fehlverhalten des Systems garantiert.

18 Preprocessor

Der *Preprocessor* ist theoretisch ein eigenständiges Programm, das vor dem eigentlichen Compiler gestartet wird. Heutzutage ist der *Präprozessor* oft in den Compiler integriert, aber das Prinzip ist immer noch dasselbe, zuerst behandelt der Präprozessor das Programm, und erst anschliessend der Compiler. Der Präprozessor hat zwei Hauptaufgaben, nämlich alle Kommentare aus dem Code zu entfernen und alle *Preprozessordirektiven* (#-Befehle) ausführen. Er ist eigentlich nichts anderes als ein vielseitiges 'Suche und Ersetze Programm', er wird dabei mit den #-Befehlen gesteuert. Der Präprozessor führt alle Ersetzungen rein Textmässig aus, er versteht nichts von C.

18.1 Die Präprozessorbefehle

Die Präprozessorbefehle beginnen mit einem **#** und müssen am Anfang einer Zeile, und alleine in einer Zeile stehen. Wenn ein Befehl zu lang für eine Zeile wird, kann er mit einem **** am Zeilenende auf der nächsten Zeile fortgesetzt werden.

Die wichtigsten Befehle:

#include<Filename>	Fügt die angegebene Datei genau an dieser Stelle in den Code ein. Die Datei wird in den im Compilerverzeichnissen gesucht (Der Suchpfad kann bei den meisten Compilern irgendwo definiert oder eingestellt werden)
#include"Filename"	Fügt die angegebene Datei genau an dieser Stelle in den Code ein. Die Datei wird zuerst im Verzeichnis der aktuellen Datei (Die mit dem Include-Befehl) gesucht, und falls sie dort nicht gefunden wird wie bei **#include<>.**
#define Such Ersetz	Ersetzt ab hier jedes auftreten von Such durch Ersetz, Ersetz kann auch leer sein. (Genaugenommen wird das Makro Such definiert)
#undef Such	Löscht das Makro Such, ab hier werden keine Ersetzungen mehr vorgenommen.
#define Square(x) ((x)*(x))	Definiert ein Makro mit einem Argument, beim Aufruf des Makros muss ein Argument angegeben werden, welches dann in die Ersetzung einfliesst. **Achtung**, der Präprozessor führt eine reine Textersetzung durch und ignoriert irgendwelche mathematischen Vorrangregeln -> Klammern setzten !!!:

	`#define Square(x)  x*x  /* Problem, ohne Klammern */`
	
	`q = Square(r+w);`
	`  /* wird zu:      q = r + w * r + w,`
	`    was als q = r + (w * r) + w gerechnet wird  */`

	Makros können auch mehr als ein Argument besitzen.
#if **#else** **#endif**	Bedingte Compilierung. Wenn die Bedingung nach **#if** wahr ist, wird der Code zwischen **#if** und **#else** normal behandelt, alles zwischen **#else** und **#endif** wie Kommentar. Wenn die Bedingung falsch ist, entsprechend umgekehrt. Das **#else** kann auch weggelassen werden. Die Bedingung nach **#if** muss aus konstanten Integerausdrücken bestehen (und **defined()**), und kann beliebige logische und arithmetische Verknüpfungen enthalten.
defined(Name)	Kann nach **#if** verwendet werden, liefert 1 wenn das genannte Makro existiert, sonst 0.
#ifdef Name	Kurzform für **#if defined(Name)**
#ifndef Name	Kurzform für **#if ! defined(Name)**
#pragma	Mit dieser Zeile können herstellerabhängige Direktiven implementiert werden. Der Compiler ignoriert **#pragma**-Zeilen, die er nicht versteht.
#error irgendwas	Gibt eine Fehlermeldung aus

18.2 Vordefinierte Makros:

Folgende Makros sind in jedem Compiler vordefiniert (Je nach Hersteller sind noch weitere möglich):

__LINE__	Eine dezimale Integerkonstante, welche der aktuellen Zeilennummer entspricht, kann für Runtime-Fehlermeldungen benutzt werden.
__FILE__	Eine Stringkonstante, welche dem aktuellen Dateinamen entspricht
__DATE__	Eine Stringkonstante, welche das aktuelle Compiledatum enthält, kann für Versionsangaben verwendet werden
__TIME__	Eine Stringkonstante, welche die aktuelle Compilezeit enthält, kann für Versionsangaben verwendet werden
__STDC__	Ist 1 wenn der Compiler dem ANSI-C-Standard entspricht, sonst 0

18.3 Beispiele

```
/* Definition einiger Makros */

#define VERSION 103

#define DEBUG
#define PI 3.1416
#define ERROR_MESSAGE(x) \
   printf("Internal error in file %s at line %d: %s", __FILE__, __LINE__, x)

int main(int argc, char *argv[])
{
   int i = 0;

/* Ausgabe von Version und Compiledatum und -zeit */
   printf("Version %d (Compiled at %s %s)\n", VERSION, __DATE__, __TIME__);

/* Bedingte Compilierung abhaenging von DEBUG */
#ifdef DEBUG
   printf("i ist %d\n", i);
#endif

/* Bedingte Compilierung, im Moment fix ausgeschaltet       */
/* Mit diesem Konstrukt koennen groessere Codeteile einfach */
/* Zu Testzwecken temporaer deaktiviert werden (Anstatt sie */
/* zu loeschen und anschliessend wieder einzufuegen         */
#if 0
   /* Testcode */
   if (i == 2) {
      i++;
   }
#endif

/* Benutzung von Makros */
   if (i == 0) {
      ERROR_MESSAGE("Division by 0!");
      i = 1;
   }
   printf(" Wert ist %f\n", PI/i);

   return 0;
}
```

19 Bibliotheksfunktionen

19.1 Mathematische Funktionen <math.h>

Die Definitionsdatei **<math.h>** vereinbart mathematische Funktionen und Makros.
Winkel werden bei trigonometrischen Funktionen im Bogenmass (Radiant) angegeben.

double sin(double x)	Sinus von **x**
double cos(double x)	Cosinus von **x**
double tan(double x)	Tangens von **x**
double asin(double x)	arcsin(**x**) im Bereich [-π/2, +π/2], x von -1 bis +1.
double acos(double x)	arccos(**x**) im Bereich [0, +π], x von -1 bis +1.
double atan(double x)	arctan(**x**) im Bereich [-π/2, π/2].
double atan2(double x, double y)	arctan(**y**/**x**) im Bereich [-π, +π].
double sinh(double x)	Sinus Hyperbolicus von **x**
double cosh(double x)	Cosinus Hyperbolicus von **x**
double tanh(double x)	Tangens Hyperbolicus von **x**
double exp(double x)	Exponentialfunktion e^{x}
double log(double x)	Natürlicher Logarithmus ln(**x**), **x** > 0.
double log10(double x)	Logarithmus zur Basis 10, log10(**x**), **x** > 0.
double pow(double x, double y)	$\mathbf{x}^{\mathbf{y}}$. Ein Argumentfehler liegt vor bei **x**=0 und **y**<0, oder bei **x**<0 und **y** ist nicht ganzzahlig.
double sqrt(double x)	Wurzel von **x**, **x** >= 0.
double ceil(double x)	Kleinster ganzzahliger Wert, der nicht kleiner als **x** ist, als double Wert.
double floor (double x)	Grösster ganzzahliger Wert, der nicht grösser als **x** ist, als double Wert.
double fabs(double x)	Absoluter Wert | **x** |
double ldexp(double x,n)	$\mathbf{x} * 2^{\mathbf{n}}$
double frexp(double x, int *exp)	Zerlegt **x** in eine normalisierte Mantisse im Bereich [½, 1], die als Resultat geliefert wird, und eine Potenz von 2, die in ***exp** abgelegt wird. Ist **x** Null, sind beide Teile des Resultats Null. (Gegenteil von ldexp())
double modf(double x, double *ip)	Zerlegt **x** in einen ganzzahligen Teil und einen Rest, die beide das gleiche Vorzeichen wie **x** besitzen. Der ganzzahlige Teil wird bei ***ip** abgelegt, der Rest ist das Resultat.
double fmod(double x, double y)	Gleitpunktrest von **x**/**y**, mit dem gleichen Vorzeichen wie **x**. Wenn **y** Null ist, hängt das Resultat von der Implementierung ab.

19.2 String Funktionen <string.h>

In der Definitionsdatei **`<string.h>`** werden zwei Gruppen von Funktionen für Zeichenketten deklariert. Die erste Gruppe hat Namen, die mit str beginnen und ist für mit **`\0`** abgeschlossene Strings gedacht; die Namen der zweiten Gruppe beginnen mit mem, diese sind für reine Speichermanipulationen vorgesehen. Sieht man von memmove ab, ist der Effekt der Kopierfunktionen undefiniert, wenn sich der Ziel- und der Quellbereich überlappen.

char *strcpy(char * s, const char * ct) Zeichenkette **ct** in Array **s** kopieren, inklusive '\0'; liefert **s**.

char *strncpy(char * s, const char * ct, size_t n)
Höchstens **n** Zeichen aus String **ct** in String **s** kopieren; liefert **s**. Mit '\0' auffüllen, wenn String **ct** weniger als **n** Zeichen hat.

char *strcat(char * s, const char * ct) Zeichenkette **ct** hinten an die Zeichenkette **s** anfügen; liefert **s**.

char *strncat(char * s, const char * ct, size_t n)
Höchstens **n** Zeichen von **ct** hinten an die Zeichenkette **s** anfügen und **s** mit '\0' abschliessen; liefert **s**.

int strcmp(const char * cs, const char * ct)
Strings **cs** und **ct** vergleichen; liefert <0 wenn **cs**<**ct**, 0 wenn **cs**==**ct**, oder >0, wenn **cs**>**ct**. (Reiner ASCII-Vergleich, nicht Lexikalisch)

int strncmp(const char * cs, const char * ct, size_t n)
Höchstens **n** Zeichen von **cs** mit der Zeichenkette **ct** vergleichen; liefert <0 wenn **cs**<**ct**, 0 wenn **cs**==**ct**, oder >0 wenn **cs**>**ct**.

char *strchr(const char * cs, char c) Liefert Zeiger auf das erste **c** in **cs** oder NULL, falls nicht vorhanden.

char *strrchr(const char * cs, char c) Liefert Zeiger auf das letzte **c** in **cs**, oder NULL, falls nicht vorhanden,

size_t strspn(const char * cs, const char * ct)
Liefert Anzahl der Zeichen am Anfang vom String **cs**, die sämtliche im String **ct** auch vorkommen.

size_t strcspn(const char * cs, const char * ct)
Liefert Anzahl der Zeichen am Anfang vom String **cs**, die sämtliche im String **ct** nicht vorkommen.

char *strpbrk(const char * cs, const char * ct)
Liefert Zeiger auf die Position in String **cs**, an der irgendein Zeichen aus **ct** erstmals vorkommt, oder NULL, falls keines vorkommt.

char *strstr(const char * cs, const char * ct)
Liefert Zeiger auf erstes Auftreten von **ct** in **cs** oder NULL, falls nicht vorhanden. (Suchen von String in anderem String)

size_t strlen(const char * cs) Liefert Länge von **cs** (ohne '\0').

char *strerror(size_t n) Liefert Zeiger auf Zeichenkette, die in der Implementierung für den Fehler mit der Nummer **n** definiert ist.

char *strtok(char * s, const char * ct) strtok durchsucht **s** nach Zeichenfolgen, die durch Zeichen aus **ct** begrenzt sind. (Zerlegt einen String in Teilstrings)

Die mem... Funktionen sind zur Manipulation von Speicherbereichen gedacht; sie behandeln den Wert \0 wie jeden anderen Wert, deshalb muss immer eine Bereichslänge angegeben werden.

void *memcpy(void * s, const void * ct, size_t n)
Kopiert **n** Zeichen/Bytes von **ct** nach **s**; liefert **s**.

void *memmove(void * s, const void * ct, size_t n)
Wie memcpy, funktioniert aber auch, wenn sich die Bereiche überlappen.

int memcmp(const void * cs, const void * ct, size_t n)
Vergleicht die ersten **n** Zeichen vom Bereich **cs** mit dem Bereich **ct**; Resultat analog zu strcmp.

void *memchr(const void * cs, char c, size_t n)
Liefert Zeiger auf das erste Byte mit dem Wert **c** in **cs** oder NULL, wenn das Byte in den ersten **n** Zeichen nicht vorkommt.

void *memset(void * s, char c, size_t n) Setzt die ersten **n** Bytes von **s** auf den Wert **c**, liefert **s**. (Speicher füllen)

19.3 Hilfsfunktionen: <stdlib.h>

Die Definitionsdatei **<stdlib.h>** vereinbart Funktionen zur Umwandlung von Zahlen, für Speicherverwaltung und ähnliche Aufgaben.

double atof(const char *s) atof wandelt den String **s** in double um. Beendet die Umwandlung beim ersten unbrauchbaren Zeichen.

int atoi(const char *s) atoi wandelt den String **s** in int um. Beendet die Umwandlung beim ersten unbrauchbaren Zeichen.

long atol(const char *s) atol wandelt den String **s** in long um. Beendet die Umwandlung beim ersten unbrauchbaren Zeichen.

double strtod(const char *s, char **endp) strtod wandelt den Anfang der Zeichenkette **s** in double um, dabei wird Zwischenraum am Anfang ignoriert. Die Umwandlung wird beim ersten unbrauchbaren Zeichen beendet. Die Funktion speichert einen Zeiger auf den ev. nicht umgewandelten Rest der Zeichenkette bei ***endp**, falls **endp** nicht NULL ist. Falls das Ergebnis zu gross ist, (also bei overflow), wird als Resultat HUGE_VAL mit dem korrekten Vorzeichen geliefert; liegt das Ergebnis zu dicht bei Null (also bei underflow), wird Null geliefert. In beiden Fällen erhält errno den Wert ERANGE.

long strtol(const char *s, char **endp, int base) strtol wandelt den Anfang der Zeichenkette **s** in long um, dabei wird Zwischenraum am Anfang ignoriert. Die Umwandlung wird beim ersten unbrauchbaren Zeichen beendet. Die Funktion speichert einen Zeiger auf den ev. nicht umgewandelten Rest der Zeichenkette bei ***endp**, falls **endp** nicht NULL ist. Hat **base** einen Wert zwischen 2 und 36, erfolgt die Umwandlung unter der Annahme, dass die Eingabe in dieser Basis repräsentiert ist. Hat **base** den Wert Null, wird als Basis 8, 10 oder 16 verwendet, je nach **s**; eine führende Null bedeutet dabei oktal und 0x oder 0X zeigen eine hexadezimale Zahl an. In jedem Fall stehen Buchstaben für die Ziffern von 10 bis **base**-l; bei Basis 16 darf 0x oder 0X am Anfang stehen. Wenn das Resultat zu gross werden würde, wird je nach Vorzeichen LONG_MAX oder LONG_MIN geliefert und errno erhält den Wert ERANGE.

unsigned long strtoul(const char *s, char **endp, int base) strtoul funktioniert wie strtol, nur ist der Resultattyp unsigned long und der Fehlerwert ist ULONG_MAX.

int rand(void) rand liefert eine ganzzahlige Pseudo-Zufallszahl im Bereich von 0 bis RAND_MAX; dieser Wert ist mindestens 32767.

void srand(unsigned int seed) srand benutzt **seed** als Ausgangswert für eine neue Folge von Pseudo-Zufallszahlen. Der erste Ausgangswert ist 1. Vor erstmaliger Benutzung von rand() sollte in jedem Programm einmal (Z.B. zu Beginn) srand() aufgerufen werden, z. B: srand(time(NULL)) (Die aktuelle Zeit reicht meistens als zufälliger Startwert).

void *calloc(size_t nobj, size_t size) calloc liefert einen Zeiger auf einen Speicherbereich für einen Vektor von **nobj** Objekten, jedes mit der Grösse **size**, oder NULL, wenn die Anforderung nicht erfüllt werden kann. Der Bereich wird mit Null-Bytes initialisiert.

void *malloc(size_t size) malloc liefert einen Zeiger auf einen Speicherbereich für ein Objekt der Grösse **size** oder NULL, wenn die Anforderung nicht erfüllt werden kann. Der Bereich ist nicht initialisiert.

void *realloc(void *p, size_t size) realloc ändert die Grösse des Objekts, auf das **p** zeigt, in **size** ab. Bis zur kleineren der alten und neuen Grösse bleibt der Inhalt unverändert. Wird der Bereich für das Objekt grösser, so ist der zusätzliche Bereich uninitialisiert. realloc liefert einen Zeiger auf den neuen Bereich oder NULL, wenn die Anforderung nicht erfüllt werden kann; in diesem Fall wird der Inhalt nicht verändert.

void free(void *p) free gibt den Bereich frei, auf den **p** zeigt; die Funktion hat keinen Effekt, wenn **p** den Wert NULL hat. **p** muss auf einen Bereich zeigen, der zuvor mit calloc, malloc oder realloc angelegt wurde.

void abort(void) abort sorgt für eine anormale, sofortige Beendigung des Programms.

void exit(int status)	exit beendet das Programm normal. atexit-Funktionen werden in umgekehrter Reihenfolge ihrer Hinterlegung aufgerufen, die Puffer offener Dateien werden geschrieben, offene Ströme werden abgeschlossen, und die Kontrolle geht an die Umgebung des Programms zurück. Wie **status** an die Umgebung des Programms geliefert wird, hängt von der Implementierung ab, aber Null gilt als erfolgreiches Ende. Die Werte EXIT_SUCCESS und EXIT_FAILURE können ebenfalls angegeben werden.
int atexit(void (*fcn)(void))	atexit hinterlegt die Funktion **fcn**, damit sie aufgerufen wird, wenn das Programm normal endet, und liefert einen von Null verschiedenen Wert, wenn die Funktion nicht hinterlegt werden kann.
int system(const char *s)	system liefert die Zeichenkette **s** an die Umgebung zur Ausführung. Hat **s** den Wert NULL, so liefert system einen von Null verschiedenen Wert, wenn es einen Kommandoprozessor gibt. Wenn **s** von NULL verschieden ist, dann ist der Resultatwert implementierungsabhängig.
char *getenv(const char *name)	getenv liefert die zu **name** gehörende Zeichenkette aus der Umgebung oder NULL, wenn keine Zeichenkette existiert. Die Details hängen von der Implementierung ab.
void *bsearch(const void *key, const void *base, size_t n, size_t size, int (*cmp)(const void *keyval, const void *datum))	bsearch durchsucht **base**[0] ... **base**[**n**-l] nach einem Eintrag, der gleich ***key** ist. Die Funktion **cmp** muss einen negativen Wert liefern, wenn ihr erstes Argument (der Suchschlüssel) kleiner als ihr zweites Argument (ein Tabelleneintrag) ist, Null, wenn beide gleich sind, und sonst einen positiven Wert. Die Elemente des Arrays **base** müssen aufsteigend sortiert sein. In **size** muss die Grösse eines einzelnen Elements übergeben werden. bsearch liefert einen Zeiger auf das gefundene Element oder NULL, wenn keines existiert.
void qsort(void *base, size_t n, size_t size, int (*cmp)(const void *, const void *))	qsort sortiert ein Array **base**[0] ... **base**[**n**-1] von Objekten der Grösse **size** in aufsteigender Reihenfolge. Für die Vergleichsfunktion **cmp** gilt das gleiche wie bei bsearch.
int abs(int n)	abs liefert den absoluten Wert seines int Arguments **n**.
long labs(long n)	labs liefert den absoluten Wert seines long Arguments **n**.
div_t div(int num, int denom)	div berechnet Quotient und Rest von **num**/**denom**. Die Resultate werden in den int Komponenten quot und rem einer Struktur vom Typ div_t abgelegt.
ldiv_t ldiv(long num, long denom)	div berechnet Quotient und Rest von **num**/**denom**. Die Resultate werden in den long Komponenten quot und rem einer Struktur vom Typ ldiv_t abgelegt.

Anwendung von Qsort:

```
typedef struct Entry {
   char Name[];
   int Age;
} Entry;

/* Vergleichsfunktion fuer qsort, sortiert nach Alter (Age) */
int EntryCompareFkt(const void *keyval, const void *datum)
{
   /* Funktion muss negativen Wert liefern wenn *Key < *datum  */
   /* Funktion muss 0 liefern wenn              *Key == *datum */
   /* Funktion muss positiven Wert liefern wenn *Key > *datum  */
   /* Cast ist noetig um aus void* wieder einen Entry* zu machen */
   return ((const Entry *) keyval)->Age - ((const Entry *) datum)->Age;
}

void UseQuickSort(struct Entry *Array, int Length)
{
   /* Das Array mit Quicksort aus der Standardbibliothek sortieren */
   qsort(Array, Length, sizeof(struct Entry), EntryCompareFkt);
}
```

19.4 Funktionen für Datum und Uhrzeit: <time.h>

Die Definitionsdatei **<time.h>** vereinbart Typen und Funktionen zum Umgang mit Datum und Uhrzeit. Manche Funktionen verarbeiten die Ortszeit, die von der Kalenderzeit zum Beispiel wegen einer Zeitzone abweicht. **clock_t** und **time_t** sind numerische Typen, die Zeiten repräsentieren, und **struct tm** enthält die Komponenten einer Kalenderzeit:

```
struct tm {
    int tm_sec;     Sekunden nach der vollen Minute (0, 61)*
    int tm_min;     Minuten nach der vollen Stunde (0, 59)
    int tm_hour;    Stunden seit Mitternacht (0, 23)
    int tm_mday;    Tage im Monat (1, 31)
    int tm_mon;     Monate seit Januar (0, 11)
    int tm_year;    Jahre seit 1900
    int tm_wday;    Tage seit Sonntag (0, 6)
    int tm_yday;    Tage seit dem 1. Januar (0, 365)
    int tm_isdst;   Kennzeichen für Sommerzeit
}
```

(*Die zusätzlich möglichen Sekunden sind Schaltsekunden)

tm_isdst ist positiv, wenn Sommerzeit gilt, Null, wenn Sommerzeit nicht gilt, und negativ, wenn die Information nicht zur Verfügung steht.

clock_t clock(void) — **clock** liefert die Rechnerkern-Zeit, die das Programm seit Beginn seiner Ausführung verbraucht hat, oder -1, wenn diese Information nicht zur Verfügung steht. clock()/CLOCKS_PER_SEC ist eine Zeit in Sekunden.

time_t time(time_t *tp) — time liefert die aktuelle Kalenderzeit oder -1, wenn diese nicht zur Verfügung steht. Wenn **tp** von NULL verschieden ist, wird der Resultatwert auch bei ***tp** abgelegt.

double difftime(time_t time2, time_t timel) — difftime liefert **time2** - **timel** ausgedrückt in Sekunden.

time_t mktime(struct tm *tp) — mktime wandelt die Ortszeit in der Struktur ***tp** in Kalenderzeit um, die so dargestellt wird wie bei time. Die Komponenten erhalten Werte in den angegebenen Bereichen. mktime liefert die Kalenderzeit oder -1, wenn sie nicht dargestellt werden kann.

Die folgenden vier Funktionen liefern Zeiger auf statische Objekte, die von anderen Aufrufen überschrieben werden können.

char *asctime(const struct tm *tp) — asctime konstruiert aus der Zeit in der Struktur ***tp** eine Zeichenkette der Form

Sun Jan 3 15:14:13 1988\n\0

char *ctime(const time_t *tp) — ctime verwandelt die Kalenderzeit ***tp** in Ortszeit; dies ist äquivalent zu asctime(localtime(tp))

struct tm *gmtime(const time_t *tp) — gmtime verwandelt die Kalenderzeit ***tp** in Coordinated Universal Time (UTC). Die Funktion liefert NULL, wenn UTC nicht zur Verfügung steht. Der Name gmtime hat historische Bedeutung.

struct tm *localtime(const time_t *tp) — localtime verwandelt die Kalenderzeit ***tp** in Ortszeit.

size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)

strftime formatiert Datum und Zeit aus ***tp** in **s** unter Kontrolle von **fmt**, analog zu einem printf-Format. Gewöhnliche Zeichen (insbesondere auch das abschliessende '\0') werden nach s kopiert. Jedes %... wird so wie unten beschrieben ersetzt, wobei Werte verwendet werden, die der lokalen Umgebung entsprechen. Höchstens **smax** Zeichen werden in **s** abgelegt. strftime liefert die Anzahl der resultierenden Zeichen, mit Ausnahme von '\0'. Wenn mehr als **smax** Zeichen erzeugt wurden, liefert strftime den Wert Null.

Umwandlungszeichen für den Formatstring fmt:

%a	abgekürzter Name des Wochentags.
%A	voller Name des Wochentags.
%b	abgekürzter Name des Monats.
%B	voller Name des Monats.
%c	lokale Darstellung von Datum und Zeit.
%d	Tag im Monat (01 - 31).
%H	Stunde (00 - 23).
%I	Stunde (01 - 12).
%j	Tag im Jahr (001 - 366).
%m	Monat (01 - 12).
%M	Minute (00 - 59).
%p	lokales Äquivalent von AM oder PM.
%S	Sekunde (00 - 61).
%U	Woche im Jahr (Sonntag ist erster Tag) (00 - 53).
%w	Wochentag (0 - 6, Sonntag ist 0).
%W	Woche im Jahr (Montag ist erster Tag) (00 - 53).
%x	lokale Darstellung des Datums.
%X	lokale Darstellung der Zeit.
%y	Jahr ohne Jahrhundert (00 - 99).
%Y	Jahr mit Jahrhundert.
%Z	Name der Zeitzone, falls diese existiert.
%%	%. (Gibt ein % aus)

Beispiel:

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

#define BUFFERSIZE 100              /* Groesse fuer Buffer definieren    */

int main(int argc, char *argv[])
{
   time_t Now;                      /* Platz fuer Zeitinfo               */
   struct tm * OurTimeNow;          /* Zeiger auf lokale Zeit            */
   char Buffer[BUFFERSIZE];         /* Platz fuer formatierte Zeit       */

   Now = time(NULL);                /* Aktuelle Zeit holen               */
   OurTimeNow = localtime(&Now);    /* Umwandeln der Zeit in lokale Zeit */
                                    /* Formatierten Text erzeugen        */
   strftime(Buffer, BUFFERSIZE, "Today is %a, the %d of %B", OurTimeNow);

   puts(Buffer);                    /* Und Text ausgeben                 */

   return 0;
}
```

19.5 Ein- und Ausgabe: <stdio.h>

Die Datei **`<stdio.h>`** vereinbart Typen und Funktionen zum Umgang mit *Streams*. Ein Stream (Datenstrom) ist Quelle oder Ziel von Daten und wird mit einer Datei oder einem Peripheriegerät verknüpft. Es werden zwei Arten von Streams unterstützt, nämlich für Text und für binäre Information, die jedoch bei manchen Systemen, und insbesondere bei UNIX identisch sind. Ein Textstrom ist eine Folge von Zeilen; jede Zeile enthält null oder mehr Zeichen und ist mit '\n' abgeschlossen. Das Betriebssystem muss möglicherweise zwischen einem Textstrom und einer anderen Repräsentierung umwandeln (also zum Beispiel '\n' als Wagenrücklauf und Zeilenvorschub abbilden).

Ein Stream wird durch Öffnen (open) mit einer Datei oder einem Gerät verbunden; die Verbindung wird durch Schliessen (close) wieder aufgehoben. Öffnet man eine Datei, so erhält man einen Zeiger auf ein Objekt vom Typ **`FILE`**, in welchem alle Information hinterlegt sind, die zur Kontrolle des Stream nötig sind.

Wenn die Ausführung eines Programms beginnt, sind die drei Streams **`stdin`**, **`stdout`** und **`stderr`** bereits geöffnet.

19.5.1 Dateioperationen

Die folgenden Funktionen beschäftigen sich mit Dateioperationen. Der Typ **`size_t`** ist der vorzeichenlose, ganzzahlige Resultattyp des sizeof-Operators. EOF ist eine vordefinierte Konstante, welche das Dateiende (End Of File) anzeigt.

FILE *fopen(const char *filename, const char *mode) fopen eröffnet die angegebene Datei und liefert einen Datenstrom oder NULL bei Misserfolg. Zu den erlaubten Werten von **mode** gehören

- "r" Textdatei zum Lesen öffnen
- "w" Textdatei zum Schreiben erzeugen; gegebenenfalls alten Inhalt wegwerfen
- "a" anfügen; Textdatei zum Schreiben am Dateiende öffnen oder erzeugen
- "r+" Textdatei zum Ändern öffnen (Lesen und Schreiben)
- "w+" Textdatei zum Ändern erzeugen; gegebenenfalls alten Inhalt wegwerfen
- "a+" anfügen; Textdatei zum Ändern öffnen oder erzeugen, Schreiben am Ende

Ändern bedeutet, dass die gleiche Datei gelesen und geschrieben werden darf; fflush oder eine Funktion zum Positionieren in Dateien muss zwischen einer Lese- und einer Schreiboperation oder umgekehrt aufgerufen werden. Enthält **mode** nach dem ersten Zeichen noch b, also etwa "rb" oder "w+b", dann wird auf eine binäre Datei zugegriffen. Dateinamen sind auf FILENAME_MAX Zeichen begrenzt. Höchstens FOPEN_MAX Dateien können gleichzeitig offen sein.

FILE *freopen(const char *filename, const char *mode, FILE *stream)
freopen öffnet die Datei für den angegebenen Zugriff **mode** und verknüpft **stream** damit. Das Resultat ist **stream** oder Null bei einem Fehler. Mit freopen ändert man normalerweise die Dateien, die mit stdin, stdout oder stderr verknüpft sind. (Neue Datei mit bestehendem Stream verknüpfen)

int fflush(FILE *stream) Bei einem Ausgabestrom sorgt fflush dafür, dass gepufferte, aber noch nicht geschriebene Daten geschrieben werden; bei einem Eingabestrom ist der Effekt undefiniert. Die Funktion liefert EOF bei einem Schreibfehler und sonst Null. fflush(NULL) bezieht sich auf alle offenen Dateien.

int fclose(FILE *stream) fclose schreibt noch nicht geschriebene Daten für **stream**, wirft noch nicht gelesene, gepufferte Eingaben weg, gibt automatisch angelegte Puffer frei und schliesst den Datenstrom. Die Funktion liefert EOF bei Fehlern und sonst Null.

int remove(const char *filename) remove löscht die angegebene Datei, so dass ein anschliessender Versuch, sie zu öffnen, fehlschlagen wird. Die Funktion liefert bei Fehlern einen von Null verschiedenen Wert.

int rename(const char *oldname, const char *newname)
rename ändert den Namen einer Datei und liefert nicht Null, wenn der Versuch fehlschlägt.

FILE *tmpfile(void) tmpfile erzeugt eine temporäre Datei mit Zugriff "wb+", die automatisch gelöscht wird, wenn der Zugriff abgeschlossen wird, oder wenn das Programm normal zu Ende geht. tmpfile liefert einen Datenstrom, oder NULL, wenn die Datei nicht erzeugt werden konnte.

char *tmpnam(char s[L_tmpnam]) tmpnam(NULL) erzeugt eine Zeichenkette, die nicht der Name einer existierenden Datei ist, und liefert einen Zeiger auf einen internen Vektor im statischen Speicherbereich. tmpnam(s) speichert die Zeichenkette in **s** und liefert auch **s** als Resultat; in **s** müssen wenigstens **L_tmpnam** Zeichen abgelegt werden können. tmpnam erzeugt bei jedem Aufruf einen anderen Namen; man kann höchstens von TMP_MAX verschiedenen Namen während der Ausführung des Programms ausgehen. Zu beachten ist, dass tmpnam einen Namen und keine Datei erzeugt.

int setvbuf(FILE *stream, char *buf, int mode, size_t size)
setvbuf kontrolliert die Pufferung bei einem Datenstrom; die Funktion muss aufgerufen werden, bevor gelesen oder geschrieben wird, und vor allen anderen Operationen. Hat **mode** den Wert _IOFBF, so wird vollständig gepuffert, _IOLBF sorgt für zeilenweise Pufferung bei Textdateien und _IONBF verhindert Puffern. Wenn **buf** nicht NULL ist, wird **buf** als Puffer verwendet; andernfalls wird ein Puffer angelegt. **size** legt die Puffergrösse fest. Bei einem Fehler liefert setvbuf nicht Null. (Der Einsatz von Puffern kann Dateizugriffe zum Teil massiv beschleunigen).

void setbuf(FILE *stream, char *buf) Wenn **buf** den Wert NULL hat, wird der Datenstrom nicht gepuffert. Andernfalls ist setbuf äquivalent zu (void) setvbuf(stream, buf, _IOFBF, BUFSIZ).

19.5.2 Formatierte Ausgabe

Die printf-Funktionen ermöglichen Ausgaben unter Formatkontrolle.

int fprintf(FILE *stream, const char *format, ...)
fprintf wandelt Ausgaben um und schreibt sie in **stream** unter Kontrolle von **format**. Der Resultatwert ist die Anzahl der geschriebenen Zeichen; er ist negativ, wenn ein Fehler passiert ist.

int printf(const char *format, ...) printf(...) ist äquivalent zu fprintf(stdout,...).

int sprintf(char *s, const char *format, ...)
sprintf funktioniert wie printf, nur wird die Ausgabe in das Zeichenarray **s** geschrieben und mit '\0' abgeschlossen. **s** muss gross genug für das Resultat sein. Im Resultatwert wird '\0' nicht mitgezählt.

vprintf(const char *format, va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)
Die Funktionen vprintf, vfprintf und vsprintf sind äquivalent zu den entsprechenden printf-Funktionen, jedoch wird die variable Argumentenliste durch **arg** ersetzt. Dieser Wert wird mit dem Makro va_start und vielleicht mit Aufrufen von va_arg initialisiert.

Die **Format-Zeichenkette** enthält zwei Arten von Objekten: gewöhnliche Zeichen, die in die Ausgabe kopiert werden, und Umwandlungsangaben, die jeweils die Umwandlung und Ausgabe des nächstfolgenden Arguments von fprintf veranlassen. Jede Umwandlungsangabe beginnt mit dem Zeichen % und endet mit einem Umwandlungszeichen. Zwischen % und dem Umwandlungszeichen kann der Reihenfolge nach folgendes angegeben werden:

Steuerzeichen (flags) (in beliebiger Reihenfolge), welche die Umwandlung modifizieren:

- `-` veranlasst die Ausrichtung des umgewandelten Arguments in seinem Feld nach links.
- `+` bestimmt, dass die Zahl immer mit Vorzeichen ausgegeben wird.
- Leerzeichen wenn das erste Zeichen kein Vorzeichen ist, wird ein Leerzeichen vorangestellt.
- `0` legt bei numerischen Umwandlungen fest, dass bis zur Feldbreite mit führenden Nullen aufgefüllt wird.
- `#` verlangt eine alternative Form der Ausgabe. Bei o ist die erste Ziffer eine Null. Bei x oder X werden 0x oder 0X einem von Null verschiedenen Resultat vorangestellt. Bei e, E, f, g und G enthält die Ausgabe immer einen Dezimalpunkt; bei g und G werden Nullen am Schluss nicht unterdrückt.

eine **Zahl**, die eine minimale **Feldbreite** festlegt. Das umgewandelte Argument wird in einem Feld ausgegeben, das mindestens so breit ist und bei Bedarf auch breiter. Hat das umgewandelte Argument weniger Zeichen als die Feldbreite verlangt, wird links (oder rechts, wenn Ausrichtung nach links verlangt wurde) auf die Feldbreite aufgefüllt. Normalerweise wird mit Leerzeichen aufgefüllt, aber auch mit Nullen, wenn das entsprechende Steuerzeichen angegeben wurde.

Ein **Punkt**, der die Feldbreite von der Genauigkeit (precision) trennt.

Eine **Zahl**, die **Genauigkeit**, welche die maximale **Anzahl von Zeichen** festlegt, die von einer Zeichenkette ausgegeben werden, oder die **Anzahl Ziffern**, die nach dem Dezimalpunkt bei e, E, oder f Umwandlungen ausgegeben werden, oder die **Anzahl signifikanter Ziffern** bei g oder G Umwandlung oder die **minimale Anzahl von Ziffern**, die bei einem ganzzahligen Wert ausgegeben werden sollen (führende Nullen werden dann bis zur gewünschten Breite hinzugefügt).

Ein **Buchstabe** als Längenangabe: h, l oder L.

- "h" bedeutet, dass das zugehörige Argument als **`short`** oder **`unsigned short`** ausgegeben wird;
- "l" bedeutet, dass das Argument **`long`** oder **`unsigned long`** ist:
- "L" bedeutet, dass das Argument **`long double`** ist.

Als Feldbreite oder Genauigkeit kann jeweils * angegeben werden; dann wird der Wert durch Umwandlung von dem nächsten oder den zwei nächsten Argumenten festgelegt, die den Typ int besitzen müssen.

Aufbau (Ohne Leerzeichen dazwischen, alles ausser dem Umwandlungszeichen kann weggelassen werden):

% Flag Zahl . Zahl Längenangabe Umwandlungszeichen

Beispiel: `printf("%+08.4d", 17);` gibt > `+0017<` aus

Die Umwandlungszeichen und ihre Bedeutung:

Zeichen	Argument	Ausgabe
d, i	**`int`**	dezimal mit Vorzeichen.
o	**`int`**	oktal ohne Vorzeichen (ohne führende Null).
x, X	**`int`**	hexadezimal ohne Vorzeichen (ohne führendes 0x oder 0X) mit abcdef bei 0x oder ABCDEF bei 0X.
u	**`int`**	dezimal ohne Vorzeichen.
c	**`int`**	einzelnes Zeichen (Buchstabe), nach Umwandlung in unsigned char.
s	**`char*`**	aus einer Zeichenkette werden Zeichen ausgegeben bis vor '\0', aber maximal so viele Zeichen, wie die Genauigkeit (im Formatstring) erlaubt.
f	**`double`**	dezimal als [-]mmm.ddd, wobei die Genauigkeit die Anzahl der d festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.
e, E	**`double`**	dezimal als [-]m.dddddde±xx oder [-]m.ddddddE±xx, wobei die Genauigkeit die Anzahl der d festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.
g, G	**`double`**	%e oder %E wird verwendet, wenn der Exponent kleiner als -4 oder nicht kleiner als die Genauigkeit ist; sonst wird %f benutzt. Null und Dezimalpunkt am Schluss werden nicht ausgegeben.
p	**`void*`**	als Zeiger (Speicheradresse, Darstellung hängt von Implementierung ab).
n	**`int*`**	die Anzahl der bisher von diesem Aufruf von printf ausgegebenen Zeichen wird im Argument abgelegt. Es findet an dieser Stelle keine Ausgabe statt.
%	-	ein % wird ausgegeben.

Wenn das Zeichen nach % kein gültiges Umwandlungszeichen ist, ist das Ergebnis undefiniert.

19.5.3 Formatierte Eingabe

Die scanf-Funktionen behandeln Eingabe-Umwandlungen unter Formatkontrolle.

int fscanf(FILE *stream, const char *format, ...)

fscanf liest von **stream** unter Kontrolle von **format** und legt umgewandelte Werte mit Hilfe von nachfolgenden Argumenten ab, die alle Zeiger sein müssen. Die Funktion wird beendet, wenn **format** abgearbeitet ist. fscanf liefert EOF, wenn vor der ersten Umwandlung das Dateiende erreicht wird oder ein Fehler passiert; andernfalls liefert die Funktion die Anzahl der umgewandelten und abgelegten Eingaben.

int scanf(const char *format, ...)

scanf(...) ist äquivalent zu fscanf(stdin,...).

int sscanf(const char *s, const char *format, ...)

sscanf(s, ...) ist äquivalent zu scanf(...), mit dem Unterschied, dass die Eingabezeichen aus der Zeichenkette **s** stammen. (Liest aus einem String, anstelle eines Streams, so können Strings umgewandelt werden. Die Kombination `gets(Buffer); sscanf(Buffer, ...)` ist einfacher um Fehleingaben abzufangen.)

Die **Format-Zeichenkette** enthält normalerweise Umwandlungsangaben, die zur Interpretation der Eingabe verwendet werden. Die Format-Zeichenkette kann folgendes enthalten:

Leerzeichen oder Tabulatorzeichen, die ignoriert werden.

Gewöhnliche Zeichen (nicht aber %), die dem nächsten Zeichen nach Zwischenraum im Eingabestrom entsprechen müssen. Sie definieren so eine Eingabemaske. (`scanf("(%d)", &i)` z. B. akzeptiert nur Eingabewerte, die in Klammern stehen, der Benutzer muss also auch die Klammern eintippen!!)

Umwandlungsangaben, bestehend aus einem %; einem optionalen Zeichen *, das die Zuweisung an ein Argument verhindert; einer optionalen Zahl, welche die maximale Feldbreite festlegt; einem optionalen Buchstaben h, l, oder L, der die Länge des Ziels beschreibt; und einem Umwandlungszeichen.

Eine Umwandlungsangabe bestimmt die Umwandlung des nächsten Eingabefelds. Normalerweise wird das Resultat in der Variablen abgelegt, auf die das zugehörige Argument zeigt. Wenn jedoch * die Zuweisung verhindern soll, wie bei %*s, dann wird das Eingabefeld einfach übergangen und eine Zuweisung findet nicht statt. Ein Eingabefeld ist als Folge von Zeichen definiert, die keine Zwischenraumzeichen sind; es reicht entweder bis zum nächsten Zwischenraumzeichen, oder bis eine explizit angegebene Feldbreite erreicht ist. Daraus folgt, dass scanf über Zeilengrenzen hinweg liest, um seine Eingabe zu finden, denn Zeilentrenner sind Zwischenraumzeichen. (Zwischenraumzeichen sind Leerzeichen, Tabulatorzeichen \t, Zeilentrenner \n, Wagenrücklauf \r, Vertikaltabulator \v und Seitenvorschub \f).

Das Umwandlungszeichen gibt die Interpretation des Eingabefelds an. Das zugehörige Argument muss ein Zeiger sein.

Den Umwandlungszeichen d, i, n, o, u und x kann h vorausgehen, wenn das Argument ein Zeiger auf **`short`** statt **`int`** ist, oder der Buchstabe l, wenn das Argument ein Zeiger auf **`long`** ist. Vor den Umwandlungszeichen e, f und g kann der Buchstabe l stehen, wenn ein Zeiger auf **`double`** und nicht auf **`float`** in der Argumentenliste steht, und L, wenn es sich um einen Zeiger auf **`long double`** handelt.

Aufbau (Ohne Leerzeichen dazwischen, alles ausser dem Umwandlungszeichen kann weggelassen werden):

% Stern(*) Feldbreite(Zahl) Modifier(l, L oder h) Umwandlungszeichen

Achtung, scanf(...) lässt alle nicht verwertbaren Zeichen im Eingabepuffer stehen, der Eingabepuffer sollte deshalb vor dem nächsten Aufruf von scanf() geleert werden, sonst wird scanf() zuerst die verbliebenen Zeichen umzuwandeln versuchen. Wenn die Zeichen erneut für die geforderte Eingabe nicht verwendbar sind, verbleiben sie weiterhin im Eingabepuffer. So kann ein ungültiges Zeichen im Eingabepuffer jeden weiteren Einlesevorgang blockieren, solange es nicht aus dem Puffer entfernt wird (Z. B. mit getchar() oder gets()).

Beispiele für sicheren scanf()-Einsatz

```
/* Einlesen mit anschliessendem Pufferleeren */
scanf("%d", &i);               /* Einlesen      */
while (getchar() != '\n') {};  /* Puffer leeren */

/* Einlesen mit gets() und scanf() */
char Buffer[200]   /* Nur sicher wenn Eingabe nicht laenger als 200 Zeichen */
gets(Buffer);
sscanf(Buffer, "%d", &i);      /* Einlesen      */
```

Zeichen	Eingabedaten	Argumenttyp	Anmerkungen
d	dezimal, ganzzahlig	`int *`	
i	ganzzahlig	`int *`	Der Wert kann oktal (mit 0 am Anfang) oder hexadezimal (mit 0x oder 0X am Anfang) angegeben sein.
o	oktal ganzzahlig	`int *`	Eingabe mit oder ohne 0 am Anfang
u	dezimal ohne Vorzeichen	`unsigned int *`	
x	hexadezimal ganzzahlig	`int *`	Eingabe mit oder ohne 0x oder 0X am Anfang
c	ein oder mehrere Zeichen	`char *`	Die nachfolgenden Eingabezeichen werden im angegebenen Array abgelegt, bis die Feldbreite erreicht ist. Voreinstellung ist 1. '\0' wird nicht angefügt. In diesem Fall wird Zwischenraum nicht überlesen, das nächste Zeichen nach Zwischenraum liest man mit %ls. Es werden genau soviel Zeichen gelesen und erwartet, wie bei Feldbreite angegeben. Wird mehr als ein Zeichen eingelesen, muss das zugehörige Argument auf ein Array passender Grösse zeigen
s	Folge von Nicht-Zwischenraum-Zeichen (ohne Anführungszeichen)	`char *`	Argument muss auf ein Array zeigen, das die Zeichenkette und das abschliessende '\0' aufnehmen kann.
e, f, g	Gleitpunkt	`float *`	Das Eingabeformat erlaubt für float ein optionales Vorzeichen, eine Ziffernfolge, die auch einen Dezimalpunkt enthalten kann, und einen optionalen Exponenten, bestehend aus E oder e und einer ganzen Zahl, optional mit Vorzeichen.
p	Zeiger	`void *`	Eingabe der Adresse wie sie printf("%p") ausgibt
n	legt im Argument die Anzahl Zeichen ab, die bei diesem Aufruf bisher gelesen wurden	`int *`	Vom Eingabestrom wird nicht gelesen, die Zählung der Umwandlungen bleibt unbeeinflusst.
[...]	erkennt die längste nicht-leere Zeichenkette aus den Eingabezeichen in der angegebenen Menge	`char *`	Dazu kommt '\0'. Die Menge []...] enthält auch].
[^...]	erkennt die längste nicht-leere Zeichenkette aus den Eingabezeichen die nicht in der angegebenen Menge enthalten sind	`char*`	Dazu kommt '\0'. Die Menge [^]...] enthält auch].
%	erkennt %	-	eine Zuweisung findet nicht statt.

19.5.4 Ein- und Ausgabe von Zeichen

Mit diesen Funktionen erfolgt die Ein- und Ausgabe von Texten und Zeichen.

int fgetc(FILE *stream)	fgetc liefert das nächste Zeichen aus **stream** als unsigned char (umgewandelt in int) oder EOF bei Dateiende oder bei einem Fehler.
char *fgets(char *s, int n, FILE *stream)	fgets liest höchstens die nächsten **n**-1 Zeichen in **s** ein und hört vorher auf, wenn ein Zeilentrenner gefunden wird. Der Zeilentrenner wird im String **s** abgelegt. Der String **s** wird mit '\0' abgeschlossen. fgets liefert **s** oder NULL bei Dateiende oder bei einem Fehler.
int fputc(int c, FILE *stream)	fputc schreibt das Zeichen **c** (umgewandelt in unsigned char) in **stream**. Die Funktion liefert das ausgegebene Zeichen oder EOF bei Fehler.
int fputs(const char *s, FILE *stream)	fputs schreibt die Zeichenkette **s** (die '\n' nicht zu enthalten braucht) in **stream**. Die Funktion liefert einen nicht-negativen Wert oder EOF bei einem Fehler. Es wird nicht automatisch ein Zeilentrenner ausgegeben (Im Gegensatz zu puts()).
int getc(FILE *stream)	getc ist äquivalent zu fgetc, kann aber ein Makro sein und dann das Argument für **stream** mehr als einmal bewerten.
int getchar(void)	getchar ist äquivalent zu getc(stdin).
char *gets(char *s)	gets liest die nächste Zeile von stdin in das Array **s** und ersetzt dabei den abschliessenden Zeilentrenner durch '\0'. Die Funktion liefert **s** oder NULL bei Dateiende oder bei einem Fehler.
int putc(int c, FILE *stream)	putc ist äquivalent zu fputc, kann aber ein Makro sein und dann das Argument für **stream** mehr als einmal bewerten.
int putchar(int c)	putchar(**c**) ist äquivalent zu putc(**c**, stdout).
int puts(const char *s)	puts schreibt die Zeichenkette **s** und einen Zeilentrenner in stdout. Die Funktion liefert EOF, wenn ein Fehler passiert, andernfalls einen nicht-negativen Wert.
int ungetc(int c, FILE *stream)	ungetc stellt **c** (umgewandelt in unsigned char) in **stream** zurück, von wo das Zeichen beim nächsten Lesevorgang wieder geholt wird. Man kann sich nur darauf verlassen, dass pro Datenstrom ein Zeichen zurückgestellt werden kann. EOF darf nicht zurückgestellt werden. ungetc liefert das zurückgestellte Zeichen oder EOF bei einem Fehler.
EOF	**EOF** ist eine vordefinierte Konstante, welche das Dateiende anzeigt. Der Wert der Konstante ist System- und Compilerabhängig, ist aber meist ein negativer Wert.

Achtung, fgets() und fputs() behandeln Zeilentrenner ('\r\n') anders als gets() und puts(). Die f... Funktionen transportieren den Text unverändert von der Quelle zum Ziel, hingegen entfernt gets() den Zeilentrenner am Ende der Zeile, und puts() fügt einen Zeilentrenner am Ende der Ausgabe hinzu.

19.5.5 Direkte (Binäre) Ein- und Ausgabe

Mit diesen Funktionen werden unformatierte Speicherblöcke gelesen und geschrieben. Das heisst dass die Daten einfach als zusammenhängende Blöcke von Bytes behandelt werden, es findet keine Interpretation oder Umwandlung statt.

size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)

fread liest aus **stream** in das Array **ptr** höchstens **nobj** Objekte der Grösse **size** ein (Es werden also **nobj** * **size** Bytes gelesen). fread liefert die Anzahl der eingelesenen Objekte; das kann weniger als die geforderte Zahl sein. Der Zustand des Datenstroms muss mit feof und ferror untersucht werden.

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)

fwrite schreibt **nobj** Objekte der Grösse **size** aus dem Array **ptr** in **stream** (Es werden also **nobj** * **size** Bytes geschrieben). Die Funktion liefert die Anzahl der ausgegebenen Objekte; bei Fehler ist das weniger als **nobj**.

19.5.6 Positionieren in Dateien

Mit diesen Funktionen kann die aktuelle Schreib oder Leseposition in der Datei abgefragt oder gesetzt werden. Dies macht vor allem bei direkter Ein- und Ausgabe Sinn.

int fseek(FILE *stream, long offset, int origin)

fseek setzt die Dateiposition für **stream**; eine nachfolgende Lese- oder Schreiboperation wird auf Daten von der neuen Position an zugreifen. Für eine binäre Datei wird die Position auf **offset** Zeichen relativ zu **origin** eingestellt; dabei können für **origin** die Werte SEEK_SET (Dateianfang) SEEK_CUR (aktuelle Position) oder SEEK_END (Dateiende) angegeben werden. Für einen Textstrom muss **offset** Null sein oder ein Wert, der von ftell stammt (dafür muss dann **origin** den Wert SEEK_SET erhalten). fseek liefert einen von Null verschiedenen Wert bei Fehler.

long ftell(FILE *stream) ftell liefert die aktuelle Dateiposition für **stream** oder -1L bei Fehler. ftell ist somit das Gegenstück zu fseek.

void rewind(FILE *stream) rewind(fp) ist äquivalent zu fseek(fp, OL, SEEK_SET); clearerr(fp); Es setzt den Dateizeiger auf den Anfang der Datei zurück. Der nächste Schreib- oder Lesezugriff erfolgt am Dateianfang.

int fgetpos(FILE *stream, fpos_t *ptr) fgetpos speichert die aktuelle Position für **stream** bei ***ptr**. Der Wert kann später mit fsetpos verwendet werden. Der Datentyp fpos_t eignet sich zum Speichern von solchen Werten. Bei Fehler liefert fgetpos einen von Null verschiedenen Wert.

int fsetpos(FILE *stream, const fpos_t *ptr)

fsetpos positioniert **stream** auf die Position, die von fgetpos in ***ptr** abgelegt wurde. Bei Fehler liefert fsetpos einen von Null verschiedenen Wert.

19.5.7 Fehlerbehandlung

Viele der Bibliotheksfunktionen notieren Zustandsangaben, z. B. wenn ein Dateiende oder ein Fehler gefunden wird. Diese Angaben können explizit gesetzt und getestet werden. Ausserdem kann der Integer-Ausdruck `errno` (der in `<errno.h>` deklariert ist) eine Fehlernummer enthalten, die mehr Information über den zuletzt aufgetretenen Fehler liefert.

void clearerr(FILE *stream) clearerr löscht die Dateiende- und Fehlernotizen für **stream**.

int feof(FILE *stream) feof liefert einen von Null verschiedenen Wert, wenn für **stream** ein Dateiende notiert ist. (Also das Dateiende erreicht wurde).

int ferror(FILE *stream) ferror liefert einen von Null verschiedenen Wert, wenn für **stream** ein Fehler notiert ist.

void perror(const char *s) perror(**s**) gibt **s** und eine von der Implementierung definierte Fehlermeldung aus, die sich auf die Fehlernummer in errno bezieht. Die Ausgabe erfolgt im Stil von fprintf(stderr, "%s: %s\n", s, "Fehlermeldung")

19.6 Aufgaben

Aufgabe 19.1

Schreiben Sie ein Programm, welches das aktuelle Datum und die aktuelle Uhrzeit im Sekundentakt ausgibt.

Aufgabe 19.2

Schreiben Sie ein Programm, welches eine eingegebene Zahl in Klartext ausgibt, also wenn zum Beispiel die Zahl 123 eingegeben wird, soll die Ausgabe `'einhundertdreiundzwanzig'` lauten.

Aufgabe 19.3

Schreiben Sie ein Programm, das Zeilen aus einer Datei liest und sie in eine zweite Datei schreibt, wobei vor jeder Zeile noch die Zeilennummer eingefügt wird. Am Schluss soll die Anzahl der bearbeiteten Zeilen ausgegeben werden. (Die Ausgabedatei hat am Schluss den gleichen Inhalt wie die Eingabedatei, nur steht vor jeder Zeile eine Zeilennummer.) Die Zeilennummer soll immer dreistellig sein und bei kleinen Zahlen führende Nullen enthalten. Die Numerierung soll mit 001 beginnen. Zwischen der Nummer und der Zeile soll mindestens 1 Leerzeichen stehen. Die Namen der Dateien dürfen Sie im Programm fest vorgeben. (Keine Eingabe durch den Benutzer).

Beispiel:

Eingabe Datei	Ausgabe Datei
Aller anfang **ist schwer** **aber Uebung** **macht** **den Meister**	**001 aller anfang** **002 ist schwer** **003 aber Uebung** **004 macht** **005 den Meister**

20 Modulares Programmieren

Jedes grössere Programm wird normalerweise in mehrere *Module* (Dateien) aufgeteilt. Ein Modul ist eine Sammlung von Funktionen oder Prozeduren, die zusammen eine logische Einheit bilden. Jedes Modul hat eine *Schnittstelle*. Die Schnittstelle definiert die Funktionen, welche von dem Modul gegen aussen zur Verfügung gestellt werden.

Ein Modul enthält üblicherweise *lokale* Variablen und Funktionen/Prozeduren, welche nur innerhalb des Moduls verwendet werden und von aussen nicht zugänglich sind, sowie *globale* Funktionen und Variablen, auf welche von anderen Modulen aus zugegriffen werden kann.

(Globale Variablen sollten allerdings vermieden werden, da sie die Wartbarkeit eines Programmes enorm erschweren).

Ein Beispiel für ein Modul wäre stdio.h. Die Funktion **`printf()`** gehört zur Schnittstelle dieses Moduls, aber innerhalb dieses Moduls gibt es noch lokale Funktionen, die z. B. die Umwandlung von Zahlen in Text vornehmen und die von aussen nicht zugänglich sind.

Ein Modul sollte möglichst wenig Elemente global definieren. Das hat den Vorteil, dass Änderungen an lokalen Funktionen keine direkten Auswirkungen auf andere Module haben können. (Das Ändern von globalen Funktionen, insbesondere wenn Namen, Typ oder Anzahl von Argumenten oder Typ des Rückgabewertes ändern, betrifft allen Code, der die entsprechenden Funktionen benutzt.)

20.1 Modulschnittstelle

Die Schnittstelle eines Moduls(.c) wird normalerweise in seiner Headerdatei (.h) definiert.

20.2 Globale Variablen

In C sind alle Variablen global, welche nicht explizit als lokal (**`static`**) definiert werden. In einem anderen Modul definierte Variablen müssen in jedem Modul in dem sie verwendet werden als **`extern`** deklariert werden (Geschieht normalerweise durch Einbinden der Headerdatei):

```
extern int GlobalVar;  /* Reserviert keinen Speicherplatz */
extern float f;        /* die Variable ist in einem anderen Modul */
```

Üblicherweise wird das Schlüsselwort **`extern`** in Headerdateien (.h) verwendet und nicht innerhalb der Implementationsdatei (.c).

20.3 Lokale Variablen.

In C werden lokale Variablen durch das Voranstellen des Schlüsselwortes **`static`** definiert:

```
static int LocalVar;  /* Modul lokale Variable, nicht innerhalb eines Blockes */
static float f;       /* definiert, gilt für die ganze Datei (Modul)          */
```

20.4 Globale Funktionen

In C sind alle Funktionen global, wenn sie nicht explizit als lokal (**`static`**) definiert werden. Funktionen, die in einem anderen Modul definiert sind, sollten in jedem Modul in dem sie verwendet werden als externer Prototyp aufgeführt werden (**`extern`** deklariert werden) :

```
extern int f1(int a, int b) /* Funktion ist woanders definiert */
```

Üblicherweise werden externe Prototypen in Headerdateien (.h) deklariert.

20.5 Lokale Funktionen.

In C werden lokale Funktionen durch das Voranstellen des Schlüsselwortes **`static`** definiert:

```
static int f2(int a);
```

20.6 Beispiel eines kleinen C-Projektes

Dieses Projekt besteht aus den beiden Modulen (Dateien) main.c und sayhello.c. Main.c benutzt die Funktion SayHelloWorld() und die globale Variable SayAllCount vom Modul sayhello.c und muss deshalb die Headerdatei sayhello.h mit '#include' einbinden. In dieser Headerdatei sind die Schnittstellen des gleichnamigen c-moduls deklariert. Lokale Funktionen und Variablen des Moduls sayhello.c sind als static deklariert und von Ausserhalb nicht zugreifbar.

sayhello.h

```
/* exported (global) function */
extern void SayHelloWorld(void);

/* exported (global) variable */
extern int SayAllCount;
```

main.c

```
#include "sayhello.h"

int main(int argc, char *argv[])
{
    while(SayAllCount < 5) {
       SayHelloWorld();
    }
    return 0;
}

/* Die Variable Count sowie die */
/* Funktionen SayHello() und    */
/* SayWorld() sind von hier     */
/* aus nicht zugreifbar         */
```

sayhello.c

```
#include <stdio.h>
#include "sayhello.h"

/* Prototypes (local functions) */
static void SayHello(void);
static void SayWorld(void);

/* Global Variables */
int SayAllCount = 0;

/* Local Variables */
static int Count = 0;

/* Function Definitions */
void SayHelloWorld(void)
{
   SayHello();
   printf(" ");
   SayWorld();
   printf("\n");
   SayAllCount++;
}

static void SayHello(void)
{
   printf("Hello");
   Count++;
}

static void SayWorld(void)
{
   printf("World");
   Count++;
}
```

21 Datei I/O

Das *Arbeiten mit Dateien* läuft grundsätzlich über FILE-Objekte. Sämtliche Dateifunktionen benötigen einen Zeiger auf ein solches Objekt. Eine vollständige Beschreibung aller Dateifunktionen befindet sich im Kapitel 19.5.1. Eine Datei wird in C grundsätzlich als eine Folge von Bytes betrachtet.

Bevor mit einer Datei gearbeitet werden kann, muss mit **`fopen()`** eine neues, mit dieser Datei verknüpftes FILE-Objekt erzeugt werden. **`fopen()`** liefert einen Zeiger auf das FILE-Objekt zurück, wenn das Öffnen der Datei erfolgreich war, und NULL wenn ein Fehler aufgetreten ist. Nach dem Öffnen einer Datei muss immer auf Fehler geprüft werden bevor mit ihr gearbeitet wird.

```
#include <stdio.h>

FILE *DieDatei;

DieDatei = fopen("c:\\Test.txt", "w");

if (DieDatei == NULL) {
   printf("Fehler, konnte die Datei nicht oeffnen!\n");
}
```

Das erste Argument der Funktion **`fopen()`** ist der Name der zu öffnenden Datei, das zweite die Zugriffsart auf die Datei:

"r"	Textdatei zum lesen öffnen
"w"	Textdatei zum Schreiben erzeugen; gegebenenfalls alten Inhalt wegwerfen
"a"	anfügen; Textdatei zum Schreiben am Dateiende öffnen oder erzeugen
"r+"	Textdatei zum Ändern öffnen (Lesen und Schreiben)
"w+"	Textdatei zum Ändern erzeugen; gegebenenfalls alten Inhalt wegwerfen
"a+"	anfügen; Textdatei zum Ändern öffnen oder erzeugen, Schreiben am Ende

Dateien können im Binär oder Textformat geöffnet werden, bei UNIX Betriebssystemen hat das Format keine Bedeutung, bei Dos und Windows wird im Textmodus der Zeilenvorschub ('\n') beim Schreiben in die Sequenz von Zeilenvorschub und Wagenrücklauf ('\n' '\r') umgewandelt, und beim Lesen umgekehrt die Sequenz '\n' '\r' wieder in ein einfaches '\n'. Beim Lesen im Textmodus wird unter Dos/Windows zudem das Zeichen mit dem ASCII-Code 26 (EOF, CTRL-Z) als Dateiende angesehen, der Rest der Datei nach diesem Zeichen kann somit nicht gelesen werden.

Um im Binärmodus zu arbeiten, muss den oben angegebenen Zugriffsarten noch ein 'b' angehängt werden, also z. B. "rb" oder "w+b".

Wenn die Datei nicht mehr benötigt wird, muss sie mit **`fclose()`** wieder geschlossen werden.

```
fclose (DieDatei);
```

Zur Arbeit mit Textdateien können ähnliche Funktionen wie zur Ein/Ausgabe über Bildschirm und Tastatur verwendet werden, den Funktionsnamen ist einfach ein f vorangestellt und sie haben einen File-Pointer als zusätzliches Argument:

Tastatur/Bildschirm	Funktion	Datei
`printf("Wert ist %d", i);`	Formatierte Ausgabe	`fprintf(DieDatei, "Wert ist %d",i);`
`scanf("%f", &f);`	Formatierte Eingabe	`fscanf(DieDatei, "%f", &f);`
`c = getchar();`	Zeichen einlesen	`c = fgetc(DieDatei);`
`putchar(c);`	Zeichen ausgeben	`fputc(c, DieDatei);`
`gets(Buffer);`	Zeile einlesen *)	`fgets(Buffer, MaxLength, DieDatei);`
`puts(Text);`	Zeile ausgeben *)	`fputs(Text, DieDatei);`

*) **Achtung**, Zeilenwechsel werden von den f... Funktionen anders behandelt (Nicht entfernt und hinzugefügt).

21.1 Die Standardkanäle stdin, stdout und stderr

Es gibt drei Dateien, die immer geöffnet sind: **`stdin`**, **`stdout`** und **`stderr`**. **`stdout`** und **`stderr`** sind normalerweise mit dem Bildschirm verbunden, **`stdin`** ist mit der Tastatur verbunden. Alle IO-Funktionen (wie printf, getchar, puts...) welche kein Dateiobjekt erwarten, benutzen implizit diese Dateien. Zum Beispiel hat fprintf(stdout, "Hallo") den selben Effekt wie printf("Hallo").

21.2 Beispiele

Beispiel für die Arbeit mit Textdateien

Das folgende Programm öffnet die Datei Test.txt, liest deren Inhalt zeilenweise und schreibt die gelesenen Zeilen mit vorangestellter Zeilennummer in die Datei Test.out.

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

int main(int argc, char *argv[])
{
   char *InputFileName  = "test.txt";
   char *OutputFileName = "test.out";
   FILE *InputFile, *OutputFile;

   /* Dateien oeffnen und auf Fehler Pruefen */
   InputFile = fopen(InputFileName, "r");
   if (InputFile != NULL) {
      OutputFile = fopen(OutputFileName, "w");
      if (OutputFile != NULL) {
         int ZeilenNummer = 1;

         /* Solange das Dateiende nicht erreicht ist */
         while (!feof(InputFile)) {
            char Buffer[200];

            /* Zeile einlesen... */
            if (fgets(Buffer, 200, InputFile) != NULL) {
               /* ...und mit vorangestellter Zeilennummer wieder ausgeben */
               fprintf(OutputFile, "%04d: %s", ZeilenNummer, Buffer);
            }
            ZeilenNummer++;
         }

         /* Dateien wieder schliessen */
         fclose(OutputFile);
      } else {
         printf("Sorry, could not open '%s' for write\n", OutputFileName);
         printf("Errorcode was %d\n", errno);
         perror("Errormessage: ");
      }
      fclose(InputFile);
   } else {
      printf("Sorry, could not open '%s' for read\n", InputFileName);
      printf("Errorcode was %d\n", errno);
      perror("Errormessage: ");
   }
   system("PAUSE");
   return 0;
}
```

Beispiel für die Arbeit mit binären Dateien

Das Programm gibt die Datensätze der Datei Test.bin aus (Falls vorhanden), und lässt anschliessend den Benutzer weitere Datensätze an die Datei anhängen.

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

typedef struct DataStruct {
   int   Value;
   char  String[40];
   float FValue;
} DataStruct;

int main(int argc, char *argv[])
{
   char *FileName  = "test.bin";
   FILE *File;
   /* Binaere Datei zum Lesen oeffnen */
   File = fopen(FileName, "rb");
   if (File != NULL) {
      int i = 0;
      /* Zwischenspeicher fuer die Datensaetze */
      DataStruct Input;
      /* Alle Datensaetze lesen und ausgeben */
      while (!feof(File)) {
         if (fread(&Input, sizeof(DataStruct), 1, File) == 1) {
            printf("\nEntry %d\n", i++);
            printf("  Value  = %d\n", Input.Value);
            printf("  String = %s\n", Input.String);
            printf("  Float  = %f\n", Input.FValue);
         }
      }
      /* Datei wieder schliessen */
      fclose(File);
   }
   /* Binaere Datei zum Anhaengen (Schreiben) oeffnen */
   File = fopen(FileName, "a+b");
   if (File != NULL) {
      /* Datensaetze eingeben lassen und in Datei schreiben */
      DataStruct Output;
      printf("Enter a integer Value: ");
      scanf("%d", &(Output.Value));
      printf("Enter a string: ");
      scanf("%s", &(Output.String));
      printf("Enter a float Value: ");
      scanf("%f", &(Output.FValue));
      if (fwrite(&Output, sizeof(DataStruct), 1, File) != 1) {
         printf("Sorry, could not write the datas");
      }
      /* Datei wieder schliessen */
      fclose(File);
   } else {
      printf("Sorry, could not open '%s' for append\n", FileName);
      printf("Errorcode was %d\n", errno);
      perror("Errormessage: ");
   }
   system("PAUSE");
   return 0;
}
```

22 Standardargumente

Wenn ein Programm gestartet wird, erhält es vom Betriebssystem Argumente. Diese Argumente werden der Funktion **`main(int argc, char *argv[])`** in den Variablen **`argc`** und **`argv`** übergeben. Die Namen sind nicht zwingend vorgeschrieben, haben sich aber eingebürgert.

In der Variablen **`argc`** (argument count) ist die Anzahl der Argumente abgelegt, und die Variable **`argv`** (argument vector) ist ein entsprechend grosses Array aus Strings, welche die Argumente enthalten. Es können also nur Strings übergeben werden. Das erste Argument ist immer der Programmname, und zwar der effektive Namen der beim Aufruf gültig ist, inklusive dem Pfad zum Programm. (Wenn die Datei umbenannt wurde, wird der neue Name geliefert, und nicht der zur Compilezeit gültige).

Die Argumente müssen dem Programm beim Aufruf via Commandline durch Leerzeichen separiert hinter dem Programmnamen angegeben werden. Ob und wie die Argumente bei nicht Commandline-Basierten Programmen übergeben werden, hängt vom Betriebssystem ab.

Das nachfolgende Programm gibt einfach alle seine Argumente auf dem Bildschirm aus:

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

int main(int argc, char *argv[])
{
  int i;

  /* Alle Argumente ausgeben */
  for (i = 0; i < argc; i++) {
     printf("<%s>\n", argv[i]);
  }
  /* Auf Tastendruck warten */
  system("Pause");
  return 0;
}
```

Starten des Programms in der Shell:

```
C:\TEMP>printarg eins hallo -23 "eins zwei drei"
```

Ausgabe des Programms:

```
<C:\TEMP\printarg.exe>
<eins>
<hallo>
<-23>
<eins zwei drei>
Taste drücken, um fortzusetzen . . .
```

So können Programmen bereits beim Start zum Beispiel Namen von Dateien, oder Optionen übergeben werden. Sehr viele Programme lassen sich über Comandlineargumente steuern, wie zum Beispiel der Befehl 'dir' in der Dos Shell.

23 Sortieren

Einer der am häufigsten gebrauchten Algorithmen ist das Sortieren. Überlegen Sie sich als einleitendes Beispiel kurz wie Sie das folgende Array mit 5 Zahlen in aufsteigender Reihenfolge sortieren würden (Als Anweisungsfolge an eine Person):

```
int Werte[5] = {9, 17, 3, 21, 5};
```

Es gibt sehr viele verschiedene bekannte Sortierverfahren, die sich durch Effizienz und Kompliziertheit voneinander unterscheiden. Die am einfachsten zu verstehenden Verfahren sind im allgemeinen auch die am wenigsten effizienten. Einige bekannte Verfahren sind in der folgenden Tabelle aufgeführt (Siehe 'The Art of Computerprogramming', Donald E.. Knuth, Addsion Wesley):

Name	Implementation	Laufzeit		Speicher bedarf
		Durchschnitt	Maximal	
Bubblesort	Einfach	$5.75N^2+N*\ln(N)$	$7.5N^2$	N+1
Straight insertion	Einfach	$1.5N^2+9.5N$	$3N^2$	N+1
Straight Selection	Einfach	$2.5N^2+3N*\ln(N)$	$3.25N^2$	N
Quicksort	Komplex	$11.6N*\ln(N)-1.7N$	$>=2N^2$	N + ln*(N)
Heapsort	Komplex	$23.N*\ln(N)$	$24.5N*\ln(N)$	N
Listmerge	Komplex	$14.4N*\ln(N)+4.9N$	$14.4N*\ln(N)$	N

N: Anzahl der zu sortierenden Elemente
ln(): Logarithmus.

Quicksort gilt allgemein als einer der am universellsten einsetzbaren Algorithmen. Wenn das Feld allerdings bereits sortiert ist, kann Quicksort sehr langsam werden. Je nach Anwendungsfall gibt es jeweils einen optimalen Algorithmus, dies kann von den Daten, der Hardware und vom verfügbaren Speicherplatz abhängen.

Für Spezialfälle (Fast sortiertes Feld, in umgekehrter Reihenfolge sortiertes Feld) können die Algorithmen bei Bedarf oft optimiert werden.

Wir werden hier **Bubblesort**, **Straight Insertion, Straight Selection** und **Quicksort** näher betrachten:
Bei allen Algorithmen werden wir in aufsteigender Reihenfolge sortieren. Selbstverständlich können diese Algorithmen durch entsprechende Modifikation auch in die andere Richtung sortieren.

23.1 Bubblesort:

Bubblesort ist einer der einfachsten Sortieralgorithmen, er ist aber auch nicht gerade sehr effizient.
Vorgehen: Wir wandern durch das zu sortierende Feld und vergleichen nacheinander immer das aktuelle Feld mit seinem Nachbarn, und wenn die Reihenfolge der beiden Elemente nicht stimmt werden sie vertauscht. Das wird solange wiederholt bis das Feld sortiert ist. Der Algorithmus heisst Bubblesort, weil in jedem Durchgang die noch verbliebene grösste Zahl wie eine Luftblase an ihren korrekten Platz aufsteigt.

Da nach jedem Durchgang ein Element mehr bereits am korrekten Platz ist, kann der Vorgang so optimiert werden, das die bereits korrekt plazierten Elemente nicht mehr weiter sortiert werden. Es wird nicht jedesmal bis zum Ende des Feldes getauscht, sondern nur bis zum Beginn des bereits sortierten Bereichs.

Nicht Optimiert

Pass	0	1	2	3	4
0	5	9	1	7	6
1.1	5	***9***	1	7	6
1.2	5	***9***	1	7	6
1.3	5	1	***9***	7	6
1.4	5	1	7	***9***	6
2.1	5	1	***7***	6	**9**
2.2	1	5	***7***	6	**9**
2.3	1	5	***7***	6	**9**
2.4	1	5	6	***7***	**9**
3.1	1	5	***6***	**7**	**9**
3.2	1	5	***6***	**7**	**9**
3.3	1	5	***6***	**7**	**9**
3.4	1	5	***6***	**7**	**9**
4.1	1	***5***	**6**	**7**	**9**
4.2	1	***5***	**6**	**7**	**9**
4.3	1	***5***	**6**	**7**	**9**
4.4	1	***5***	**6**	**7**	**9**

Optimiert

Pass	0	1	2	3	4
0	5	9	1	7	6
1.1	5	***9***	1	7	6
1.2	5	***9***	1	7	6
1.3	5	1	***9***	7	6
1.4	5	1	7	***9***	6
2.1	5	1	***7***	6	**9**
2.2	1	5	***7***	6	**9**
2.3	1	5	***7***	6	**9**
3.1	1	5	***6***	**7**	**9**
3.2	1	5	***6***	**7**	**9**
4.1	1	***5***	**6**	**7**	**9**

Aktueller Vergleich
Bereits Sortiert
Grösstes Element in diesem Durchlauf

Der Algorithmus braucht im optimierten Fall (N-1)*(N / 2) Vergleiche sowie im Durchschnitt halb so viele Vertauschungen (Im Schnitt muss nur jedes 2. Mal vertauscht werden). Eine Vertauschung kostet allerdings meistens 2-3mal mehr Rechenzeit als ein Vergleich.

Das Feld ist sortiert, sobald in einem Durchlauf keine Vertauschung mehr durchgeführt werden musste. Dies ist garantiert nach N Durchläufen der Fall.

23.2 Straight Insertion:

Direktes Einfügen ist ebenfalls einer der einfacheren Algorithmen. Er ist etwas effizienter als Bubblesort.

Vorgehen: Das sortierte Feld wächst von vorne nach hinten. Wir wandern durch das zu sortierende Feld und fügen jedes Element an seiner korrekten Position im bereits sortierten Teilfeld ein. Das Teilfeld wird von der Einfügeposition an nach hinten verschoben.

Wenn wir das letzte Element eingefügt haben, ist das Feld sortiert.

Der Algorithmus heisst **Straight Insertion**, weil jedes Element direkt dort eingefügt wird wo es hingehört.

Pass	0	1	2	3	4	5
0	5	*9*	1	7	6	4
1.1	**5**	***9***	1	7	6	4
2.1	**5**	**9**	***1***	7	6	4
2.2	**5**	**9**	***1***	7	6	4
2.3	**5**	**9**	***1***	7	6	4
3.1	**1**	**5**	**9**	***7***	6	4
3.2	**1**	**5**	**9**	***7***	6	4
3.3	**1**	**5**	**9**	***7***	6	4
3.4	**1**	**5**	**9**	***7***	6	4
3.5	**1**	**5**	**9**	***7***	*6*	4
4.1	**1**	**5**	**7**	**9**	***6***	4
4.2	**1**	**5**	**7**	**9**	***6***	4
4.3	**1**	**5**	**7**	**9**	***6***	4
4.4	**1**	**5**	**7**	**9**	***6***	4
4.5	**1**	**5**	**7**	**9**	***6***	4
5.1	**1**	**5**	**6**	**7**	**9**	***4***
5.2	**1**	**5**	**6**	**7**	**9**	***4***
5.3	**1**	**5**	**6**	**7**	**9**	***4***
5.4	**1**	**5**	**6**	**7**	**9**	***4***
	1	**4**	**5**	**6**	**7**	**9**
	1	**4**	**5**	**6**	**7**	**9**

Bereich zum Schieben
Aktueller Vergleich
Bereits Sortiert
aktuelles Element

Der Algorithmus braucht durchschnittlich (N-1)*(N / 4) Vergleiche und muss im Durchschnitt etwa ebensoviele Elemente verschieben. (Das Suchen der Einfügeposition könnte mit binary search erfolgen, damit wären nur noch etwa log2(N!) *) Vergleiche nötig, aber die Anzahl der Verschiebungen bleibt nach wie vor bei (N-1) * (N / 4)).

*) log2(N!) = log2(1) + log2(2) + log2(3) + ... + log2(N)

23.3 Straight Selection:

Direktes Auswählen gehört auch zu den einfachen Algorithmen. Auch dieser Algorithmus ist etwas effizienter als Bubblesort.

Vorgehen: Das sortierte Feld wächst von vorne nach hinten. Wir suchen jeweils im unsortierten Teil des Feldes den kleinsten Wert, und hängen diesen hinten an den sortierten Teil des Feldes an. Das Anhängen geschieht, indem wir das Element direkt hinter dem sortierten Feld mit dem neu gefundenen Element vertauschen.

Wenn wir das zweitletzte Element des Restfeldes eingefügt haben, ist das Feld sortiert.

Der Algorithmus heisst **Straight Selection**, weil immer das Element zum Anhängen ausgewählt wird, das auch genau dorthin gehört.

Pass	0	1	2	3	4	5
0	5	*9*	7	1	6	4
1.1	5	*9*	7	***1***	6	4
2.1	**1**	*9*	7	5	6	4
2.2	**1**	*9*	7	5	6	***4***
2.3	**1**	**4**	7	5	6	9
3.1	**1**	**4**	7	***5***	6	9
3.2	**1**	**4**	**5**	7	6	9
3.3	**1**	**4**	**5**	7	***6***	9
3.4	**1**	**4**	**5**	**6**	7	9
3.5	**1**	**4**	**5**	**6**	***7***	9
4.1	**1**	**4**	**5**	**6**	**7**	9
4.2	**1**	**4**	**5**	**6**	**7**	**9**
	1	**4**	**5**	**6**	**7**	**9**

Aktueller Tausch

Bereits Sortiert

aktuell kleinstes Element

Der Algorithmus braucht durchschnittlich (N-1)*(N / 4) Vergleiche zum Suchen des kleinsten Elementes und muss etwa (N-1) Elemente Vertauschen.

23.4 Hinweis

Die drei hier vorgestellten Algorithmen haben alle ein gemeinsames Problem, nämlich dass die benötigte Rechenzeit quadratisch zur Anzahl der zu sortierenden Elemente zunimmt. Es gibt bessere Algorithmen, deren Rechenaufwand nur mit N*log(N) zunimmt. (Z. B. Quicksort oder Heapsort).

23.5 Aufgabe 23.1

Implementieren Sie im Projekt Sortieren **eine** der Sortierfunktionen im Modul Sortfun.c. Das Projekt finden Sie unter …\Data\Aufgaben\Sortieren.

23.6 Quicksort:

Quicksort ist einer der effizientesten und wahrscheinlich auch am häufigsten eingesetzten Sortieralgorithmen. Die Funktionsweise dieses Sortierverfahrens ist nicht allzukompliziert, aber es ist dennoch nicht so einfach, dieses auch fehlerfrei zu implementieren. Deshalb greift man meist auf entsprechende Bibliotheksfunktionen zurück (z. B. qsort() in stdlib.h).

Die Idee des Algorithmus ist sehr einfach:

Man sucht sich zufällig ein Element aus dem Datenfeld aus (Oft als 'Pivot' bezeichnet), und zerteilt das Feld anschliessend in zwei Teile, und zwar so, dass im einen Teil nur Zahlen grösser als der Pivot, und im anderen nur Zahlen kleiner oder gleich dem Pivot enthalten sind. Anschliessend werden die beiden Teile mit dem genau gleichen Verfahren weiterbehandelt. Dies wiederholt man so oft bis die Grösse des Teilfelder 1 wird. Somit lässt sich Quicksort am einfachsten rekursiv implementieren.

Im Optimalfall würde das Feld in jedem Schritt exakt halbiert werden, was log2(N) Schritte ergeben würde, und jedem Schritt müssen etwa N-1 Vergleiche zwischen Zahlen und Pivot durchgeführt werden. Im optimalen Fall sind somit nur N*log2(N) Vergleiche nötig. Da das Feld jedoch im Allgemeinen nicht in der Mitte halbiert wird, sondern an einer zufälligen Stelle, ergeben sich im Durchschnitt ln(N) * N oder 1.38*N*log2(N) Vergleiche, also etwa 38% mehr Aufwand (Siehe Robert Sedgewick, Algorithmen in C++).

Im schlimmsten Fall wird immer das kleinste oder grösste Element als Pivot gewählt, somit wird das Feld nicht geteilt, sondern nur um 1 vermindert. In diesem Fall benötigt der Algorithmus N*N Vergleiche. Es gibt einige Verbesserungen, um diese Fälle zu vermeiden (Sedgewick, Knuth).

Der Algorithmus heisst **Quicksort**, weil es eines der schnellsten Sortierverfahren ist.

Nachfolgend sind die Arbeitsschritte von Quicksort an einem Beispiel dargestellt. Es wird immer das mittlerste Element des Feldes als Pivot verwendet.

Pass	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1.0	10	13	2	17	12	1	21	8	20	5	**6**	11	19	18	15	4	9	3	7	14	16
1.1	3	4	2	5	1	**6**	12	21	8	20	17	11	19	18	15	13	9	10	7	14	16
2.0	3	4	**2**	5	1	**6**	12	21	8	20	17	11	19	**18**	15	13	9	10	7	14	16
2.1	1	**2**	4	5	3	**6**	12	16	8	14	17	11	7	15	13	9	10	**18**	19	20	21
3.0	1	**2**	4	**5**	3	**6**	12	16	8	14	17	**11**	7	15	13	9	10	**18**	19	**20**	21
3.1	1	**2**	4	**5**	3	**6**	10	9	8	7	**11**	17	14	15	13	16	12	**18**	19	**20**	21
4.0	1	**2**	**4**	3	**5**	**6**	10	**9**	8	7	**11**	17	14	**15**	13	16	12	**18**	19	**20**	21
4.1	1	**2**	3	**4**	**5**	**6**	7	8	**9**	10	**11**	12	14	13	**15**	16	17	**18**	19	**20**	21
5.0	1	**2**	3	**4**	**5**	**6**	**7**	8	**9**	10	**11**	12	**14**	13	**15**	**16**	17	**18**	19	**20**	21
5.1	1	**2**	3	**4**	**5**	**6**	**7**	8	**9**	10	**11**	12	13	**14**	**15**	**16**	17	**18**	19	**20**	21
6.0	1	**2**	3	**4**	**5**	**6**	**7**	8	**9**	10	**11**	**12**	13	**14**	**15**	**16**	17	**18**	19	**20**	21
6.1	1	**2**	3	**4**	**5**	**6**	**7**	8	**9**	10	**11**	**12**	13	**14**	**15**	**16**	17	**18**	19	**20**	21
7.0	1	**2**	3	**4**	**5**	**6**	**7**	8	**9**	10	**11**	**12**	13	**14**	**15**	**16**	17	**18**	19	**20**	21
7.1	1	**2**	3	**4**	**5**	**6**	**7**	8	**9**	10	**11**	**12**	13	**14**	**15**	**16**	17	**18**	19	**20**	21

Bereits am richtigen Platz / Sortiert

Pivot

Pivot aus früherer Runde, am richtigen Platz/Sortiert

Ein Problem von Quicksort ist, dass es nicht stabil ist, d.h. eine ursprüngliche Ordnung bleibt nicht erhalten, man kann also nicht zuerst nach Vornamen, und dann nach Nachnamen sortieren um Leute mit gleichem Nachnamen nach Vornamen sortiert zu erhalten.

24 Suchen

Suchen von Werten innerhalb von Datenbeständen gehört wie das Sortieren zu den wichtigsten Algorithmen in der Informatik.
Der Wert nach dem gesucht wird, bezeichnet man meist als Schlüssel.

24.1 Sequentielles Suchen

Die einfachste, und manchmal auch die einzige Möglichkeit, einen Eintrag in einem Feld zu suchen ist die sequentielle Suche.
Dabei werden schlichtweg alle Elemente vom Anfang bis zum Ende mit dem gesuchten Wert verglichen. Im Durchschnitt sind dafür N/2 Vergleiche nötig, wenn das Element im Feld enthalten ist, und immer N Vergleiche um festzustellen dass der Wert nicht im Feld enthalten ist.

Dieser Algorithmus kann noch optimiert werden: Anstelle jedes mal innerhalb der Suchschlaufe zu prüfen ob man das Ende des Feldes erreicht hat, kann man am Ende des Feldes den gesuchten Wert anhängen. Damit wird die Schlaufe auf jeden Fall beendet, ohne dass ein zusätzlicher Test auf erreichen des Ende des Feldes nötig ist. Das kann einen merklichen Geschwindigkeitszuwachs zur Folge haben.

Wenn der Datenbestand nicht sortiert ist, ist das Sequentielle Suchen die einzige Suchmöglichkeit, und auch in Listen kann nur sequentiell gesucht werden

24.2 Sequentielles Suchen in sortierten Feldern

Sequentielles Suchen in sortierten Feldern funktioniert wie das normale sequentielle Suchen, nur dass man die Suche auf jeden Fall abbrechen kann, sobald man ein Element grösser oder gleich dem gesuchten Wert erreicht. Im Durchschnitt sind hier N/2 Vergleiche nötig, unabhängig davon ob das gesuchte Element im Feld enthalten ist oder nicht.

24.3 Binäres Suchen in sortierten Feldern

Bei sortierten Feldern drängt sich das binäre Suchen direkt auf, es ist analog zum Suchen im Lexikon: Man beginnt nicht zuvorderst, sondern in der Mitte und blättert in immer kleineren Schritten vorwärts oder rückwärts.

Man vergleicht den zu suchenden Wert (Schlüssel) mit dem Wert in der Mitte des Feldes. Wenn der Wert grösser ist, sucht man in der vorderen Hälfte weiter, sonst in der hinteren Hälfte. Man wendet dazu einfach das vorige Vorgehen erneut auf die passende Hälfte an. Das wird solange wiederholt bis man den Wert gefunden hat, oder sich das Feld nicht mehr halbieren lässt. Dieser Algorithmus braucht maximal log2(N) Vergleiche. Er funktioniert jedoch nur bei bereits sortierten Feldern.

Als Beispiel suchen wir in der folgenden Tabelle den Wert 68

7	12	13	33	39	41	42	43	48	50	51	53	57	67	68	74	77	83	85	87	96
7	12	13	33	39	41	42	43	48	50	**51**	53	57	67	68	74	77	83	85	87	96
7	12	13	33	39	41	42	43	48	50	51	53	57	67	68	74	**77**	83	85	87	96
7	12	13	33	39	41	42	43	48	50	51	53	57	**67**	68	74	77	83	85	87	96
7	12	13	33	39	41	42	43	48	50	51	53	57	67	68	**74**	77	83	85	87	96
7	12	13	33	39	41	42	43	48	50	51	53	57	67	**68**	74	77	83	85	87	96
7	12	13	33	39	41	42	43	48	50	51	53	57	67	68	74	77	83	85	87	96

Wir vergleichen 68 zuerst mit dem mittleren Element (**51**) der Tabelle und stellen fest das wir in der oberen Hälfte des Feldes weitersuchen müssen. Wir vergleichen 68 erneut mit dem mittleren Element (**77**) des verbliebenen Feldes und stellen fest, das wir in dessen unterer Hälfte weitersuchen müssen. Und so landen wir nacheinander noch bei **67** und **74** bis wir im fünften Schritt den gesuchten Wert **68** gefunden haben.

Die Effizienz dieses Algorithmus beruht darauf, dass das Feld in jedem Suchschritt halbiert wird und die Suche sich so sehr schnell dem Ziel nähert.

Dieser Algorithmus lässt sich sehr schön rekursiv Implementieren: Solange das Element nicht gefunden ist, wendet die Funktion sich selbst auf die entsprechende übriggebliebene Hälfte an.

Aufgabe 24.1:

Implementieren Sie ein Modul Binary.c mit der Funktion BinarySearch(), welche in einem sortierten Array den angegebenen Wert sucht und einen Zeiger auf sein erstmaliges Auftreten zurückliefert.

25 Rekursion

Rekursion bedeutet, dass sich ein Programm selbst aufruft. Viele Probleme lassen sich sehr elegant rekursiv Lösen, insbesondere alle, die sich auf einen vorgängig zu bestimmenden Wert beziehen.

Ein bekanntes, wenn auch eher sinnloses Beispiel ist das Berechnen der Fakultät nach der *Rekursionsformel* n! = n * (n-1)! und der Anfangsbedingung 0! = 1.Dies würde wie folgt implementiert werden:

```
int Fakultaet(int n)
{
   if (n > 0) {
      return n*Fakultaet(n-1);  /* Rekursiver Aufruf */
   }else {
      return 1;
   }
}
```

Allerdings ist in diesem Fall eine Iteration (Schlaufe) eindeutig die bessere Lösung.

Achtung: Eine Rekursion kann sehr viel Stack-Speicher benötigen, und bei Systemen mit wenig Stackspeicher zum Programmabsturz führen. Iterative Ansätze sind normalerweise (Geschwindigkeits-) Effizienter als Rekursionen, Rekursionen sind dagegen oft einfacher zu Implementieren und zu Verstehen.

Jede Rekursion kann mit dem entsprechenden Aufwand in eine Iteration umgewandelt werden. In sicherheitskritischen Anwendungen (z. B. Autopilot) sollte auf Rekursion verzichtet werden.

Eine Rekursion muss eine garantierte Abbruchbedingung besitzen, damit sie ein Ende findet und nicht ewig läuft (Und somit garantiert abstürzt).

Eine Rekursion ist im Gegensatz zu einer Schlaufe durch den verfügbaren Stack-Speicher limitiert. Wenn eine Rekursion länger läuft als Speicher zur Verfügung steht, stürzt das Programm unweigerlich ab.

Ein weiteres Beispiel für eine Rekursion ist die Ausgabe einer Zahl. Die Einerstelle einer Zahl kann mit der Modulo-Operation sehr einfach bestimmt werden, nur müssen zuvor noch die restlichen Ziffern der Zahl ausgegeben werden. Also wenn man 4285 Ausgeben will ergibt sich folgender Ablauf:

Ablauf

```
4285 muss noch ausgegeben werden
   5 ist die Einerstelle, 428 muss noch ausgegeben werden
      8 ist nun die Einerstelle, 42 muss noch ausgegeben werden
         2 ist nun die Einerstelle, 4 muss noch ausgegeben werden
            4 ist nun die Einerstelle, keine weiteren Ausgaben nötig
            4 ausgeben      4
         2 ausgeben         42
      8 ausgeben            428
   5 ausgeben               4285
```

Die Einrückung entspricht der Verschachtelung der rekursiven Auf-

Code

```
void PrintZahl(int x)
{
  if (x > 0) {
    PrintZahl(x/10);
    putchar('0' + x % 10);
  }
}

int main (int c, char *a[])
{
   PrintZahl(4285);
   return 0;
}
```

Aufgabe 25.1:

Schreiben Sie ein Programm, das eine Zeile einliest und umgekehrt wieder ausgibt. Verwenden Sie dazu Rekursion. (Tip: Lesen Sie die Zeile zeichenweise [getchar()]).

Eingabe: **`Hallo Welt`** *Ausgabe:* **`tleW ollaH`**

25.1 Turm von Hanoi

Ein schönes Beispiel für die Anwendung der Rekursion ist das Problem vom Turm von Hanoi:
Es soll ein Stapel von immer kleiner werdenden Scheiben von einem Platz auf einen anderen Platz verschoben werden. Dazu steht ein weiterer Platz als Zwischenablage zur Verfügung. Die Scheiben dürfen nur auf einem der drei Plätze abgelegt werden, es darf immer nur eine Scheibe auf einmal verschoben werden und es darf nie eine grössere Scheibe auf einer kleineren zu liegen kommen. Zu Beginn steht der Turm auf dem linken Platz, am Ende soll er auf dem rechten Platz stehen.

Das Problem lässt sich elegant rekursiv Formulieren:

> Um den Turm der Höhe n auf das rechte Feld zu bekommen, muss zuerst der Teilturm aus den oberen (n-1) Scheiben in die Mitte plaziert werden, dann die unterste Scheibe auf den rechten Platz verschoben werden und anschliessend wird der Teilturm von der Mitte auch auf den rechten Platz verschoben.

Das bedeutet, um einen Turm der Höhe n zu verschieben, müssen wir zweimal einen Turm der Höhe (n-1) und einmal eine Scheibe verschieben. Das Verschieben eines Teilturmes kann für sich alleine wie das Verschieben des ganzen Turms betrachtet werden und deshalb auch mit der oben beschriebenen Methode behandelt werden, ausser die Höhe des Teilturms ist 0, aber das Verschieben eines Turmes der Höhe 0 ist trivial, es muss schlicht nichts gemacht werden.

Diese Vorschrift kann auch von menschlichen Spielern zum Lösen des Problems benutzt werden.

Aufgabe 25.2:
Implementieren Sie ein Programm, welches das Problem des Turmes von Hanoi für beliebige Turmhöhen löst. Das Ausgangsprojekt finden Sie unter …\Data\Aufgaben\Hanoi. Sie müssen nur die Funktion MoveTower() in der Datei hanoi.c implementieren.

25.2 Weitere Einsatzgebiete für Rekursionen

Einige Beispiele zum Einsatz von Rekursion sind:

Lexikalische Analysen (Parser, lesen und umsetzen von mathematischen Formeln oder auch Programmen, z. B. C-Code bei einem C-Compiler),

Zuggeneratoren bei Spielen wie z. B. Schachprogrammen, bei denen der Computer Zugfolgen von Spieler und Computer bis zu einer gewissen Tiefe durchspielt und den besten Zug auswählt.

Wegsuche in einem Labyrinth und andere Backtracking-Algorithmen. (Bei Backtracking wird im Prinzip etwas ausprobiert bis es nicht mehr geht, dann wird die letzte Entscheidung rückgängig gemacht und wieder alles durchprobiert, wenn es immer noch nicht geht die zweitletzte rückgängig gemacht usw. bis das Ziel erreicht ist oder man wieder am Anfang steht).

Allgemein alle Probleme welche in kleinere, gleichartige Teilprobleme zerlegt werden können (Wird oft als 'Teile und herrsche' bezeichnet.) Ein Beispiel dafür ist Quicksort).

26 Dynamische Speicherverwaltung

Häufig ist beim Schreiben und Compilieren eines Programmes nicht bekannt, wieviel Daten dereinst von diesem Programm bearbeitet werden. Deshalb ist es bei den internen Datenstrukturen oft auch nicht möglich, im Voraus eine vernünftige, fixe Grösse für den Speicherplatz zu wählen.

Man kann natürlich alle Arrays auf Vorrat masslos überdimensionieren (wie zum Beispiel: **`char TextVonEditor[1000000]`**), nur ist damit immer noch nicht garantiert dass nun in jedem Fall genügend Platz vorhanden ist, und zudem wird dadurch im Normalfall viel zuviel Speicher verschwendet.

Es muss deshalb möglich sein, zu Laufzeit des Programmes Speicher vom Betriebssystem anzufordern wenn er benötigt wird, und diesen Speicher auch wieder zurückzugeben wenn er nicht mehr benutzt wird.

Dieses dynamische Anfordern und Freigeben von Speicher bezeichnet man als *dynamische Speicherverwaltung*. In C stehen dazu grundsätzlich die Funktionen **`malloc()`** zum Allozieren von Speicher, und **`free()`** zum Freigeben von Speicher zur Verfügung. (Zur Anforderung von Speicher gibt es für spezielle Fälle noch die Funktionen **`realloc()`** und **`calloc()`**). Der Speicherbereich, aus dem dieser dynamisch verwaltete Speicher entnommen wird, bezeichnet man als '*Heap*'.

Der Funktion **`malloc()`** muss die Grösse des gewünschten Speichers übergeben werden, Sie liefert einen Zeiger auf den nun reservierten Speicherbereich zurück. Wenn das System nicht mehr genügend Speicher frei hat, um die Anforderung zu erfüllen, wird ein **`NULL`**-Zeiger zurückgegeben. Es ist deshalb unerlässlich, nach **jeder** Anforderung von Speicher zu überprüfen, ob man den Speicher auch erhalten hat.

Mit der Funktion **`realloc()`** kann ein bereits reservierter Speicherblock 'erweitert' werden. Dabei wird vom System ein neuer Block mit der neuen Grösse reserviert, soviel Daten wie möglich (Maximal soviel wie der neue Block aufnehmen kann, resp. im alten enthalten ist) vom alten Block in den neuen Block kopiert, der alte Block freigegeben und ein Zeiger auf den neuen Block zurückgeliefert. Der Funktion muss ein Zeiger auf den alten Block, sowie die gewünschte neue Grösse mitgegeben werden. Diese Funktion liefert ebenfalls einen **`NULL`** Zeiger zurück, wenn der angeforderte Speicher nicht zur Verfügung gestellt werden konnte (Der alte Block wird in diesem Fall aber nicht freigegeben). Diese Funktion kann auch nachgebildet werden, indem man das Allozieren, Kopieren und Freigeben selbst programmiert, nur kann realloc in gewissen Fällen effizienter sein (Wenn z.B. der Speicher hinter diesem Block noch frei ist, kann der Block einfach erweitert werden, und es muss nichts kopiert werden.)

Wenn man einen reservierten Speicherblock nicht mehr benötigt, muss man ihn mit der Funktion **`free()`** wieder freigeben, damit er anderen Programmen zur Verfügung gestellt werden kann. Der Funktion **`free()`** muss man den Zeiger auf den freizugebenden Speicherblock übergeben (Derselbe Zeiger, den man von **`malloc()`**, **`calloc()`** oder **`realloc()`** erhalten hat).

Achtung, wenn ein Zeiger übergeben wird, der nicht auf einen von **`malloc()`** gelieferten Speicherblock zeigt, oder denselben Block zweimal freigibt wird die Speicherverwaltung *korrupt*, und es muss mit Absturz oder verfälschten Daten gerechnet werden. (Um solche Probleme zu finden gibt es spezielle Debug-Speicherverwaltungen, welche zusätzliche Überprüfungen vornehmen und Fehlermeldungen ausgeben.)

Wichtig, jeder Speicherblock, der angefordert wird, muss auch wieder freigegeben werden, sonst entsteht ein sogenanntes *Speicherleck*. Das heisst, dass der verfügbare Speicher ständig abnimmt, bis plötzlich kein freier Speicher mehr übrig ist.

Innerhalb von Interruptroutinen und zeitkritischen Codestücken sollten **`malloc()`** und **`free()`** nicht eingesetzt werden. (Speicher vorher oder nachher allozieren resp. freigeben).

Ein einfacheres Beispiel für die Benutzung von dynamischem Speicher:

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

int main(int argc, char*argv[])
{
   double *Werte;
   int     Size;
   int     i;
   double  Eingabe;

   /* Herausfinden wieviel Platz benoetigt wird */
   printf("Wieviele Zahlen moechten Sie eingeben? ");
   scanf("%d", &Size);
   if (Size < 0) {
      return 0;
   }

   /* Den benoetigten Platz reservieren */
   Werte = malloc(sizeof(double) * Size);
   if (Werte == NULL) {
      printf("Error, out of memory");
      return 0;
   }

   /* Die Werte einlesen */
   for (i = 0; i < Size; i++) {
      printf("Bitte geben Sie die Zahl %d (von %d) ein:", i, Size);
      scanf("%lf", &Eingabe);
      Werte[i] = Eingabe;
   }

   /* und in umgekehrter Reihenfolge wieder ausgeben */
   for (i = Size-1; i >= 0; i--) {
      printf("Die Zahl %d lautet: %f\n", i, Werte[i]);
   }

   /* und zum Schluss den Speicherblock wieder freigeben */
   free(Werte);
   return 0;
}
```

Ein grösseres Beispiel für die Benutzung von dynamischem Speicher:

*Liest eine beliebige Anzahl Zeichen ein, bis * eingegeben wird und gibt den Text anschliessend wieder aus.*

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

char *TextEinlesen(void)
{
   char *Text;
   char Zeichen;
   int  Length = 0;
   int  MaxLength = 100;

   /* Platz fuer 100 Zeichen reservieren */
   /*( + 1 Reserve fuer abschliessende '\0') */
   Text = malloc(sizeof(char) * (MaxLength + 1));
   if (Text == NULL) {
      printf("Error, out of memory");
      return NULL;
   } else {
      printf("Bitte Text eingeben, abschluss mit <*>\n");
      Zeichen = getchar();

      /* Solange bis <*> gedrueckt wird */
      while (Zeichen != '*') {

         /* Block erweitern, wenn zu falls klein */
         if (Length >= MaxLength) {
            Text = realloc(Text, MaxLength+100);
            if (Text == NULL) {
               printf("Error, out of memory");
               return NULL;
            }

            /* wurde erweitert */
            MaxLength += 100;
         }

         /* Zeichen abspeichern */
         Text[Length++] = Zeichen;

         /* und neues Zeichen einlesen */
         Zeichen = getchar();
      }
      Text[Length++] = '\0'; /* Buffer mit '\0' abschliessen */
   }
   return Text;
}

int main(int argc, char*argv[])
{
   char *TextBlock;

   /* Textblock beliebiger groesse einlesen */
   TextBlock = TextEinlesen();
   if (TextBlock != NULL) {
      /* ...wieder ausgeben */
      puts("Sie haben folgenden Text eingegeben:\n\n");
      puts(TextBlock);
      /* und Speicherblock wieder freigeben */
      free(TextBlock);
   }
   return 0;
}
```

26.1 Selbstgeschriebene Speicherverwaltung

Gelegentlich ist es Erforderlich, aus Effizienzgründen selbst eine Speicherverwaltung zu schreiben. Diese kann dabei auf Geschwindigkeit oder Speicherausnutzung optimiert werden.

Die standardmässige Speicherverwaltung führt meistens eine Liste der freien Speicherblöcke, und sucht sich bei der Anforderung von Speicher einen Block passender Grösse, oder zerteilt einen grösseren Block falls kein passender Block vorhanden ist. Bei der Freigabe wird ensprechend versucht, den Block mit einem benachbarten freien Block in der Liste zusammenzufassen, sonst wird er in die Liste der freien Blöcke aufgenommen. Um die Verwaltung zu vereinfachen werden dabei die Blockgrössen auf ein vielfaches einer Basisblockgrösse aufgerundet, dadurch ergibt sich ein Mittelweg zwischen Geschwindigkeit und effizienter Speicherausnutzung.

Wenn man eine sehr schnelle Speicherverwaltung benötigt, geht das am einfachsten mit einem Array, dabei muss man aber die Einschränkung einer fixen Blockgrösse in Kauf nehmen.

```
static char MyMemory[50][128] = {0};  /* 50 Blöcke a 128 Bytes */

void *FastMalloc(void)   /* Keine Grössenangabe, da fixe Blockgrösse */
{
   int i;
   for (i = 0; i < 50; i++) {       /* Freien Block suchen             */
      if (MyMemory[i][0] == 0) {
         MyMemory[i][0] = 1;        /* Block als Belegt markieren      */
         return &(MyMemory[i][0]);  /* Zeiger auf reservierten Block   */
                                    /* zurueckliefern                  */
      }
   }
   return NULL;                     /* keinen freien Speicher gefunden */
}


void FastFree(void *Mem)
{
   *((char *)(Mem)) = 0;            /* Block als Frei markieren        */
}
```

Bei dieser Version gilt die Einschränkung, das im ersten Byte des Speicherblocks nie der Wert 0 abgelegt werden darf, weil sonst der Block als Frei angesehen wird. In vielen Anwendungen ist das kein Problem, und sonst muss einfach ein Zeiger auf das zweite Byte des Blocks zurückgegeben werden, dann hat der Anwender keinen Zugriff auf das Flag-Byte am Anfang:

```
/* Aenderung in FastMalloc: */
        return &(MyMemory[i][1]);  /* Zeiger auf Zweites Byte zurueck */

/* Aenderung in FastFree: */
  *(((char *)(Mem))-1) = 0;        /* Block als Frei markieren        */
```

Wenn man mehrere Blockgrössen benötigt, kann man einfach für jede Blockgrösse ein eigenes Array benutzen, und die grösse der Arrays dem zu erwartenden Speicherbedarf anpassen. Speicheranforderungen des Anwenders werden jeweils auf die nächstgrösste Blockgrösse aufgerundet, und der Block aus dem entsprechenden Array angefordert.

Bei grossen Arrays kann anstelle eines Flagbytes eine Indexliste aufgebaut werden. In jedem freien Block ist im vordersten Byte/Wort der Index des naechsten freien Blocks eingetragen, so wird eine Liste der freien Blocks aufgebaut welche schnelleres finden eines freien Blocks erlaubt. Die Speicherverwaltung muss sich den Index des ersten freien Blocks merken (Head der Liste).

Freigegebene Blöcke werden zuvorderst in die Liste eingehängt, und angeforderte Blöcke werden wiederum vom Listenanfang genommen (Somit ist nur ein 'Head-Pointer' notwendig).

27 Listen

Arrays sind die einfachste Form, um eine Menge von Daten zu verwalten. Sobald aber die einzelnen Datenelemente eine gewisse Grösse erreichen oder häufig Daten zwischen bestehenden Elementen eingefügt oder entfernt werden sollen, wird sehr viel Zeit mit kopieren von Daten verbracht.

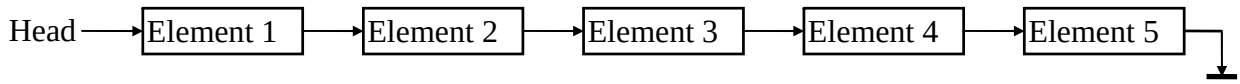
Eine Lösung für dieses Problem bietet die Verwendung von Listen. Eine Liste kann man sich wie eine Kette vorstellen, jeder Datensatz ist wie ein Kettenglied jeweils mit seinen direkten Nachbarn verbunden. Wenn man nun einen Datensatz einfügen oder entfernen will, muss dazu nicht die ganze Liste geändert werden, es betrifft nur die direkten Nachbarn.

Diese Verbindungen der Nachbarn untereinander werden in C normalerweise mit Zeigern realisiert.

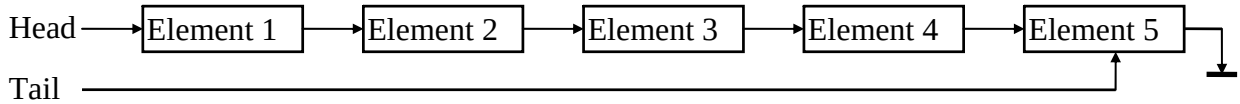
27.1 Einfach verkettete Liste

Bei einer *einfach verketteten Liste* hat jedes Datenelement einen Zeiger, der auf seinen Nachfolger zeigt. Der Zeiger des letzten Elements zeigt auf **`NULL`**.

Im einfachsten Fall hat die Liste noch einen Zeiger (*Head*) der auf den Anfang der Liste zeigt.



Wenn man häufig auf Elemente am Listenende zugreifen will (*Queue*) empfiehlt sich der Einsatz eines weiteren Zeigers, der ständig auf das letzte Element der Liste zeigt.



Ein Datenelement einer Liste weist in C üblicherweise folgende Struktur auf:

```
struct Listenelement {

   /* Hier kommen die Daten die gespeichert werden sollen       */
   /* (Z.B. struct Student Daten; oder wie hier ein simpler int */

   int Wert;

   /* Und hier das wichtigste: Einen Zeiger auf den Nachfolger  */
   struct Listenelement *Next;
}
```

Und nun die 'Verankerung' der Liste: Mindestens der Zeiger auf den Anfang ist unverzichtbar, sonst weiss man ja nicht wo die Liste beginnt.

```
struct Listenelement * Head;  /* Zeiger auf den Listenanfang */
struct Listenelement * Tail;  /* Zeiger auf das Listenende   */
```

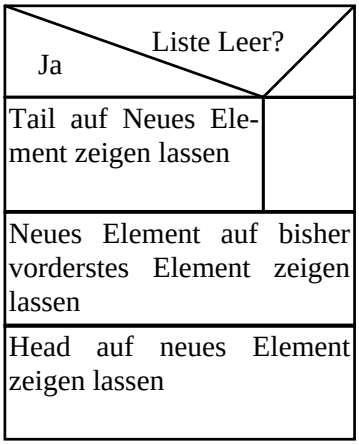
Die beiden Zeiger müssen bei einer leeren Liste auf **`NULL`** gesetzt werden.

Neue Listenelemente werden bei bedarf mit **`malloc()`** erzeugt und mit **`free()`** wieder freigegeben (Mit einem typedef könnte man sich einige Schreibarbeit ersparen):

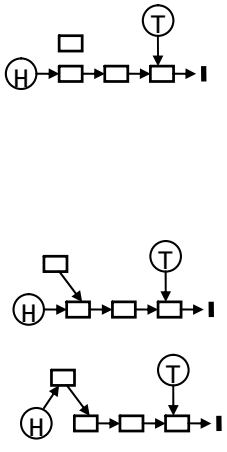
```
struct Listenelement * Neues;    /* Zeiger auf neues Element*/
Neues = (struct Listenelement *) malloc(sizeof(struct Listenelement ));
....
free(Neues);
```

Um ein Element am Listenanfang einzufügen, ist wie folgt vorzugehen:

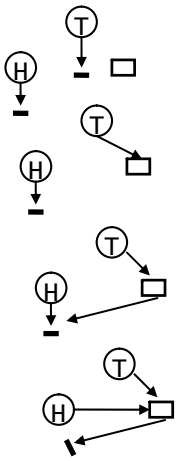
Struktogramm



Graphisch



Leere Liste

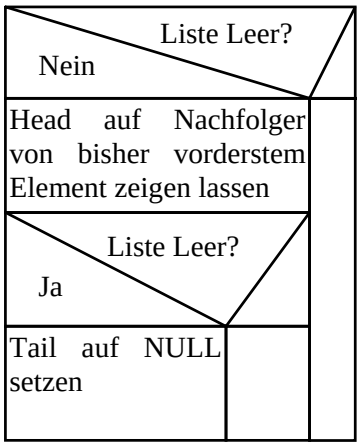


C-Code

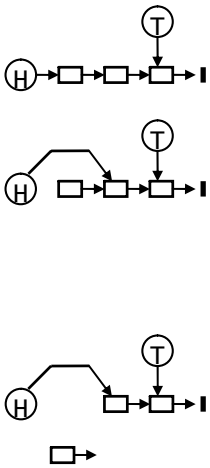
```
if (Head == NULL) {
   Tail = NewElement;
}
NewElement->Next =
Head;
Head = NewElement;
```

Um ein Element vom Listenanfang zu entfernen, ist wie folgt vorzugehen:

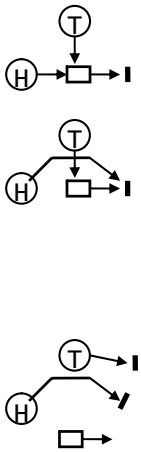
Struktogramm



Graphisch



Leere Liste



C-Code

```
if (Head != NULL) {
   Head = Head->Next;
   If (Head == NULL) {
      Tail = NULL;
   }
}
```

Um Elemente am Listenende einzufügen oder zu entfernen müssen entsprechend modifizierte Operationen durchgeführt werden.

Achtung: Der Speicherplatz von Listenelementen, die aus einer Liste entfernt werden und nicht mehr benötigt werden, muss mit **`free()`** wieder freigegeben werden, falls zum Erzeugen der Listenelemente **`malloc()`** verwendet wurde. Das Freigeben oder Erzeugen von Listenelementen ist jedoch nicht unbedingt Aufgabe der Einfüge und Entfernfunktionen.

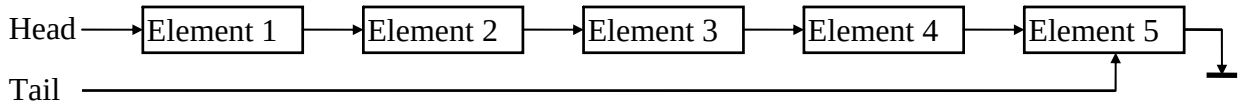
Aufgabe 27.1:
Entwerfen Sie zwei C-Funktionen (Inklusive Design/Struktogramm), die Elemente am Listenende einfügen resp. entfernen.

27.2 Stack / Fifo / Queue

Stack (*LIFO* [= Last in First Out]) und *Queue* (*FIFO* [=First In First Out]) sind spezielle Arten von Listen, bei denen entweder nur am Anfang angefügt oder entfernt wird (Stack) oder an einem Ende angehängt und am anderen Entfernt wird (Queue). Bei einer Queue erhält man die Elemente in derselben Reihenfolge wieder, in der sie abgespeichert wurden, beim Stack in umgekehrter Reihenfolge. Eine Queue könnte man z. B. zum Zwischenspeichern von über die serielle Schnittstelle empfangenen Daten verwenden, damit die Reihenfolge erhalten bleibt.

Bei einer einfach verketteten Liste lassen sich Elemente sehr einfach am Anfang oder am Ende anfügen oder entfernen, es ist aber viel aufwendiger, Elemente in der Mitte hinzuzufügen oder zu entfernen.

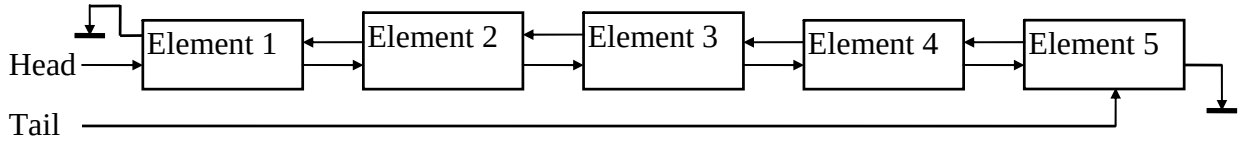
Aufgabe 27.2:
Überlegen Sie sich, wie Sie vorgehen müssten, um bei einer einfach verketteten Liste ein Element in der Mitte zu entfernen, und wie um eines in der Mitte hinzuzufügen. Besprechen Sie sich mit Ihrem Nachbarn.



Überlegen Sie sich, mit welchen Modifikationen der Listenstruktur man diese Funktionen vereinfachen könnte.

27.3 Doppelt verkettete Liste

Einfüge und Löschoperationen lassen sich bei einer Liste deutlich vereinfachen, wenn man bei den Datenelementen einen Rückwärtszeiger einführt. Damit hat man von jedem Element aus auch Zugriff auf seinen Vorgänger.



Ein Datenelement einer doppelt verketteten Liste könnte in C wie folgt definiert werden:

```
struct Datenelement {

   /* Hier kommen die Daten die gespeichert werden sollen       */
   /* (Z.B. struct Student Daten; oder wie hier ein simpler int */

   int Wert;

   /* Und hier das wichtigste: Einen Zeiger auf den Nachfolger  */
   struct Datenelement *Next;
   /* und einen auf den Vorgänger  */
   struct Datenelement *Previous;
}
```

Die Listenoperationen gestalten sich eigentlich ähnlich wie bei der einfach verketteten Liste, nur muss jetzt immer auch der Rückwärtszeiger behandelt werden.

Um ein Element am Listenanfang einzufügen, ist somit wie folgt vorzugehen:

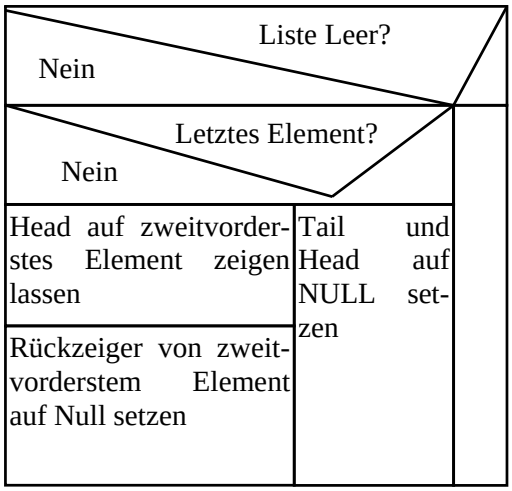
Struktogramm

Liste Leer? Ja	
Tail auf Neues Element zeigen lassen	Rückzeiger von bisher vorderstem Element auf neues Element zeigen lassen
Neues Element auf bisher vorderstes Element zeigen lassen	
Head auf neues Element zeigen lassen	
Rückzeiger von neuem Element auf Null setzen.	

C-Code

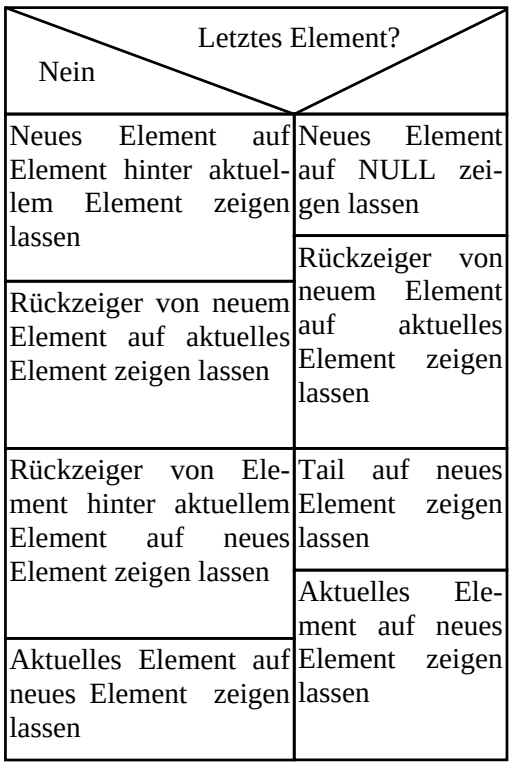
```
if (Head == NULL) {
   Tail = NewElement;
} else {
   Head->Previous =
NewElement;
}
NewElement->Next =
Head;
Head = NewElement;
NewElement->Previous =
                  NULL;
```

Ein Element lässt sich folgendermassen vom Listenanfang entfernen:




```
if (Head != NULL) {

  If (Head->Next == NULL){
    Tail = NULL;
    Head = NULL;
  } else {
    Head = Head->Next;


    Head->Previous = NULL;
  }
}
```

Um ein Element hinter einem gegebenen Element anzuhängen, ist wie folgt vorzugehen:




```
if (Aktuell->Next != NULL)
{

  NewElement->Next =
           Aktuell->Next;

  NewElement->Previous =
                   Aktuell;

  Aktuell->Next->Previous
=

  Aktuell->Next =
               NewElement;

} else {
  NewElement->Next = NULL;

  NewElement->Previous =
                   Aktuell;

  Tail = NewElement;


  Aktuell->Next =
               NewElement;
}
```

Achtung: Der Speicherplatz von Listenelementen, die aus einer Liste entfernt werden und nicht mehr benötigt werden, muss mit **`free()`** wieder freigegeben werden, falls zum Erzeugen der Listenelemente **`malloc()`** verwendet wurde. Das Freigeben oder Erzeugen von Listenelementen ist jedoch nicht unbedingt Aufgabe der Einfüge und Entfernfunktionen.

Aufgabe 27.3: Überlegen Sie sich die Vorgehensweise zum Löschen eines beliebigen Elementes aus der Liste. Sie erhalten einen Zeiger auf das zu löschende Element (Bitte Spezialfälle auch behandeln: letztes, erstes, einziges).

27.3.1 Arbeiten mit Listen

Auf Listen können selbstverständlich auch Sortier und Suchalgorithmen angewendet werden. Weil man auf Listenelemente aber keinen *wahlfreien*, sondern nur *sequentiellen* Zugriff hat, können nicht mehr alle Algorithmen effizient angewendet werden.

Binäres Suchen ist z.B. nicht möglich, weil man nicht direkt in die Mitte der Liste zugreifen kann, und auch Sortierverfahren wie Quicksort fallen aus demselben Grund weg. Bubblesort, Straight Selection und Straight Insertion hingegen können gut eingesetzt werden.

Um jedes Element einer Liste zu untersuchen, bietet sich folgende Schlaufe an:

```
struct Datenelement* Current;

for (Current = Head; Current != NULL; Current = Current->Next) {
  /* Hier der Code fuer was auch man immer machen moechte z.B. Ausgabe*/
  printf("Wert ist %d\n", Current->Wert);
};
```

27.3.2 Nun ein komplettes Modul zur Listenbearbeitung:

```
typedef struct ListElement {

      /* Hier die Datenelemente einfuegen */

      struct ListElement *Next;
      struct ListElement *Previous;
} ListElement;

ListElement *Head = NULL;
ListElement *Tail = NULL;

void PushFront (ListElement *NewElement) {
      NewElement->Next = Head;
      NewElement->Previous = NULL;
      if (Head == NULL) {
            Tail = NewElement;
      } else {
            Head->Previous = NewElement;
      }
      Head = NewElement;
}

void PushBack (ListElement *NewElement) {
      NewElement->Next = NULL;
      NewElement->Previous = Tail;
      if (Tail == NULL) {
            Head = NewElement;
      } else {
            Tail->Next = NewElement;
      }
      Tail = NewElement;
}

ListElement *PopFront(void) {
      ListElement *Temp = NULL;
      if (Head != NULL) {
            Temp = Head;
            Head = Head->Next;
            if (Head == NULL) {
                  Tail = NULL;
            } else {
                  Head->Previous = NULL;
            }
      }
      return Temp;
}

ListElement *PopBack(void) {
      ListElement *Temp = NULL;
      if (Tail != NULL) {
            Temp = Tail;
            Tail = Tail->Previous;
            if (Tail == NULL) {
               Head = NULL;
            } else {
               Tail->Next = NULL;
            }
      }
      return Temp;
}
```

```
void InsertBehind(ListElement *Where, ListElement *What) {
      if (Where == NULL) {
            PushBack(What);
      } else {
            What->Previous = Where;
            What->Next = Where->Next;
            if (Where->Next == NULL) {
                  Tail = What;
            } else {
                  Where->Next->Previous = What;
            }
            Where->Next = What;
      }
}

void InsertBefore(ListElement *Where, ListElement *What) {
      if (Where == NULL) {
            PushFront(What);
      } else {
            What->Previous = Where->Previous;
            What->Next = Where;
            if (Where->Previous == NULL) {
                  Head = What;
            } else {
                  Where->Previous->Next = What;
            }
            Where->Previous = What;
      }
};

ListElement *Erase(ListElement *Which) {
      if ((Head == NULL)||(Which == NULL)) {
         return (NULL);
      }
      if (Which->Next == NULL) {
            Tail = Which->Previous;
      } else {
            Which->Next->Previous = Which->Previous;
      }
      if (Which->Previous == NULL) {
            Head = Which->Next;
      } else {
            Which->Previous->Next = Which->Next;
      }
      return Which->Next;
}

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

      ListElement *Tmp = (ListElement *) malloc(sizeof(ListElement));
      if (Tmp != NULL) {
            PushFront (Tmp);
      }
      Tmp = PopBack();
      free(Tmp);
}
```

Um mit mehreren Listen gleichzeitig arbeiten zu können, kann das Modul dahingehend geändert werden, das Tail und Head in eine Struktur gepackt werden, und allen Funktionen als zusätzliches Argument ein Zeiger auf eine solche Struktur übergeben wird.

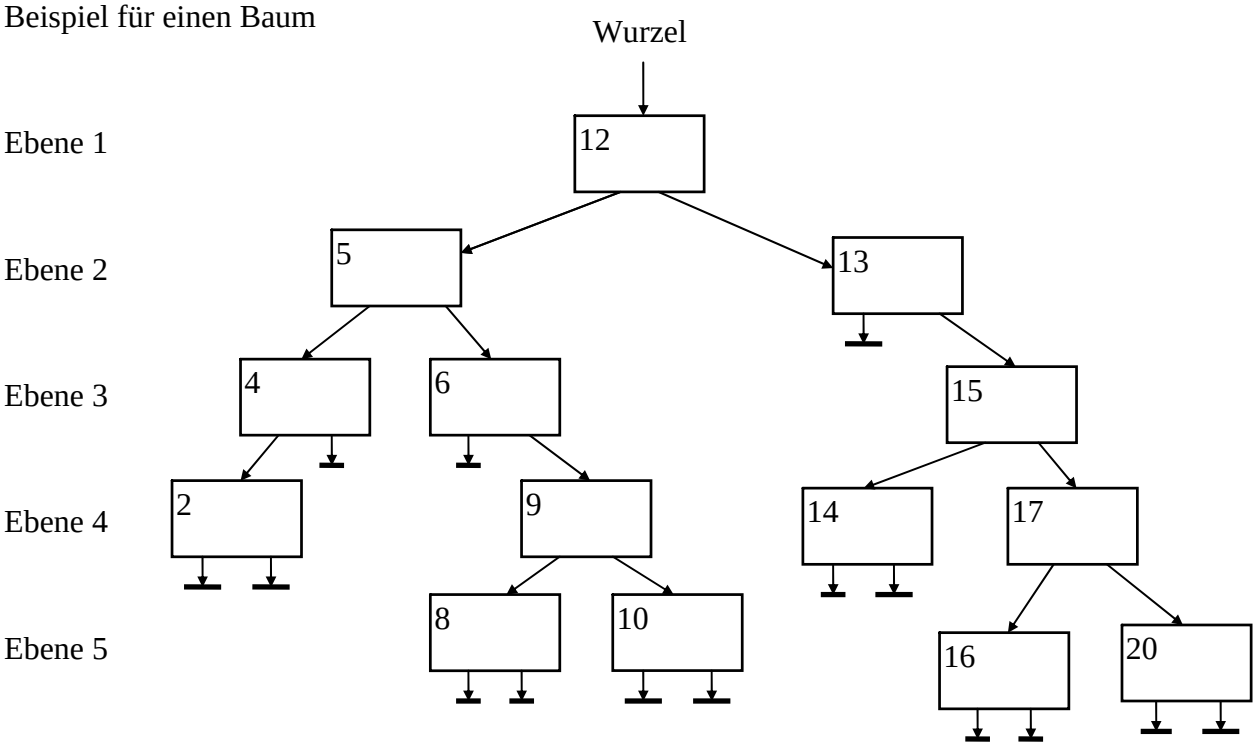
Jede Liste hat so ihre eigene Listenverwaltung (Listenkopf und Ende).

28 (Binäre) Bäume

Wenn man auf schnelles Suchen angewiesen ist, können Listen nicht verwendet werden. Als alternative dazu bieten sich *Bäume* an. Im Gegensatz zu einer Liste ist ein Baum eine *Hierarchische Struktur*, bei der effizientes Suchen möglich ist. Ein Baum ist von Natur aus immer sortiert.

Bei einem binären Baum hat jedes Element zwei Nachfolger, auf der linken Seite das kleinere Element, und auf der rechten Seite das grössere. (Links und Rechts sind willkürlich gewählt).

Beispiel für einen Baum



Der oberste *Knoten* eines Baumes heisst *Wurzel*. Ein Element, das keine Nachfolger hat, wird als *Blatt* bezeichnet, ein Element mit Nachfolger als Knoten. Jedes Blatt und jeder Knoten ist genau einem Knoten untergeordnet. Jeder Knoten bildet zusammen mit seinen untergeordneten Elementen einen *Teilbaum*.

Bei einem optimal verteilten Baum haben bis auf die Knoten der letzten und der zweitletzten Ebene alle Knoten zwei Nachfolger. Dies wird als *ausgeglichener Baum* bezeichnet. Bei einem ausgeglichenen Baum findet man jedes Element nach spätestens log2(N) vergleichen.

Ein Datenelement eines Baumes könnte in C wie folgt definiert werden:

```
typedef struct Datenelement {

   /* Hier kommen die Daten die gespeichert werden sollen       */
   /* (Z.B. struct Student Daten; oder wie hier ein simpler int */

   int Wert;

   /* Und hier das wichtigste: Die Zeiger auf die Nachfolger  */
   struct Datenelement *Left;
   struct Datenelement *Right;
} TreeNode;
```

Und nun noch der Aufhänger für die Wurzel des Baumes:

```
TreeNode* Wurzel;  /* Zeiger auf die Baumwurzel */
```

Da ein Baum eine *rekursive Datenstruktur* ist (Ein Baum besteht aus Teilbäumen), lässt sich das Einfügen von Elementen sehr elegant rekursiv lösen:

```
void InsertNode(TreeNode **Root, TreeNode *Node)
{
   /* Falls kein Element, neuen Knoten Anhängen  */
   if (*Root == NULL) {
      Node->Left  = NULL;
      Node->Right = NULL;
      *Root = Node;
   } else {
      /* Prüfen ob im linken oder im rechten Teilbaum anhängen */
      if (Node->Value > (*Root)->Value) {
         /* Wert ist grösser, im rechten Teilbaum einfügen */
         InsertNode(&( (*Root)->Right), Node);
      } else {
         /* Wert ist kleiner, im linken Teilbaum einfügen */
         InsertNode(&( (*Root)->Left), Node);
      }
   }
}
```

Root ist ein Zeiger auf die Wurzel des aktuellen Teilbaums

Node ist ein Zeiger auf den einzufügenden Knoten

Aufruf der Einfügeoperation:

```
TreeNode *NewNode;
NewNode = (TreeNode *) malloc( sizeof(TreeNode) );
NewNode->Wert = Value;
InsertNode(&Wurzel, NewNode);    /* Zeiger auf Wurzel und Zeiger auf */
                                 /* neuen Knoten uebergeben          */
```

Das sortierte Ausgeben des Bauminhaltes lässt sich auch sehr leicht rekursiv lösen:

```
void PrintTree(TreeNode *Root)
{
  /* Nur falls gültiges Element */
  if (Root != NULL) {

    /* Zuerest alle kleineren Elemente ausgeben (linker Teilbaum) */
    PrintTree(Root->Left);

    /* Dann den Wert dieses Knotens */
    printf("\n%d", Root->Wert;

    /* und dann alle grösseren Elemente ausgeben (rechter Teilbaum) */
    PrintTree(Root->Right);
  }
}
```

Aufruf der Ausgabeoperation:

```
PrintTree(Wurzel);
```

Das Suchen kann rekursiv, oder auch sequentiell erfolgen:

Rekursiv:

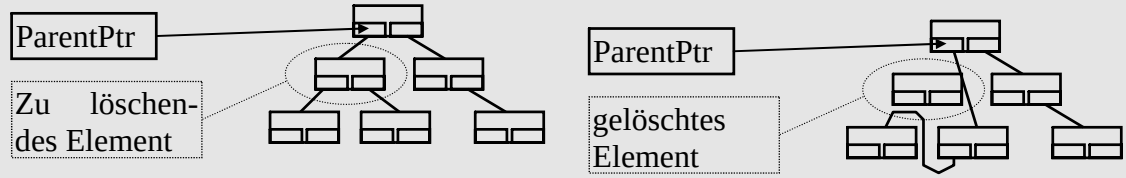
```
TreeNode *FindNode(int Wert, TreeNode *Node)
{
  /* Falls kein Element, nicht gefunden */
  if (Node== NULL) {
    return NULL;
  } else {
    /* Prüfen ob gefunden, sonst in Teilbaeumen weitersuchen */
    if (Node->Value == Wert) {
      return Node;
    } else {
      /* Prüfen ob im rechten oder im linken Teilbaum weitersuchen*/
      if (Wert > Node->Value) {
        /* Wert ist grösser, im rechten Teilbaum weitersuchen*/
        return FindNode(Wert, Node->Right);
      } else {
        /* Wert ist kleiner, im linken Teilbaum weitersuchen*/
        return FindNode(Wert, Node->Left);
      }
   }
}
```

Sequentiell:

```

TreeNode *FindNode(int Wert, TreeNode *Root)
{
  TreeNode *p;

  /* Falls kein Element, nicht gefunden */
  if (Node== NULL) {
    return NULL;
  } else {
    /* Prüfen ob gefunden, sonst in Teilbaeumen weitersuchen */
    for(p = Root; p != NULL; ) {
      if (p->Value == Wert) {
        return p;
      }
      /* Prüfen ob im rechten oder im linken Teilbaum weitersuchen*/
      if (Wert > p->Value) {
        /* Wert ist grösser, im rechten Teilbaum weitersuchen*/
        p = p->Right);
      } else {
        /* Wert ist kleiner, im linken Teilbaum weitersuchen*/
        p = p->Left;
      }
   }
}
```

Das Löschen eines Knotens ist deutlich komplizierter, insbesondere wenn ein Knoten mit zwei Nachfolgern gelöscht werden soll, dann müssen beide Teilbäume dieses Knotens an der korrekten Stelle des Restbaumes eingefügt werden. Bei Bäumen empfiehlt es sich deshalb besonders, auf bereits existierende (und gründlich getestete) Bibliotheken zurückzugreifen.

```
/* Der Funktion muss die Addresse des Zeigers, der auf den zu löschenden  */
/* Knoten zeigt, übergeben werden. Dies ist nötig, weil ebendieser Zeiger */
/* auch verändert werden muss. (Suchfunktion entsprechend ändern)         */
```




```
void RemoveNode(TreeNode **ParentPtr) {
  TreeNode *Node = *ParentPtr;
  TreeNode *p, *p2;
  if(Node == NULL) {
    return;    /* element not found; */
  } else {
    if((Node->left == NULL) && (Node->right == NULL)) {
      /* Knoten hat keine Nachfolger, einfach entfernen */
      free(Node);
      *ParentPtr= NULL;
    }
    else if(Node->left==NULL) {
      /* Knoten hat einen Nachfolger, einfach bei Parent anhaengen */
      p = Node->right;
      free(Node);
      *ParentPtr= p;
    }
    else if(Node->right == NULL) {
      /* Knoten hat einen Nachfolger, einfach bei Parent anhaengen */
      p = Node->left;
      free(Node);
      *ParentPtr= p;
    else {
      /* hat zwei Nachfolger, zusammenfuegen und bei Parent anhaengen */
      p2 = Node->right;
      p = Node->right;
      /*Zusammenfügeposition suchen (wo linken Zweig in rechten einfuegen)*/
      while(p->left) p = p->left;
      p->left = Node->left;
      free(Node);
      *ParentPtr= p2;
    }
  }
}
```

Bei binären Bäumen muss man darauf achten, dass sie nicht *entarten* (Viele Knoten haben nur einen Nachfolger -> Baum wird zu einer Liste). Um das zu verhindern gibt es '*balanced Trees*' (*Ausgeglichene Bäume*). Dabei wird bei jeder Einfüge und Löschoperation durch Überprüfung der Baumstruktur und eventueller Umordnung des Baumes versucht, den Baum in einer möglichst optimalen Form zu behalten.

Es gibt noch viele weitere Abarten von Bäumen, einige davon haben auch mehr als zwei Nachfolger pro Knoten. Jede Variante hat ihre Vor- und Nachteile, welche jeweils gegeneinander abgewogen werden müssen.

29 Hashtabellen

Um Daten schnell zu finden werden oft *Hashtabellen* eingesetzt. Eine Hashtabelle ist ein gewöhnliches Array, aber die Position eines jeden Elementes in diesem Array wird aus dem *Schlüssel* (= Wert nach dem gesucht wird) des Elements selbst berechnet. Diese Berechnung wird als *Hashfunktion* bezeichnet. Bei Strings kann im einfachsten Fall einfach die Quersumme über alle ASCII-Codes der enthaltenen Buchstaben gebildet werden, oder auch aufwendigere Berechnungen, die auch Buchstabenverdreher berücksichtigen.

Es passiert selbstverständlich hin und wieder, das zwei unterschiedliche Einträge denselben Hashwert erhalten, und so eigentlich am selben Platz in der Tabelle stehen müssten. Dies bezeichnet man als *Kollision*. Das Problem kann auf verschiedene Arten umgangen werden.

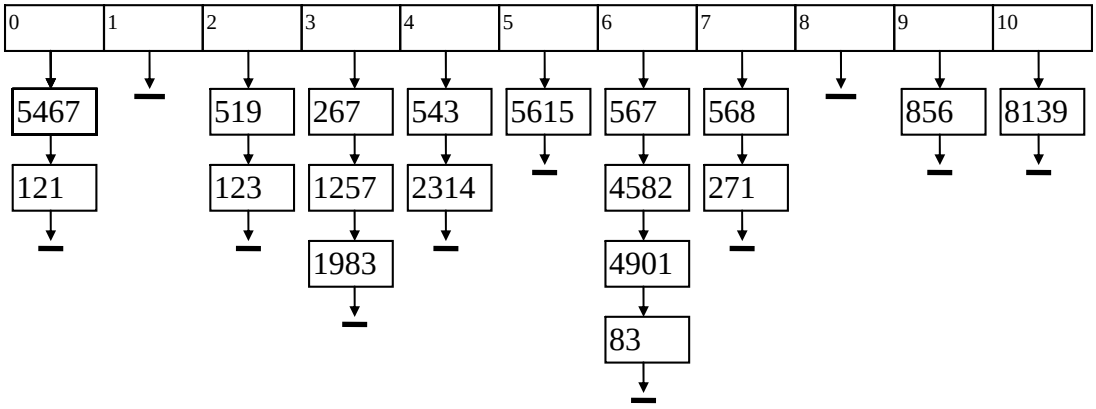
Entweder man bildet Listen bei den einzelnen Tabellenplätzen, so können mehrere Einträge 'am selben' Platz stehen, oder man nimmt einfach den nächsten freien Platz in der Tabelle.

Die grösste Schwierigkeit bei Hashtabellen ist es, eine möglichst optimale Hashfunktion zu finden. Sie sollte einerseits schnell sein, aber andererseits Kollisionen möglichst optimal vermeiden und die Schlüssel möglichst gleichverteilt in die Tabelle abbilden. Zudem müssen die Schlüssel positiv sein, da sie im Allgemeinen als Arrayindex verwendet werden.

Beispiele einiger Hash-Funktionen (Key ist der Wert, nach dem gesucht wird, Text oder Zahl):

```
#define FACTOR 7           /* Sollte Ungerade sein       */
#define TABLESIZE 256      /* 2-er Potenzen sind schnell */
                           /* Primzahlen ideal           */


unsigned int Hash(char *Key)
{
   unsigned int Value = 0;

   while (*Key != '\0') {
      Value = (Value * FACTOR + *Key++) % TABLESIZE;
   }
   return Value;
}


unsigned int Hash(int Key)
{
   return (Key % TABLESIZE);
}


/* Berücksichtigt nur die ersten beiden Zeichen */
unsigned int Hash(char *Key)
{
   if (*Key != '\0') {
      return (Key[0] + 265*Key[1] % TABLESIZE);
   } else {
      return (Key[0] % TABLESIZE);
   }
}
```

Beispiel einer Hashtabelle (Hashfunktion h = Wert % 11)



Beispielcode für eine Hashtabelle mit Einfüge und Suchfunktionen. (Doppelte Einträge werden nicht gefunden, aber auch nicht verhindert. Dazu müsste man beim Einfügen zuerst nach dem Wert suchen, und nur dann Einfügen wenn der Wert nicht gefunden wurde.).

```
typedef struct Entry {
   int Value;
   struct Entry *Next;
} Entry;

Entry* HashTable[11] = {NULL, NULL, NULL, NULL, NULL, NULL,
                                    NULL, NULL, NULL, NULL, NULL};

void Insert(int Value)
{
   Entry* Place;
   Entry* NewEntry;
   unsigned int Hash = Value % 11;

   /* Neues Element erzeugen */
   NewEntry = (Entry *) malloc(sizeof (Entry));
   if (NewEntry != NULL) {
      /* Und am passenden Platz in der Hashtabelle eintragen */
      NewEntry->Value = Value;
      NewEntry->Next  = HashTable[Hash];
      HashTable[Hash] = NewEntry;
   }
}

Entry *Search(int Value)
{
   Entry* Ptr = NULL;
   unsigned int Hash = Value % 11;

   /* Die passende Eintragsliste in der Hashtabelle durchsuchen */
   for(Ptr = HashTable[Hash]; Ptr != NULL; Ptr = Ptr->Next) {
      if (Ptr->Value == Value) {
         return Ptr;
      }
   }
   return NULL;
}
```

Suchfunktionen werden im optimalen Fall bei der Benutzung von Hashtabellen um den Faktor der Tabellengrösse beschleunigt (Im Vergleich zum Suchen in linearen Listen).

30 Software Engineering

Software Engineering heisst, bei der Programmierung von Softwaresystemen ein ingenieurmässiges Vorgehen anzuwenden.

Das heisst zu überlegen:

Was wann getan werden muss

Wie es getan werden muss

Womit es getan werden muss

Und es setzt voraus:

Einen Vorgehensplan (Phasenmodell)
- Zeitliches Vorgehen
- Inhaltliche Schwerpunkte, Aktivitäten

Ein Methodenkonzept
- Art und Weise der Aufgabenerledigung
- Inhaltliche Schwerpunkte, Aktivitäten

Werkzeuge und Techniken
- Arbeitshilfsmittel, Unterstützung zur Darstellung und zur Erzeugung der Ergebnisse

30.1 Zielsetzung des Software Engineering

Die Zielsetzung des Software Engineering ist, die Qualität des Softwaresystems zu verbessern und die Kosten zu optimieren.

Qualität:
- Wartbarkeit
- Korrektheit
- Zuverlässigkeit
- Effizienz
- Ressourcen
- Ergonomie (Benutzerschnittstelle)
- Portabilität
- Wiederverwendbarkeit
- Modularität
- Flexibilität (Weiterentwicklung, Modifikation)

Kosten:
- Entwicklungskosten
- Personalkosten
- Wartungskosten
- Amortisation

30.2 Der Projektablauf / Phasenmodell

Die Aufgabe des *Phasenmodells* ist es, die zeitliche Dimension eines Projektes zu gliedern.

Dabei kann zwischen sachlicher Aufteilung (Gliedern des Gesamtprozesses in überschaubare Teilaktivitäten) und zeitlicher Gliederung (Zerlegung der Projektzeit in absehbare Intervalle) unterschieden werden.

Die Anwendung eines Phasenmodells (Auch Vorgehensmodell genannt) erlaubt es, während der Projektzeit Aussagen über das Projekt zu machen. In diesem Sinne kann das Phasenmodell als Messinstrument für die Projektleitung verstanden werden:

Wer macht was?	Zuständigkeit und Verantwortlichkeiten für einzelne Aktivitäten und Phasen regeln
Was ist gemacht worden?	Projektfortschritt anhand erreichter Ergebnisse kontrollieren
Was ist noch zu tun?	Aufwand für verbleibende Aktivitäten abschätzen

30.2.1 Die 5 Hauptphasen der Softwareentwicklung

1. Analyse (30%)
Benutzeranforderungen und technische Randbedingungen in einem Anforderungskatalog zusammenstellen. Systemanalyse durchführen. Daraus ergibt sich die Spezifikation mit einem genauen Bild der Problemstellung. Eventuell rudimentären Prototypen erstellen. Dokumentation erstellen.

2. Entwurf (30%)
Lösungskonzepte erarbeiten, d.h. Analyse verfeinern, Module spezifizieren, Schnittstellen ausarbeiten. Für kritische Bereiche Prototypen erstellen. Dokumentation nachführen.

3. Realisierung (20%-30%)
Detailspezifikation erarbeiten, Teilproblemen in Programmiersprache umsetzen, Tests der Prozeduren/Funktionen/Module durchführen. Getestete Module zusammenfügen. Dokumentation nachführen.

4. Test (10% - 20%)
Gesamtsystem testen, Produkt anhand der Spezifikation (Zielvorgaben) überprüfen. Dokumentation nachführen.

5. Betrieb
Produkt einführen, Benutzer schulen, Nachkontrolle und Korrekturen durchführen, Produkt warten und modifizieren (Eventuell in Form eines Nachprojektes). Dokumentation nachführen.

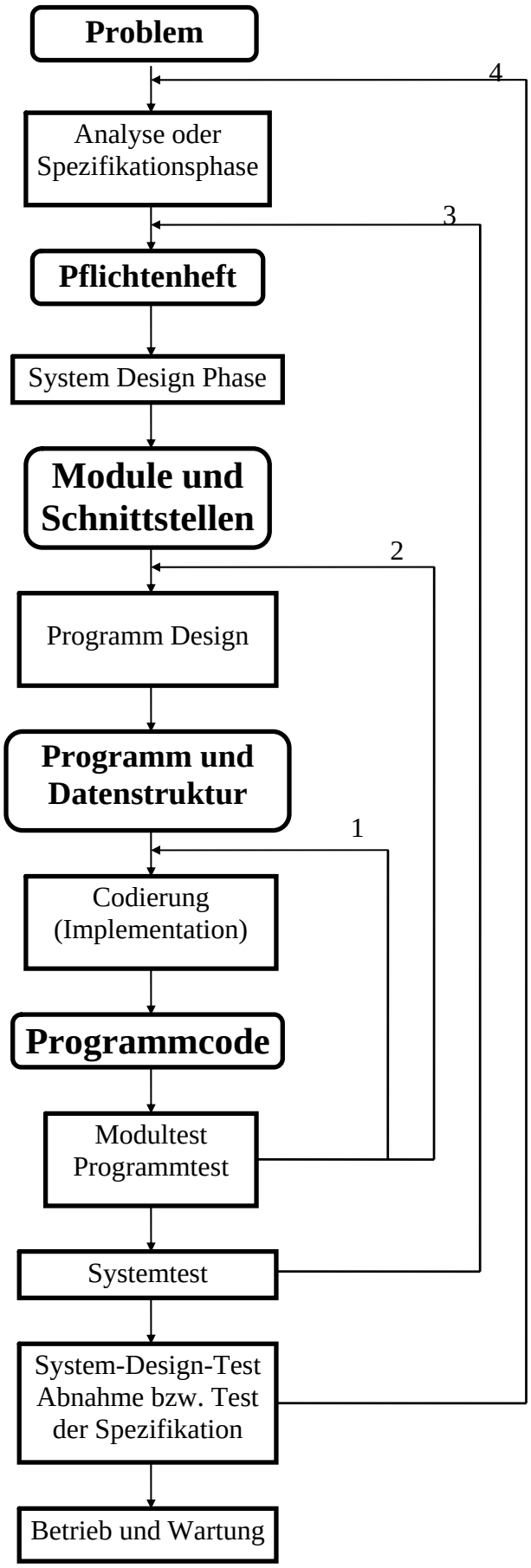
Diese fünf Phasen bilden nicht einen starren, streng linearen Ablauf, vielmehr sind sie Teile eines iterativen Prozesses, bei dem einzelne oder mehrere Phasen bei negativen Resultaten mehrfach durchlaufen werden. Jede Phase wird zuerst initialisiert, das heisst die Vorarbeiten werden ausgeführt, danach folgt die eigentliche Phasenaktivitätsteil gefolgt von den Abschlussarbeiten. Jede Phase hat eine Schnittstelle zu der vorangehenden und der nachfolgenden Phase. Diese Schnittstellen bestehen aus einer Reihe von Dokumenten und Anforderungen, die erfüllt, bzw. vorgelegt werden müssen (siehe dazu untenstehende Graphik).

Initialisierung:
Feststellen, ob aus der Vorphase alle notwendigen Ergebnisse vorliegen.

Durchführung:
Alle Aktivitäten ausführen, die Bestandteil dieser Phase sind.

Abschluss:
Ergebnisse der Phase auf Richtigkeit und Vollständigkeit überprüfen. Erst wenn alles zufriedenstellend ist, soll die nächste Phase angegangen werden. Fehler innerhalb dieser Phase (oder von Vorgängerphasen), die an diesem Punkt nicht erkannt werden, stellen die Ergebnisse aller nachfolgenden Phasen in Frage.

Das Iterative Phasenmodell (Wasserfall Modell):



1. **Codierfehler** .
   Billig zu beheben

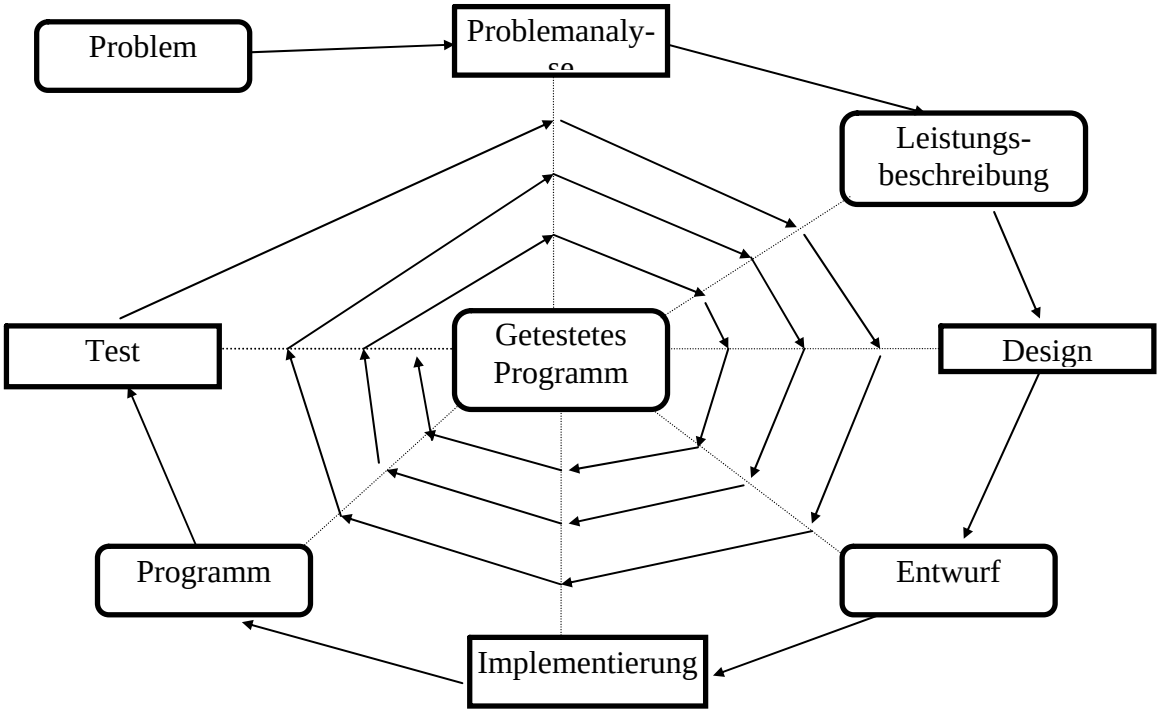
2. **Programmdesignfehler**
   Benötigen mehr Aufwand zur Korrektur und sind mit mehr Kosten verbunden.

3. **Systemdesignfehler**
   Benötigen sehr viel Aufwand und sind teuer zum Beheben.

4. **Spezifikationsfehler**.
   Sind sehr teuer zum Beheben, möglicherweise wird das ganze System in Frage gestellt, unter Umständen muss komplett neu begonnen werden.

Das Spiral Modell:

Da das *Wasserfallmodell* die Realität nicht korrekt abbildet, wurden weitere Modelle entworfen, eines davon ist das nachfolgend dargestellte *Spiraldiagramm*. Dieses Diagramm zeigt die sukzessive Verbesserung und Erweiterung eines Programmes, bis die Zielvorstellung nahezu erreicht worden ist. Das Ziel wird nie erreicht, nur bei jedem Schritt besser angenähert. Der Zyklus muss somit irgendwann einmal abgebrochen werden, damit das Projekt ein Ende findet.



30.3 Extreme Programming (XP)

Ein völlig neuer Ansatz ist das *Extreme Programming (XP)*. Dieser Ansatz widerspricht in vielem der gängigen Lehrmeinung und wird kontrovers diskutiert, ist aber eine interessante Alternative.

Modularisiertes und Diszipliniertes Vorgehen sind auch bei diesem Ansatz die wichtigsten Voraussetzungen.

Beim Extreme Programming wird nicht mehr das ganze System auf einmal entworfen und implementiert, sondern das ganze Projekt durchläuft kurze (1-3 Wochen) Iterationszyklen. Zu Beginn eines jeden Iterationszyklus wird mit dem Kunden besprochen, welche Funktionalität in diesem Zyklus realisiert werden soll. Am Ende eines jeden Zyklus erhält der Kunde ein stabiles, getestetes System, in dem alle bisher implementierten, sowie die neuen Features eingebaut sind und fehlerfrei funktionieren. Die Iterationszyklen werden solange wiederholt, bis das Produkt den Kundenwünschen entspricht.

Beim XP gelten folgende, für alle verbindlichen 12 Regeln

1. **The Planning Game:** Management und Entwicklung arbeiten zusammen um den maximalen Verkaufswert so schnell wie möglich zu erreichen. Das Spiel kann auf verschiedenen Ebenen ablaufen, die grundlegenden Regeln sind immer dieselben:
   1. Das Management erstellt eine Liste mit den gewünschten Funktionen für das System.
   2. Die Entwicklung schätzt den Aufwand für die Funktionen, und wieviel davon in einer Iteration implementiert werden kann.
   3. Das Management entscheidet, welche Funktionen in welcher Reihenfolge zu Implementieren sind.
2. **Small Releases:** Beginne mit der kleinsten nutzbringenden Menge von Funktionen. Gib früh und oft neue Versionen frei, füge nur wenig neue Funktionalität pro Schritt hinzu.
3. **System Metaphor:** Jedes Projekt hat eine organisierte Ausdrucksweise, welche für leicht lernbare Regeln für Namensgebung sorgt.
4. **Simple Design:** Benutze immer den einfachst möglichen Entwurf der den Zweck erfüllt. Die Anforderungen können schon morgen ändern, deshalb mache nur was für die aktuelle Iteration Ziel nötig ist.
5. **Continuous Testing:** Bevor die Entwickler eine neue Funktionalität hinzufügen wird ein Test dafür geschrieben. Wenn die Applikation den Test erfüllt ist die Arbeit erledigt. Tests können grundsätzlich in 2 Formen erfolgen.
   1. Unit Tests sind automatische Tests, die von den Entwicklern geschrieben werden und die neuen Funktionen testen sobald sie implementiert sind. Jeder Unit-Test testet eine Klasse, ein Modul oder eine kleine Gruppe von Klassen.
   2. Acceptance Tests werden vom Kunden definiert und testen das gesamte System auf die Funktion gemäss den Spezifikationen. Wenn alle Akzeptanztests für eine bestimmte Funktion erfüllt sind, gilt diese als implementiert.
6. **Refactoring:** Entferne durch Umbau das Systems Codeduplikate, die während der Codierung entstanden sind. Dank den Tests kannst Du sicherstellen, das dabei keine bereits implementierte Funktionalität verloren geht.
7. **Pair Programming:** Jedes Codestück wird von zwei Programmieren gemeinsam geschrieben, die zusammen vor dem selben Rechner sitzen. Dadurch wird jedes Codestück noch während dem Schreiben einem Review unterzogen.
8. **Collective Code Ownership:** Kein Modul gehört einer bestimmten Person. Jeder Entwickler darf jederzeit an jedem Modul arbeiten und Änderungen vornehmen.
9. **Continuous Integration:** Alle Änderungen werden mindestens täglich ins Gesamtsystem integriert. Die Tests müssen vor und nach der Integration zu 100% erfüllt werden.
10. **40-Hour Work Week:** Programmierer gehen pünktlich nach Hause. In kritischen Phasen ist während einer Woche Überzeit erlaubt. Mehrere aufeinanderfolgende Wochen mit Überzeit sind ein Zeichen für ernsthafte Probleme mit der Methode.
11. **On-site Customer:** Das Entwicklungsteam hat ständigen Kontakt zu einem realen Kunden, also jemanden der das System benutzen wird. Der Kunde kann auch ein Vertreter des Managements sein.
12. **Coding Standards:** Jeder benutzt die gleichen Codier-Richtlinien. Man sollte idealerweise einem Stück Code nicht ansehen, von wem es stammt.

Extreme Programming eignet sich besonders, wenn die Anforderungen an das System nicht vollständig bekannt sind, oder fortlaufend ändern. Die Methode eignet sich für Teams von bis zu 12 Entwicklern, alle Entwickler müssen zur gleichen Zeit am selben Ort arbeiten.

Ein ungelöstes Problem bei diesem Ansatz ist besonders die Dokumentation, da keine Analyse des Gesamtsystems erfolgt, und das System iterativ ständig erweitert wird. Weitere Schwierigkeiten ergeben sich durch die gemeinsame Codebasis, da jeder jederzeit Änderungen vornehmen kann stimmt die Ausgangslage für einen anderen plötzlich nicht mehr. Die ständigen Änderungen und Erweiterungen erfordern besonders sorgfältiges aktualisieren der Tests, da bei den Tests Korrektheit zwingend vorausgesetzt wird. Trotz all diesen Problemen wurde XP schon bei vielen Projekten erfolgreich eingesetzt.

31 Analyse/Design

Durch die *Analyse* wird das Problem untersucht und erfasst. Das Lösungskonzept entsteht durch Synthese während der *Designphase*. Die *Implementierung* (Codierung, Realisierung) stellt die Umsetzung der Problemstellung in eine Programmiersprache dar.

Beim Übergang von Analyse zu Design ergeben sich häufig Lücken, weil nicht alles genau spezifiziert ist oder werden konnte. Programmierer müssen diese Lücke im Normalfall durch ihre Erfahrung überbrücken.

Analysephase

Das wichtigste Resultat dieser Phase ist der Anforderungskatalog:

- Die Beschreibung der Gesamtaufgabe des Systems
- Die Beschreibung aller Verarbeitungen, die das System erbringen soll.
- Die Beschreibung aller Daten, die dafür einzugeben sind.
- Die Beschreibung aller dafür bereitzuhaltenden Daten.
- Die Beschreibung wesentlicher Konsistenzbedingungen wie Wertebereiche und Plausibilitäten
- Die Beschreibung der Zugriffsrechte auf die Daten: Wer darf was lesen, eingeben, verändern oder löschen
- Die Beschreibung der Mensch-Maschinen-Schnittstelle (MMS/MMI): Welche Eingaben hat der Benutzer wie zu tätigen.
- Die Beschreibung aller Ausgaben des Systems mit Ausgabemedium und Format.
- Die Beschreibung der Daten, die über Schnittstellen übernommen werden.

Unter Daten verstehen wir dabei auch Dinge wie Werte von Sensoren, Zustände von Schaltern oder Steuersignale.

Designphase

In der Designphase wird ausgehend vom Anforderungskatalog ein (oder mehrere) Entwurf für das Gesamtsystem erstellt.

Oft eingesetzte Methoden:

Topdown:

Das System als Gesamtes betrachten und in ständig kleinere Einheiten aufteilen, bis ein detaillierter Entwurf erreicht ist.

Objektorientierter Entwurf:

Das System als Ansammlung von miteinander kommunizierenden Objekten betrachten, Beziehungen zwischen den Objekten finden und modellieren.

Datengesteuerter Entwurf:

Das System widerspiegelt die Struktur der zu verarbeitenden Daten.

31.1 Analyse und Designstrategien

Während der Analyse- und Designphase sind viele Entscheidungen zu treffen. Methoden beinhalten Strategien, die aufzeigen wie wann welche Fragen zu stellen sind. Die beantworteten Fragen führen zu den Entscheidungen. Methoden wenden eine oder mehrere der nachfolgend formulierten Grundstrategien auf.

Welche Entscheidungen müssen getroffen werden
Wie werden solche Entscheidungen getroffen
In welcher Reihenfolge werden solche Entscheidungen getroffen.

Einige der gebräuchlichsten Strategien:

Strategie	**Welche Entscheidung**	**Wie**	**Welche Reihenfolge**
Top-Down	Was ist die Grösstmögliche Menge des Gesamtsystems	Innerhalb einer Ebene, Einsatz von weiteren Methoden	Von höherer Ebene zu tieferer Ebene
Outside-In	Was ist die Grösstmögliche Menge des Gesamtsystems	Innerhalb einer Ebene, Einsatz von weiteren Methoden	Von der äusseren (Benutzer, Peripherie) zu der inneren (Implementierung) Ebene
Bottom-Up	Prozessorientiert	Optimierung des lokalen Prozesses in Abhängigkeit der unteren Ebene	Von tieferer Ebene zu höherer Ebene
Most-Critical-Component-First	Kritische Teile des Systems	Innerhalb eines Teils, Einsatz weiterer Methoden	Vom Kritischen zum Unkritischen

In der Praxis wird oft eine Kombination dieser Strategien eingesetzt.

31.2 Methoden

Eine Kombination von Methoden zur Lösung eines Problems wird als Methodologie bezeichnet. Eine Analyse/Design-Mehodologie umfasst typisch die folgenden Elemente:

- Projektmanagement-Technik
- Dokumentationsprozeduren
- Designwerkzeuge (ev. CASE-Tools)
- Standards, Regeln

Die Anwendung einer Methodologie ergibt folgende Vorteile:

- Aufdeckung von Widersprüchen
- Erhöhung der Qualität
- verbesserte Kommunikation zwischen den Projektpartnern
- kürzere Einarbeitungszeit
- überprüfbare Resultate

Die Verwendung von Methoden / Methodologien soll nicht als ein einengendes Korsett verstanden werden, sondern als Hilfsmittel / Werkzeug um den Denkprozess zu unterstützen, eine klare Vorgehensweise zu definieren, die Kommunikation mit Kunden und Mitarbeitern zu erleichtern und das Abweichen von geordneten Bahnen (Abdriften ins Chaos) zu vermeiden.

Wichtig ist, dass allen Beteiligten die Regeln der verwendeten Methoden mit ihren Vor- und Nachteilen bekannt sind.

31.3 Einsatz von Methoden

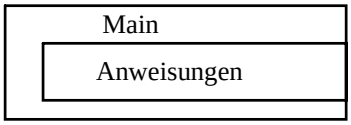
Phase	**Methode**
Spezifikation, Analyse und Anforderungsdefinition, Pflichtenheft	Klartext CRC-Cards (Class, Responsibilities and Collaboration) Use Cases SADT (Für Grosse Zusammenhänge, Organisationen usw.) SA nach Yourdan / De Marco Petri-Netze State-Event (Beschreibung von Automaten) Entity-Relationship-Modelle (E-R-Modelle) für Datenmodelle / Datenbanken UML (OOA)
Konzept und Systemdesign	Structure Chart (Modul Hierarchie), Modul Life Cycles Jackson (Datenstrukturen) Nassi Schneidermann (Struktogramm) State-Event (Automaten, reaktive Systeme) MASCOT (Prozesse und Prozesskommunikation) Entity-Relationship-Modelle UML (OOD)
Detailliertes Systemdesign, Programmdesign	Entity-Relationship-Modelle (Beschreibung von Datenbanken, Datennormalisierung) State-Event (Automaten, reaktive Systeme) Petri-Netze (Automaten, reaktive Systeme, Kommunikation von Prozessen, Erkennen von Deadlocks) Structure Charts (Modul Hierarchie), Modul Life Cycles Jackson (Datenstrukturen) Nassi Schneidermann (Struktogramm) Entscheidungstabellen UML (OOD)

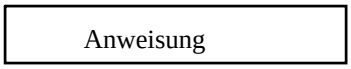
32 Designmethoden

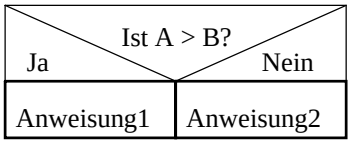
32.1.1 Struktogramme (nach Nassi-Schneiderman).

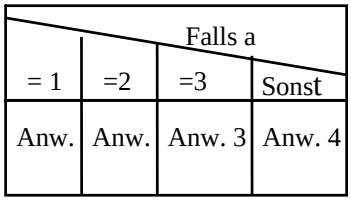
Struktogramme bilden jeden elementaren Strukturblock der strukturierten Programmierung als eindeutig erkennbares graphisches Einzelsymbol ab. Es gibt eindeutige Vorschriften zur Darstellung der einzelnen Elemente und ihrer Kombination. Struktogramme unterstützen damit den Entwurf strukturierter Programme in direkter Weise und erfordern die genaue Planung ihres Einsatzes. So sollte die Struktur des Programmes oder wenigstens die des gerade bearbeiteten Programmbausteins (Modul) bereits feststehen, bevor man mit dem Struktogramm beginnt. Dieser Zwang zur exakten Durchstrukturierung von Programmen sowie die damit verbundene Notwendigkeit zur Modularisierung ist die grosse Stärke der Struktogrammdarstellung (für ein Struktogramm steht im Normalfall nur eine A4-Seite zur Verfügung, wird das Programm hierfür zu komplex, muss es in einzelne kleinere Module unterteilt werden).

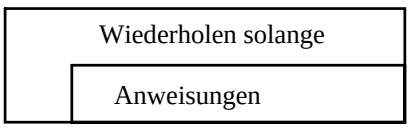
Symbol	Beschreibung	C-Code

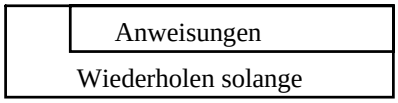
	Modul/Funktionsblock, fasst eine Menge von Blöcken zu einem Benannten Modul, einer Prozedur oder einer Funktion zusammen	int FunktionA (int a, int b)
		{
		Anweisungen
		return 0;
		}

	Anweisung	a = b + c;

	Verzweigung	if (A > B) {
		Anweisung1
		} else {
		Anweisung2
		}

	Fallunterscheidung, Mehrfachverzweigung	switch(a) {
		case 1: Anweisung1 break;
		case 2: Anweisung2 break;
		case 3: Anweisung3 break;
		default: Anweisung4 break;
		}

	Schlaufe mit Anfangsprüfung
	
	(For und while), for-Schlaufen müssen in eine Whileschlaufe umgeformt werden.
		while (i < 10) {
		Anweisung
		}
		for (i= 0; i < 5; i++) {
		Anweisungen
		}

	Schlaufe mit Endeprüfung	do { Anweisung } while (i < 10);

	Unterprogrammaufruf
	
	Funktionsaufruf
		printf("Hallo");
		
		b = FunktionA(c, d);

Achtung: Struktogramme unterstützen das früher oft eingesetzte "GoTo" nicht. Dieser gewollte "Mangel" ist eine Konsequenz aus den Ideen der strukturierten Programmierung. Denn dadurch wird ein unbegrenztes, willkürliches Verzweigen (GoTo!) in der Programmlogik von vornherein unmöglich gemacht.

Aufbau von Struktogrammen

Alle Elementar-Strukturblöcke können auf einfache Weise zusammengesetzt werden. Das Ergebnis ist wiederum ein überschaubarer Strukturblock mit genau einem Eingang und einem Ausgang (Zweipol). Für die Konstruktion zusammengesetzter Strukturblöcke gibt es dem Block-Konzept folgend zwei einfache Regeln:

1. Ein Strukturblock wird an einen anderen gereiht, indem die **gesamte Ausgangskante** des vorstehenden Strukturblock mit der **gesamten Eingangskante** des nachfolgenden Strukturblocks zusammengelegt wird. Ein durch eine solche Aneinanderreihung entstandener Strukturblock wird Sequenz genannt.
2. In die Block-Felder (Anweisungs-Felder) der (elementaren) Strukturblöcke kann jeder beliebige elementare oder zusammengesetzte Strukturblock eingesetzt werden.

Vorgehensweise

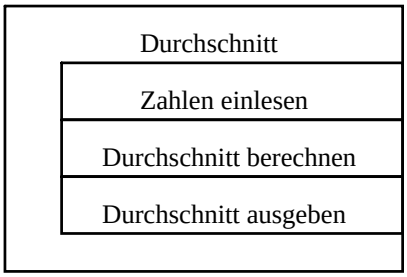
In Struktogrammen wird meist Pseudocode verwendet (Eine Art formale Umgangssprache), damit sie Sprachunabhängig sind. Ein Struktogramm sollte noch nicht festlegen, in welcher Programmiersprache der entsprechende Code implementiert wird.

Bei grösseren Problemen erstellt man zuerst ein grobes Struktogramm, welches nur den grundsätzlichen Programmablauf darstellt, und verfeinert es anschliessend. Es können dabei mehrere Verfeinerungsstufen folgen. In der feinsten Stufe kann bei Bedarf bereits Code der Zielsprache anstelle von Pseudocode eingesetzt werden.

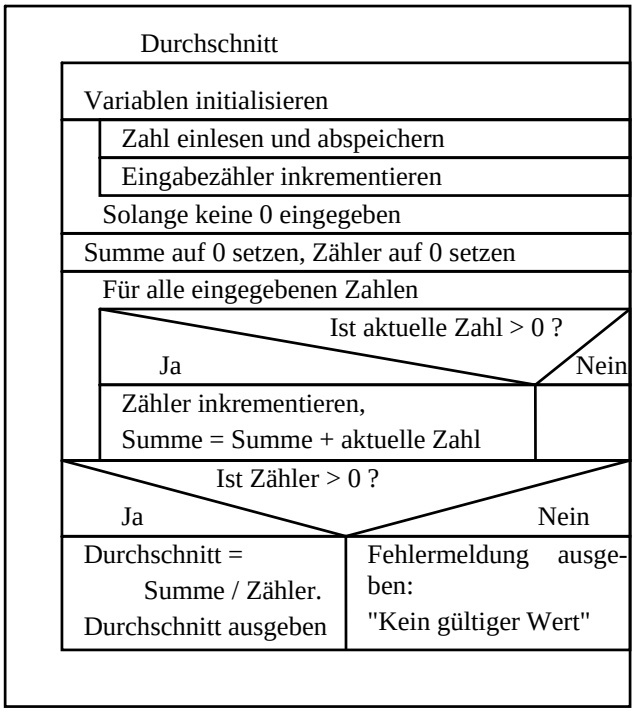
Beispiel:

Aufgabe: Es soll ein Programm entworfen werden, das eine beliebige Anzahl Werte einliest und daraus den Durchschnitt berechnet, negative Zahlen sollen ignoriert werden, eine 0 zeigt den Abschluss der Eingaben an.

Grobstruktogramm:



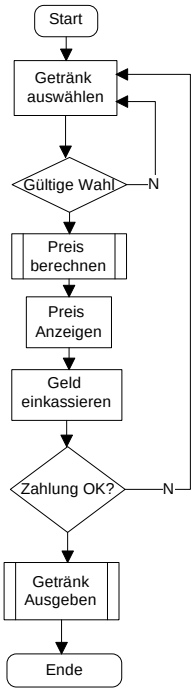
Feinstruktogramm

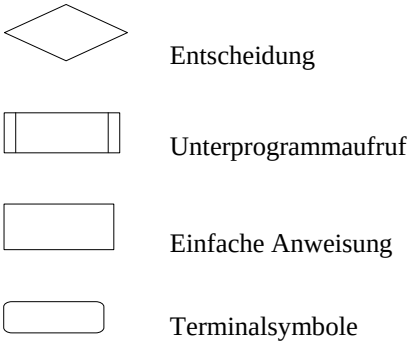


32.1.2 Flussdiagramme

Flussdiagramme werden oft verpönt und als erster Schritt in das Chaos angesehen, aber ein sauberes Flussdiagramm kann unter Umständen das Wesentliche eines Ablaufs deutlicher hervorheben als andere Methoden.

Beispiel Getränkeautomat

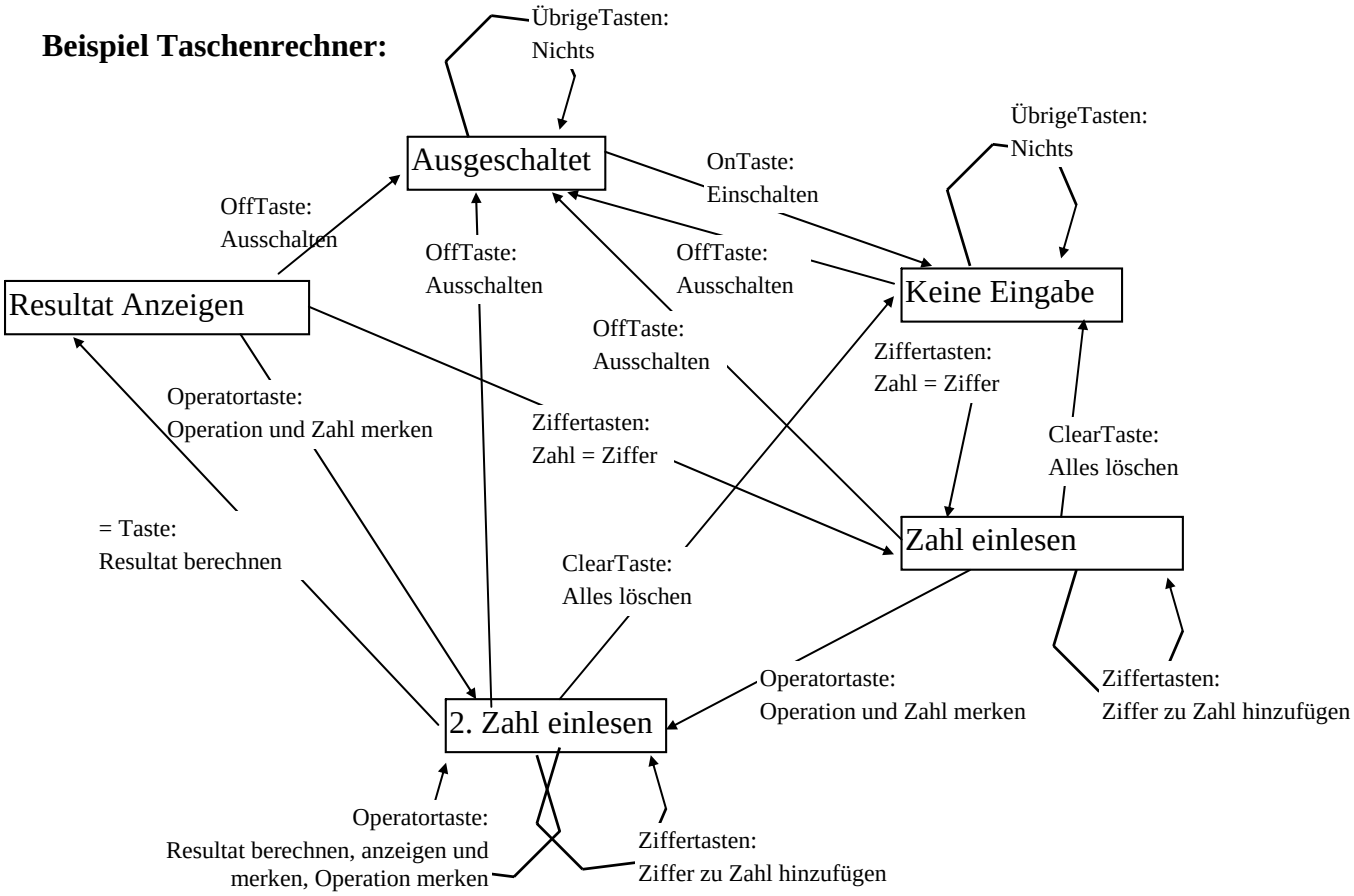




32.1.3 State Event Technik

Die State-Event-Technik eignet sich sehr gut zur Modellierung von ereignisorientierten Prozessen. Eine State-Machine hat eine Menge von Zuständen, in denen sie sich befinden kann, und sie reagiert auf Ereignisse. Ein Ereignis kann je nach Zustand verschiedene Reaktionen, sowie möglicherweise einen Zustandswechsel hervorrufen.

Beispiel Taschenrechner:

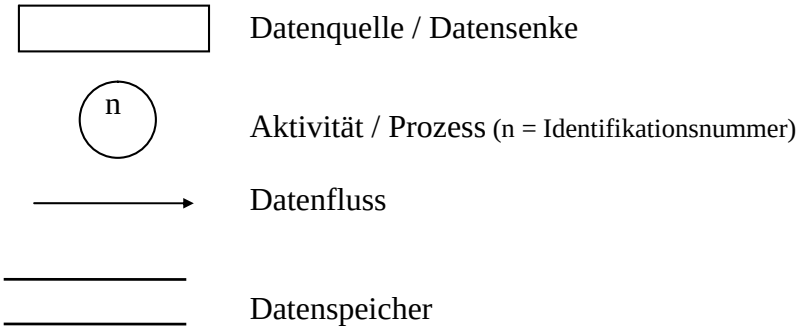


Als Tabelle (Nicht aufgeführte Ereignisse ändern weder den Zustand noch haben sie Aktionen zur Folge):

Zustand	Ereignis	Aktion	Neuer Zustand
Ausgeschaltet	OnTaste	Einschalten	Keine Eingabe
Keine Eingabe	Off Taste	Ausschalten	Ausgeschaltet
	Ziffertaste	Zahl auf Wert von Ziffer setzen	Zahl einlesen
Zahl Einlesen	Off Taste	Ausschalten	Ausgeschaltet
	Ziffertaste	Ziffer zu Zahl hinzufügen	Zahl einlesen
	ClearTaste	Zahl auf 0 setzen	Keine Eingabe
	Operatortaste	Operation & Zahl merken	2. Zahl einlesen
2. Zahl Einlesen	Off Taste	Ausschalten	Ausgeschaltet
	Ziffertaste	Ziffer zu Zahl hinzufügen	2. Zahl einlesen
	ClearTaste	Zahl auf 0 setzen	Keine Eingabe
	Operatortaste	Resultat berechnen, anzeigen und merken, Operation merken	2. Zahl einlesen
	= Taste	Resultat berechnen und anzeigen	Resultat anzeigen
Resultat anzeigen	Off Taste	Ausschalten	Ausgeschaltet
	Ziffertaste	Zahl auf Wert von Ziffer setzen	Zahl einlesen
	ClearTaste	Zahl auf 0 setzen *(Fehlt in Diagramm)*	Keine Eingabe
	Operatortaste	Operation & Zahl merken	2. Zahl einlesen

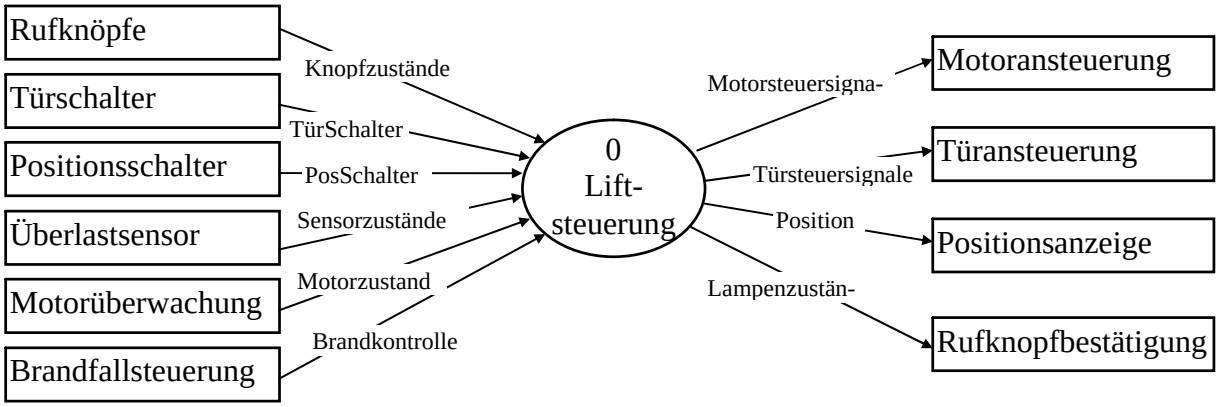
32.1.4 Datenflussdiagramme

Es wird gezeigt, woher die Daten kommen, wohin sie gehen und wo sie wie verarbeitet werden. Diese Darstellung eignet sich auch gut für parallele Prozesse.



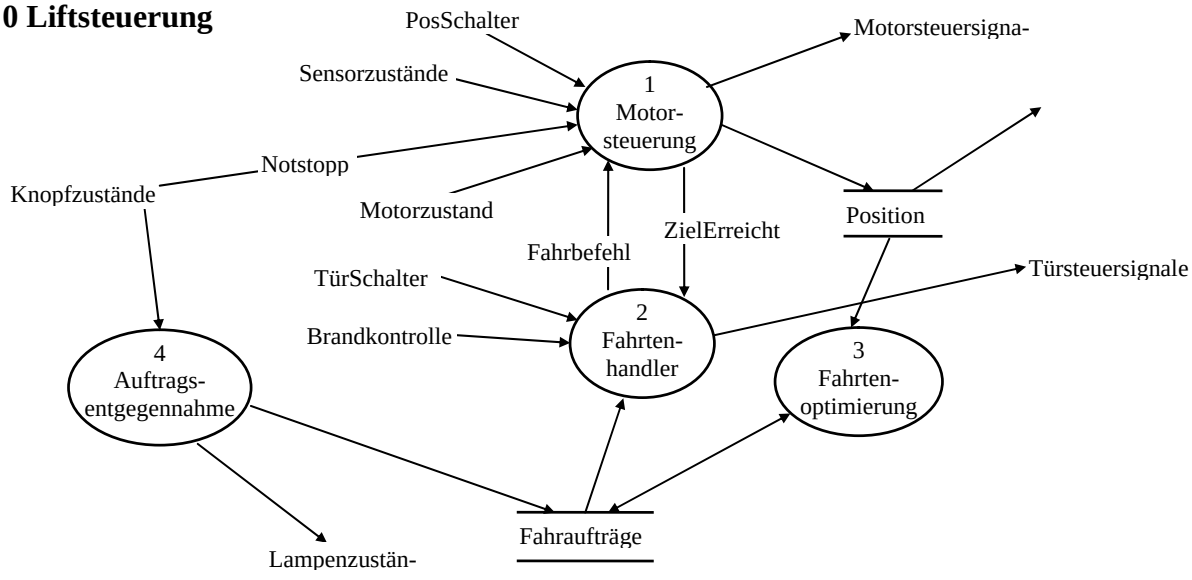
Kontext Diagramm

Das Kontextdiagramm stellt das System in seiner Umgebung dar. Es enthält nur einen einzigen Prozess, der das System enthält.



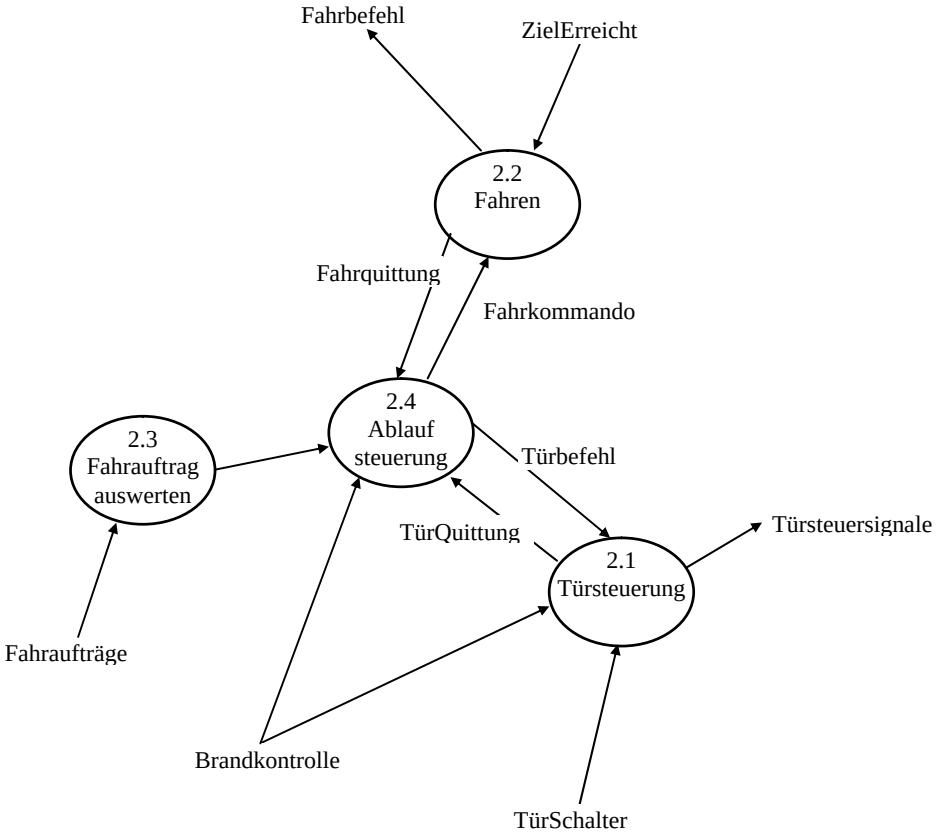
Im nächsttieferen Diagramm (Dem 0-Diagramm) wird der Prozess aus dem Kontextdiagram weiter aufgeteilt. Sämtliche Datenflüsse aus dem Kontextdiagramm müssen hier auch enthalten sein.

0 Liftsteuerung



Die einzelnen Prozesse können wiederum in detailliertere Datenflussdiagramme zerlegt werden. Diese Verfeinerung wird solange wiederholt, bis die Funktion des Systems klar wird.

2 Fahrtenhandler



32.1.5 CRC

Für jedes Modul (Eigentlich Klasse = Class) wird klar definiert wie es heisst, es wird beschrieben für was es verantwortlich ist (Responsibility), und mit welchen anderen Modulen (Klassen) es zusammenarbeitet (Collaborators)

Aufbau einer CRC-Card:

Modulname	
Modulaufgaben/ Funktionalitäten/ Verantwortlichkeit	Benötigte Module

Beispiel Liftsteuerung

Motorsteuerung	
Motor auf definierte Position bewegen. Motor auf Überlasst überwachen Motor auf Blockieren überwachen Motorgeschwindigkeit regeln	MotorHardwareAnsteuerung Sensorhandler Strommessung

Liftkabine Positionieren	
Sollposition bestimmen. Optimalen Geschwindigkeitsverlauf bestimmen. Lift auf Sollposition bewegen Istposition verfolgen Im Notfall auf nächstem Stock oder sofort halten, je nach Dringlichkeit.	Motorsteuerung Sensorhandler

32.1.6 Pseudocode

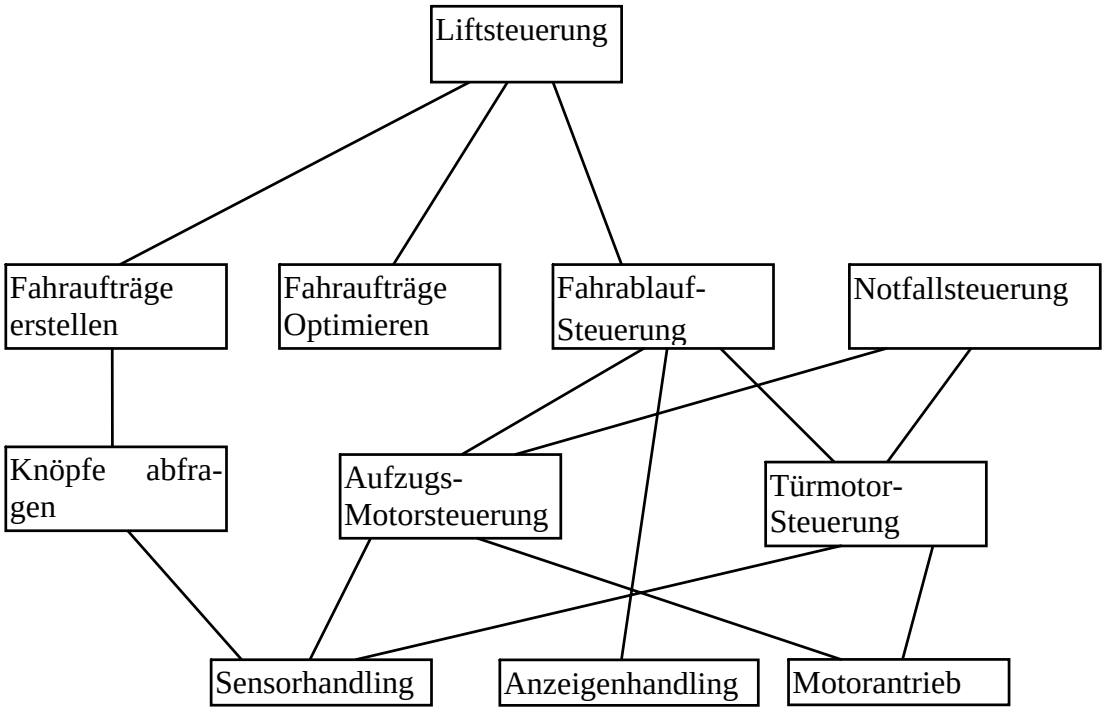
Die Lösung des Problems wird in einer Art Pseudoprogrammiersprache beschrieben:

Funktion FindeAusreisser

Lese alle Daten ein und speichere Sie ab
Berechne den Durchschnitt der Werte
Trage alle Werte, die mehr als 10% vom Durchschnitt abweichen in eine Liste ein.
Gib die Liste mit den Abweichenden Werten an das aufrufende Programm zurück.

32.1.7 Structured Design

Aufteilung des Projektes in einzelne Module mit definierten Verantwortlichkeiten und Darstellung der gegenseitigen Abhängigkeiten. Diese Darstellung ist vorallem für sequentielle Prozesse geeignet.



32.1.8 UML, Objektorientierte Analyse/Design

Bei der objektorientierten Analyse sucht man nach Objekten, deren Eigenschaften und Fähigkeiten sowie den Beziehungen der Objekte untereinander. Der objektorientierte Ansatz ist eine Weiterentwicklung der modularen Programmierung.

Bei diesem Vorgehen steht nun nicht mehr die Funktionalität, also das wie, sondern die Daten, also das was im Vordergrund. Ein Objekt ist eine Einheit, die bestimmte Eigenschaften (Attribute) aufweist, und bestimmte Befehle (Methoden) ausführen kann. Ein Objekt kann eine konkrete Entsprechung in der realen Welt haben, aber auch etwas abstraktes wie einen Algorithmus umfassen.

Beim objektorientierten Entwurf sucht man zuerst alle am Problem beteiligten Objekte, bestimmt anschliessend deren Eigenschaften (Attribute) und Aufgaben und findet heraus, wie diese Objekte miteinander in Beziehung stehen. Die zu den Daten gehörenden Funktionen (Aufgaben) werden dabei in die Objekte integriert. Man spricht dabei nicht mehr von Funktionen, sondern von Memberfunktionen oder Methoden. Die Daten werden entsprechend als Attribute (Eigenschaften) oder Member bezeichnet. Die Objekte kommunizieren untereinander um die Aufgabe des Gesamtsystems zu erledigen.

Funktionsaufrufe resp. Methodenaufrufe werden in der Literatur oft als 'Botschaften' an ein Objekt bezeichnet. (Z. B. Objekt A sendet die Botschaft 'ZeichneDich()' an Objekt B).

Jedes Objekt gehört zu einer Klasse, gleichartige Objekte gehören zur selben Klasse. (Gleichartige Objekte besitzen alle die gleichen Eigenschaften, nur können sie anders ausgeprägt sein. Zwei Objekte können z. B. die Eigenschaft Farbe besitzen, wobei sie beim einen rot, und beim anderen grün ist)

Die objektorientierte Methode kann in jeder Programmiersprache angewendet werden (Sogar Assembler), aber es gibt Sprachen, die speziell dafür entwickelt worden sind wie Smalltalk, C++ oder Java. (Das heisst aber noch nicht, dass man bereits objektorientiert Programmiert, sobald man eine dieser Sprachen benutzt, die Sprache ist nur das Werkzeug)

Die wichtigsten Grundsätze der objektorientierten Programmierung sind:

Kapselung	Auf die Daten (Attribute) eines Objektes kann von aussen nicht zugegriffen werden, sie sind in das Objekt eingepackt. Nur Funktionen die zum Objekt gehören können auf seine Daten zugreifen.
Vererbung	Eine Klasse übernimmt grundsätzlich einmal alles von einer bestehenden Klasse (Erbt von der Klasse, wird von dieser Klasse abgeleitet), und wird dann um zusätzliche Eigenschaften oder Fähigkeiten erweitert, oder bestehende Fähigkeiten werden angepasst.
Polymorphismus	Heisst, dass ein Objekt mehrere Gestalten annehmen kann. Zur Codierungs und zur Compilezeit ist nicht mehr genau festgelegt, welcher Objekttyp jetzt wirklich verwendet wird. Das Objekt muss nur die an dieser Stelle nötigen Fähigkeiten besitzen, zur Laufzeit können die Objekte dynamisch ersetzt werden, solange sie nur die benötigten Fähigkeiten haben. (Z. B. für eine graphische Ausgabe wird ein Ausgabeobjekt benötigt. Dabei ist zur Compilezeit vielleicht gar nicht bekannt, auf welches Objekt genau ausgegeben wird, es muss nur den Befehl Zeichne() verstehen. Es könnte ein Laserdrukker, ein Plotter oder nur ein Fenster auf dem Bildschirm sein, je nach dem, wo der Benutzer gerade seine Ausgabe haben möchte).

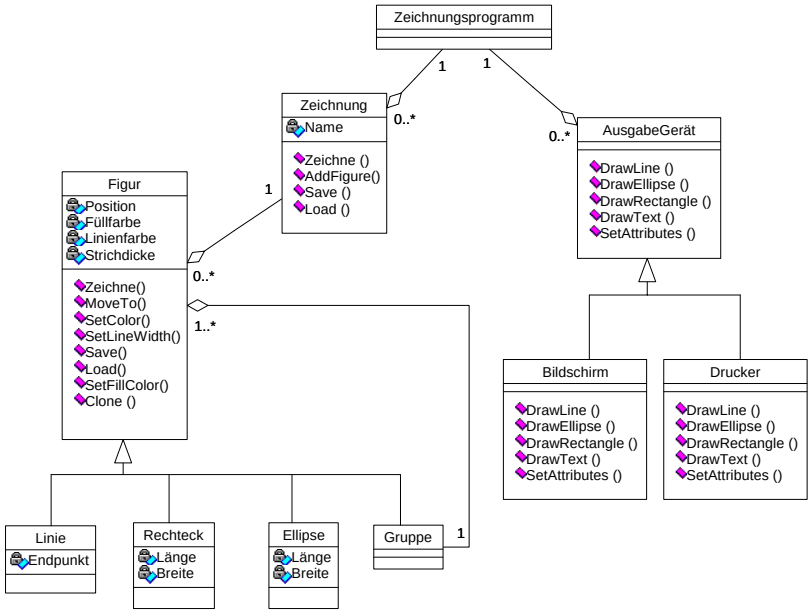
Beispiel:

In einem Graphikprogramm können Zeichnungen erstellt werden. Die Zeichnung kann Figuren wie Linien, Rechtecke und Ellipsen enthalten. Bei allen Figuren kann die Strichdicke, die Strichfarbe und die Füllfarbe eingestellt werden. Mehrere Figuren können zu einer Gruppe zusammengefasst werden, welche anschliessend wie eine Figur behandelt werden kann. Figuren können gezeichnet, verschoben, kopiert, gespeichert, geladen und gelöscht werden. Die Eigenschaften der Figuren (Farbe, Strichdicke) können jederzeit verändert werden. Die Zeichnung wird auf dem Bildschirm dargestellt, kann aber auch auf einem Drucker ausgegeben werden. Eine Zeichnung kann abgespeichert und wieder geladen werden. Das Ausgabegerät (Bildschirm, Drucker) stellt Funktionen zum Zeichnen der Grundfiguren (Linie, Rechteck, Ellipse) zur Verfügung.

Es können mehrere Zeichnungen gleichzeitig geladen sein. Jede Zeichnung hat einen eindeutigen Namen, unter dem sie auch abgespeichert wird.

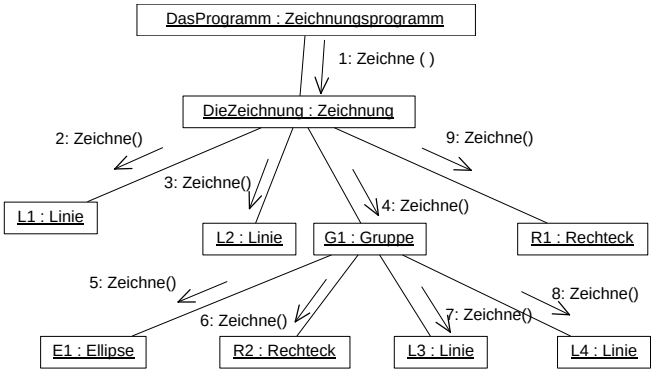
Klassendiagramm

Im Klassendiagramm werden die Klassen mit ihren Beziehungen untereinander, sowie ihre Methoden und Attribute dargestellt.

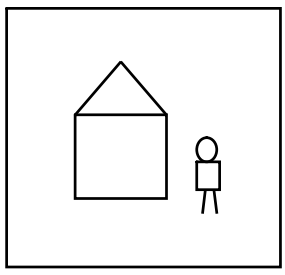


Kollaborationsdiagramm

Im Kollaborationsdiagramm wird ein bestimmter Zustand des Systems mit allen beteiligten Objekten, sowie ein Kommunikationsablauf für ein bestimmtes Ereignis dargestellt. Damit können Abläufe im System durchgespielt und der Entwurf auf Korrektheit und Vollständigkeit überprüft werden.



Dem Diagramm entsprechende Zeichnung



33 Systematisches Testen von Software

Bevor ein Softwaresystem seinem Verwendungszweck übergeben werden kann, muss verifiziert werden, ob das System seinen Spezifikationen entspricht. Ein System sollte aber nicht erst am Ende der Entwicklungphase getestet werden, sonst sind viele Fehler nur sehr kostspielig zu beheben.

Es ist also anzustreben, Fehler möglichst frühzeitig zu finden. Konzeptionelle Fehler sollten dabei schon in der Analyse/Designphase erkannt und behoben werden, da sonst das gesamte Projekt in Frage gestellt wird. Auch bereits getestete Module sollten wieder überprüft werden, wenn sie in einem neuen Kontext verwendet werden. Auf jeden Fall müssen die Spezifikationen der Module mit den jeweiligen Anforderungen verglichen werden.

Der Absturz der ersten ARIANE 5 im Jahre 1996 ist auf ein ungeprüftes Übernehmen von Software der Ariane 4 zurückzuführen. Es wurde das Trägheitsnavigationssystem der ARIANE 4 übernommen, dabei wurde auf ausführliche Tests verzichtet, da sich das System in der ARIANE 4 jahrelang bewährt hatte. Leider hatte die ARIANE 5 aber eine höhere horizontale Geschwindigkeit, was zu einem Zahlenüberlauf führte, und dies wiederum ziemlich schnell zum Absturz (Resp. kontrollierten Sprengung) der Rakete. Schadensumme: 1.7 Miliarden DM. Wenn das System nicht einfach unbesehen übernommen worden wäre, hätte man das Problem mit grosser Sicherheit erkannt. (Weitere Informationen dazu unter http://www.esa.int/export/esaCP/Pr_33_1996_p_EN.html)

Viele Fehler äussern sich nicht so spektakulär, aber die Gesamtschadenssumme die jährlich durch nicht sorgfältig durchgeführte Tests entsteht dürfte enorm hoch sein.

Grundsätzlich unterscheidet man zwei Arten von Tests: Statische Tests und Funktionstests.

33.1 Statische Tests (Reviews)

Zu den *statischen Tests* gehört die *Reviewtechnik*, dabei werden in jeder Phase alle Dokumente kritisch auf Fehler und Inkonsistenzen untersucht. Diese Technik kann auf Diagramme, Spezifikationen, Pflichtenhefte und effektiven Code (Listings) angewendet werden. Dabei wird kein Code ausgeführt, sondern es findet eine reine visuelle Kontrolle statt.

Ein Review wird dabei nach vorher festgelegten Richtlinien (Checklisten) durchgeführt. Es kann genauso auf Einhaltung des Codierungstandards (Komentare, Namensgebung, Verschachtelungstiefe, Funktionsgrösse) wie auf die Codierung selbst geprüft werden.

In einem Review werden gefundene Fehler oder Unstimmigkeiten nicht behoben, sondern nur in einem Bericht festgehalten. Anschliessend an das Review können zusammen mit dem Autor Lösungsvorschläge diskutiert werden.

Das Review kann zusammen mit dem Autor, oder auch ohne den Autor des jeweiligen Dokumentes stattfinden. Wichtig, bei einem Review sind weder Schuldzuweisungen noch Verteidigungen angebracht, es werden einfach Erkenntnisse sachlich festgehalten.

In einem Projekt sollte jedes Dokument nach seiner Fertigstellung einem Review unterzogen werden.

Ein positiver Seiteneffekt der Reviewtechnik ist zudem die breitere Streuung des Projektwissens über die Projektmitglieder. Da die Projektmitglieder sich jeweils gegenseitig reviewen, weiss jedes Mitglied ungefähr was das andere macht, dadurch wird es leichter, Aufgaben umzuverteilen und neue Mitarbeiter einzubinden. Zudem wird über das ganze Projekt der Programmierstil einheitlicher, weil man sieht wie andere ihre Arbeit erledigen.

Wichtig, die Ergebnisse der Reviews dürfen nicht zur Qualifikation der Mitarbeiter verwendet werden, sie dienen nur der Qualitätssicherung innerhalb der Projekte. Sobald sie für Mitarbeiterqualifikation missbraucht werden, verlieren sie ihr Effizienz ('Gefälligkeitsgutachten').

33.2 Funktionstests

Beim *Funktionstest* wird geprüft, ob ein System oder Teilsystem korrekt funktioniert und seinen Spezifikationen entspricht, also das tut was von ihm gefordert wird. Das Ziel des Tests ist es *Fehler zu finden.*

Es ist wissenschaftlich belegt, das es unmöglich ist, die Fehlerfreiheit eines Programmes zu beweisen. Daraus folgt auch, das vollständiges Testen unmöglich ist. Aber durch systematisches Vorgehen kann man ein optimales Ergebnis erreichen.

Planloses Ausprobieren ist keine brauchbare und zuverlässige Testmethode

Testen ist eine kreative und anspruchsvolle Tätigkeit. Der Tester muss das Produkt gut kennen, und sein Ziel muss sein, Fehler zu finden, nicht die Korrektheit zu bestätigen. Das Testen ist selbst ein Projekt, welches geplantes Vorgehen erfordert. Der Tester muss sich überlegen was getestet werden soll, und wie es getestet werden kann. Er muss sich zudem überlegen, was alles schiefgehen kann. Dabei wird für jede in Frage kommende Möglichkeit ein *Testfall* spezifiziert.

Für jeden Testfall muss definiert werden, in welchem Ausgangszustand das System sein soll, welche Eingabedaten an das System erfolgen, und was die erwartete Reaktion des Systems ist. Tests müssen reproduzierbar sein, d.h. dass der gleiche Test wiederholt ausgeführt werden kann (Nur so kann eine Fehlerbehebung verifiziert werden). Tests können oft automatisiert werden, so dass der Tester von Fleissarbeit entlastet wird. Für Tests muss häufig eigene Hard und Software entwickelt werden.

Wichtig: Testen und Fehlerbehebung sind zwei verschiedene und getrennte Tätigkeiten. Das Ergebnis eines Tests ist ein Fehlerprotokoll, die Fehlerkorrektur erfolgt anschliessend durch den(die) Programmierer aufgrund des Protokolls.

Für das Amt des Testers eignen sich besonders Leute, die ein grosses Interesse daran haben, möglichst viele Fehler zu finden. Deshalb sind im Allgemeinen die Entwickler des Systems ungeeignet. Besser eignen sich Leute, die das System später Warten oder in Betrieb nehmen müssen, zukünftige Anwender oder Leute aus der Qualitätssicherung.

Es gibt grundsätzlich zwei Methoden für Funktionstests: *Whiteboxtest* und *Blackboxtest*.

33.3 Blackboxtest

Beim Blackboxtest sind die Interna des Systems unbekannt, das System wird als Blackbox angesehen, und die Tests werden aus den Anforderungen (Use Cases) an das System gewonnen. Blackbox Tests werden meist gegen Ende des Projektes zum Test von Teilsystemen und des Gesamtsystems eingesetzt. Da Blackboxtests auf den Anforderungen an das System basieren, können die Testfälle parallel zur Entwicklung des Gesamtsystems entworfen werden. Das Testteam muss nicht bis zum Projektende mit der Ausarbeitung der Tests warten.

33.4 Whiteboxtest

Beim Whitebox Test sind die Interna das Systems (Programmcode) bekannt. Beim Erstellen der Testfälle wird versucht, möglichst alle Programmflusspfade abzudecken, also die Testfälle so zu wählen, dass alle Fälle eines Cases oder einer Verzweigung mindestens einmal durchlaufen werden, und dass Schlaufen an ihren Grenzfällen betrieben werden (Einmal, keinmal, maximal). Diese Art von Tests werden vor allem bei Modultests eingesetzt.

33.5 Systematisches erzeugen von Testfällen

Da die Anzahl der möglichen Kombinationen von Eingabedaten meist nahezu unendlich ist, muss beim Entwurf von Testfällen systematisch vorgegangen werden um möglichst alle Fälle abzudekken. Ein zufälliges Auswählen von Werten ergibt keinen zuverlässigen Test.

Eine gute Möglichkeit ist die Bildung von *Äquivalenzklassen*. Dabei werden Gruppen von Werten gebildet, von denen man ähnliches Verhalten und ähnliche Ergebnisse erwartet.

Mögliche Klassen wären: Alle positiven Werte, Alle negativen Werte, Alle Werte oberhalb/unterhalb des spezifizierten Wertebereichs, Alle Kombinationen die denen Wert A grösser als Wert B ist (Bargeldbezug wenn Kontostand kleiner als gewünschter Wert ist), Rückgeld grösser als Geldreservoir ist, ...

Anschliessend wählt man die Testfälle so, das Werte aus allen Äquivalenzklassen zum Zuge kommen. Meist werden einige Werte zufällig irgendwo aus der Mitte der jeweiligen Äquivalenzklassen, sowie Grenzwerte (Um eins daneben, gehört gerade noch dazu/nicht mehr dazu) der Klassen gewählt.

Damit lässt sich die Menge der Testfälle drastisch reduzieren, ohne die Qualität des Tests massiv zu vermindern. Es ist aber wichtig, gute Äquivalenzklassen zu bilden.

33.6 Testabbruch

Die schwierigste Entscheidung beim Testen ist immer, wann die Testphase beendet wird. Es gibt im Allgemeinen keine Möglichkeit mit Sicherheit zu wissen, ob alle Fehler beseitigt sind, also müssen andere Kriterien herangezogen werden.

Einfach eine bestimmte Zeitdauer festzulegen ist sicherlich das schlechteste Kriterium. Bessere Möglichkeiten basieren auf einer *Restfehler-Abschätzung*, d.h. versuchen zu schätzen, wieviele Fehler noch übrig sind.

Dazu bietet sich die Fehlerfinderate an. (Gefundene Fehler pro Zeiteinheit, ev. noch nach Schweregrad gewichtet). Der Test wird abgebrochen, wenn die Fehlerfinderate eine gewisse Grenze unterschreitet. Diese Grenze wird vor dem Testbeginn festgelegt.

Eine anderer Ansatz ist die Kostenrechnung, der Test wird abgebrochen, wenn der durchschnittliche Aufwand (Die Kosten) um einen Fehler zu finden, eine vorbestimmte Grenze überschreitet.

Eine Möglichkeit, die Restfehler zu schätzen besteht daran, mehrere Gruppen unabhängig voneinander dasselbe System testen zu lassen. Aus den Ergebnissen der unterschiedlichen Gruppen lassen sich Rückschlüse auf die nicht gefundenen Fehler ziehen.

Beispiel:

Von beiden Gruppen gefundene Fehler:	80
Nur von Gruppe A gefundene Fehler:	10
Nur von Gruppe B gefundene Fehler:	8
Von keiner Gruppe gefundene Fehler:	??

Aus diesen Werten kann man schliessen, dass noch ein unentdeckter Fehler im System steckt.

Idee: Gruppe B hat von den 90 Fehlern der Gruppe A 80 gefunden, die Erfolgsquote von B ist also 8:9. Wenn man die selbe Erfolgsquote auf die nur von Gruppe B gefundenen Fehler anwendet, ergibt sich, dass noch ein Fehler unentdeckt ist. Diese Methode funktioniert aber nur, wenn beide Gruppen völlig unabhängig voneinander testen (Keine Kommunikation, auch nicht in der Kaffeepause oder nach Feierabend).

Damit solche Abschätzungen wirklich aussagekraftig sind, muss statistisch korrekt vorgegangen und ausgewertet werden, also die Wahrscheinlichkeitsrechnung herangezogen werden.

34 Projektorganisation und Projektleitung

Es hat sich gezeigt, dass die *Projektorganisation* von grosser Wichtigkeit ist. Es muss darauf geachtet werden, dass die Aufgabenverteilung und die Kompetenzen von Anfang an klar sind und während der Projektzeit eingehalten werden.

Zu jedem Projekt gehört ein Projektleiter. Der *Projektleiter* wacht über die Einhaltung der Kompetenzen und verteilt die Aufgaben an seine Mitarbeiter.

Bei kleinen Projekten stellt die Projektleitung einen Teilzeitjob dar. Der Leiter arbeitet in der Restzeit als gleichberechtigtes Mitglied in Team mit. Diese Konstellation erfordert von den Mitarbeitern und dem Leiter Flexibilität, da Rollenwechsel stattfinden

Der Projektleiter wacht im weiteren über die Einhaltung des *Zeitplans*, des *Budgets*, der *Qualität* und des *Pflichtenhefts* und vertritt das Projekt gegen aussen. Beim Nichteinhalten von Vorgaben ergreift der Projektleiter geeignete Massnahmen.

Das Hauptführungsinstrument ist die *Teamsitzung*. Die Leitung übernimmt der Projektleiter. Er ist dafür verantwortlich, dass Entscheidungen gefällt und umgesetzt werden. Der Projektleiter kann zur eigenen Entlastung Aufgaben delegieren.

Software wird heute fast ausnahmslos im Team, d.h. in Zusammenarbeit mit anderen erstellt. Die dabei auftretenden gruppendynamischen Prozesse dürfen nicht unterschätzt werden. Das Teamgefüge entscheidet oft über Erfolg oder Misserfolg eines Projekts. Softwaretechnische Kenntnisse können im Gegensatz zu menschlichen Qualitäten um einiges einfacher verbessert werden.

Es ist Aufgabe des Projektleiters bei Teamschwierigkeiten Massnahmen zu ergreifen: z.B. Aussprache, Umverteilung von Kompetenzen, Überwachung...

34.1 Zeitplanung

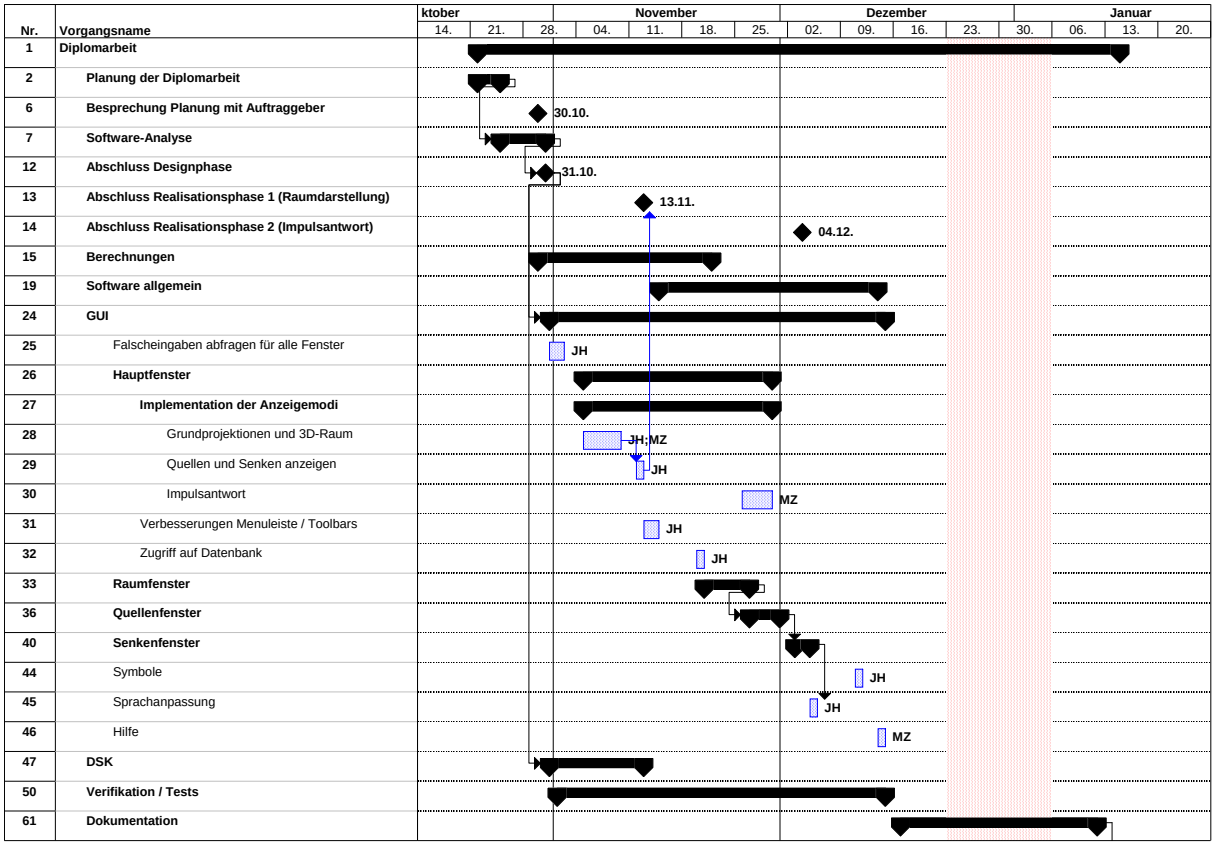
Um ein Projekt erfolgreich durchführen zu können, ist eine *Zeitplanung* unerlässlich. Dazu wird das Projekt in einzelne Arbeitsschritte aufgeteilt, welche nacheinander oder nebeneinander ausgeführt werden können. Anschliessend wird für jeden Arbeitsschritt der benötigte Zeitaufwand abgeschätzt. Unter der Berücksichtigung von gegenseitigen Abhängigkeiten und verfügbaren Mitarbeitern wird anschliessend der Gesamtzeitaufwand ersichtlich, oder wenn der Einführungszeitpunkt bekannt ist, kann die Anzahl der benötigten Mitarbeiter abgeschätzt werden. (Achtung: Eine Verdoppelung der Mitarbeiter hat nicht eine Verdoppelung der Kapazität zur folge, da der Kommunikations- und Organisationsaufwand zunimmt. Zudem können nicht alle Arbeiten parallel zueinander ausgeführt werden.)

Wenn ein Projekt zuviel Unbekannte (Meist sind schon 2 eine zuviel) enthält, ist es oft gar nicht möglich, eine zuverlässige Zeitschätzung zu machen, es muss auf Erfahrungswerte zurückgegriffen werden. Aber auch in diesem Fall ist der Terminplan ein wichtiges Hilfsmittel, weil Schätzungsfehler damit schon früh erkannt werden und frühzeitig die nötigen Massnahmen ergriffen werden können.

Innerhalb eines Projektes werden zudem *Meilensteine* definiert, das sind Zeitpunkte, zu welchen bestimmte Ziele des Projektes erreicht worden sein sollten. Spätestens zu diesen Zeitpunkten findet ein Vergleich zwischen Soll und Ist statt, und bei Abweichungen wird über das weitere Vorgehen entschieden (Teile weglassen, auf später verschieben, vorziehen, Team vergrössern/Verkleinern, Budget überarbeiten). Meilensteine dienen bei grösseren Projekten auch als überblickbare *Zwischenziele*, um nicht nur den Endtermin in 5 Jahren vor Augen zu haben.

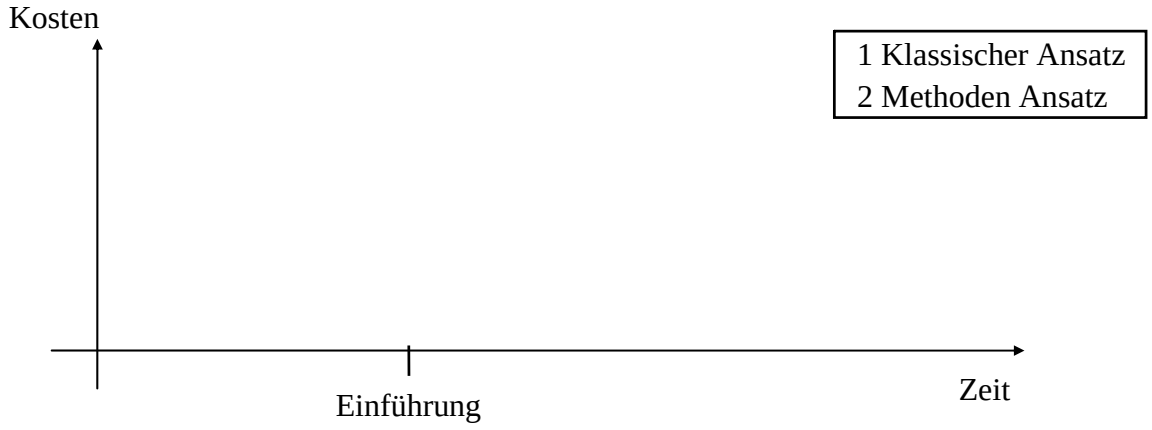
Bei einer guten Zeitplanung können Abweichungen von der kalkulierten Projektzeit sehr früh erkannt werden, ohne Zeitplanung wird meist erst viel zu spät erkannt, dass ein Projekt aus dem Ruder läuft.

Beispiel eines Terminplanes:



Die einzelnen Punkte werden soweit verfeinert, bis eine Abschätzung des Zeitaufwandes möglich ist. Die Länge eines Balkens ist durch den Zeitaufwand, die Anzahl der gleichzeitig daran arbeitenden Personen und den Beteiligungsgrad dieser Personen gegeben. Es muss beachtet werden, das die Gesamtarbeitslast einer Person zu keinem Zeitpunkt 100% überschreitet.

34.2 Verlauf der Projektkosten



34.3 Versionsverwaltung

Ein weiterer wichtiger Punkt in grösseren und/oder langlebigen Softwareprodukten ist die *Versionsverwaltung*. Ohne Versionsverwaltung ist es oft nicht mehr möglich, den Stand von älteren Programmversionen wiederherzustellen. Aber gerade zu Wartungs- und Fehlerbehebungszwecken ist es oft unumgänglich, die der Releaseversion des fehlerverursachenden Programmes entsprechenden Sourcefiles zur Verfügung zu haben.

Mit einer guten Versionsverwaltung ist es jederzeit möglich, jeden beliebigen Versionstand des Sourcecodes wiederherzustellen.

Im einfachsten Fall wird bei jedem Release schlicht der gesamte Sourcecode archiviert. Unter Verwendung von Versionsverwaltungssystemen wie z. B. RCS oder CVS kann das Verwalten von Versionen vereinfacht und automatisiert werden, es wird dabei auch überwacht, dass nicht mehrere Personen gleichzeitig an derselben Datei arbeiten. Solche Systeme speichern meist nicht den gesamten Code jedesmal vollständig ab, sondern jeweils nur die Differenzen zum letzten Release.

Grosse oder langlebige Softwareprojekte können ohne Versionsverwaltung kaum vernünftig verwaltet werden, insbesondere weil ständig an irgend einem Teil weiterentwickelt wird und es kaum möglich ist, den Überblick zu behalten wenn jeder unkontrolliert den Sourcecode ändert.

Übungsaufgabe 33.1:

Es soll eine Applikation entworfen werden, welche in einer Adressdatenbank doppelte Einträge erkennt und entfernt. Es soll eine Logdatei über alle entfernten Einträge geführt werden, sowie alle entfernten Einträge in einer separaten Datei abgelegt werden. Als Doppeleinträge sollen auch unterschiedliche Schreibweisen, Tippfehler, und Verdreher erkannt werden. Die Applikation soll in Form einer Bibliothek entworfen werden, welche bei unterschiedlichen Datenbanken eingesetzt werden kann.

Analysieren sie diese Aufgabe, entwerfen Sie einen Grobdesign und einen Terminplan. Den Terminplan können Sie für eine oder mehrere Personen erstellen.

35 Anhang A, weiterführende Literatur

Programmieren in C, 2. Ausgabe **Brian Kernighan, Dennis Ritchie** Verlag Hanser ISBN 3-446-15497-3	Das Standardwerk (Erfinder von C), nicht unbedingt ein Lehrbuch.
The C programming Language, 2nd Edition **Brian Kernighan, Dennis Ritchie** Verlag Prentice Hall ISBN 0-13-110362-8 Taschenbuch ISBN 0-13-110370-9 Hardcover	Englische Originalausgabe
Goto C Programmierung **Guido Krüger** Verlag Addison Wesley ISBN 3-8273-1368-6	Gutes Einsteigerbuch
Informatik für Ingenieure, 3, vollst. überarb. Aufl. **Gerd Küveler, Dietrich Schwoch** Verlag Vieweg ISBN 3-528-24952-8	Breitgefasstes Einsteigerbuch zum Thema Informatik, inkl. Einführung in C und C++
Algorithmen in C **Robert Sedgewick** Verlag Addison Wesley ISBN 3-8931-9376-6	Grosse Sammlung von Algorithmen, es gibt auch Varianten des Buchs für Pascal und C++ ('Algorithmen in C++' und 'Algorithmen')
The Art of Computer Programming **Donald E. Knuth** **Volume 1, Fundamental Algorithms**, 3rd Edition ISBN 0-201-89683-4 **Volume 2, Seminumerical Algorithms**, 3rd Edition ISBN 0-201-89684-2 **Volume 3, Sorting and Searching**, 2nd Edition Verlag Addison Wesley ISBN 0-201-89685-0	Die Referenz für Algorithmen und Datenstrukturen.
Software Engineering, 6. Auflage **Ian Sommerville** Verlag Pearson Studium (Addison Wesley) ISBN 3-8273-7001-9	Umfassendes Buch zum Thema SW-Engineering. Sehr empfehlenswert wenn man in die SW-Entwicklung einsteigen will.
Die C++ Programmiersprache, 4. aktualisierte Auflage **Bjarne Stroustrup** Verlag Addison Wesley ISBN 3-8273-1660-X	Das Standardwerk zu C++, (Erfinder von C++) nur wer mehr über OOP erfahren möchte.

36 Anhang B, Debugging Tips und Tricks

Compilerwarnungen beachten.

Ein Modul sollte sich möglichst ohne Warnungen compilieren lassen, jede verbleibende Warnung überprüfen, nicht einfach ignorieren. Bei den Compileroptionen möglichst alle Warnungen einschalten.

Auf nicht initialisierte Variablen achten.

Probleme mit Programmteilen, die einmal laufen und anschliessend nie mehr, oder einmalig unkorrekt und anschliessend fehlerfrei laufen, lassen sich oft auf uninitialisierte Variablen (Insbesondere Pointer) zurückführen.

Auf Überschreiten von Arraygrenzen achten

In C kann problemlos über Arraygrenzen hinaus geschrieben werden (oben und unten), was oft zu Totalabstürzen, manchmal aber nur zur Veränderung von benachbarten Variablen führt (Schwer lokalisierbarer Fehler !!!). Möglicherweise in Debugversion vor jedem Arrayzugriff Index auf Bereich überprüfen (Achtung, ineffizienter Code!)

Selbstüberprüfenden Code schreiben.

Zu Beginn jeder Funktion die Argumente auf Plausibilität prüfen, insbesondere NULL-Pointer. Testfunktionen schreiben, die Datenstrukturen (Listen, Bäume, ...) auf ihre Integrität hin untersuchen. Diese Testroutinen beim Debuggen an strategisch wichtigen Punkten aufrufen.

Fehler eingrenzen.

Durch Ausgaben, Statusanzeigen oder grobschrittiges Debuggen Fehler eingrenzen, bis fehlererzeugende Funktion oder Codezeile lokalisiert ist. Eventuell Hilfscode schreiben um Fehler weiter einzugrenzen.

Fehler isolieren.

Funktionen, die als Fehlerquelle in Frage kommen durch vereinfachte Versionen ersetzen und prüfen ob Fehler verschwindet, so kann fehlerhafte Funktion gefunden werden.

Funktionen separat testen.

Verdächtige Funktionen aus der Applikation entfernen und in ein einfaches Testprogramm einbauen, das die Funktion während längerer Zeit mit Testdaten oder Zufallsdaten aufruft und das Ergebnis jeweils überprüft, dieser Test kann mit einer Integritätskontrolle kombiniert werden.

Statusinformationen ausgeben

Systemzustand ausgeben, ev. auf LED oder freie Portbits, wenn keine andere Ausgabemöglichkeit vorhanden ist. So kann der Lauf des Programms verfolgt und unerwünschtes Verhalten erkannt werden. Auch Timingprobleme können so analysiert werden; dies ist meist sogar die einfachste Möglichkeit, Timingprobleme zu erkennen.

Bei Sporadischen Fehlern

Ein Logfile oder einen Tracebuffer (Grosses Array mit ausreichend Platz für Einträge, als Ringbuffer aufgebaut der immer die n letzten Ereignisse bereit hält) einfügen, worin ständig Systemzustände oder wichtige Ereignisse (Funktionsaufrufe, Argumente, Interrupts, Benutzereingaben, Systemzeit in ms/us) abspeichert werden. Im Fehlerfall den Inhalt von Logfile oder Tracebuffer sichern. Die Analyse dieser Daten kann helfen, den Fehler zu lokalisieren.

Fehler in Speicherverwaltung

Bei Verdacht auf Fehler in Speicherverwaltung, oder auch als Vorsichtsmassnahme kann eine eigene Speicherverwaltung benutzt werden, die auf typische Fehler in der Speicherbenutzung achtet: Rückgabe nicht allozierten Speichers, doppelte Rückgabe desselben Speicherblocks, Allozieren aber nie Freigeben eines Speicherblockes, beschränkt auch das Überschreiten der Speicherblockgrenzen (Block am Anfang und am Ende grösser machen und diese Schutzbereiche mit Testmuster füllen, bei Rückgabe prüfen ob diese Testmuster unversehrt sind).

Spezialcode für Fehlerfall schreiben

Extracode schreiben, der beim Erkennen eines Fehlers aufgerufen wird und möglichst viele Informationen des aktuellen Systemzustandes ausgibt oder in ein File schreibt, oder einen Breakpoint auf diesen Extracode setzen, so dass im Fehlerfall der Debugger zum Zuge kommt.

Bei HW-naher Programmierung auf volatile und Stacküberlauf achten

Variablen die von der HW oder Interrupts verändert werden, müssen als volatile deklariert werden. Lokale Variablen von Funktionen werden auf dem Stack abgelegt, lokale Arrays können bei Mikrokontrollern schnell zu Stacküberlauf führen. Unerklärliches Fehlverhalten oder Abstürze können auch auf einen Stacküberlauf zurückzuführen sein.

37 Index