Otto-von-Guericke-Universität Magdeburg

Grundlagen der Informatik für Ingenieure

Begleitmaterial zur Vorlesung

Georg Paul Venzislav Nikolov Dirk Jesko Torsten Mähne

2., überarbeitete Ausgabe
7. September 2001

Doz. Dr.-Ing. habil. Georg Paul
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Institut für Technische- und Betriebliche Informationssysteme

Dr. Venzislav Nikolov
Universität für Chemische Technologie und Metallurgie Sofia
Abteilung Informatik

Dipl.-Inf. Dirk Jesko
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Institut für Technische- und Betriebliche Informationssysteme

Torsten Mähne
Otto-von-Guericke-Universität Magdeburg
Fakultät für Elektrotechnik und Informationstechnik

Vorwort

Das vorliegende Skript begleitet die gleichnamige Vorlesung, die von Dozent Dr.-Ing. habil. Georg Paul für Ingenieure und weitere Studiengänge angeboten wird. Der Inhalt des Teiles 1 wird in 2 Semesterwochenstunden vermittelt. Zur Vertiefung des Stoffes werden Laborübungen durchgeführt. Der erste Teil beinhaltet als Schwerpunkte eine Einführung und wesentliche Gedanken zu Algorithmen, um dann darauf aufbauend in die Programmierung in C/C++ einzusteigen. Hierbei werden Datenstrukturen aufgebaut und geeignete Algorithmen darauf ausgeführt. Zahlreiche Anwendungsbeispiele unterstützen die Lernarbeit. Teil 2 wird in einer 14-tägigen, zweistündigen Vorlesung fortgeführt, wobei Aspekte der Grafik, von Datenbanksystemen und Softwaretechnologie besprochen werden. Die Vorlesung wird begleitet durch eine umfangreiche Sammlung von Übungsaufgaben und Lösungen. Das Skript wurde auch auf Anregung vieler Studierender überarbeitet.

Magdeburg, 30. August 2001 Dr.-Ing. habil. Georg Paul

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ASCII American Standard Code for Information Interchange

ANSI American National Standards Institute

BNF Backus-Naur-Form

CAx Computer Aided x (Rechnerunterstützung in der Domäne x)

CAD Computer Aided Design

CISC Complex Instructions Set Computer

CPU Central Processing Unit

DB Datenbank

DBMS Datenbank-Management-System

DBS Datenbanksystem

DDL Data Definition Language

DML Data Manipulation Language

DV Datenverarbeitung

DVS Datenverarbeitungssystem

EBNF Erweiterte Backus-Naur-Form

EDV Elektronische Datenverarbeitung

ERM Entity Relationship Modell

GKS Graphic Kernel System

GLU OpenGL Utility Library

GLUT OpenGL Utility Toolkit

GPL GNU Public License

IT Informationstechnik

KI Künstliche Intelligenz

OpenGL Open Graphics Library

PAP Programmablaufplan

PPS Produktionsplanung und -steuerung

QL Query Language

RAM Random Access Memory

RISC Reduced Instruction Set Computer

ROM Read Only Memory

RTL Röhren-Transistor-Logik

SEP Softwareentwicklungsprozeß

Si Silizium

SPARC Standard Planning And Requirement Commitee

SQL Structured Query Language

TTL Transistor-Transistor-Logik

ULSI Ultra Large Scale Integration

VLSI Very Large Scale Integration

1 Einführung

Die Lehrveranstaltung „Grundlagen der Informatik für Ingenieur“ wurde erstmals vom Vorlesenden 1988 angeboten. In den vergangenen Jahrzehnten hat die Informatik bzw. die Informations- und Kommunikationstechnik eine rasante Entwicklung genommen, die auch auf die Anwendungsfelder der Informatik durchschlägt. Voraussetzung dafür waren vor allem die enormen Fortschritte in der Hardwareentwicklung. Mußte vorher mit jedem Speicherbit oder jeder Sekunde Rechenzeit gegeizt werden, so sind heute Rechner verfügbar, die umfangreiche Softwareprodukte mit kurzen Antwortzeiten bedienen. Damit haben sich zum Teil auch die Erwartungshaltung der Anwender an die Informatik gewandelt. Diese drückt sich z.B. darin aus, daß der Anwender der Informationstechnik (im Maschinenbau, in der Elektrotechnik, in der Sporttechnik, u.a.) zumeist vorgefertigte Tools nutzen möchte, die sein technisches Problem lösen helfen, ohne daß er selbst tief in die „Geheimnisse“ der Informatik eindringen muß. Diese Forderung ist akzeptabel, schließt aber eine Beschäftigung mit der Informationstechnik aus mindestens 3 Gründen nicht aus:

- Der Anwender muß die Einsatzbedingungen der vorgefertigten Softwarewerkzeuge einschließlich ihrer Schnittstellen selbst erschließen können.
- Er muß diese durch eigene Anwendungen erweitern können.
- Er muß ein Grundverständnis für die Informatik entwickeln, um im Team die Sprache des Partners zu verstehen.

Aus diesen Zielstellungen heraus wurde diese Vorlesung ständig überarbeitet.

1.1 Information als Element der Technik

„Mit der weiteren Entfaltung der wissenschaftlich-technischen Revolution wird das Verhalten des Menschen zu den kulturellen Bildungsgütern, dem sich entwickelnden Weltwissen, den Alltagsinformationen und zu der individuellen geistigen Welt selbst durch die moderne Informationstechnik wesentlich beeinflußt ... Der Mensch im Mittelpunkt stehend, wird sich des Computers auf allen Gebieten gesellschaftlichen Lebens bemächtigen. Da aber nun alles, was der Mensch beabsichtigt, den Weg durch seinen Kopf gehen muß, steht für alle Zeiten der Existenz des Menschen die Frage nach den Inhalten und Prozessen seines Bewußtseins.“ [Job87, S. 135]

Diese bereits 1987 aufgestellte These hat durch die rasante Entwicklung der Informationstechnik im letzten Jahrzehnt und die damit verbundene Herausforderung an die geistigen

Fähigkeiten des Menschen ihre Bestätigung gefunden. Die Entwicklung der Persönlichkeit wird in diesem Sinne wesentlich vom Bildungserwerb abhängen, der zugleich Erwerb geistiger Fähigkeiten ist. Ist dieser nun an praktische Manipulationen von Geräten gebunden, so sind auch Fertigkeiten zu erwerben. Die wirkliche Nutzung der jeweiligen Möglichkeiten der Computer verlangt jedoch die Einheit von Fähigkeiten und Fertigkeiten. Der Bildungsinhalt der Lehrveranstaltung „Grundlagen der Informatik“ soll zur Ausprägung dieser persönlichen Eigenschaften beitragen.

1.1.1 DV - alte Notwendigkeit und neue Möglichkeit

Die elektronische Datenverarbeitung (EDV) hat während der letzten Jahrzehnte bereits tiefgreifende Wirkungen in der Technik, Wirtschaft, Verwaltung und im Alltag hervorgerufen. Diese Entwicklung geht mit rasanter Geschwindigkeit weiter. Durch wirtschaftlichere Prozessoren, Speicher und komplette Verarbeitungssysteme dringt die EDV in ihrer Anwendung in immer neue Bereiche vor. Dies wird durch die Tendenz gefördert, daß eine Verschiebung der Arbeitsinhalte und -strukturen weg von manuell-technischen Fähigkeiten hin zur Be- und Verarbeitung von Informationen seit über 100 Jahren im Gange ist.

Die zunehmende Menge von Informationen und damit verbundener Rechenarbeit führte bereits im 17. Jahrhundert dazu, daß Wissenschaftler und Erfinder immer wieder versuchten, Informationen mit Hilfe von maschinellen Einrichtungen zu verarbeiten, zu transportieren und zu übermitteln. Die ersten Ergebnisse dieser Bemühungen (siehe [RL99]) sind:

1623 Schickard baut die erste Rechenmaschine (Addition, Subtraktion, Multiplikation, Division).

1641 Pascal konstruiert eine Zweispeziesmaschine („Pascaline“).

1673 Leibniz konstruiert die erste Vierspeziesmaschine.

Weitere Entwicklungsetappen stellen dar:

1805 Jacquard konstruiert den lochkartengesteuerten mechanischen Webstuhl, der die Lochungen mit Nadeln abtastet.

1840 Babbage entwirft das Konzept einer mechanischen programmgesteuerten Rechenmaschine mit den Einheiten Speicher, Steuereinheit und Verarbeitungseinheit. Die Anlage kann aufgrund der begrenzten Leistungsfähigkeit der ausschließlich mechanischen Bauteile nicht zur vollständigen Funktionstüchtigkeit entwickelt werden.

1886 Hollerith konstruiert die elektrische Lochkartenmaschine, die in den USA zur Auswertung der Volkszählung verwendet wird.

1941 Zuse entwirft und konstruiert die Anlage Z3, die erste programmgesteuerte Rechenmaschine mit ca. 2600 Relais und 15-20 arithmetische Operationen je Sekunde.

1944 Aiken entwickelt den Relaisrechner Mark I, mit dem eine Addition in 0,3 sec, eine Multiplikation in 6 sec und eine Division in 11 sec möglich wird.

1946 Eckert und Mauchly geben die Ideen zum Bau des Elektronenröhrenrechners ENIAC (electric integrater and numerical computer).

1949 Beginn der industriellen Rechnerproduktion nach der Konzeption des speicherprogrammierbaren Rechners von Johann von Neumann (1946).

Diese Ausführungen zeigen, daß die EDV keineswegs eine zufällige Erfindung war, sondern das Ergebnis langen Suchens sowohl nach technischen Lösungen als auch nach geeigneten funktionalen Prinzipien. Funktionsprinzipien in Verbindung mit den dafür geeigneten Technologien erschließen der EDV laufend neue Gebiete, die Wirkungen erzeugen, die weit über die ursprünglichen Bedürfnisse hinausgehen.

1.1.2 Die neue Stufe in Technik und Wissenschaft

Obwohl der Computer als Werkzeug zur DV das Ergebnis einer zielstrebigen Forschung und Entwicklung war und ist, überraschen seine weitreichenden Anwendungen selbst Fachleute immer wieder. Die Ursachen liegen im Zusammentreffen mehrerer grundsätzlich neuer Dinge, u.a. sind dies:

- das gespeichertes Programm,
- das kybernetisches Prinzip der informationellen Steuerung auf der Basis digital dargestellter Informationen,
- die Entwicklung einer Jahrhunderttechnologie in Form integrierter Schaltkreise auf Si-Basis.

Die Verbindung neuer funktionaler Prinzipien und kaum begrenzter Leistungstechnologien bewirken den bedeutungsvollen Schritt in das Informationszeitalter. Während die Si-Technik größte Beachtung erfährt, ist es ungleich schwieriger, die revolutionierende Bedeutung der geistig-logischen Prinzipien an geeigneter Stelle darzustellen. In dieser Reihe sind besonders zu erwähnen:

Leibniz entwickelte die Verallgemeinerung der Ziffernschreibweise im Dualprinzip sowie eine formale, universelle Sprache (1680-1690).

Boole entwickelte die Algebra der Logik (1815-1864).

Turing entwickelte die mathematische Logik weiter zum Konzept der Berechenbarkeit und ihrer Formulierung in einem Automatenmodell (1936).

Den entscheidenden Durchbruch gab es mit der bereits erwähnten Konzeption des *speicherprogrammierbaren Rechners* durch den ungarischen Mathematiker Johann von Neumann (1946). Das gespeicherte Programm schafft eine informationsverarbeitende Maschine, die die Möglichkeit hat, nicht nur Nutzinformationen, sondern auch ihre eigene Arbeitsfolge, die als Informationen ebenfalls gespeichert ist,

- zu *verarbeiten*,

- zu *verändern*,
- den Ablauf der Arbeit damit selbst zu *steuern*.

Dieser Prozeß war bisher in der Technik nicht bekannt. Der dadurch praktizierte Übergang zur informatorischen Steuerung hat viel grundsätzlichere Bedeutung, da verschiedenartige Steuerungsvorgänge abstrahiert und auf einheitliche Verarbeitungsmethoden zurückgeführt werden. Hierin liegt auch das Erfolgsgeheimnis der Mikroprozessoren. Die Entwicklung integrierter Schaltkreise in Si-Technik stellt eine wissenschaftlich-technische Kombinationsleistung dar, die zusammen mit der Kernenergie- und Raumfahrttechnik die Geschichte der Technik im 20. Jahrhundert prägte. Die technische Perfektion der Si-Technik führt zur Verbesserung des Preis/Leistungsverhältnisses. Waren anfänglich geringe Produktionsausbeuten aufgrund ungenügend beherrschter Herstellungsprozesse zu verzeichnen, so gelang es mittels zuverlässiger Technologien, die Schaltkreise auf immer kleineren Si-Chips unterzubringen und damit die Packungsdichte zu erhöhen. Höhere Ausbeute, schnellere und leistungsfähigere Schaltkreise führen hinsichtlich der Computerleistung zu einem Effekt in dritter Potenz. Die alte Ingenieurerfahrung, nach der Fortschritt an der einen Stelle seinen Preis an anderer Stelle hat, tritt in diesem Falle in mehrfacher Wirkung auf. Dafür gibt es in der Naturwissenschaft nur wenige Parallelen (z.B. Nutzbarmachung des Feuers, Erfindung der Buchdruckerkunst).

1.1.3 Äquivalenz von Steuerung und Informationen

Auf der Suche nach fundamentalen Grundsätzen in der Wissenschaft war es stets nützlich, Quelle und Entwicklung des heute gesicherten Wissens im historischen Rückblick zu analysieren. Wissenschaft und Technik sind oft dadurch einen großen Schritt vorangebracht worden, daß große Denker unabhängig voneinander bekannte Gesetzmäßigkeiten in anderen Erscheinungen gefunden haben. Dieser Sachverhalt wird durch den Begriff „Äquivalenz“ ausgedrückt, z.B.:

Isaac Newton 1683: Schwerkraft-Massenanziehung,

Robert Meyer 1842: Wärme und Energie,

Albert Einstein 1907/17: Masse und Energie,

Johann von Neumann 1946: Information und Steuerung.

Das Prinzip des als Information gespeicherten Programms präsentiert sich als weitreichendes Prinzip, dem sich fast täglich neue Anwendungen eröffnen. Alle seit der von Neumann'schen Entdeckung entwickelten Computeranlagen haben dieses Prinzip als Kernstück. Daher ist die Organisation der informatorischen Steuerung sowohl in zentralen als auch in verteilten Systemen zu einer Primäraufgabe der Softwareentwicklung und -architektur geworden. Dies erfordert, daß sich die Ingenieurgenerationen im verstärkten Maße der Ausarbeitung und Beherrschung logischer Konzeptionen von Systemen widmen müssen. Die Möglichkeit, ursprünglich technische Prozesse als informatorische Strukturen zu erkennen und zu betrachten, führt dazu, daß die Informationsverarbeitung mehr und mehr als eine

Periode	Zeitraum	Elektronische Basis	Charakteristik
1	1945	Röhren	$1ms$ bis Sekunden
2	1955	Transistoren, RTL	$1\mu s$
3	1965	Integrierte Schaltkreise, TTL	$1ns$, geringe Baugröße, Magnetkernspeicher
4	1975	Prozessoren, VLSI, MOS	Softwareentwicklung
5	1987	ULSI, Gesamtsystem	Künstliche Intelligenz
6	1990	CISC und RISC-Prozessoren	Mehrprozessor-, Verbundsysteme

Tabelle 1.1: Überblick über die Rechnergenerationen

Wechselwirkung zwischen Programmen und Informationskomplexen zu verstehen ist. Die von Neumann'sche Architektur markiert den „Königsweg“ der EDV zu ihrem zivilisationsgeschichtlichen Universalanspruch.

Vor diesem Hintergrund vollzieht sich gegenwärtig ein dramatischer Technologiewandel. Nach Rembold ist die Entwicklung der elektronischen Rechnermaschinen dadurch gekennzeichnet, daß sich jede der sechs Perioden durch typische Software- und Hardwareentwicklungsstufen auszeichnet [RL99, S. 25 ff.] (die Zeitangaben differieren in den einzelnen Quellen, die Merkmale sind jedoch zumeist eindeutig). In den Tabellen 1.1 und 1.2 sind die Perioden einschließlich ihrer Charakteristiken aufgeführt.
Allen Beteiligten ist klar, daß sich eine revolutionäre Neuorientierung in der Computerentwicklung vollziehen wird. Wir müssen heute jedoch noch feststellen, daß der größte und schnellste Computer der Welt nicht in der Lage ist, Denkarbeit zu leisten, z.B. im Sinne des Sprach- und Bildverständnisses. Doch man denkt jetzt bereits ernsthaft über das Ende der Neumann-Ära nach.

1.1.4 Herausforderung an Wissenschaft und Technik

Mit der Information als neues Element und mit der Informationsverarbeitung durch Automaten hat sich die Informatik als Wissenschaftsbereich herausgebildet, die interdisziplinär ihren Platz zwischen Mathematik, Logik, Ingenieurwissenschaften und Linguistik finden wird. Da wir auch wissen, daß sogar biologische Prozesse wesentlich informatorisch bestimmt sind, wird jeder Versuch einer Abgrenzung fehlschlagen. Die große Reichweite des neuen Gebietes führt dazu, daß sich bereits heute eine ganze Reihe wissenschaftlich-technischer Disziplinen darum ranken, u.a.:

- Digitale Schaltkreistechnik,
- Computerarchitektur,
- Programmierung, Software-Engineering,
- Simulation,

Periode (Zeitraum)	Hardware	Software	Einsatz
I (1953-58)	Vakuumröhren, Magnetbänder, Magnettrommeln als Externspeicher, Zugriffszeit 10^{-3} sec.	keine unterstützenden Betriebssysteme bzw. Compiler	vor allem im Rechnungswesen
II (1958-66)	Transistoren, verbesserte Kernspeicher, 10^{-6} sec.	Betriebssystem und Compiler (Cobol, Fortran) vorhanden	wissenschaftliche Rechnungen, Betriebsüberwachung
III (1966-74)	integrierte Schaltungen, Mikroprozessoren, $2 \cdot 10^{-9}$ sec., Halbleiterspeicher, Kleinrechner, Rechnerverbund	Softwarekrise, da mit Hardwareentwicklung nicht schritthaltend	Rechner als Konstruktionshilfe (CAD)
IV (1974-82)	hochintegrierte Schaltungen, Verkleinerung der Schalkreise (>130.000 Transistoren), 64 KBit-Speicher, Super- und Kleinrechner	integrierte Softwarelösungen	CAD/CAM-Anlagen, Netze
V (1982-90)	höchstintegrierte Schaltkreise, 16 MBit-Speicherchips, 64 MBit in Arbeit, Workstation	Expertensysteme, KI	Automatisierung von Industrieprozessen mit Prozeßrechnern
VI (seit 1990)	64 Bit-Prozessoren, 64 Mbit DRAM, PC-Dominanz	Entwicklung für Verbundsysteme	EDV durchdringt fast alle Bereiche

Tabelle 1.2: Merkmale der Rechnergenerationen

- Computergeometrie, -grafik, Bildverarbeitung, Animation, virtuelle Realität,
- Datenbanken,
- Kommunikationstechnologie, Netze, Internet,
- wissensbasierte Systeme, Lernsysteme,

- Neuronale Netze, Fuzzy-Systeme.

All diese Fachgebiete tragen noch unverkennbare Züge einer jungen Wissenschaft. Sie sind zum Teil noch geprägt durch viele empirische Fakten und Verfahren, die ständig in der Veränderung begriffen sind. Hier gilt jedoch der dialektische Grundsatz, daß aus der wachsenden Quantität gesetzmäßig eine neue Qualität hervorgehen wird. In dem Maße, wie es der Informatik gelingt, invariante Prinzipien herauszuarbeiten, wird sich aus diesem Wissenschaftszweig eine Basiswissenschaft und Ingenieurwissenschaft zugleich entwickeln. Hierbei geht es um die Ableitung von Grundfunktionen. Der heutige Erkenntnisstand basiert zumeist noch auf Erfahrungen, die im praktischen Umgang mit Informationsverarbeitungsprozessen gesammelt wurden. Trotzdem ist zu erwarten, daß künftige Generationen von Informatikern mit sehr viel weniger Lehrstoff ein sehr viel breiteres Anwendungsspektrum der Informationsverarbeitung abdecken können. Aus der Sicht des Technikers beinhaltet dieses Spektrum Tätigkeiten, wie:

- Prozesse simulieren und steuern,
- umfangreiche Rechenoperationen ausführen,
- Daten speichern, umarbeiten, verarbeiten,
- informationelle Arbeiten durchführen,
- Experimente unterstützen,
- Objekte beschreiben, speichern, manipulieren, darstellen, ausgeben.

1.2 Grundsätzliches zur Datenverarbeitung

Für Nutzer, die sich der Rechentechnik als Arbeitsmittel bedienen, ohne an ihrer Entwicklung und Konstruktion beteiligt zu sein, gilt es, folgende Grundhaltung zu beherzigen:

Der Nutzer arbeitet zwar am Computer,
kommuniziert aber immer mit einem Programm!

Dieser Zusammenhang ist deshalb von Bedeutung für die praktische Arbeit, weil sofort nach dem Einschalten des Computers automatisch eine Komponente des Betriebssystems aufgerufen wird, die die Kommunikation mit der Umwelt sichert. Dabei werden dem Computer Informationen in Form von Signalen mitgeteilt. Werden diese Signale so kodiert, daß sie von einem Computer akzeptiert werden können, und steht ein Programm(system) zur Verfügung, das diese verarbeiten kann, sind diese Signale oder Signalfolgen programmgerecht dargestellt und werden als *Daten* bezeichnet.

Die *Daten* sind demnach eine Teilmenge aller darstellbaren Informationen. Daten sind Zeichenketten (kodierte Signalfolgen), die Informationen repräsentieren und zur Speicherung und Verarbeitung in Computern bestimmt sind. Sie bestehen aus beliebigen Zeichen des rechnerexternen Alphabets. Mit ihnen dürfen Transport- und Vergleichsoperationen ausgeführt werden.

Die *Grundfunktionen der Datenverarbeitung* (DV) können nachfolgend zusammengefaßt werden: DV-Systeme (oft als Computer bezeichnet) sind Anlagen, die dazu dienen, Daten, die in das System eingegeben werden oder dort digital gespeichert sind,

- zu *verknüpfen*,
- zu *verarbeiten*, um die Ergebnisse wieder
- *abzuspeichern* oder dem Benutzer in geeigneter Form
- *zur Verfügung zu stellen*.

Vorgänge im *Datenverarbeitungssytem* (DVS) lassen sich in 5 Kategorien von Grundfunktionen einteilen:

Ein-/Ausgabe (input,output): Führt einem DVS alle Daten zu, die verarbeitet werden sollen und stellt die Ergebnisse einem Nutzer zur Verfügung.

Transport (transfer): Ermöglicht das Zusammenspiel der einzelnen Funktionseinheiten.

Speicherung (storage): Sichert die Verfügbarkeit der Daten zwischen den Ein- und Ausgabevorgängen.

Verknüpfung (processing): Beinhaltet die eigentliche Verarbeitung von Daten.

Steuerung (control): Bewirkt das folgerichtige Zusammenspiel aller Funktionen.

Aufeinanderfolgende Steuerschritte, die verschiedene zusammenhängende Funktionsanweisungen in einer maschinell verarbeitbaren Sprache (Maschinensprache) darstellen, nennt man *Programm*. Programmieren bedeutet, Algorithmen[1] zu kodieren auf:

- Maschinen-Niveau,
- maschinenorientiertem Niveau,
- problemorientiertem Niveau (=virtuelle Betrachtungsweise einer Maschine).

Werden diese Grundfunktionen durch technische Einrichtungen bewirkt, so spricht man von *Hardware*. Erfolgt dies durch den Ablauf von „informatorischen“ Funktionsanweisungen, so werden diese zur *Software* gerechnet.

Software: ist die Gesamtheit aller Verarbeitungsprogramme, d.h. aller festgelegten Funktionsabläufe der DV.

Programme: sind formgebundene, stofflich konkretisierte Geistesschöpfungen.

Soll eine Anwendung auf dem Rechner erledigt werden, so ergibt sich diese Problemlösungskette:

Problem - Algorithmus - Programm

[1] Algorithmen und Algorithmierung werden ausführlich in Abschnitt 2 behandelt

1.3 Allgemeine Computeranwendungen

Die meisten Menschen wissen, daß Computer eine bedeutende Rolle im Leben spielen. Die wenigsten sind sich jedoch bewußt, wie durchdringend diese Rolle ist. Computer spielen nahezu in alle Sphären des Alltags hinein: Von der Ladenkasse bis zur Ampelsteuerung, vom Führen von Bankkonten bis zur Warenverteilung, von der Wettervorhersage bis zur Produktion von Tageszeitungen, von der medizinischen Diagnostik bis in die Freizeit. Diese Liste ist nahezu endlos und jeder Versuch, sie erschöpfend darzustellen, wäre nicht mehr informativ. Auf zwei besondere Anwendungsbereiche soll an dieser Stelle eingegangen werden:

- *Datenverarbeitung*, weil diese mehr Computerzeit belegt als irgendein anderer Anwendungsbereich,
- *Wissensbasierte Systeme*, weil sie immer mehr an Bedeutung gewinnen.

Im eigentlichen Sinne sind alle Computeranwendungen als Datenverarbeitung zu bezeichnen, da die meisten Algorithmen Daten in der einen oder anderen Form verändern. Dieser Begriff der DV assoziiert zunächst die Vorstellung von der Bewältigung großer Datenmengen, von denen nur ein Bruchteil im Speicher gehalten werden kann. Hunderttausende Datensätze und -elemente abrufen, sortieren, transformieren, ausdrucken, einlesen, prüfen, selektieren, verdichten usw. sind die elementaren Operationen der DV. Vielfältige Methoden wurden für all diese Funktionen entwickelt und als Software implementiert. Diese Anwendungen werden uns ständig begleiten. In jüngster Zeit werden diese Informationen weltweit über Netzdienste angeboten, wobei der Teilnehmerkreis an dieser Kommunikation sich ständig erweitert, so daß dieses Verständigungsmittel auch bald zum Alltag gehören wird.

Die *künstliche Intelligenz* (KI), heute besser unter dem Oberbegriff *wissensbasierte Systeme* geführt, gehört zu jenen Computeranwendungen, die mehr als alle anderen Anwendungen die Vorbehalte Außenstehender erzeugt haben. Die Gründe dafür sind aus diesen drei Überschriften ableitbar:

> „Elektronengehirn schlägt Schachmeister“
>
> „Roboter läuft Amok“
>
> „Können Maschinen denken?“

Diese Diskussion wird häufig mit zu wenig Sachkenntnis geführt. Die Frage auf Grund welcher Mittel und Methoden die heutigen und morgigen Computersysteme intelligenter als die gestrigen sind, wurde bezeichnenderweise sehr selten gestellt. Deren Kenntnis allein eröffnet aber den Weg der künstlichen Intelligenz in die gesellschaftliche Praxis der Gegenwart und Zukunft. Nach [Job87, S. 50 ff.] sind mindestens 3 Entwicklungslinien des 19. und 20. Jahrhunderts von grundlegender Bedeutung für die Entwicklung der KI:

1. Die Mathematische Logik, die den Nachweis führte, daß das logische Schließen als ein Teilgebiet des menschlichen Denkens formalisierbar ist (Boole, Frege, Whitehead, Russel u.a.),

2. Zunehmendes Verständnis des Berechenbarkeits- und Algorithmusbegriffs als Bindeglied zu den Computern (Turing, Church u.a.),

3. Konstruktiv-technische Entwicklung der Computer (Babbage, Turing, Zuse, von Neumann).

Weitere Teilgebiete der KI sind:

- Problemlösen, Steuermethoden, Suche (Spielprogrammierung),
- Frage der Wissensdarstellung (Expertensysteme),
- Verarbeitung der natürlichen Sprache,
- Schaffung von Programmiersprachen und Software für KI (LISP, PROLOG),
- Theorie des menschlichen Problemlöseverhaltens (Überprüfung von Erfahrungen, Modellierung von Emotionen, Schlußfolgerungen über Meinungen und Motivationen, Natur von Erfahrungen, Schlußfolgerungen über Ereignisse und Handlungen, Entscheidungsprobleme, Gedächtnis- und Bewußtseinsmodelle),
- Computervision = Bildverarbeitung (Bildaufnahme und -darstellung, Klassifikation von Bildern, Verstehen von Bildern),
- maschinelles Lernen (Entwicklung von Lernprozessen zur Wissensgewinnung und Wissensverbesserung),
- Robotik.

Bei dieser Entwicklung wird die Frage nach der Rolle des Menschen zu stellen sein, da ja durch technische Mittel Funktionen des menschlichen Gehirns mehr oder weniger perfekt simuliert werden können. Solche Fragen berühren moralische Sachverhalte, wie die besondere Stellung des Menschen in der Welt, seine Subjektivität als Voraussetzung zielsetzender und verwirklichender Aktivität und menschlicher Selbstbestimmung. Eine Teilantwort kann aus der Sicht der Übertragbarkeit menschlicher Entscheidungen auf den Computer und der Verantwortbarkeit für deren Folgen gegeben werden. Eine Entscheidung fällen heißt, aus einem Feld real gegebener und erkannter Möglichkeiten jene auszuwählen, die unter den gegebenen Bedingungen der Zielfunktion am besten entspricht. Damit wird ausgesagt, daß Entscheidungen fällen als vermittelndes Glied zwischen Erkennen und Handeln ein Prozeß geistiger Tätigkeit ist. Die dazu notwendige Verarbeitung von Informationen kann zweifelsfrei von künstlich intelligenten Systemen ausgeführt werden.

Welche Entscheidungen kann der Mensch an die Technik delegieren? Welche Entscheidungen bleiben ihm vorbehalten? Zunächst kann festgehalten werden, daß es zwei Niveaustufen computergestützter Entscheidungen gibt:

1. Die moderne Informations-Technik fungiert als Basis und Stützung menschlicher Entscheidungsfindung.

2. Entscheidungen werden im Rahmen von Steuerfunktionen an intelligente, technische Systeme delegiert.

Der Entscheidungsspielraum muß wissenschaftlich erschlossen sein und ist durch das erkannte Möglichkeitsfeld akzeptabler Varianten abzugrenzen. Entscheidungen bleiben dem Menschen prinzipiell dort überlassen, wo sie nicht auf logische Operationen zu reduzieren sind, sondern auch interessengebundene Wertungen implizieren.

Spezielle Computeranwendungen finden wir in fast allen Lebensbereichen. Aus der Sicht des Zuhörerkreises sind dies zum Beispiel Softwareprodukte zur Unterstützung der Produktion materieller Produkte:

- Computer Aided Systems (CAx, CAD, CAP, CAM, ...),
- Produktionsplanungssysteme (PPS),
- Produktdatenverwaltungssysteme (PDM/EDM),
- Workflowsysteme,
- Data Warehouse Systeme,
- Computertomografie,
- Krankenhausmanagementsysteme,
- Umweltinformationssysteme.

Softwareprodukte für den Konsumbereich:

- Computerspiele,
- Filmproduktion,
- Medientechnik (Text, Video, Audio).

2 Algorithmierung

Ohne Algorithmus kein Programm.
Ohne Programm keine Ausführung auf dem Rechner!

Dieser Grundsatz bestimmt den Umgang mit dem Computer. Es versteht sich, daß alles, was die Entwicklung und die Umsetzung von Algorithmen betrifft, erhöhte Aufmerksamkeit verlangt.

2.1 Algorithmierung, Formalismen, Begriffe

2.1.1 Algorithmierung - eine allgemeine Einführung

Wir leben im Zeitalter der Computerrevolution. Wie jede Revolution ist sie umfassend, durchdringend und wird bleibende, fundamentale Auswirkungen für die gesellschaftliche Entwicklung haben. Sie wirkt sich insbesondere auf die Denk- und Lebensweise jedes Einzelnen aus.

Industrielle Revolution bedeutete im wesentlichen eine Steigerung der *körperlichen* Kräfte des Menschen, der Muskelkräfte:

- Druck auf den Knopf veranlaßt die Maschine, ein Muster in ein Metallblech zu stanzen.
- Zug an einem Hebel bewegt eine schwere Baggerschaufel durch eine Kohlenmasse.
- Maschinen übernehmen bestimmte, sich wiederholende körperliche Tätigkeiten.

Computerrevolution als tragende Säule der technischen Revolution bedeutet Steigerung der *geistigen* Kräfte des Menschen:

- Druck auf einen Knopf kann eine Maschine veranlassen, verzwickte Berechnungen durchzuführen, komplizierte Entscheidungen zu fällen oder Informationsmengen zu speichern, auffinden und verarbeiten.
- Bestimmte, sich wiederholende geistige Tätigkeiten werden von Maschinen übernommen.

Was ist ein *Computer*, daß er solche revolutionären Auswirkungen besitzt?

Prozeß	Algorithmus	Typische Schritte im Algorithmus
Pullover stricken	Strickmuster	stricke Rechtsmasche, stricke Linksmasche
Modellflugzeug bauen	Montageanleitung	leime Teil A an den Flügel B
Kuchen backen	Rezept	nimm 3 Eier, schaumig schlagen
Einkaufen	Einkaufszettel	Suche Butter
Kleider nähen	Schnittmuster	nähe seitlichen Saum
Sonate spielen	Notenblatt	spiele Note

Tabelle 2.1: Algorithmen für Alltagsprozesse

Ein Computer ist eine Maschine, *die geistige Routineaufgaben ausführt*, indem sie einfache Operationen mit hoher Geschwindigkeit vornimmt, wobei die *Einfachheit der Operationen* (z.B. Addition oder Vergleich zweier Zahlen) mit Geschwindigkeit ausgeglichen wird und eine *hohe Zahl von Operationen ausführbar* ist.

Definition des Algorithmusbegriffs

Einen Computer dahin zu bringen, daß er eine Aufgabe ausführt, bedeutet, ihm mitzuteilen, welche Operationen er ausführen soll – man muß beschreiben, wie die Aufgabe auszuführen ist. Solch eine Beschreibung nennt man *Algorithmus*. Er beschreibt demzufolge die Methode, mit der eine Aufgabe gelöst wird. Der Algorithmus besteht aus einer Folge von Schritten, deren korrekte Abarbeitung die gestellte Aufgabe löst. Diesen Vorgang bezeichnet man als *Prozeß* [GL90, S. 11].

Weitere Definitionen

- Ein *Algorithmus* liegt genau dann vor, wenn gegebene Größen (Eingabegrößen, Eingabeinformationen, Aufgaben, etc.) aufgrund eines Systems von Regeln (Umformungsregeln) eindeutig in andere Größen (Ausgabegrößen, Ausgabeinformationen, Lösungen, etc.) umgeformt oder umgearbeitet werden können.
- Ein *Algorithmus* dient stets zur Lösung einer Klasse von Aufgaben einheitlichen Typs.
- Ein *Algorithmus* ist ein eindeutig bestimmtes Verfahren unter Anwendung von Grundoperationen über primitiven (gegebenen) Objekten [SH87, S. 27].

Der Algorithmus ist keine Besonderheit der Informatik. Viele Alltagsvorgänge lassen sich durch Algorithmen beschreiben (siehe Tabelle 2.1) [GL90].

Beispiel 2.1
Algorithmus für das Suchen der größten Zahl aus einer Menge von positiven ganzen Zahlen; die Zahl -1 kennzeichnet das Ende der Menge.

1. Verbale Beschreibung

1: lies die erste Zahl
2: initialisiere z mit der gelesenen Zahl (die Zahl der Variablen z zuweisen)
3: lies die nächste Zahl x
4: wenn diese Zahl x größer z, dann setze z auf diese Zahl
5: wenn noch Zahlen vorhanden, dann gehe nach Schritt 3
6: gib z aus

2. Pascal-Programm

```
program maxnr;

var z, x : integer;

begin
   readln(z);              { Schritt 1 und 2 }
   repeat
      readln(x);           { Schritt 1 und 3 }
      if x>z then z:=x;    { Schritt 4, Vergleich }
   until x=-1;             { Schritt 5, Abbruch }
   writeln(z);             { Schritt 6 }
   readln;                 { Ausgabebildschirm bleibt,}
                           { bis eine Taste gedrückt wird }
end.
```

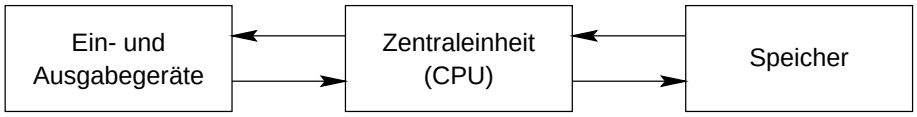


Abbildung 2.1: Komponenten eines typischen Computers

Personen oder Einheiten, die solche Prozesse ausführen, nennt man Prozessoren. Der Computer ist ein spezieller Prozessor, der im wesentlichen aus 3 Hauptkomponenten besteht, die die Hardware bilden (vgl. Abbildung 2.1):

1. *Zentraleinheit* (CPU, central processing unit), die die Basisoperationen ausführt,
2. *Speicher* (memory), der
   a) die auszuführenden Operationen als Algorithmus und
   b) die Daten, auf denen die Operationen wirken, enthält,
3. *Ein- und Ausgabegeräte* (input and output devices), über die der Algorithmus und die Daten in den Hauptspeicher gebracht werden und über die der Computer die Ergebnisse seiner Tätigkeit mitteilt.

Merkmale, die einen Computer kennzeichnen, sind:

1. Geschwindigkeit (Operationen/Zeiteinheit),
2. Zuverlässigkeit (Fehlerhäufigkeit),
3. Speicherfähigkeit (Menge der Informationseinheiten),
4. Kostenaufwand (Preis/Leistungsverhältnis).

An einen Algorithmus wird der Anspruch gestellt, daß er so ausgedrückt wird, daß der Prozessor ihn versteht und ausführen kann. Der Prozessor muß den Algorithmus interpretieren können, indem er

1. versteht, was jeder Schritt bedeutet und
2. die jeweilige Operation ausführen kann.

Zum Beispiel muß der Pianist Noten lesen und spielen können, der Koch muß ein Rezept umsetzen und der Strickende muß Nadeln und Wolle handhaben können. Ist der Prozessor ein Computer, muß der Algorithmus in Form eines Programmes ausgeführt werden. Dazu bedarf es einer geeigneten Programmiersprache. Um den Algorithmus als Programm zu formulieren, muß man programmieren. Jeder Schritt eines Algorithmus wird durch eine Anweisung (instruction) beschrieben. Ein Algorithmus besteht somit aus einer Folge von Anweisungen, von denen jede Operationen angibt, die der Computer ausführen soll. Abbildung 2.2 vermittelt die Stufen der Algorithmusausführung mittels Computer.

Worin besteht nun die Bedeutung von Algorithmen? Wie dargelegt, erfordert die Durchführung eines Prozesses auf einem Computer, daß

1. ein Algorithmus entworfen wird,
2. der Algorithmus in einer geeigneten Programmiersprache ausgedrückt wird,
3. der Computer das Programm ausführt.

Die Rolle der Algorithmen ist grundlegend:

Ohne Algorithmus kein Programm,
ohne Programm keine Ausführung.

Algorithmen sind weiterhin sowohl unabhängig von der Programmiersprache als auch vom Computertyp. Als Analogie zum Alltag: Ein Rezept für einen Obstkuchen kann in Deutsch oder Englisch ausgedrückt werden, der Algorithmus ist derselbe. Falls das Rezept gewissenhaft befolgt wird, entsteht der gleiche Kuchen, unabhängig vom Code. Technisch ausgedrückt heißt das, alle Computer (wie alle Köche) können die gleichen Grundoperationen ausführen, obwohl sich diese in Details unterscheiden. Daraus resultiert der Schluß, daß Algorithmen unabhängig von der „Tagestechnologie“ erzeugt und studiert werden können, die Ergebnisse bleiben trotz neuer Computermodelle und Programmiersprachen gültig.

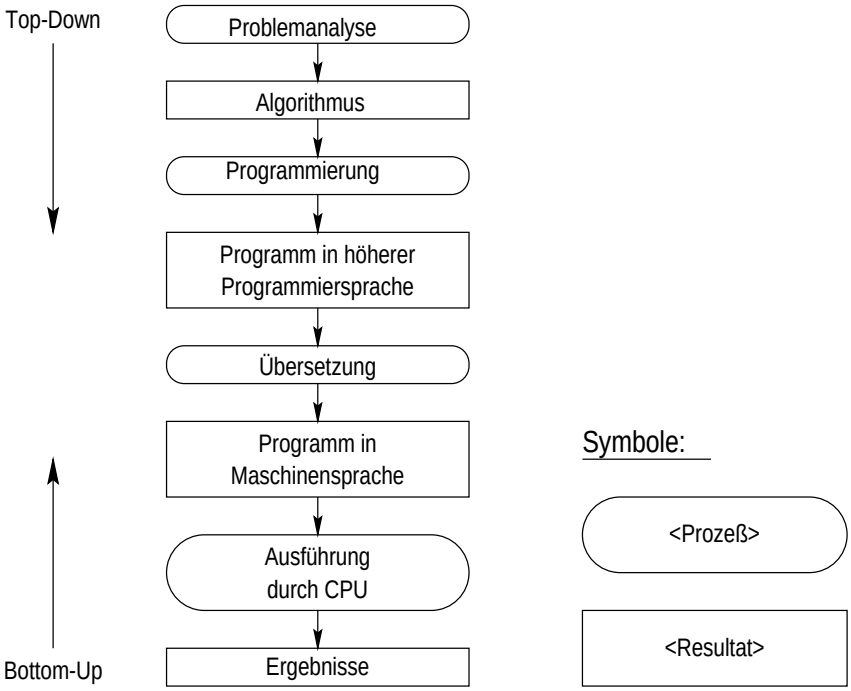


Abbildung 2.2: Stufen der Algorithmenausführung

Programmiersprachen und Computer sind Mittel, um Algorithmen in Form von Prozessen auszuführen. Computertechnologie und Programmiersprachen bestimmen jedoch entscheidend die schnellere, billigere und zuverlässigere Ausführung von Algorithmen (Beispiel: Möglichkeit der computergestützten Wettervorhersage). Vier allgemeine Merkmale von Algorithmen sind:

1. Ein Algorithmus muß von einer Maschine durchgeführt werden können. Die für den Ablauf des Algorithmus benötigte Information muß zu Beginn vorhanden sein.

2. Ein Algorithmus muß allgemeingültig sein. Die Größe der Datenmenge, auf die der Algorithmus angewandt wird, darf nicht eingeschränkt sein.

3. Der Algorithmus besteht aus einer Reihe von Einzelschritten und Anweisungen über die Reihenfolge. Jeder Schritt muß in seiner Wirkung genau definiert sein.

4. Ein Algorithmus muß nach einer endlichen Zeit (und nach einer endlichen Zahl von Schritten) enden. Für das Ende des Algorithmus muß eine Abbruchbedingung formuliert sein.

Nach der Erläuterung der fundamentalen Bedeutung der Algorithmen für die Informatik muß nun die Frage nach der Zielrichtung des Studiums derselben gestellt werden. Welche Gesichtspunkte und welche Eigenschaften sind zu beachten?

1. Aspekt: Wie entwirft man Algorithmen?

Es gibt weder eine allgemeingültige Regel, noch einen Algorithmus zum Entwurf von Algorithmen. Alle diesbezüglichen Bemühungen werden unter der Überschrift Berechenbarkeit eingeordnet. Leichtgläubig könnte man annehmen, daß Computer alle Probleme lösen können. Überraschenderweise ist das Gegenteil der Fall. Computer können die meisten Dinge nicht. Dies resultiert daraus, daß es sogenannte *berechenbare* und *nichtberechenbare* Probleme gibt. Ein klassisches Beispiel für ein nichtberechenbares Problem ist es nachzuweisen, ob ein Programm P, daß mit den Daten D bedient wird, in endlicher Zeit endet oder in eine sogenannte Endlosschleife übergeht. Dieses Problem ist unter dem Begriff *Halteproblem* bekannt. Ohne hier den Beweis anführen zu wollen, müssen wir feststellen, daß es nicht gelingt, dieses Problem zu lösen (siehe [GL90, S. 83 ff.]).

2. Aspekt: Wie bewertet man Algorithmen, wie trifft man eine alternative Auswahl?

Unter dieser Fragestellung sind eingeordnet:

- die Bestimmung der erforderlichen Ressourcen,
- die Optimierung der Algorithmen,
- die Entscheidung über die Durchführbarkeit (maximal polynomialer Zeitbedarf).

Diese Themen sind unter dem Begriff der Komplexität von Algorithmen zusammengefaßt.

3. Aspekt: Wie steht es mit der Korrektheit der Algorithmen?

Hierunter sind z.B.

- die Fehlersuche und -korrektur (Debugging),
- das Testen und Beweisen

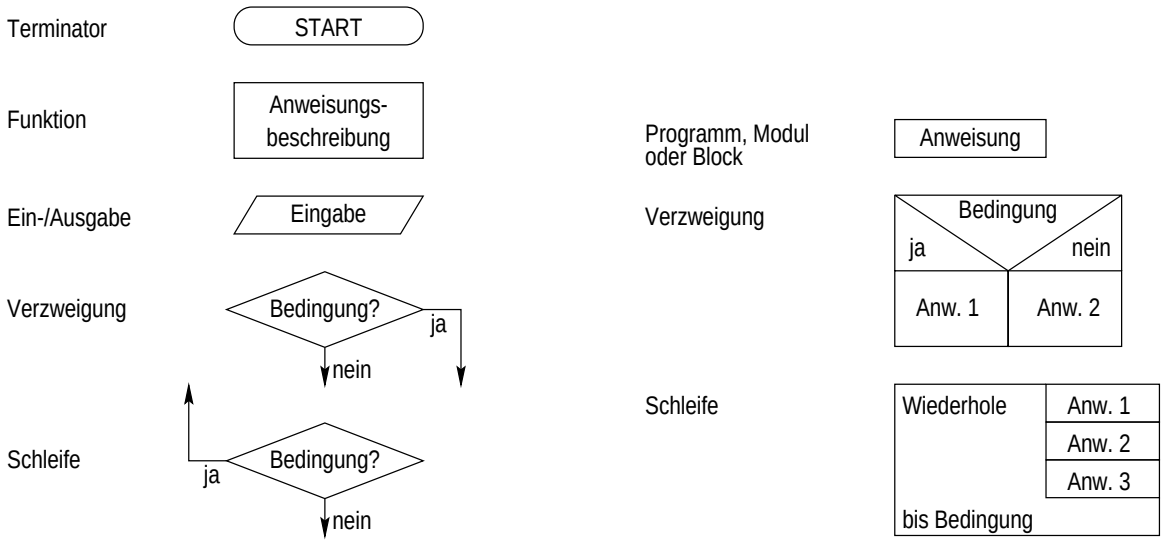
zu verstehen.

2.2 Formalismen zur Beschreibung von Algorithmen

Im Abschnitt 2.1.1 wurde die zentrale Bedeutung des Algorithmenbegriffs für die Informatik behandelt. Bevor wir uns dem Entwurf von Algorithmen zuwenden, werden noch einige Mittel zur Darstellung von Algorithmen vorgestellt.

Die wichtigsten sind:

- allgemeine verbale Beschreibung,
- Programmablaufplan,
- Programmlinienmethode,
- Struktogrammtechnik,



(a) Programmablaufplan

(b) Struktogramm

Abbildung 2.3: Notation in Programmablaufplänen und Struktogrammen

- strukturierte verbale Anweisungsfolge (SVA, Pseudocode).

Jede dieser Methoden wird ihre Nutzer finden. Im Rahmen dieser Lehrveranstaltung entscheiden wir uns für eine Kombination der Struktogrammtechnik, Programmablaufpläne (PAP) und der allgemeinen verbalen Beschreibung. In den Abbildungen 2.3(a) und 2.3(b) dargestellt.

Beispiel 2.2
Algorithmus zum Einschlagen eines Nagels.

1. verbale Beschreibung

Require: Hammer, Nägel, Flaschen mit Bier
Ensure: Entscheidung, ob Nagel eingeschlagen wurde oder nicht.

1: ERFOLG:=0
2: Hammer ergreifen
3: Nagel ergreifen
4: Schlag
5: falls (Daumen ist blau), gehe nach 11
6: falls (Nagel ist fest) setze ERFOLG:=1 und gehe nach 9
7: falls (Nagel ist gerade) gehe nach 4
8: falls (Nägel sind noch vorhanden) gehe nach 3
9: Trinken einer Flasche Bier
10: falls (Flaschen sind noch vorhanden) gehe nach 9
11: Fluch

12: Falls ERFOLG=1, ist der Nagel wie gewünscht eingeschlagen, falls ERFOLG=0, wurde das Ziel nicht erreicht.

2. Programmablaufplan (PAP)

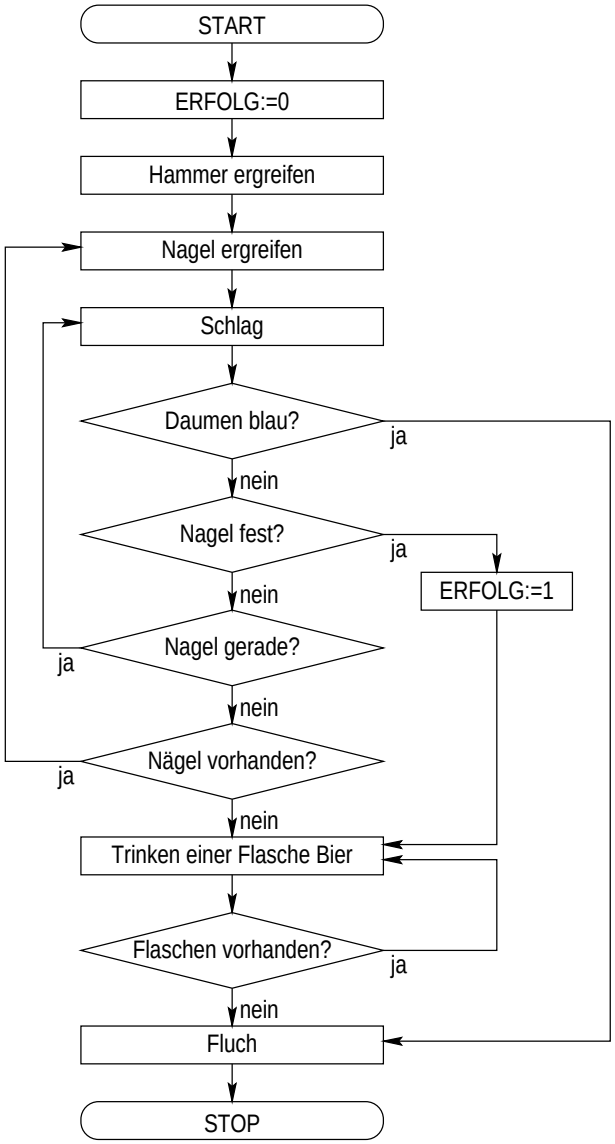


Abbildung 2.4: Programmablaufplan für Algorithmus *„Einschlagen eines Nagels“*

Beispiel 2.3
Algorithmus zur Bestimmung der Teiler einer positiven ganzen Zahl mit der Ausgabe von sechs Teilerwerten je Zeile.

1. verbale Beschreibung

Require: positive ganze Zahl
Ensure: Teiler dieser Zahl
1: Lies eine Zahl z ein
2: falls $z < 1$ gehe nach 1
3: Beginne mit Teiler $t = 1$
4: Berechne r als Rest der Division $z \div t$
5: falls $r \neq 0$ gehe nach 7
6: Ausgabe: r (r ist ein Teiler von z)
7: Erhöhe t um 1 (Test der nächsten Zahl)
8: falls $t < \frac{z}{2}$ gehe nach 4 (Alle Zahlen $> \frac{z}{2}$ können keine Teiler sein)

2. Struktogramm

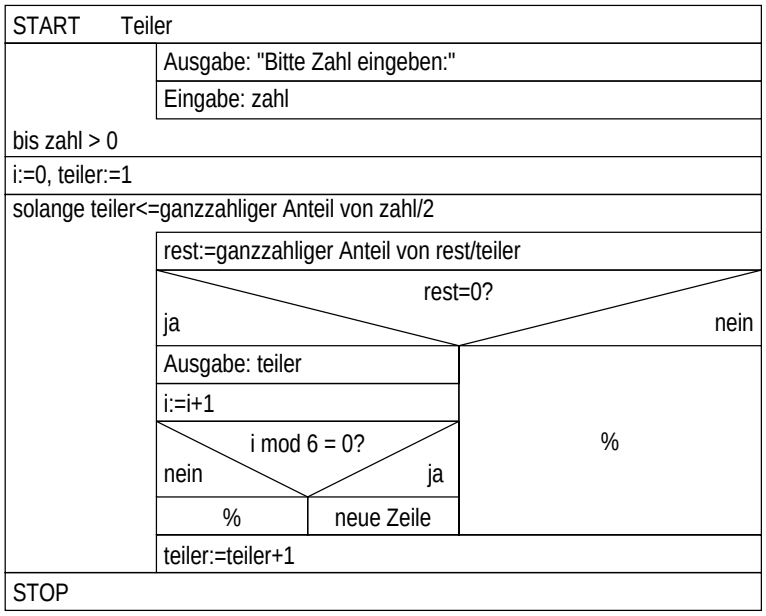


Abbildung 2.5: Struktogramm für Algorithmus *„Teiler einer positiven ganzen Zahl“*

Da die Programmlinienmethode als vereinfachte Darstellungsform von Programmabläufen zu verstehen ist und demzufolge möglichst nicht zur Erarbeitung von Systemunterlagen benutzt werden sollte, wollen wir es bei diesem Verweis belassen.

2.2.1 Algorithmen, Programme, Programmiersprachen

Ein Algorithmus entspricht einer Beschreibung, wie eine Aufgabe oder ein Prozeß auszuführen ist. *Prozesse* interagieren mit ihrer Umgebung. Sie nehmen *Eingaben* (input) entgegen und erzeugen *Ausgaben* (output). Die Ausgabe ist meist die Zielrichtung des Prozesses. In Tabelle 2.2 sind Beispiele für Ein- und Ausgaben von Prozessen dargestellt. Ein computergesteuerter Prozeß erfordert Eingaben in Form von Daten und erzeugt Ausgaben als Daten.

Prozeß	Eingabe	Ausgabe
Pullover stricken	Wolle	Pullover
Flugmodell bauen	Balsaholz	Flugmodell
Kuchen backen	Mehl, Eier, Zucker	Kuchen
Kleider nähen	Stoff	Kleid
Wöchentliche Lohnermittlung	Stundensätze, geleistete Arbeitsstunden	Auszahlungsbeträge

Tabelle 2.2: Beispiele für Ein- und Ausgaben von Prozessen

Der Prozeß *Wöchentliche Lohnermittlung* in Tabelle 2.2 ist ein entsprechendes Beispiel. Zu beachten ist, daß Ein- und Ausgabe als Algorithmusbestandteile unabhängig vom Prozessor sind.

Alle bisherigen Prozeßbeispiele sind endlich, d.h. sie sind terminiert. Es gibt aber auch unendliche bzw. endlose Prozesse, z.B.:

1. Bibliothekskatalog fortschreiben,
2. glücklich bleiben,
3. Ampelanlage steuern,
4. Patienten auf Intensivstation betreuen.

Endlichkeit (bzw. *Unendlichkeit*) ist Hauptmerkmal eines Prozesses. Ein Hauptfehler ist, daß ein Prozeß, der als endlich vorausgesetzt wurde, häufig nicht endet. Beispiel: Märchen Aschenputtel: Prinz verfolgte Algorithmus des passenden Schuhs. Wenn Aschenputtel gestorben, sucht er heute noch.

Bisherige Prozesse wurden durch Algorithmen beschrieben, die sich als Ausdrucksform der menschlichen Umgangssprache bedienen. Es gibt zwei Gründe, die diese Ausdrucksform für die Computerarbeit nicht erlauben:

1. Die Umgangssprache umfaßt ein enormes Vokabular und komplizierte grammatikalische Regeln, für die noch keine Algorithmen entwickelt wurden.
2. Das Verständnis des Satzes hängt nicht nur von der Grammatik ab, sondern auch wesentlich vom Umfeld (Kontext), z.B.:
   - „Sich regen bringt Segen.“
   - „Das war ein Wink mit dem Zaunpfahl.“

Als Konsequenz müssen Computeralgorithmen in einfacher Form als Programm ausgedrückt werden. Dazu bedarf es Programmiersprachen. In Tabelle 2.3 sind Beispiele höherer Programmiersprachen aufgelistet. Die Ursache für die Entwicklung verschiedener Programmiersprachen ist in 3 Punkten zu erklären :

Name (Einführungsjahr)	Langbezeichnung	Neue Leistungen
FORTRAN (1953)	**For**mula **Tran**slation	Struktur für Arithmetik
COBOL (1956)	**Co**mmon **B**usiness **O**riented **L**anguage	kommerzielle und ökonomische Berechnungen
ALGOL 60 (1960)	**Algo**rithmic **L**anguage	math.-nat. Berechnungen
PL/1 (1965)	Programming Language	Struktur für Prozeduren, wiss.-techn. und ökon. Berechnungen
BASIC (1969)	**B**eginners **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode	Heranführung an die Programmierung eines Computers
Pascal (1970	nach Blaise Pascal benannt	math., naturwiss. und ökon. Berechnungen, Lehrsprache
C (1972)		Programmierung aller Aufgaben einschl. Betriebssystemprogramme
Prolog (1972)	**Pro**gramming in **Log**ic	log. Programmiersprache geeignet für wiss.-bas. Systeme
Smalltalk (1980)		Pioniersprache des objektorientierten Denkmodells
C++ (1983)		Erweiterung von C um objektorientierte Konzepte
Java (1996)	aus OAK von Sun Microsystems entwickelt	plattformunabhängige OO-Programmiersprache

Tabelle 2.3: Beispiele höherer Programmiersprachen

1. Computerprogrammierung ist eine vergleichsweise junge Tätigkeit, die ständig in Fluß ist und von neuen Ideen lebt.

2. Computer werden zu vielfältigen Zwecken genutzt. Daraus folgt, daß die angewandten Algorithmen verschiedener Art sind und nicht in jeder Programmiersprache gleichermaßen gut darstellbar sind (z.B. Noten zum Spielen eines Musikinstruments und die Schreibweise von Strickmustern unterscheiden sich grundsätzlich). Spezialsprachen sind das Resultat dieser Bemühungen (COBOL, CORAL, LISP).

3. Es gibt eine unumgängliche Tendenz, das Rad neu zu erfinden. Diesem Syndrom liegt der Glaube zugrunde, daß das eigene Rad besser ist als das des anderen.

Jede Programmiersprache hat ihr eigenes Vokabular und eigene grammatikalische Regeln, die in der Regel aus mathematischen Symbolen und englischen Wörtern (**if**, **then**, **else**, **while**, **repeat**, **until**, **do**) bestehen. Die Unterschiede im Vokabular und in der Grammatik

führen in den verschiedenen Programmiersprachen zu unterschiedlichen Formen erlaubter Ausdrücke.

Beispiel 2.4
Darstellung eines Rechenalgorithmus

in COBOL: **MULTIPLY** Preis **BY** Menge **GIVING** Kosten

in PASCAL: Kosten:=Preis∗Menge;

in C: Kosten=Preis∗Menge;

Der Entwicklung von Programmiersprachen liegen mehrere Ziele zugrunde:

1. Die Sprache muß eine einfache und knappe Darstellung des Algorithmus in dem Anwendungsbereich gestatten, für den sie entworfen ist.
2. Die Sprache muß für einen Computer leicht verständlich sein.
3. In dieser Sprache geschriebene Programme müssen leicht verständlich für Menschen sein, so daß sie bei Bedarf leicht geändert werden können.
4. Die Sprache sollte die Fehlermöglichkeiten bei der Umsetzung eines Algorithmus in ein Programm minimieren.
5. Die Durchsicht des Programmtextes sollte zeigen, daß die Programmausführung tatsächlich den auszuführenden Prozeß wiedergibt.

Diese Ziele sind zum Teil nicht vollständig miteinander vereinbar, z.B. effiziente, rechnerorientierte Darstellung und leichte Verständlichkeit für den Menschen.

2.2.2 Syntax und Semantik von Programmiersprachen

Es wurde bereits darauf hingewiesen, daß ein Prozessor in der Lage sein muß, einen Algorithmus zu interpretieren, um den von ihm beschriebenen Ablauf auszuführen. Dazu muß der Prozessor in der Lage sein,

1. die Darstellung, in der der Algorithmus ausgedrückt ist, zu verstehen (z.B. ein Strickmuster oder ein Notenblatt) und
2. die entsprechenden Operationen auszuführen.

Betrachten wir den ersten Schritt näher, so stellen wir fest, daß dieser in 2 Teilschritte zerfällt:

1. Der Prozessor muß in der Lage sein, die Symbole, in denen der Algorithmus dargestellt ist, zu erkennen und ihnen eine Bedeutung zuzuordnen (Worte, Abkürzungen, mathematische Symbole, Noten eines Musikstückes, usw.). Dafür muß der Prozessor Kenntnisse über das Vokabular und die Grammatik der Sprache besitzen, in der der Algorithmus ausgedrückt ist, z.B.:

- „Ärmel“ ist ein deutsches Wort.
- „=“ ist eine Beziehung zwischen Zahlen.

Folgende Kombinationen sind Verletzungen der jeweiligen Sprachregeln und müssen vom Prozessor erkannt werden:

- Ärmel die Säume
- $a + c = b$

Die Menge der grammatikalischen Regeln, die bestimmen, wie die Symbole in der Sprache korrekt zu benutzen sind, heißt *Syntax* der Sprache. Ein Programm, das die Syntax der Sprache, in der es ausgeführt ist, befolgt, heißt syntaktisch korrekt. Eine Abweichung von der Sprachsyntax heißt *Syntaxfehler*. Syntaktische Korrektheit ist normalerweise eine notwendige Voraussetzung für die Interpretation eines Computerprogrammes.

2. Der zweite Teilschritt, einen Algorithmus zu verstehen, verlangt, jedem Schritt des Algorithmus eine Bedeutung in Form von Operationen zuzuordnen, die der Prozessor ausführen kann, z.B.:

   - „1 rechts“ beim Stricken bedeutet, Nadel und Wolle in bestimmter Weise zu behandeln.
   - die Bedeutung von Kosten:=Preis∗Menge; ist darin zu sehen, daß zwei Zahlen, mit Preis und Menge bezeichnet, zu multiplizieren sind und somit als Ergebnis eine dritte Zahl mit der Bezeichnung Kosten ergeben.

   Die Bedeutung besonderer Ausdrucksformen einer Sprache heißt *Semantik* der Sprache. In der Umgangssprache sind Syntax und Semantik sehr komplex und oft aufeinander bezogen. Hier ist es möglich, syntaktisch richtige Sätze zu bilden, die jedoch inhaltlich sinnlos sind, z.B.:

   - Farblose grüne Ideen schlafen wild.
   - Der Elefant aß die Erdnuß. (sinnvoll) jedoch
   - Die Erdnuß aß den Elefanten. (sinnlos)

   Ähnlich können Schritte im Algorithmus syntaktisch korrekt jedoch semantisch falsch sein, z.B.:

   - Schreibe den Namen des 1. Monats im Jahr. (richtig)
   - Schreibe den Namen des 13. Monats im Jahr. (falsch)

Programmiersprachen sind hinsichtlich Syntax und Semantik relativ einfach gestaltet, so daß ein Programm ohne Bezug auf die Semantik syntaktisch analysiert werden kann. Die Aufdeckung semantischer Unstimmigkeiten beruht auf Kenntnissen über die angegebenen Objekte. Insbesondere baut sie auf Kenntnissen über Eigenschaften (Attribute) dieser Objekte und ihren Zusammenhängen auf. Das heißt, daß zum Beispiel obige Absurditäten

(Erdnuß ⟶ Elefant, 13. Monat) erkannt werden. Für den Prozessor folgt daraus, daß er genügend über die Objekte, auf die der Algorithmus Bezug nimmt, wissen muß. Nicht ausreichende Kenntnisse führen zu Inkonsistenzen, die eventuell erst bei der Ausführung des Programms zum Vorschein kommen. Schwerwiegendere Fehler in der Semantik treten auf, wenn Unstimmigkeiten aus früheren Algorithmusschritten herrühren, z.B.:

1. Denke Dir eine Zahl von 1 bis 13 aus.

2. Bezeichne diese Zahl mit n.

3. Schreibe den n-ten Namen des Monats im Jahr.

Die Unstimmigkeit der möglichen Zahl 13 tritt erst bei der Ausführung des Algorithmus auf. Es gibt kaum eine Chance, diesen Fehler vorher zu entdecken.

2.2.3 Zusammenfassung

1. Algorithmen sind fundamental für die Informatik.

2. Sie zeichnen sich durch grundlegende Merkmale aus.

3. Der Entwurf verlangt Kreativität und Einsicht.

Zur Interpretation eines jeden Schrittes eines Algorithmus muß ein Prozessor eines Rechners in der Lage sein:

1. die Symbole, in denen der Algorithmusschritt ausgedrückt ist, zu verstehen,

2. dem Algorithmusschritt in Form von Operationen eine Bedeutung zuzuordnen,

3. die auftretenden Operationen auszuführen.

Syntaxfehler werden in Stufe 1, bestimmte Semantikfehler in Stufe 2, andere Semantikfehler erst in Stufe 3 festgestellt. Stufe 1 und 2 werden von einem Übersetzer vollzogen. Logische Fehler werden vom Prozessor nicht erkannt.

2.3 Beschreibung des Sprachumfangs ANSI-C

Die Programmiersprache C [KR90] wurde Anfang der 70er Jahre von Dennis Ritchie an den Bell Laboratories in den USA entwickelt. 1989 wurde diese Sprache vom amerikanischen Normungsausschuß (ANSI) zum Standard erhoben. Im späteren Teil der Vorlesung werden wir mit C++ eine um objektorientierte Ansätze Erweiterung von C kennenlernen.

2.3.1 Grundelemente der Sprache

Zeichensatz

Das „Grundvokabular“ von C basiert auf mehreren Klassen von Grundsymbolen.

```
<Buchstabe>     ::= A | B | ... | Z | a | b | ... | z |
<Ziffer>        ::= 0 | 1 | ... | 9 |
<Sonderzeichen> ::= ( | ) | [ | ] | { | } | < | > | + | - |
                    * | / | % | ^ | ~ | & | | | _ | = | ! |
                    ? | # | \ | , | . | ; | : | ’ | "
```

Weiterhin gehören zum Zeichensatz Leerzeichen, Zeilenendezeichen, horizontaler und vertikaler Tabulator und Seitenvorschub.

Operatoren

1. primäre Operatoren

   ( ) Klammer
   [ ] Feld
   . Struktur
   −> Zeiger auf Strukturen

2. Inkrement und Dekrement

   ++ Erhöhung um 1 x++ oder ++x
   −− Verringerung um 1 x−− oder −−x

   Hinweis: x++ und x−− sind verkürzte Schreibweisen für x=x+1 und x=x−1. Zwischen x++ und ++x besteht ein Unterschied, wenn dieser Ausdruck in einem komplexeren Ausdruck verwendet wird. In diesem Fall wird durch die Position von ++ bzw. −− bestimmt, wann das inkrementieren bzw. dekrementieren erfolgen soll. Wir werden diesen Unterschied zu einem späteren Zeitpunkt noch genauer betrachten.

3. Arithmetische Operatoren

   + Addition
   − Subtraktion
   ∗ Multiplikation
   / Division
   % Modulo (Divisionsrest)

4. Zuweisungsoperatoren

=	einfache Zuweisung	x=y
+=	Zuweisung mit Addition	x+=y
−=	Zuweisung mit Subtraktion	x−=y
∗=	Zuweisung mit Multiplikation	x∗=y
/=	Zuweisung mit Division	x/=y
%=	Zuweisung mit Modulo	x%=y

Hinweis: x+=y ist eine verkürzte Schreibweise von x=x+y. Das gilt auch für die übrigen Zuweisungen mit Operation.

5. Vergleichsoperatoren

   == Gleichheit
   != Ungleichheit (verschieden von, nicht gleich)
   <= kleiner gleich
   >= größer gleich
   < kleiner als
   > größer als

   Hinweis: Ein häufiger Fehler besteht darin, daß anstelle von == bei Vergleichen nur = geschrieben wird. Dies ist syntaktisch richtig und wird auch ausgeführt, liefert aber in der Regel ein falsches Ergebnis. Wir werden auf diese Fehlerquelle und deren Auswirkungen im Zusammenhang mit der **if**-Anweisung noch genauer eingehen.

6. logische Operatoren

   && UND-Verknüpfung
   || ODER-Verknüpfung
   ! logisches NICHT

7. Bitoperatoren

   & bitweise UND-Verknüpfung
   | bitweise ODER-Verknüpfung
   ^ bitweise XOR-Verknüpfung
   ~ bitweiser NOT-Operator
   >> Bitverschiebung nach rechts
   << Bitverschiebung nach links

8. weitere Operatoren

   (typ)A Typumwandlungsoperator
   * Dereferenzierungsoperator bei Verwendung von Zeigern
   & Adreßoperator
   sizeof Speicherbedarfsoperator

   Hinweis: Mit dem Typumwandlungsoperator kann der Typ eines Ausdrucks, z.B. einer Variablen, innerhalb eines Ausdrucks geändert werden. Im Zusammenhang mit den Datentypen werden wir noch genauer betrachten, warum und wann dies notwendig ist.

Reservierte Worte (Schlüsselwörter)

Folgende Worte sind in C fest definiert und dürfen nur in dieser entsprechenden Semantik verwendet werden.

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

Bezeichner und Namen

Bezeichner dienen der eindeutigen Identifizierung von Objekten innerhalb eines Programms. Für Bezeichner in C-Programmen gilt:

- Ein Bezeichner besteht aus einer Folge von Buchstaben, Ziffern oder dem Zeichen _, wobei das erste Zeichen keine Ziffer sein darf.
- Ein Bezeichner darf beliebig lang sein, wobei jedoch nur eine bestimmte Länge signifikant ist. Diese Länge variiert von Compiler zu Compiler.
- Schlüsselwörter dürfen nicht als Bezeichner genutzt werden.

In C-Programmen dienen Namen als Bezeichner von Variablen, Konstanten, Funktionen, Marken, usw. Namen können nur in einer Datei benutzt werden (interne Namen), wobei dieselben bis zu 31 signifikante Zeichen aufweisen können, z.B.:

`I, a5, _Simulation`

Werden mit Namen auch Objekte in mehreren Dateien angesprochen (externe Namen), so ist die signifikante Länge mindestens 6 Zeichen.

Formate in C-Programmen

C-Programme können formatfrei geschrieben werden, d.h., daß sie keine bestimmte Zeilenstruktur haben müssen. Es empfiehlt sich jedoch im Sinne einer besseren Übersicht, eine Strukturierung vorzunehmen.

Standardbezeichner

Neben den Schlüsselwörtern enthält die Sprache Bibliotheksroutinen, die Standardbezeichner erhalten. Dazu gehören neben mathematischen Routinen, Routinen zur Arbeit im Text- oder Grafikmodus, usw. Diese Bezeichner sollten auch nur in diesem semantischen Zusammenhang verwendet werden.

Kommentare

Kommentare dienen der besseren Lesbarkeit von Programmen und sollten demzufolge häufig verwandt werden. Zwischen den Zeichen /∗ und ∗/ können beliebig viele Zeichen stehen. Der Compiler betrachtet diese Einfügung als Trennzeichen, z.B.:

/∗ Dies ist ein Kommentar ∗/

2.3.2 Nutzerdefinierte Sprachelemente

Neben den vordefinierten Sprachelementen, die eine Programmiersprache anbietet, gibt es die Möglichkeit, eigene Sprachelemente zu definieren, die aber bestimmten Syntaxregeln genügen müssen. Für die Darstellung der Syntaxregeln einer Sprache gibt es verschiedene Beschreibungsmittel. Im folgenden werden zwei Möglichkeiten vorgestellt.

Backus-Naur-Form

Die Backus-Naur-Form (BNF) wurde in den 60er Jahren im Zusammenhang mit der Programmiersprache Algol eingeführt. Später wurde sie um zusätzliche Beschreibungsmittel zur Erweiterten Backus-Naur-Form (EBNF) erweitert. Diese Erweiterungen bieten keine neuen Konzepte für die Darstellung von Syntaxregeln, verringern aber den Umfang der für die Definition einer Grammatik notwendigen Regeln. Zeichen mit Sonderbedeutung:

`::=` Trennt linke Seite von der rechten Seite einer syntaktischen Regel.

`< >` Dient zum Einschluß von Nichtterminalsymbolen (Objekte, die durch syntaktische Regeln beschrieben werden, die man aber selber definieren kann, z.B.: Variablen, Anweisungen).

`|` Dient zur Darstellung von Alternativen auf der rechten Seite von syntaktischen Regeln.

Terminalsymbole (vordefinierte, feststehende Symbole, z.B.: begin, end) sind entweder einzelne Zeichen (z.B.: Klammern, Semikolon) oder unterstrichene Zeichenketten, z.B.:

```
<programm> ::= <kopf> <block> <ende>
<anweisungsteil> ::= begin <anweisung> end
```

Die Backus-Naur-Form wurde hier nur der Vollständigkeit wegen erwähnt und kurz vorgestellt. Im weiteren Verlauf der Vorlesung wird die zweite Möglichkeit, die Beschreibung der Syntax einer Programmiersprache durch Syntaxdiagramme, genutzt.

Syntaxdiagramm

Die Syntaxdiagramme wurden erstmals mit der Programmiersprache PASCAL eingeführt. Sie bieten gegenüber der BNF den Vorteil, daß die syntaktischen Regeln grafisch wiedergegeben werden und somit für den Menschen oft besser nachzuvollziehen sind, als die textuelle Notation der BNF. Folgende Abbildungsvorschriften bilden die Grundlage der Syntaxdiagramme:

- Ein Syntaxdiagramm ist ein knotenmarkierter, gerichteter Graph.
- Jedes Syntaxdiagramm hat eine Bezeichnung und zwei ausgezeichnete Knoten; genau einen Eingangs- und genau einen Ausgangsknoten.
- Zwei Arten von Knoten:
  - *Rechtecke* (Abbildung 2.6(a)) markieren Nichtterminalsymbole.
  - *Ovale* und *Kreise* (Abbildung 2.6(b)) markieren Terminalsymbole.

Jeder Weg durch ein Syntaxdiagramm liefert durch Aneinanderreihen der dabei erreichten Knotenmarkierungen eine zulässige syntaktische Struktur. In Abbildung 2.7 sind zwei Beispiele von Syntaxdiagrammen dargestellt.

Beispiel 2.5
Syntaxdiagramm für Bezeichner.

(a) Nichtterminale (b) Terminale

Abbildung 2.6: In Syntaxdiagrammen verwendete Notation

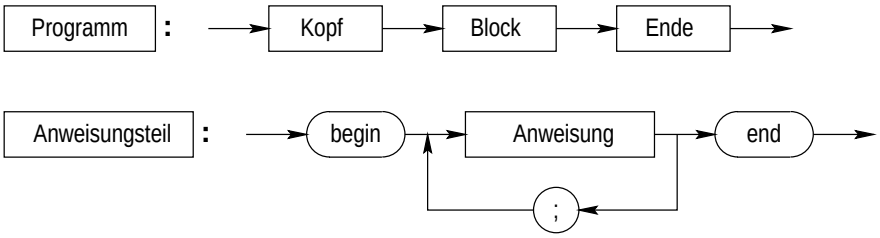


Abbildung 2.7: Beispiele für Syntaxdiagramme

Bezeichner werden zur Bezeichnung von Marken, Konstanten, Typen, Variablen und Funktionen verwendet. Das Syntaxdiagramm für Bezeichner ist in Abbildung 2.8 dargestellt. Nach diesem Syntaxdiagramm zulässige Bezeichner sind beispielsweise: Otto, B1 und u_16. Demgegenüber sind 1x und Bsp 1 keine zulässigen Bezeichner.

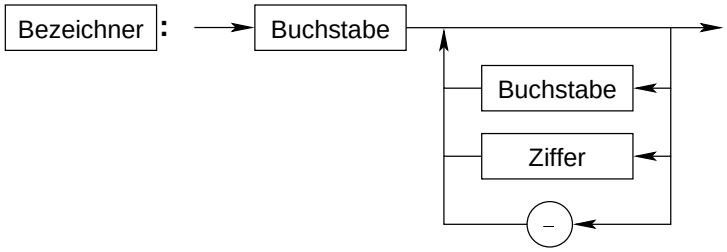


Abbildung 2.8: Syntaxdiagramm für Bezeichner

2.3.3 Informationsdarstellung

Der Rechner bekommt von der Außenwelt Signale zumeist in elektrischer Form mitgeteilt. Diese kennzeichnen einen bestimmten Zustand (high oder low, 0 oder 1, vorhanden oder nichtvorhanden). Diese Zustände sind binär, d.h. es existieren genau zwei. Der Informationsgehalt eines 0-1 - Zustandes wird als *Bit* (binary digit) bezeichnet. Eine Gruppe von

Bitkombination	Code 1	Code 2	Code 3
000	0	A	do1
001	1	B	re
010	2	C	mi
011	3	D	fa
100	4	E	sol
101	5	F	la
110	6	G	si
111	7	H	do2

Tabelle 2.4: Beispiele für verschiedene Zeichencodes unter Verwendung von Bittriaden (n=3)

8 Bits ergeben 1 *Byte*, 2^{10}=1024 Byte ergeben ein Kilobyte, 1024 KByte = 1048576 Byte ergeben 1 Megabyte, 1024 MByte = 1073741824 Byte ergeben 1 Gigabyte, usw. Mit n Bit können 2^n Informationen dargestellt werden. Zur Darstellung der Informationen stehen uns, wie bereits erwähnt, entsprechende Zeichen zur Verfügung:

- Ziffern (numerische Zeichen),
- Buchstaben (alphanumerische Zeichen),
- Sonderzeichen.

Wie werden diese in den Computer gebracht? Der Computer hat nur zwei Zustände zur Verfügung, das Minimalalphabet 1 Bit. Eine Erweiterung dieses Alphabets gelingt durch Bitkombinationen nach der Gleichung

$$m = 2^n$$

D.h., mit n Bit lassen sich m unterschiedliche Zeichen darstellen. Auf diesen binären Code werden alle weiteren gebräuchlichen Codes zurückgeführt. Zur Darstellung eines Zeichens werden somit n Bit lange Bitkombinationen gebildet und verarbeitet. In Tabelle 2.4 sind Beispiele für Codes angegeben, die mit 3 Bit dargestellt werden können. An der Schnittstelle zum Nutzer (extern) werden Codes benutzt, die überschaubarer sind. Nach der Anzahl der darstellbaren Grundsymbole m unterscheidet man

binären Code	{0,1}	n = 1	m = 2
oktalen Code	{0,1,2,3,4,5,6,7}	n = 3	m = 8
hexadezimalen Code	{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}	n = 4	m = 16

Für die Codierung von Zeichen werden meist Codes mit einer Länge von 8 Bit = 1 Byte verwendet, z.B. der erweiterte ASCII-Code[1] (American Standard Code for Information Interchange) und ANSI-Code (American National Standards Institute). Damit können 256 verschiedene Zeichen dargestellt werden. Diese Codes benötigen zum einen relativ wenig

[1]Der ursprüngliche ASCII-Code verwendete 7 Bit je Zeichen.

Alphanum. Zeichen	Dezimal	Binär	Oktal	Hexadezimal
1	49	00110001	061	31
2	50	00110010	062	32
.	46	00101110	056	2E
!	33	00100001	041	21
?	63	00111111	077	3F
a	97	01100001	141	61
b	98	01100010	142	62
[	91	01011011	133	5B
(	40	00101000	050	28

Tabelle 2.5: Ausschnitt aus dem ASCII-Code

Speicher, bieten andererseits aber genügend „Platz“ für die wesentlichen Zeichen. Heute werden aber auch zunehmend Codes mit einer Länge von 16 Bit verwendet, z.B. der Unicode. Ein Hauptgrund für die Einführung dieser Codes ist die zunehmende Internationalisierung, d.h. Software soll in verschiedenen Ländern verwendet werden, die verschiedene Schriftzeichen verwenden. Dafür reichen die im ASCII- und ANSI-Code zur Verfügung stehenden Zeichen nicht mehr aus. Eines haben aber alle Codes zur Darstellung von Zeichen gemeinsam: Das von uns benutzte externe Alphabet wird durch eine Transformation in einen entsprechenden Code in ein rechnerinternes Alphabet umgewandelt. Dies ist in Tabelle 2.5 am Beispiel des ASCII-Codes dargestellt.

Werden bestimmte Bitkombinationen (Binärwörter) als Ziffern gedeutet, dann lassen sich damit Zahlen darstellen. Verantwortlich für die Deutung der Bitfolge als Zeichenkette oder Zahl ist der vereinbarte Typ.

Beispiel 2.6
Codierung positiver ganzer Zahlen im Binärsystem.

$$
\begin{array}{cccccccc|c}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 \\
 & & & & \vdots & & & & \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 10 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 11 \\
 & & & & \vdots & & & & \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 254 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 255
\end{array}
$$

3 Grundsätzliches zum Programmieren in C

Die Sprache C ist eine höhere Programmiersprache, die es uns erspart, Programme in der umständlichen Maschinensprache abzufassen. Ein C-Programm besteht aus einer Folge von Anweisungen. Dieser Quellcode wird anschließend durch einen *Compiler* in Maschinensprache übersetzt. Der Vorgang vom *Quellprogramm* bis zum ausführbaren Programm läuft wie in Abbildung 3.1 dargestellt ab [Het93, S. 8]. Die Headerdateien enthalten Deklarationen von Funktionen, Makros, Konstanten usw., die bestimmte Funktionalitäten bereitstellen, z.B. Funktionen für die Ein- und Ausgabe in `stdio.h`. In den Headerdateien werden aber nur die Prototypen der Funktionen, d.h. Name, Parameter und Rückgabewert, angegeben, nicht aber die Anweisungen. Die Funktionalität der Funktionen wird in *Libraries* in vorkompilierter Form bereitgestellt. Damit ist es möglich, Funktionen zu nutzen, deren Umsetzung nicht bekannt sein müssen.

3.1 Programmaufbau

Aufgabe eines Programms: Ist die Manipulation von Daten.

Daten: Sind Diejenigen Programmbereiche, in die durch Befehle des Anwenderprogramms Informationen ein- und ausgetragen werden.

Datenobjekte: Sind benennbare Speichereinheiten eines bestimmten Datentyps.

Jedes C-Programm wird in der Regel nach folgender Grobstruktur gebildet:

- Include-Dateien, die externe Quellcodedateien einbinden,
- Definitionsteil für Konstanten und Makros,
- Deklarationsteil für globale Variablen und Konstanten,
- Funktionen (In C-Programmen gibt es den Unterprogrammtyp Prozedur im Gegensatz zu anderen problemorientierten Programmiersprachen nicht.),
- Hauptprogramm.

Das Syntaxdiagramm dieses grundlegenden C-Programmes ist in Abbildung 3.2 dargestellt. Ein entsprechender Quellcode könnte folgendermaßen aussehen:

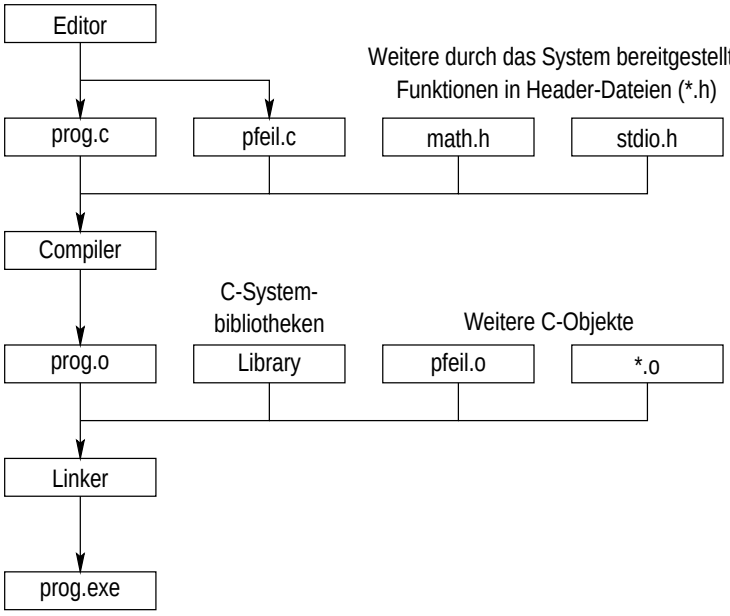


Abbildung 3.1: Phasen der Programmentwicklung

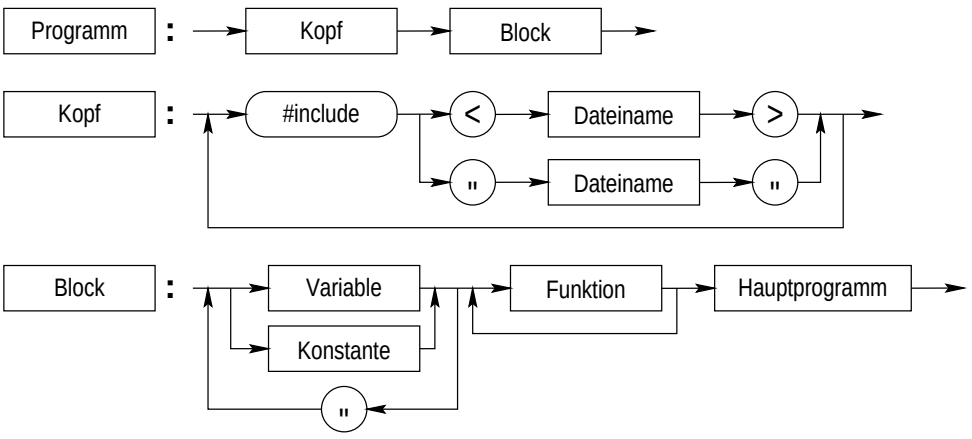


Abbildung 3.2: Syntaxdiagramm eines C-Programms

```
#include < ... >      /* externe Quellcodedateien */
#define  ...          /* Definition von Konstanten und Makros */

int ...               /* globale Variablen und Datentypen */

void funk1() {        /* Funktion 1 */
  ⋮
}

int funk2() {         /* Funktion 2 */
  ⋮
}

int main() {          /* Hauptprogramm */
  ⋮
}
```

Beispiel 3.1
Berechnung der größeren Zahl aus einem Zahlenpaar.

```
/* Einführendes Beispiel
 * Berechnung der größeren Zahlen aus einem Zahlenpaar
 * Abbruch mit x=-1
 */

/* Datei stdio.h für i/o-Operationen */
#include <stdio.h>

/* Hauptprogramm */
int main()
{
  int z, x;
  /* Eingabe organisieren */
  printf("Größte Zahl\nZahl z= ");
  scanf("%i", &z);
  printf("\nZahl x= ");
  scanf("%i", &x);
  /* Schleife zur Abfrage */
  do {
    if (x > z)
      z = x;
    printf("Ergebnis z = %i", z);
    printf("\nneues x = ");
    scanf("%i", &x);
  }
  while (x != -1);
  return 0;
}
/* Ende des Programmes */
```

3.2 Variablen, Konstanten und elementare Datentypen

3.2.1 Variablen

Variablen und Konstanten bilden das Datenmodell als Träger der Informationen. Eine *Variable* ist ein Datenobjekt mit veränderlichem Wert, das Namen und Datentyp besitzt und ab einer bestimmten Adresse im Speicher Platz belegt. Variablen müssen vor ihrer Verwendung deklariert werden. Die Syntax für die Definition einer Variablen lautet:

Syntax (Variablenname):

```
datentyp varname1 [, varname2];
```

3.2.2 Konstanten

Eine *Konstante* ist ein Datenobjekt mit unveränderlichem Wert. Dieses Objekt kann eine Zahl, ein Zeichen oder eine Zeichenkette sein. Es wird zwischen folgenden Typen von Konstanten unterschieden:

- Ganzzahlkoonstanten (**int**, **short**, **long**),
- Gleitkommakonstanten (**float** oder **double**),
- Zeichenkonstanten (**char**),
- Zeichenkettenkonstanten (Feld vom Typ **char**),
- Aufzählungskonstanten (enumerators, intern **int**).

Konstanten können auch symbolisch definiert werden, z.B.:

```
#define PI 3.141529
#define Begin {
#define End }
```

3.2.3 Elementare Datentypen

Bevor wir ein Programm schreiben, um darin Daten zu verarbeiten, müssen wir Speicherplatz für die Variablen vereinbaren. Dies geschieht über Datentypen. In C unterscheidet man in Basisdatentypen und deren Modifikationen durch Voranstellen der Typ-Modifizierer **short**, **long**, **double**, **signed** und **unsigned**. Darüber hinaus kann der Programmierer erweiterte Datentypen festlegen (Felder, Strukturen, Unions), die später behandelt werden. In Abbildung 3.3 sind die wichtigsten von C bereitgestellten Datentypen zusammen mit ihrem Speicherplatzbedarf angegeben. Dieser Platzbedarf ist abhängig von Rechnerarchitektur, Betriebssystem und Compiler. Der ANSI-Standard legt nur Mindestgrößen für die Basisdatentypen fest. Im folgenden beziehen wir uns auf den GNU-Compiler unter Windows 98.

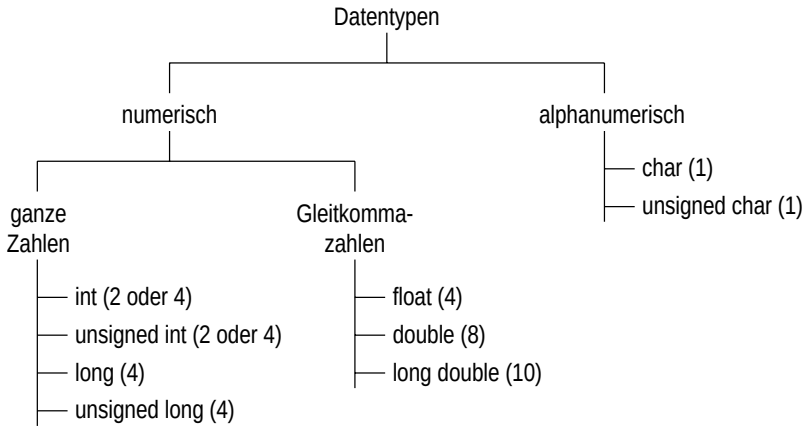


Abbildung 3.3: Einteilung und Speicherplatzbedarf elementarer Datentypen

Datentyp	Wertebereich	Speicherplatzbedarf
signed char	-128 ... 127	1 Byte
char	0 ... 255	1 Byte
short	-32.768 ... 32.767	2 Byte
unsigned short	0 ... 65.535	2 Byte
int	-2.147.483.648 ... 2.147.483.647	4 Byte
unsigned int	0 ... 4.294.967.295	4 Byte
long	-2.147.483.648 ... 2.147.483.647	4 Byte
unsigned long	0 ... 4.294.967.295	4 Byte

Tabelle 3.1: Wertebereiche verschiedener Integer-Datentypen

Ganze Zahlen

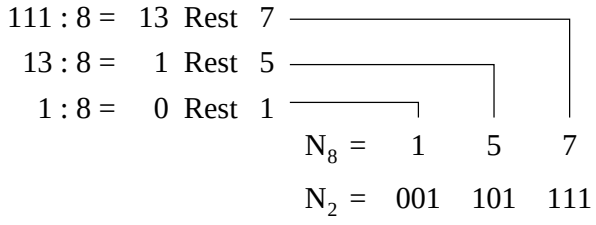
Für die Speicherung werden Integer-Datentypen verwendet. In Tabelle 3.1 sind die Wertebereiche und der benötigte Speicher verschiedenener Integer-Datentypen dargestellt.
Die Darstellung natürlicher Zahlen im Dualsystem erfolgt nach dem *Stellenwertverfahren*. Mittels folgender Formel läßt sich aus einer Zahl in einem Zahlensystem der Basis B die entsprechende Zahl im Dezimalsystem errechnen:

$$N_B = \sum_{i=0}^{n-1} Z_i * B^i = Z^{n-1} * B^{n-1} + \ldots + Z^1 * B^1 + Z^0 * B^0$$

Dabei ist Z_i die Ziffer an der i-ten Stelle in der Zahl zur Basis B.
Beispiel: $N_8 = 43721 = 4 * 8^4 + 3 * 8^3 + 7 * 8^2 + 2 * 8^1 + 1 * 8^0$

Beispiel 3.2
Umwandlung der Dezimalzahl $N_{10} = 111$ in Dualzahl mit $B = 2$

Algorithmus: $N_{10} \longrightarrow N_8 \longrightarrow N_2$, d.h. dezimal $\longrightarrow$ oktal $\longrightarrow$ binär



Dualzahl: $0 * 2^7 + 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$

Bitposition	7	6	5	4	3	2	1	0
Stellenwertigkeit	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Bitbelegung	0	1	1	0	1	1	1	1

1	*	1												=	1
1	*	2												=	2
1	*	2	*	2										=	4
1	*	2	*	2	*	2								=	8
0	*	2	*	2	*	2	*	2						=	0
1	*	2	*	2	*	2	*	2	*	2				=	32
1	*	2	*	2	*	2	*	2	*	2	*	2		=	64
0	*	2	*	2	*	2	*	2	*	2	*	2	*	2	=	0
															111

Reelle Zahlen

Bei der internen Darstellung numerischer Daten muß man selbstverständlich berücksichtigen, daß nicht nur ganze Zahlen sondern auch reelle Zahlen zu verarbeiten sind. Dieser Tatsache wird mit der unterschiedlichen internen Darstellung von Zahlen

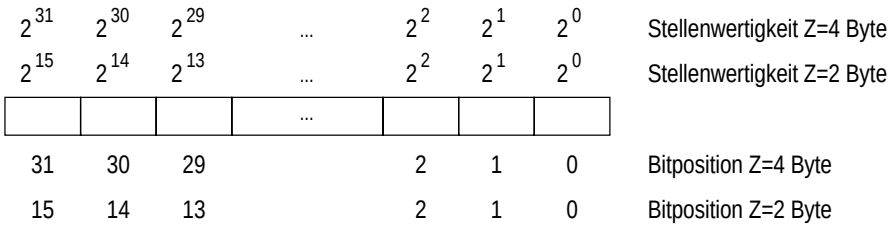


Abbildung 3.4: Darstellung von Festkommazahlen

- als Festkommazahl (Festpunktzahl) oder
- als Gleitkommazahl (Gleitpunktzahl)

entsprochen.

Festkommazahlen werden durch Bitfolgen mit einer festen Länge von Z Byte dargestellt. In Abbildung 3.4 ist diese Darstellung veranschaulicht. Zur Darstellung negativer Zahlen wird das Bit mit der höchsten Wertigkeit als Vorzeichen verwendet, d.h. Bitposition 15 bei Z = 2 Byte, Bitposition 31 bei Z = 4 Byte, usw. Dabei entspricht 0 positiv (+) und 1 negativ (-). Negative Zahlen werden als Zweierkomplement dargestellt. Die Ermittlung des Zweierkomplements erfolgt durch Invertierung aller Bits (außer dem Vorzeichenbit) und Addition von 1 an der Stelle 2^0.

Beispiel 3.3
Zahlenbeispiel zur Festkommadarstellung.

Dual																Dezimal
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32767
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1

Gleitkommazahlen Reelle Zahlen werden intern als Gleitkommazahlen abgebildet. Die Bestimmungsgleichung in halblogarithmischer Form lautet :

$$Zahl = (+/-)m * B(+/-)exp \qquad \text{mit } 0 < m < 1, exp \in \mathbb{N} \cup \{\varnothing\}$$

m - Mantisse
B - Basis
exp - Exponent

Die interne Darstellung einer Gleitkommazahl erfolgt in drei Teilen, einer Mantisse (m), einer Charakteristik (CH) und einem Vorzeichen (V). Für das Vorzeichen wird immer ein Bit verwendet. Die Länge der anderen beiden Komponenten hängt vom konkreten Datentyp ab. Die Charakteristik berechnet sich wiederum aus zwei Bestandteilen:

$$CH = e + k$$

wobei e der vorzeichenbehaftete Exponent und k ein konstanter Faktor sind.

Beispiel 3.4
Interne Darstellung von **float***-Werten.*

Für die Darstellung einer Zahl vom Datentyp **float** werden 32 Bit (4 Byte) verwendet. Davon entfallen 23 Bit auf die Mantisse (m), 8 Bit auf die Charakteristik (CH) und 1 Bit auf das Vorzeichen (V). Abbildung 3.5 verdeutlicht diese Aufteilung. Die Darstellung von Zahlen der Typen **double** und **long double** erfolgen auf ähnliche Weise. Für die Mantisse und die Charakteristik stehen dann nur entsprechend mehr Bits zur Verfügung.

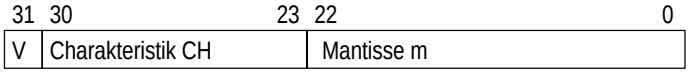


Abbildung 3.5: Darstellung von **float**-Zahlen

Beispiel 3.5
Darstellung der Dezimalzahl +26.5 als normierte Gleitpunktzahl.

1. Schritt: Umwandlung Dezimalzahl in Dualzahl

$$26.5_{10} = 11010.101_2$$

2. Schritt: Normierung der Dualzahl

$$11010.101 * 2^0 = 0.11010101 * 2^5$$
$$\rightsquigarrow m = 0.11010101$$

3. Schritt: Exponent 5 entspricht $e = 101$. Mit $k = 10000000$ ergibt sich:

$$CH = k + e = 10000000 + 101 = 10000101$$

Es folgt demnach:

$$26.5_{10} \hat{=} 0\ 10000101\ 11010101000000000000000_{\text{float}}$$

In einem C-Programm können Gleitkommazahlen auf 2 Arten dargestellt werden:

1. als Dezimalzahl. Das Syntaxdiagramm ist in Abbildung 3.6 dargestellt. Beispiele:

   3.14, 73.234 richtig
   .656, 65. falsch

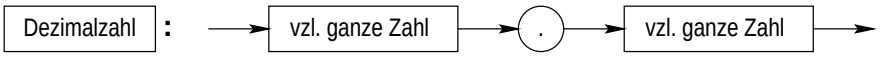


Abbildung 3.6: Syntaxdiagramm für Gleitkommazahlen als Dezimalzahl

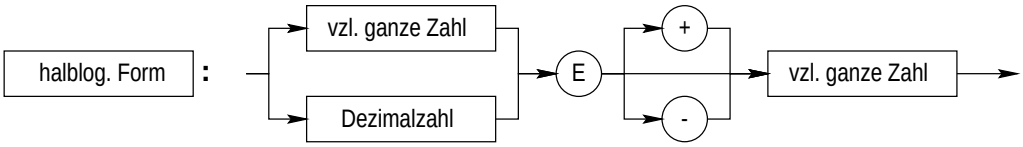


Abbildung 3.7: Syntaxdiagramm für Gleitkommazahlen in halblogarithmischer Form

2. in halblogarithmische Form. Das Syntaxdiagramm ist in Abbildung 3.7 dargestellt. Beispiele:

3.21E7, 0.5E+8, 111.11E-1, 3E-11	richtig
E10, 16E	falsch

Das vollständige Syntaxdiagramm für reelle Zahlen ist in Abbildung 3.8 dargestellt. Wird explizit keine andere Formatierung angegeben, dann wird bei Ausgaben die halblogarithmische Darstellung verwendet, z.B.: 0.12E-2.

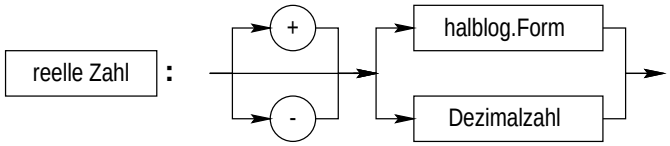
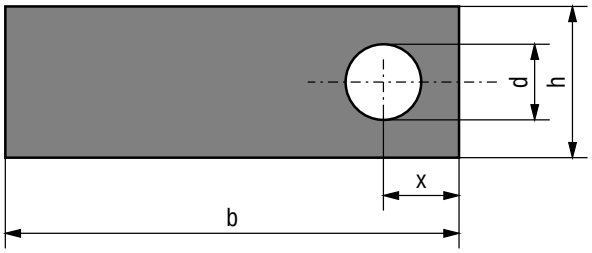


Abbildung 3.8: Syntaxdiagramm für reelle Zahlen

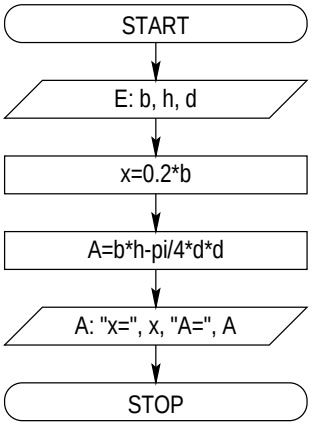
Beispiel 3.6
Berechnung der Fläche der in folgender Abbildung dargestellten Figur.



Gegeben ist weiterhin:

$$x = 0.2 * b$$
$$A = b * h - \frac{\pi}{4} * d^2$$

1. Programmablaufplan



2. Programm

```
/* Beispiel: Verwendung reeller Typen */

#include <stdio.h>

#define PI 3.14159265358979323846

int main()
{
  float b, h, d, x, a;

  scanf("%f%f%f", &b, &h, &d);
  x = 0.2 * b;
  a = b * h - PI / 4 * d * d;
  printf("\nx=%f A=%f\n", x, a);
  return 0;
}
```

3. Ergebnis

```
2.0
0.2E1
0.15

x=0.400000 A=3.982329
```

---

boolean-Typ

Dieser Typ ist in C als logischer Datentyp nicht vorgesehen. In C wird bei logischen Operationen ein Ergebniswert verschieden von Null als wahr und gleich Null als falsch gewertet.

char-Typ

Der Datentyp **char** ist vordefiniert und beschreibt die Menge von endlich vielen Zeichen, die ein Computersystem für die Kommunikation über das Terminal (Tastatur) benutzt. Zur Zeichenmenge gehören:

1. 'A' ... 'Z' Ordnungsnummer 65 - 90 im ASCII-Code,
2. 'a' ... 'z' Ordnungsnummer 97 - 122,
3. '0' ... '9' Ordnungsnummer 48 - 57,
4. Sonderzeichen Ordnungsnummer 32 - 47.

Als Ordnung gilt: 'A'<'Z', 'a'<'z' und '0'<'9'. Als Speicherplatz wird 1 Byte benötigt!

Beispiel 3.7
Anwendung des Datentyp **char**.

```
char z1, z2;
z1 = 'A';
scanf("%c", &z2);
printf("\nz1 = %c, z2 = %c", z1, z2);
```

Der Datentyp **char** ist die Grundlage für die Zeichenverarbeitung (Textverarbeitung). Eine Zeichenkette wird durch den strukturierten Datentyp Feld (vgl. Abschnitt 5.1) gebildet, z.B.:

```
char zk[15];
```

In dieser Variablen läßt sich eine Zeichenkette mit 14 Zeichen und einem Endezeichen abbilden.

3.3 Ein-und Ausgabefunktionen

Wir haben in unseren Beispielen bereits stillschweigend die Funktionen scanf und printf benutzt. Sie sind in der Datei `stdio.h` enthalten und organisieren standardmäßig die Eingabe über Tastatur und Ausgabe über Bildschirm. Bei der Eingabe müssen dabei sowohl der Datentyp und die Speicherbereiche der Variablen angegeben werden.

Beispiel 3.8
Anwendung der Ein- und Ausgabefunktionen scanf und printf.

```
#include <stdio.h>

int main() {
  int var_i;
  long var_l;

  printf("Variable var_i: ");
  scanf("%d", &var_i);
```

```
  printf("\nDezimal: %d", var_i);
  printf("\nOktal:   %o", var_i);
  printf("\nHexa:    %x", var_i);

  printf("\nVariable var_l: ");
  scanf("%ld", &var_l);
  printf("\nDezimal: %ld", var_l);
  printf("\nOktal:   %lo", var_l);
  printf("\nHexa:    %lx\n", var_l);

  return 0;
}
```

3.4 Steuerstrukturen

Die Programmiersprache C erlaubt eine modulare Programmierung und sieht deshalb die geeigneten Steuerstrukturen (Abbildung 3.9) für die Einbindung solcher Module in das Gesamtprogramm vor.

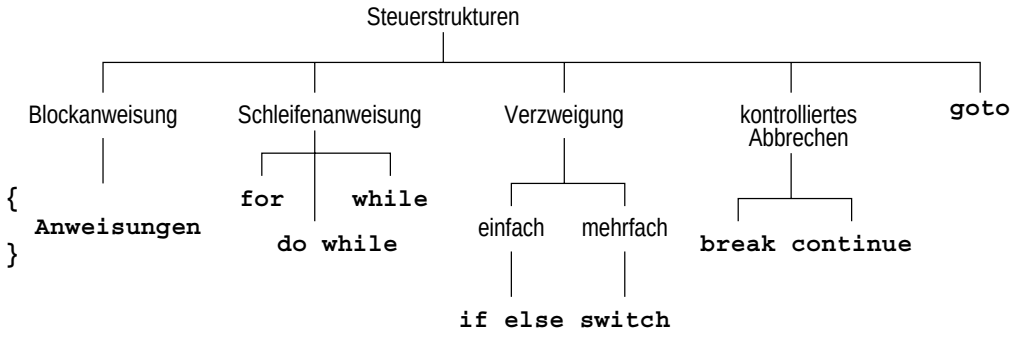


Abbildung 3.9: Steuerstrukturen in C

3.4.1 Schleifenanweisungen

Schleifen- oder auch Zyklen-Anweisungen sind die wohl wichtigsten Steuerkonstrukte in einem Programm. Mit ihnen können ganze Teile oder auch nur einzelne Anweisungen in Programmen wiederholt werden. Merkmale für Schleifen sind die Anzahl der Durchläufe (Schleife mit bekannter und unbekannter Dauer) sowie die Art der Abprüfung des Abbruchkriteriums. Dabei wird zwischen *anfangsgeprüften* oder *abweisenden Schleifen* und *endgeprüften* oder *nichtabweisenden Schleifen* unterschieden.

Zählschleife oder for-Schleife

Die Zählschleife ist eine Schleife bekannter Dauer, da die Anzahl der Wiederholungen von Anfang an bekannt ist. Der Ablauf einer **for**-Schleife ist in Abbildung 3.10 in Form eines PAP dargestellt.

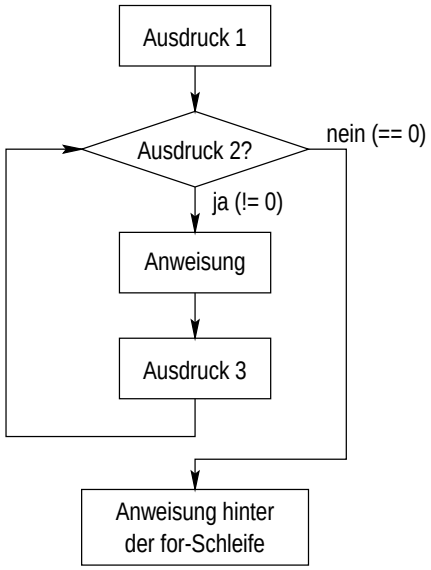


Abbildung 3.10: PAP einer **for**-Schleife

Syntax (for):

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

Ausdruck 1: Initialisierung der Schleifenvariablen

Ausdruck 2: Abbruchkriterium

Ausdruck 3: Reinitialisierung der Schleifenvariablen

Im Gegensatz zu anderen Programmiersprachen ist in C die Schrittweite frei wählbar, z.B.:

for (i=0; i<10; i+=2)	Schrittweite 2	i = 0,2,4,6,8
for (i=10; i>0; i−−)	negative Schrittweite -1	i = 10,9, ... 1
for (i=1; i<1024; i∗=2)	multiplikative Schrittweite	i = 1,2,4,8 ... 512

Beispiel 3.9
Alle Elemente eines Feldes auf 0 setzen.

Oftmals kommt es vor, daß eine Menge gleichartiger Elemente zu einem Feld zusammengefaßt werden soll. Nun sollen diese Feldelemente zunächst mit Null initialisiert werden.

1. Programmausschnitt:

```
int laufvar, feld[30];
for (laufvar=0; laufvar<30; laufvar++)  /* Schleifenkopf */
  feld[laufvar]=0;                      /* Schleifenkörper */
```

Beispiel 3.10
Summierung der n ungeraden Zahlen zwischen 1 und einem oberen Wert.

Programmlösung nach [Wil95, S. 244]. Vom Anwender ist die Zahl für den oberen Wert vorzugeben. Es ist eine **for**-Schleife zu verwenden.

1. Programm

```
#include <stdio.h>

int main() {
  long s_odd;              /* Summe der ungeraden Zahlen */
  int k;                   /* Schleifenkontrollvariable */
  int max;                 /* Oberwert */
  int n;                   /* Anzahl der Summanden */

  printf("Dieses Programm berechnet die Summe der\n");
  printf("ungeraden Zahlen von 1 bis n (1-30000).\n");
  printf("n = ");
  scanf("%d", &max);
  s_odd = 0;               /* Initialisierung der Summenvariablen */
  n = 0;                   /* Initialisierung des Summenzählers */
  for(k=1;k<=max;k+=2) { /* solange k <= Oberwert */
    s_odd += k;            /* Summieren */
    n++;                   /* Summanden zählen */
  }
  printf("\n\nEs wurden die %d ungeraden Zahlen von 1 bis ", n);
  printf("%d addiert.", max);
  printf("\n\nDie Summe dieser Zahlen ist: %ld\n", s_odd);
  return 0;
}
```

Code-Reduzierung: Die Schleifenvariablen s_odd und n können auch im Schleifenkopf initialisiert werden:

```
for (k=1, s_odd=0, n=0; k<=max; k+=2) {
  s_odd = s_odd + k;
  n++;
}
```

Das Komma als Trennoperator bewirkt, daß die ersten 3 Ausdrücke als ein Ausdruck zusammengefaßt werden.

do−while-Schleife

Der Zyklus der do-while-Schleife wird mindestens einmal durchlaufen. Der Grund liegt darin, daß das Abbruchkriterium am Ende erstmalig abgeprüft wird. Diese „Wiederholebis"-Schleife ist demzufolge nichtabweisend und von unbekannter Dauer. Der PAP einer **do−while**-Schleife ist in Abbildung 3.11 dargestellt. Da das Abbruchkriterium (auch Bedingung genannt) im Schleifenkörper erstmalig abgefragt wird, kann die Initialisierung der Schleifenbedingung vor Beginn der Schleife entfallen. Dieser Schleifentyp ist immer dann anzuwenden, wenn bewußt mindestens ein Durchlauf einkalkuliert wird.

Syntax (do−while):

```
do
  Anweisung;
while (Bedingung);
```

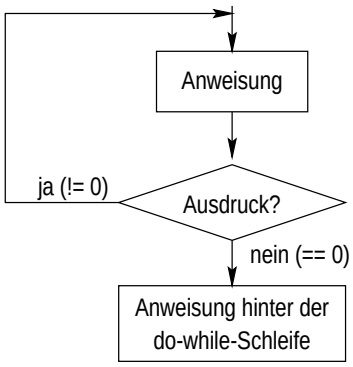


Abbildung 3.11: PAP einer **do−while**-Schleife

Beispiel 3.11
Bildschirmsteuerung: Es werden für einen Lösungsprozeß, z.B. zur Dateiverarbeitung verschiedene Operationen vorgesehen, die über den Bildschirm zu steuern sind.

1. Programm

```
#include <stdio.h>

int main() {
  char sel;                /* Schleifenkontrollvariable,
                            * nicht initialisiert
                            */
  do {                     /* die folgenden Anweisungen ausführen */
    printf("\nStammdatenverwaltung SIM 1\n\n");
    printf("\t1 = Anlegen\n");
    printf("\t2 = Ändern\n");
    printf("\t3 = Löschen\n");
    printf("\t4 = Drucken\n");
    printf("\t0 = Ende\n\n");
    printf("\tIhre Wahl: ");
    sel = getchar();       /* Wahl einlesen */
    getchar();             /* Enter lesen */
    switch (sel) {
    case '1':
      printf("\n\nHier ist die Anlegen-Simulation.\n");
      break;
    case '2':
      printf("\n\nHier ist die Ändern-Simulation.\n");
      break;
    case '3':
      printf("\n\nHier ist die Löschen-Simulation.\n");
      break;
    case '4':
      printf("\n\nHier ist die Drucken-Simulation.\n");
      break;
    case '0':              /* Programm beenden */
      printf("\n\n\t\t***\tEnde der Simulation\t***\n");
      break;
```

```
    default:              /* Falsche Eingabe */
      printf("\n\nFalsche Eingabe\n");
      break;
    }                     /* Ende switch */
  } while (sel != '0'); /* solange Eingabe nicht 0 */
  return 0;
}                         /* Ende main */
```

2. Ergebnis

```
Stammdatenverwaltung SIM 1

        1 = Anlegen
        2 = Ändern
        3 = Löschen
        4 = Drucken
        0 = Ende

        Ihre Wahl: 2

Hier ist die Ändern-Simulation.

Stammdatenverwaltung SIM 1

        1 = Anlegen
        2 = Ändern
        3 = Löschen
        4 = Drucken
        0 = Ende

        Ihre Wahl: 0

                ***     Ende der Simulation     ***
```

---

while-Schleife

Die **while**-Schleife ist eine anfangsgeprüfte, unter Umständen abweisende Wiederholung unbestimmter Dauer. Im Grenzfall wird bereits der erste Zyklus abgewiesen. Natürlich muß im Schleifenkörper auch die Schleifenbedingung geändert werden, um nicht in eine Endlosschleife zu verfallen. Auch die Initialisierung der Schleifenvariable(n) ist vor dem Eintritt in die Schleife durchzuführen, da ansonsten die zufällige Speicherbelegung dieser Variable(n) u.U. zu einem absolut falschen Ergebnis führen kann. Die Reinitialisierung erfolgt im Schleifenkörper. Der PAP der **while**-Schleife ist in Abbildung 3.12 dargestellt.

Syntax (while):

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

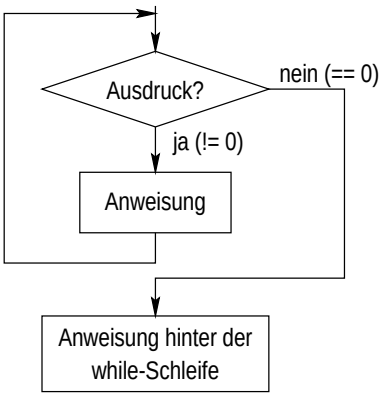


Abbildung 3.12: PAP einer **while**-Schleife

Beispiel 3.12
Das folgende Programm liest mittels einer **while***-Schleife so lange gerade Zahlen ein, bis eine ungerade eingegeben wird. Ausgegeben wird die Anzahl der eingegebenen geraden Zahlen [Wil95, S. 233].*

1. Programm

```
#include <stdio.h>

int main() {
  /* Mit gerader Zahl initialisieren, damit Schleifenbedingung wahr. */
  long n=2;
  /* zählt eingegebene gerade Zahlen, Initialisierung mit 0 */
  int cnt=0;

  printf("Das Programm schluckt gerade ganze Zahlen, bis Sie eine ");
  printf("ungerade\nZahl eingeben.\n\n");
  printf("Sie erfahren dann, wieviele gerade Zahlen Sie ");
  printf("eingegeben haben.\n\n");
  /* solange eine gerade Zahl eingegeben wurde */
  while (n % 2==0) {
    printf("Eine gerade Zahl bitte: ");
    /* Reinitialisierung der Schleifenvariablen */
    scanf("%ld", &n);
    cnt++; /* Anzahl der eingegebenen Werte zählen */
  }
  /* Achtung: Der Schleifenzähler ist um 1 größer, als die
   * tatsächliche gezählte Menge.
   */
  printf("\nSie gaben %d gerade Zahlen ein.\n", cnt-1);
  return 0;
}
```

2. Ergebnis

```
Das Programm schluckt gerade ganze Zahlen, bis Sie eine ungerade
Zahl eingeben.
```

```
Sie erfahren dann, wie viele gerade Zahlen Sie eingegeben haben.

Eine gerade Zahl bitte: 2
Eine gerade Zahl bitte: 4
Eine gerade Zahl bitte: 18
Eine gerade Zahl bitte: 9

Sie gaben 3 gerade Zahlen ein.
```

Geschachtelte Schleifen

Oftmals verlangt der Lösungsansatz die Anwendung der Schleifenkonstrukte in sogenannter verschachtelter Form, d.h., es werden mehrere Zyklen abhängig voneinander gekoppelt. Diese Aufgaben werden uns z.B. bei der Bearbeitung mehrdimensionaler Felder gestellt. Die Tiefe der Verschachtelungen hängt von der Semantik, aber auch von der Lesbarkeit der Lösung ab.

Beispiel 3.13
Es soll eine Matrix mit n Zeilen und m Spalten mit Anfangswerten über Tastatureingabe gefüllt werden. Der Lösungsansatz besteht aus 2 **for***-Schleifen.*

$$Matrix = \begin{bmatrix} 2 & 3 & 5 \\ 7 & -1 & 9 \end{bmatrix}$$

1. Programm

```
int Matrix[2][3];
int i, j;

for (i=0; i<2; i++)                /* Zeilen */
  for (j=0; i<3; j++)              /* Spalten */
    scanf("%d", &Matrix[i][j]);    /* einlesen */
```

3.4.2 Verzweigung, Alternative

Das Kontrollstrukturelement der Alternative gestattet die Auswahl (Selektion) von Lösungen in Abhängigkeit vom Wahrheitswert der Bedingung. Im einzelnen stehen zur Verfügung:

- Die bedingte Anweisung (**if**)
- die bedingte Anweisung mit Alternative (**if else**),
- geschachtelte bedingte Anweisungen (**if else if else**),
- Mehrfachanweisungen (**switch**).

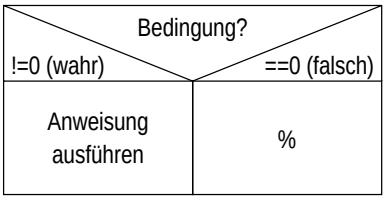


Abbildung 3.13: Struktogramm der bedingten Anweisung (**if**)

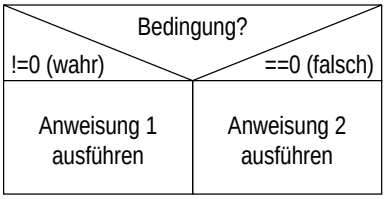


Abbildung 3.14: Struktogramm der bedingten Anweisung mit Alternative (**if**−**else**)

Bedingte Anweisung

Die bedingte Anweisung gestattet es, bestimmte Programmteile nur dann auszuführen, wenn eine bestimmte Bedingung erfüllt ist. In C steht zu diesem Zweck die **if**-Anweisung zur Verfügung. Die Bedingung kann dabei einfacher oder komplexer Art sein. Es kommt nur darauf an, daß die Auswertung auf den Wert 0 oder verschieden von 0 führt. Jeder Wert des Ausdrucks, der verschieden von 0 ist oder bei Vergleichsoperationen den Wert wahr liefert, wird auf !0 gesetzt, d.h., daß die nachfolgende Anweisung ausgeführt wird. Dagegen wird jedes Ergebnis des Ausdruckes mit dem Wert 0 oder falsch auf 0 gesetzt und die nachfolgende Anweisung (oder Anweisungsblock) wird übersprungen. Das Struktogramm der **if**-Anweisung ist in Abbildung 3.13 dargestellt. Die Syntax lautet:

Syntax (if):

```
if (Bedingung)
  Anweisung;
```

Bedingten Anweisung mit Alternative

Die bedingte Anweisung mit Alternative wird dann verwendet, wenn auch dann Anweisungen ausgeführt werden sollen, falls die Bedingung nicht erfüllt ist. Zu diesem Zweck bietet C mit **else** eine entsprechende Erweiterung der **if**-Anweisung an. Das Struktogramm ist in Abbildung 3.14 dargestellt, die Syntax lautet:

Syntax (if−else):

```
if (Bedingung)
  Anweisung 1;
else
  Anweisung 2;
```

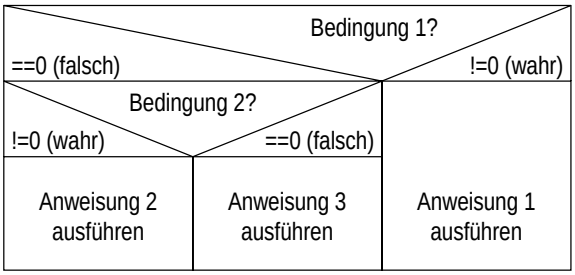


Abbildung 3.15: Struktogramm der geschachtelten bedingten Anweisung (**if −else if**)

Geschachtelte bedingte Anweisungen

Geschachtelte, bedingte Anweisungen werden u.a. verwendet, um sogenannte Filter aufbauen zu können. Das Struktogramm ist in Abbildung 3.15 dargestellt, die Syntax lautet:

Syntax (if −else if):

```
if (Bedingung 1)
  Anweisung 1;
else if(Bedingung 2)
  Anweisung 2;
else
  Anweisung 3;
```

Beispiel 3.14
Im Rahmen einer Volkszählung soll eine Auswertung hinsichtlich der Altersstruktur vorgenommen werden, indem die Personen in bestimmten Altersbereichen aufsummiert werden. Die Lösung verwendet eine **if −else if** *- Konstruktion.*

1. Programm

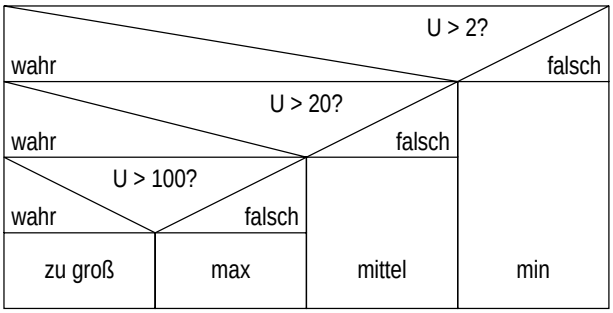
```
if (alter<20)
  alter1++;       /* alter1: jünger als 20 Jahre */
else if (alter>=20 && alter<30)
  alter2++;       /* alter2: 20-29 Jahre */
else if (alter>=30 && alter<40)
  alter3++;       /* alter3: 30-39 Jahre */
else if (alter>=40 && alter<50)
  alter4++;       /* alter4: 40-49 Jahre */
else if (alter>=50 && alter<60)
  alter5++;       /* alter5: 50-59 Jahre */
else
  alter6++;       /* alter6: alle ab 60 Jahre */
```

Beispiel 3.15
Es soll ein gemessener Spannungswert auf die Bereiche

100	$<$	*U*			*zu groß*
20	$<$	*U*	$<=$	*100*	*max*
2	$<$	*U*	$<=$	*20*	*mittel*
		U	$<=$	*2*	*min*

gefiltert werden. Die Lösung verwendet `if else` *- Verschachtelungen.*

1. Struktogramm



2. Programm

```
if (U>2)
  if (U>20)
    if (U>100)
      printf("zu groß");
    else
      printf("max");
  else
    printf("mittel");
else
  printf("min");
```

Mehrfachentscheidungen

Mehrfachentscheidungen kann man mit **if else**-Anweisungen in beliebiger Tiefe aufbauen. Der Nachteil daran ist aber, daß man relativ schnell die Übersicht verliert. Ein übersichtliches Einrücken des Quelltextes ist durch die Spaltenzahl des Bildschirmes ebenfalls begrenzt. Für eine solche Fallauswahl (Mehrfachentscheidung) ist die **switch**-Anweisung vorgesehen. Das Struktogramm ist in Abbildung 3.16 dargestellt, die Syntax ist:

Syntax (switch):

```
switch (Variable) {
  case Konstante 1: Anweisung 1;
  case Konstante 2: Anweisung 2;
  ⋮
  case Konstante n: Anweisung n;
  default: Anweisung x;
}
```

Der Ausdruck muß einen ganzzahligen Wert liefern, d.h., es können alle abzählbaren Datentypen (**int**, **char**, ...) benutzt werden. Die Konstanten 1 bis n nehmen dann einen bestimmten Wert aus dem Wertebereich dieses Datentyps an. Trifft dieser Wert zu, wird der nach dem Doppelpunkt stehende Anweisungsblock ausgeführt. Nach dem Schlüsselwort **default** (bedeutet in etwa Vorgabe oder Standard) können Anweisungen stehen, die in dem Falle ausgeführt werden, wenn kein **case** zutrifft. Die **default**-Marke ist optional. Bei der

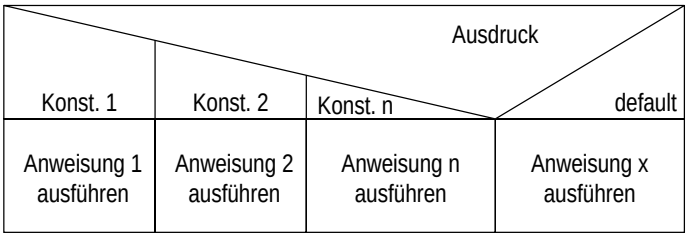


Abbildung 3.16: Struktogramm der **switch**-Anweisung

Anwendung der **switch**-Anweisung muß man beachten, daß alle Anweisungen, die innerhalb dieser Mehrfachentscheidung dem ausgewählten Fall folgen, ebenfalls ausgeführt werden. Um dies zu verhindern, sollte in jede **case**-Marke als letzte Anweisung ein **break** stehen.

Beispiel 3.16
Steuerung der Darstellung verschiedener Rechenergebnisse [Wil95, S. 222].

1. Programm

```
/* swtchcal.c demonstriert eine fünffache Auswahl mit Hilfe der
 * switch-Anweisung. Es wird wahlweise die Summe, die Differenz, das
 * Produkt, der Quotient zweier Zahlen berechnet. Darüber hinaus wird
 * ein möglicher Fehler infolge einer Division durch Null abgefangen.
 */
#include <stdio.h>
#include <stdlib.h>   /* für exit */

int main() {
  float x, y;         /* Eingabewerte */
  int selection;      /* Wahl der Rechenoperation */

  printf("Das Programm ermittelt Summe, Differenz,\n");
  printf("Produkt oder Quotient zweier Zahlen.\n");
  printf("Ihre beiden Zahlen:\n");
  scanf("%f %f", &x, &y);
  printf("\nWelche Rechenoperation?\n\n");
  printf("\t\ta\tfür Addition\n");
  printf("\t\ts\tfür Subtraktion\n");
  printf("\t\tm\tfür Multiplikation\n");
  printf("\t\td\tfür Division\n\n");
  printf("Ihre Wahl: ");

  getchar();          /* Enter lesen */
  selection = getchar();

  /* Eingabe überprüfen und Ergebnis ausgeben. */
  switch(selection) { /* Welche Rechenoperation gewählt? */
  case 'a':
  case 'A':
    printf("\nDie Summe der Zahlen ist: %f\n", x+y);
    break;
  case 's':
```

```
  case 'S':
    printf("\nDie Differenz der Zahlen ist: %f\n", x-y);
    break;
  case 'm':
  case 'M':
    printf("\nDas Produkt der Zahlen ist: %f\n", x*y);
    break;
  case 'd':
  case 'D':
    if(y == 0) {
      printf("\nDivision durch Null. Programmabbruch.\n");
      exit(1);
    } else {
      printf("\nDer Quotient der Zahlen ist: %f\n", x/y);
    }
    break;
  default:
    printf("\nFalsche Eingabe.\n");
  }
  return 0;
}
```

2. Ergebnis

```
Das Programm ermittelt Summe, Differenz,
Produkt oder Quotient zweier Zahlen.
Ihre beiden Zahlen:
12 18

Welche Rechenoperation?

     a    für Addition
     s    für Subtraktion
     m    für Multiplikation
     d    für Division

Ihre Wahl: m

Das Produkt der Zahlen ist: 216.000000

Das Programm ermittelt Summe, Differenz,
Produkt oder Quotient zweier Zahlen.
Ihre beiden Zahlen:
12 0

Welche Rechenoperation soll durchgeführt werden?

     a    für Addition
     s    für Subtraktion
     m    für Multiplikation
     d    für Division

Ihre Wahl: d
```

```
Division durch Null. Programmabbruch.
```

3.4.3 Kontrolliertes Abbrechen und goto

Zum Abschluß dieses Abschnitts wollen wir uns noch mit den Steuerkonzepten des kontrollierten Abbrechens und des unbedingten Sprungs auseinandersetzen.

break-Anweisung

Die **break**-Anweisung kann in Schleifen und in der **switch**-Anweisung vorkommen. Sie bewirkt eine vorzeitige Beendigung dieser Anweisungen. In einer Schleife ruft das Setzen eines **break** einen sofortigen Schleifenabbruch hervor, wenn die Steuerung bei dieser Anweisung angelangt ist. Die Syntax lautet:

Syntax (break):

```
break;
```

continue-Anweisung

Diese Anweisung ist nur in Schleifen zu verwenden. Sie bewirkt den Abbruch des gerade durchlaufenen Schleifendurchgangs und gibt nach Reinitialisierung der Schleifenvariablen die Steuerung an den Anfang des folgenden Schleifendurchgangs ab. Die Syntax lautet:

Syntax (continue):

```
continue;
```

Beispiel 3.17
Das Programm berechnet mit einer **while***-Schleife den Kehrwert beliebiger Zahlen. Der aktuelle Schleifendurchgang wird mit einer* **continue***-Anweisung abgebrochen, wenn der Wert 0 eingegeben wird. Anschließend kann ein neuer Wert eingegeben werden.*

1. Programm

```
#include <stdio.h>

int main() {
  float x;              /* Eingabewert */

  /* Schleifenkontrollvariable mit Initialisierung */
  char reply = 'j';
  while(reply == 'j') {
    printf("Kehrwertberechnung für alle Zahlen außer 0.\n");
    printf("Ihre Zahl: ");
    scanf("%f", &x);
    if(!x)              /* falls x gleich 0 ist */
      /* Abbruch des aktuellen Schleifendurchgangs zur Verhinderung
       * einer Division durch 0.
       */
```

```
      continue;
    printf("\nKehrwert der eingegebenen Zahl ist %f\n", 1/x);
    printf("Noch einen Kehrwert berechnen? (j/n): ");
    getchar();          /* Enter lesen */
    reply = getchar(); /* Kontrollvariable re-initialisieren */
  }                     /* Ende while */
  return 0;
}                       /* Ende main */
```

goto-Anweisung

Die **goto**-Anweisung bewirkt einen unumgänglichen (unbedingten) Sprung an die Stelle eines Programms, die durch die entsprechende Einsprungsmarke gekennzeichnet ist. Die Syntax lautet:

Syntax (goto):

```
label: Anweisung;
goto label;
```

Dabei trägt label irgendeinen zulässigen Namen. Die Sprunganweisung **goto** und die Markenanweisung müssen sich innerhalb einer Funktion befinden. **goto**-Anweisungen haben einen schlechten Ruf, da ihre häufige Anwendung zu sogenannten Spaghettiprogrammen führen kann.

Beispiel 3.18
Berechnung der Summe der ersten einhundert natürlichen Zahlen.

1. Programm mit goto

```
int x=1, s=0;

loop: s=s+x;
x++;
if (x<=100)
  goto loop;
```

2. Programm mit for-Schleife

```
for (x=1; x<=100; x++)
  s=s+x;
```

4 Einfache Anwendungen

Dieses Kapitel beinhaltet die Umsetzung der bisher vermittelten Erkenntnisse anhand von vier einfachen Beispielen. Dabei werden möglichst alle Phasen des Lebenszyklus einer Anwendung durchlaufen. Die Entwicklung von Programmen unter Verwendung einfacher Algorithmen, Datenmodelle und Steuerstrukturen wird geübt.

4.1 Berechnung einer Kreisfläche

Die Kreisfläche soll in Abhängigkeit vom Eingabewert als Radius, Durchmesser oder Umfang berechnet werden.

4.1.1 Problemanalyse

mathematisches Modell

Die Formeln für die Berechnung der Fläche lauten:

$$A = \pi * r^2 \qquad \text{oder} \qquad A = \pi * \frac{d^2}{4} \qquad \text{oder} \qquad A = \frac{u^2}{4\pi}$$

Die Formeln für die Berechnung des Umfangs lauten:

$$u = 2 * \pi * r \qquad \text{oder} \qquad u = \pi * d$$

Datenmodell

Eingabewert Wert, Fläche A, Steuerzeichen STZ, Wertebereich ganzzahlig, Zeichen oder Gleitkommazahl.

Lösungsalgorithmen

Berechnung der Fläche in einer Schleife mit dem entsprechenden Steuerzeichen als Auswahlparameter (Schleife, **switch**-Anweisung) Teilalgorithmus für Eingabe, Ausgabe.

4.1.2 Struktogramm

4.1.3 Programm

START Kreisberechnung			
Headerdateien einbinden			
Konstante PI definieren			
Ausgabe: Erklärung der Aufgabe			
Ausgabe: Aufforderung zum Einlesen eines Steuerzeichens			
Eingabe: Steuerzeichen STZ			
solange STZ!=4			
	STZ=? / 1	2	3
	Aus: Radius	Aus: Durchmesser	Aus: Umfang
	Ein: Wert	Ein: Wert	Ein: Wert
	A=PI*Wert*Wert	A=PI*Wert*Wert/4	A=Wert*Wert/(4*PI)
	Aus: A	Aus: A	Aus: A
	Ausgabe: Neues Steuerzeichen		
	Eingabe: STZ		
	Bildschirm löschen		
Ausgabe: Ende der Berechnung			
STOP			

Abbildung 4.1: Struktogramm zur Flächenberechnung

```
/* Kreisflächenberechnung mit Eingabewert Radius, Durchmesser oder
 * Umfang. Nutzung von Schleife und switch-Anweisung.
 */

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

#define PI 3.14159265358979323846

int main() {
  int STZ;
  double Wert, A;

  printf("Kreisflächenberechnung\n");
  printf("\tRadius:\t\tSTZ 1 eingeben\n");
  printf("\tDurchmesser:\tSTZ 2 eingeben\n");
  printf("\tUmfang:\t\tSTZ 3 eingeben\n");
  printf("\tEnde:\t\tSTZ 4 eingeben\n");

  /* Steuerzeichen initialisieren */
  printf("\nSteuerzeichen STZ eingeben: ");
  scanf("%i", &STZ);
  /* While-Schleife anlegen */
  while(STZ!=4) {
    switch(STZ) {
    case 1:
      printf("\nRadius eingeben: ");
```

```
      scanf("%lf", &Wert);
      A=PI*pow(Wert, 2);
      printf("\nFläche= %f", A);
      break;
    case 2:
      printf("\nDurchmesser eingeben: ");
      scanf("%lf", &Wert);
      A=PI*pow(Wert,2)/4;
      printf("\nFläche= %f",A);
      break;
    case 3:
      printf("\nUmfang eingeben: ");
      scanf("%lf", &Wert);
      A=pow(Wert,2)/(4*PI);
      printf("\nFläche= %f", A);
      break;
    }
    printf("\nNeues Steuerzeichen: ");
    scanf("%i", &STZ);
  }

  printf("\nEnde des Programmes\n");
  return 0;
}
```

4.1.4 Ergebnis

```
Kreisflächenberechnung
    Radius: STZ 1 eingeben
    Durchmesser: STZ 2 eingeben
    Umfang: STZ 3 eingeben
    Ende: STZ 4 eingeben

Steuerzeichen STZ eingeben: 1

Radius eingeben: 10

Flaeche= 314.000000
 Neues Steuerzeichen2

Durchmesser eingeben: 20

Flaeche= 314.000000
 Neues Steuerzeichen: 3

Umfang eingeben: 62.8

Flaeche= 314.000000
 Neues Steuerzeichen: 4

Ende des Programmes
```

4.2 Geradlinig begrenzte Fläche

Ein Viereck wird durch vier Geraden begrenzt. Es ist zu ermitteln, ob zufällig gewählte Koordinatenpunkte innerhalb des Vierecks liegen.

4.2.1 Problemanalyse

mathematisches Modell

Das Viereck wird durch folgende Geradengleichungen eingegrenzt (vgl. Abbildung 4.2):

$$\begin{aligned} y_1 &= -0,5x + 10 \\ y_2 &= 0,5x + 4 \\ y_3 &= 2x + 2 \\ y_4 &= -x + 10 \end{aligned}$$

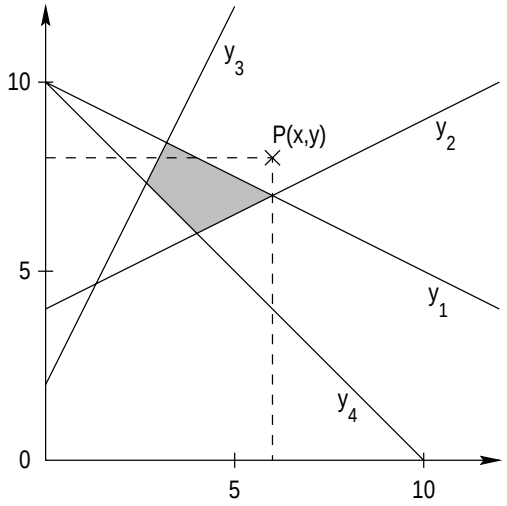


Abbildung 4.2: Viereck

Datenmodell

Punktkoordinaten x und y, Steuerzeichen stz, Wertebereich Gleitkomma, ganzzahlig oder char

Algorithmen

Eingabe: P(x,y), Steuerzeichen stz

Ausgabe: Treffer oder kein Treffer

Berechnung, Test: Ein Punkt liegt in der Fläche, wenn gilt:

$$y < y_1 \quad \text{und} \quad y > y_2 \quad \text{und} \quad y < y_3 \quad \text{und} \quad y > y_4$$

Verbale Beschreibung des Programmablauf

1. Eingabe Koordinaten
2. Berechnung + Test
3. Ausgabe Ergebnis
4. weitere Berechnung 'J' oder 'N' → ENDE

4.2.2 Programm

```
/* Programm Flächentest
 * Nutzung logischer Verknüpfungen
 */

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

int main() {
  int z;
  float x, y;

  printf("Blindekuhspiel\n\n");
  printf("Lösung durch logische UND-Verknüpfung");

  do {
    printf("\nPunktkoordinaten eingeben\n");
    printf("x=");
    scanf("%f",&x);
    printf("y=");
    scanf("%f",&y);
    if((y<-0.5*x+10) && (y>0.5*x+4) && (y<2*x+2) && (y>-x+10))
      printf("\nTreffer");
    else
      printf("\nPunkt liegt ausserhalb");
    printf("\nSteuerzeichen eingeben, 1 oder 0: ");
    scanf("%i",&z);
  } while(z!=0);

  printf("\nEnde des Programmes\n");
  return 0;
}
```

4.2.3 Ergebnis

```
Blindekuhspiel

Lösung durch logische UND-Verknuepfung
Punktkoordinaten eingeben
x=4.8
y=7.2

Treffer
```

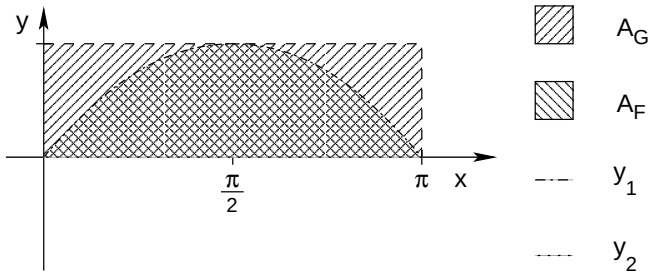


Abbildung 4.3: Skizze zur Flächenberechnung

```
Steuerzeichen eingeben, 1 oder 0: 1

Punktkoordinaten eingeben
x=2
y=4

Punkt liegt ausserhalb
Steuerzeichen eingeben, 1 oder 0: 0

Ende des Programmes
```

4.3 Flächenberechnung

Die Fläche, die durch eine Funktion und durch die x-Achse gebildet wird, ist in einem Bereich durch

1. Integration
2. mittels Monte-Carlo-Verfahren zu berechnen.

4.3.1 Problemanalyse

mathematisches Modell

Die Fläche wird durch folgende Funktionen begrenzt (vgl. Abbildung 4.3):

$$y_1 = \sin\ x \qquad y_2 = 0 \qquad \text{mit} \qquad 0 \leq x \leq \pi$$

Lösung mittels Integration:

$$A_r = \int_0^\pi \sin\ x\, dx = |-\cos\ x|_0^\pi = 2$$

Lösung mittels Monte-Carlo-Verfahren:

Der Lösungsansatz nach der Monte-Carlo-Methode geht davon aus, daß die Flächen mit einem Netz von Punkten, die durch einen gleichverteilten Zufallsgenerator erzeugt werden, belegt werden können. Das Verhältnis der Anzahl der Punkte, die zwischen der Sinuskurve

und der Abszisse liegen, und der Gesamtzahl der Punkte im Bereich der Grundfläche A_G ist ein Maß für die gesuchte Fläche. Ein Punkt in A_G ist bestimmt durch:

$$P(x, y) = P(Z_1 * \pi, Z_2 * 1) \quad \text{mit Zufallszahlen} \quad Z_1, Z_2 \in [0 \ldots 1]$$

Als Bedingung für die Trefferverteilung gilt:

$$\text{Wenn } y < \sin\ x \text{ und } y > 0 \text{ dann } T = T + 1$$

Wobei T die Anzahl der Treffer enthält. Die Anzahl der Punkte ist N. Damit ergibt sich für die Monte-Carlo-Fläche:

$$A_F = \frac{T}{N} * A_G$$

Datenmodell

Anzahl der Versuche N, Treffer T, Monte-Carlo Fläche AM, Integralfläche AI, Gesamtfläche AG, Steuerzeichen STZ, Differenz D, Koordinaten x und y, Begrenzer y1, Laufvariable j

4.3.2 Programm

```
/* Berechnung einer Fläche nach der Integrationsmethode, sowie der
 * Monte-Carlo-Methode
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define PI 3.14159265358979323846

int main() {
  float D, AI, AM, y, y1, x;
  int j, n, t, z1, z2;
  int stz;

  printf("Flächenberechnung: ");
  printf("Integral- und Monte-Carlo-Methode\n");

  /* Integralmethode */
  printf("Integral y=sin(x) mit 0<=x<=PI, yi=-cos(x)\n");
  AI=-(cos(PI)-cos(0));
  printf("Integrationsergebnis AI = %f\n\n", AI);

  /* Monte-Carlo-Methode */
  srand(time(NULL));
  do {
    printf("Anzahl der Versuche eingeben!\n");
    scanf("%i", &n);
    t=0; /* Treffer initialisieren */
    for(j=1;j<=n;j++) {
```

```
      z2=rand()%1001;
      y=z2/1000.0;
      z1=rand()%1001;
      x=z1*PI/1000; y1=sin(x);
      if((y<y1) && (y>0))
        t=t+1;
    }
    printf("Treffer: %i\n",t);
    AM=t*PI/n;
    printf("Monte-Carlo-Methoden-Ergebnis AM: %f\n",AM);
    D=(AI-AM)*100/AI;
    printf("Die Abweichung beträgt %f Prozent.\n", D);
    printf("<1> für Wiederholen, <0> für Ende eingeben! ");
    scanf("%i", &stz);
  } while(stz != 0);
  return 0;
}
```

4.3.3 Ergebnis

```
Flächenberechnung: Integral- und Monte-Carlo-Methode
Integral y=sin(x) mit 0<=x<=PI, yi=-cos(x)
Integrationsergebnis AI = 2.000000

Anzahl der Versuche eingeben!
1000
Treffer: 632
Monte-Carlo-Methoden-Ergebnis AM:1.985428
Die Abweichung betraegt 0.728601 Prozent
<1> fuer Wiederholen, <0> fuer Ende eingeben!1

Anzahl der Versuche eingeben!
10000
Treffer: 6350
Monte-Carlo-Methoden-Ergebnis AM:1.994853
Die Abweichung betraegt 0.257373 Prozent
<1> fuer Wiederholen, <0> fuer Ende eingeben!0
```

4.4 Der fleißige Tierhändler

Ein Tierhändler verkauft 100 Tiere (Hunde, Katzen, Mäuse) zu insgesamt 100 DM. Die Preise je Tier sind:

15,- DM/Hund

1,- DM/Katze

0,25 DM/Maus

Frage: Wieviel Hunde, Katzen und Mäuse werden verkauft?

4.4.1 Problemanalyse

mathematisches Modell

Mengenbilanz:	Hunde + Katzen + Mäuse = 100
Geldbilanz:	15 * Hunde + 1 * Katzen + 0,25 * Mäuse = 100
Fazit:	3 Unbekannte (Anzahl der Hunde, Katzen und Mäuse) aber nur 2 Gleichungen. $\rightsquigarrow$ Unterbestimmtes Gleichungssystem

Datenmodell

Hunde(H), Katzen(K), Mäuse(M), Menge i, Geld j, Wertebereich ganzzahlig

Algorithmus

Iterative Lösung über Schleifen:

1. **for**-Schleife:
   Schleifenbedingung: Anzahl Hunde (H) $1 < H < 6$ da $7 * 15 > 100$
2. **for**-Schleife:
   Schleifenbedingung: Anzahl Katzen (K) $1 < K < 100 - H$
3. **do**−**while**-Schleife:
   Schleifenbedingung: Anzahl Mäuse (M) $4 < M < 100 - H - K$

4.4.2 Programm

```
/* Schleifen zur Auflösung eines unterbestimmten Gleichungssystems */

#include <stdio.h>

int main() {
  int i, j, h, k, m;

  printf("Beim Tierhändler :\n");
  for(h=1;h<7;h++) {
    for(k=1;k<(100-h);k++) {
      m=0;
      do {
        i=h+k+m;
        j=15*h+k+(m/4);
        if((i==100) && (j==100)) {
          printf("Fuer %d DM bekommt man:\n", j);
          printf("%d Hunde\n", h);
          printf("%d Katzen\n", k);
          printf("%d Maeuse.\n", m);
          printf("Das sind genau %i Tiere.\n", i);
        }
        m+=4;
      } while((h+k+m)<=100);
```

```
    }
  }
  return 0;
}
```

4.4.3 Ergebnis

```
Beim Tierhaendler :
Fuer 100 DM bekommt man :
3 Hunde
41 Katzen
56 Maeuse.
Das sind genau 100 Tiere.
```

5 Zusammengesetzte Datentypen

Es ist semantisch sicherlich richtig, logisch zusammenhängende Werte auch in einer Variablen mit einem bestimmten zusammengesetzten (strukturierten) Datentyp zu beschreiben. Solche zusammenhängenden Werte findet man beispielsweise in einer Tabelle oder in einer Information, die sich aus mehreren Teilen unterschiedlichen Wertebereichs (Typ) zusammensetzt. Letzteres kann z.B. die Adresse für eine Person sein, die sich in der Regel aus der Straße mit Hausnummer, der Postleitzahl und dem Ort bildet. Der Vorteil dieser Vorgehensweise besteht in der Reduzierung des Codes und im Umgang mit dieser Variablen z.B. bei der Ein- und Ausgabe. Die Sprache C bietet drei solcher Typen an: Array, Struktur und Union. Arrays (Felder) enthalten Datenelemente gleichen Typs. Strukturen können aus Datenelementen verschiedenen Typs zusammengesetzt werden.

5.1 Arrays (Felder)

Arrays fassen Datenelemente gleichen Typs zusammen und speichern diese unmittelbar hintereinander. Als Typen der Elemente kommen in Frage:

- Elementare Datentypen wie **char**, **short**, **int**, **long**, **float**, **double**,
- Arrays,
- Strukturen und Unions,
- Zeiger, die zur Speicherung von Adressen von Datenobjekten verwendet werden.

Arrays können eindimensional sein, dann spricht man auch von Vektoren. Setzen sich die Felder aus Spalten und Zeilen zusammen, so ist eine Tabelle gemeint. Felder können auch mehrdimensional sein. Wenn man beispielsweise ein Feld von Punkten mit den jeweiligen 3 Raumkoordinaten darstellen will, so könnte man jede Koordinate als eine Dimension betrachten. Sind Feldelemente wiederum Arrays, so spricht man von geschachtelten Arrays.

5.1.1 Eindimensionale Arrays

Syntax (Felddefinition):

```
datentyp arrayname[elementezahl];
```

Die Definition einer Variablen feld mit 10 Elementen vom Typ **short** erfolgt beispielsweise mit der Anweisung **short** feld [10];. Im Speicher werden die Elemente eines Feldes hintereinander abgelegt (Abbildung 5.1). Dadurch läßt sich die Adresse jedes Feldelementes aus

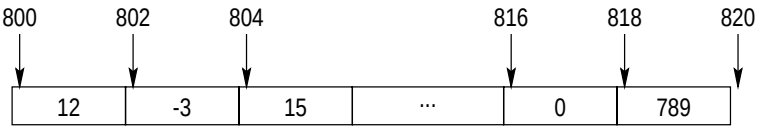


Abbildung 5.1: Speicheradressen in einem Array

der Anfangsadresse und der Nummer des entsprechenden Elementes bestimmen. Im Falle des Beispiels sind alle Elemente 2 Byte (**short**) lang.

Da die Elementezahl in der Definition mit angegeben werden muß, bleibt diese auch während des gesamten Programms *statisch* erhalten. Wir werden später kennenlernen, wie man mittels Zeigertechnik dynamische Felder anlegen kann.

Operationen auf Arrays

Typische Operationen, die auf Feldern ausgeführt werden, sind Indizierung, Zugriff und Initialisierung. Wenn man mit Feldern arbeiten will, muß man den Zugriff auf die Feldelemente organisieren. Jedes Feldelement ist durch Angabe des Namens des Feldes und durch einen Index, der die Position des Elementes im Feld angibt, erreichbar. Beispielsweise kann mit `feld[0]` auf das erste Element und mit `feld[10]` auf das elfte Element eines Feldes mit dem Namen `feld` zugegriffen werden. Der Indexwert kann auch durch eine abzählbare Variable angegeben werden, z.B. `feld[k]`. Der Zugriff auf die Werte eines einzelnen Feldelementes und die ausführbaren Operationen sind identisch mit der Verfahrensweise auf den Basistyp, z.B.:

`feld[0] = 10;`	Weist dem 1. Element des Feldes `feld` den Wert 10 zu.
`k = feld[1] + 3;`	Addiert 3 zum Wert des 2. Feldelements und weist das Ergebnis der Variablen `k` zu.
`printf("'%i", feld[5]);`	Gibt den Wert des 5. Feldelementes als ganze Zahl aus.

Die Initialisierung eines Feldes kann auch sofort mit der Definition vorgenommen werde. Dazu werden einfach Konstanten des gewählten Typs an die Felddefinition angehängt, z.B.:

```
float zahl[100] = {12.0, -8.13, 3.14};
```

Diese Definition initialisiert die ersten 3 Elemente mit den angegebenen Zahlenwerten und setzt die restlichen auf 0. Sollte das Feld zahl generell mit 0 initialisiert werden, kann man dies durch folgende Initialisierung erreichen:

```
float zahl[100] = {0.0};
```

Ein- und Ausgabeoperationen mit Feldelementen

Die Sprache C kennt keine Befehle, die auf Arrays als Ganzes zugreifen. Deshalb muß auch die Ausgabe und Eingabe elementweise erfolgen. Dazu nutzt man in der Regel das Konstrukt der Schleife. Mit folgender **for**-Schleife werden beispielsweise für die ersten 5 Elemente eines Feldes Werte von der Tastatur eingelesen und anschließend wieder ausgegeben:

```
for (k=0; k<5; k++) {
  scanf("%d", &v[k]);
  printf("%d", v[k]);
}
```

Programmbeispiele

Beispiel 5.1
Ermitteln der Größe, d.h. des Speicherplatzbedarfs verschiedener Felder [Wil95, S. 293].

1. Programm

```
#include <stdio.h>

int main() {
  double d[10];
  float  f[10];
  long   l[10];
  int    i[10];
  short  s[10];
  char   c[10];

  printf("%5d%5d%5d%5d%5d%5d\n",
         sizeof(d), sizeof(f), sizeof(l),
         sizeof(i), sizeof(s), sizeof(c));
  printf("%5d%5d%5d%5d%5d%5d\n",
         sizeof(d[1]), sizeof(f[1]),
         sizeof(l[1]), sizeof(i[1]),
         sizeof(s[1]), sizeof(c[1]));
  return 0;
}
```

2. Ergebnis

```
   80   40   40   40   20   10
    8    4    4    4    2    1
```

Beispiel 5.2
Zunächst liest das Programm [Wil95, S. 305] mit einer Schleife einige statistische Werte (Fahrzeughäufigkeiten pro Tag) in ein Feld ein. Die Werte werden über die Tastatur eingegeben, summiert und zur Kontrolle, inklusive ihrer Gesamtsumme wieder ausgegeben.

1. Programm

```
#include <stdio.h>

int main() {
  /* Anzahl der Fahrzeuge für jeden der sieben Tage.  8 Elemente
   * statt 7. Für die Tage 1 bis 7 werden die Elemente vehicles[1]
   * bis vehicles[7] benutzt.  vehicles[0] bleibt frei.
   */
  long vehicles[8];
  long s = 0;              /* Gesamtsumme der gezählten Fahrzeuge */
```

```
  short i;                /* Kontrollvariable */

  printf("Geben Sie die Fahrzeugzahlen für die Tage ");
  printf("1 - 7 ein.\n");
  for(i=1;i<8;i++) {
    printf("%hd. Tag: ", i);
    scanf("%ld", &vehicles[i]);
    s = s + vehicles[i]; /* Gesamtsumme bilden */
  }
  printf("\nEs wurden folgende Werte eingegeben:\n\n");
  for(i=1;i<8;i++)
    printf("%hd. Tag\t", i);
  printf("\n");
  for(i=1;i<8;i++)
    printf("%ld\t", vehicles[i]);
  printf("\n\nInsgesamt gezählte Fahrzeuge: %ld\n", s);
  return 0;
}
```

2. Ergebnis

```
Geben Sie die Fahrzeugzahlen für die Tage 1 - 7 ein:
1. Tag: 12
2. Tag: 16
3. Tag: 45
4. Tag: 16
5. Tag: 3
6. Tag: 25
7. Tag: 67

Es wurden folgende Werte eingegeben:

1. Tag  2. Tag  3. Tag  4. Tag  5. Tag  6. Tag  7. Tag
12      16      45      16      3       25      67

Insgesamt gezählte Fahrzeuge: 184
```

5.1.2 Mehrdimensionale Arrays

Auch auf mehrdimensionale Felder läßt sich der bisher erläuterte Mechanismus übertragen. Felder sind im Speicher nach dem letzten Index geordnet, d.h. beispielsweise, daß ein zweidimensionales Feld zeilenweise abgespeichert wird.

Syntax (Feld):

```
datentyp arrayname[e1][e2] ... [en];
```

Die Definition eines dreidimensionalen Feldes mit 27 Elementen erfolgt beispielsweise mit folgender Anweisung:

```
int feld[3][3][3]={1,2,3,4,5,6,7,8,9, ... ,27};
```

oder mit

```
int feld[3][3][3]={{{ 1, 2, 3},{ 4, 5, 6},{ 7, 8, 9}},
                   {{10,11,12},{13,14,15},{16,17,18}},
                   {{19,20,21},{22,23,24},{25,26,27}}};
```

Damit wird ein „Würfel“ festgelegt, wobei das erste Element feld [0][0][0] den Wert 1 und das letzte Element feld [2][2][2] den Wert 27 zugewiesen bekommt. Mit diesen Beispielen wurden auch die Zugriffsform über die indizierten Feldelemente und die mögliche Initialisierung über die Definition erklärt. Die Ein- und Ausgabeoperationen werden über geschachtelte Schleifen organisiert.

Beispiel 5.3
Speicherung von Daten der Mitarbeiter einer Firma [Wil95, S. 315]. Das Programm liest Werte in ein zweidimensionales Array ein und gibt sie wieder aus.

1. Programm

```
#include <stdio.h>

int main() {
  int k[3][4];          /* 2-D-Array für Mitarbeiterzahlen */
  int i, j;             /* Kontrollvariablen */
  long s = 0;           /* Gesamtzahl Mitarbeiter */

  printf("Mitarbeiterzahlen eines Konzerns eingeben.\n");
  printf("3 Unternehmen zu je 4 Filialen.\n\n");

  /* Werte einlesen */
  for(i=0;i<3;i++) {    /* Arrayzeilen */
    for(j=0;j<4;j++) { /* Arrayspalten */
      printf("%d. Unternehmen %d. Filiale: ", i+1, j+1);
      scanf("%d", &k[i][j]);
    }
  }

  /* Gesamtzahl der Mitarbeiter berechnen */
  for(i=0;i<3;i++)      /* Arrayzeilen */
    for(j=0;j<4;j++)    /* Arrayspalten */
      s = s + k[i][j];

  /* Werte ausgeben */
  printf("\n\tF0\tF1\tF2\tF3\n");
  for(i=0;i<3;i++) {    /* Arrayzeilen */
    printf("U%d\t", i);
    for(j=0;j<4;j++)    /* Arrayspalten */
      printf("%d\t", k[i][j]);
    printf("\n");
  }
  printf("\nGesamtzahl der Mitarbeiter: %ld\n", s);
  return 0;
}
```

2. Ergebnis

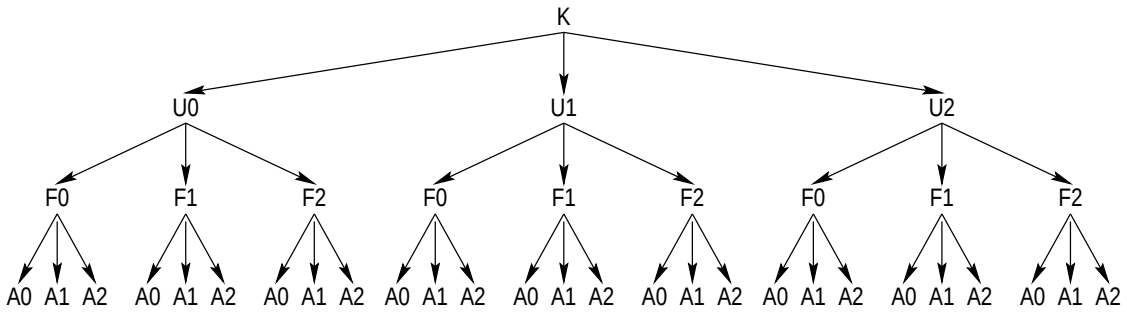


Abbildung 5.2: Strukturbaum eines Konzerns

```
Mitarbeiterzahlen eines Konzerns eingeben.
3 Unternehmen zu je 4 Filialen.

1. Unternehmen 1. Filiale: 24
1. Unternehmen 2. Filiale: 13
1. Unternehmen 3. Filiale: 12
1. Unternehmen 4. Filiale: 6
2. Unternehmen 1. Filiale: 28
2. Unternehmen 2. Filiale: 17
2. Unternehmen 3. Filiale: 70
2. Unternehmen 4. Filiale: 3
3. Unternehmen 1. Filiale: 26
3. Unternehmen 2. Filiale: 78
3. Unternehmen 3. Filiale: 15
3. Unternehmen 4. Filiale: 20

        F0      F1      F2      F3
U0      24      13      12      6
U1      28      17      70      3
U2      26      78      15      20

Gesamtzahl der Mitarbeiter: 312
```

Häufig werden mehrdimensionale Felder genutzt, um gewisse Strukturen abzubilden. Nehmen wir an, daß der im obigen Beispiel behandelte Konzern sich in Unternehmen, Filialen und Abteilungen gliedert, so kann man diese Struktur als dreidimensionales Feld abspeichern:

```
int konzernstruktur[3][3][3];
```

Mit dieser Struktur könnten somit die Beschäftigten auf Konzern-, Unternehmens-, Filial- und Abteilungsebene gespeichert und berechnet werden.

5.1.3 Zeichenketten (Strings)

Zeichenketten oder Strings sind Zeichenfolgen, die aus Zeichen des darstellbaren Zeichensatzes bestehen. Wir haben bisher in der printf-Funktion nur Zeichenkettenkonstanten benutzt.

Zeichenketten können jedoch auch als Variablen abgebildet werden. Dazu definieren wir eindimensionale **char**-Felder. Beispielsweise definiert **char** s[13] eine Variable, die 12 Zeichen und eine Endemarkierung (\0) aufnehmen kann. Die Endemarkierung dient dazu, beim Durchmustern des Strings das Ende zu finden. Da es sehr umständlich ist, Zeichenkettenfelder ständig elementweise zu verarbeiten, hat man vordefinierte Funktionen geschrieben, die die Zeichenketten quasi in einem Stück behandeln.

Operationen auf Zeichenketten

Wie wir bereits oft praktiziert haben, können Variablen auf verschiedene Art initialisiert werden. Gleiches trifft auch auf String-Variablen zu:

```
char s[5]= {'A', 'B', 'E', 'R', '\0'};
```

oder

```
char s[ ]="ABER";
```

Ein zweidimensionales Feld kann wie folgt anfangsbelegt werden:

```
char s2[4][6] = {"ALPHA", "BETA", "GAMMA", "DELTA"};
```

Damit ergibt sich im Speicher folgende Belegung:

s2[0]	A	L	P	H	A	\0
s2[1]	B	E	T	A	\0	\0
s2[2]	G	A	M	M	A	\0
s2[3]	D	E	L	T	A	\0

Die Ein- und Ausgabe von Zeichenketten kann mit den scanf- bzw. printf-Funktionen erfolgen. Zu beachten ist, daß bei der Eingabefunktion der Adreßoperator „&“ nicht erforderlich ist. Der Grund liegt darin, daß der Compiler mit dem Namen des Zeichenkettenfeldes immer die Anfangsadresse des ersten Elementes verbindet:

```
scanf("%s", name);
printf("%s", name);
```

oder mit Angabe der Zeichenzahl:

```
scanf("%10s", name);
printf("%10s", name);
```

Als Alternative stehen die Funktionen gets(name) für die Eingabe von der Tastatur und puts(name) für die Ausgabe über den Bildschirm zur Verfügung. Mit zweidimensionalen **char**-Feldern kann man z.B. Namenslisten verwalten. Zunächst definieren wir eine Liste mit:

```
char names[10][20];
```

Damit können 10 Namen mit bis zu 19 Zeichen verarbeitet werden. Mit der Anweisung:

```
gets(names[0]);
```

bzw.

```
puts(names[0]);
```

wird eine Zeichenkette in das erste Element des Feldes eingelesen, bzw. es wird das erste Element des Feldes ausgegeben.

Da die Arbeit mit Zeichen und Zeichenketten quasi die Grundlage der Textverarbeitung darstellt, sind in der Bibliothek `string.h` weitere Funktionen verfügbar (vgl. Tabelle 5.1). Bei der Anwendung der Funktionen strcpy und strcat gilt es zu beachten, daß die Funktionen keinen Speicher allokieren. Dieser muß vom aufrufenden Programm in ausreichender

Kopieren	strcpy(s1, s2)	Der String s2 wird in den String s1 kopiert. Der Speicher für s1 muß vorher vom Programm allokiert werden.
Verkettung	strcat(s1, s2)	String s2 wird an s1 angehängt. Dabei muß s1 genügend groß deklariert sein, um die zusätzlichen Zeichen aus s2 aufnehmen zu können
Vergleich	strcmp(s1, s2)	Beiden Strings werden verglichen. Das Ergebnis wird als **int**-Wert zurückgegeben. Das Resultat ist dabei kleiner 0, wenn s1<s2, gleich 0, wenn s1=s2 und größer 0, wenn s1>s2.
Stringlänge	strlen(s)	Liefert die Länge der Zeichenkette in Byte, d.h. die Anzahl der Zeichen. Das abschließende 0-Byte wird dabei nicht mitgezählt.

Tabelle 5.1: Funktionen zur Bearbeitung von Zeichenketten in `string.h`

Größe bereitgestellt werden. Soll beispielsweise eine Zeichenkette kopiert werden, müssen im Hauptprogramm zwei Felder, eines für den Quell- und eines für den Zielstring, definiert werden. Wird dies nicht beachtet, kann es zu Laufzeitfehlern kommen, da die Kopierfunktion auf einen nicht definierten Speicherbereich zugreift.

Beispiel 5.4
Ein- und Ausgabe einer Namensliste [Wil95, S. 337].

1. Programm

```
#include <stdio.h>

int main() {
  char names[10][20];
  int i=0;

  printf("Das Programm liest bis zu zehn Namen in eine ");
  printf("Liste ein und gibt sie wieder aus.\n");
  printf("Ende mit \"#\".\n\n");

  /* Eingabeschleife */
  do {
    printf("Name: ");
    gets(names[i++]); /* i nach Gebrauch inkrementieren */
  } while((i<10) && (names[i-1][0]!='#'));

  printf("\nDie Liste umfaßt folgende Namen:\n");
  i = 0;
  /* Ausgabeschleife */
  while((i<10) && (names[i][0]!='#'))
    puts(names[i++]);
  return 0;
}
```

2. Ergebnis

```
Das Programm liest bis zu zehn Namen in eine Liste ein
und gibt sie wieder aus.
Ende mit "#".

Name: Paul
Name: Georg
Name: Katrin
Name: Gudrun
Name: Janett
Name: #

Die Liste umfaßt folgende Namen:
Paul
Georg
Gudrun
Katrin
Janett
```

Beispiel 5.5
Lexikographischer Vergleich zweier Strings [Wil95, S. 349].

1. Programm

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

int main() {
  char first[81];
  char second[81];

  printf("Das Programm vergleicht zwei Strings");
  printf("lexikographisch.\n");
  printf("1. String: ");
  gets(first);
  printf("2. String: ");
  gets(second);

  /* Ergebnis auswerten */
  if(strcmp(first, second) > 0)
    printf("\nString 1 ist größer als String 2.\n");
  else if(strcmp(first, second) < 0)
    printf("\nString 1 ist kleiner als String 2.\n");
  else
    printf("\nString 1 ist gleich String 2.\n");
  return 0;
}
```

2. Ergebnis

```
Das Programm vergleicht zwei Strings lexikographisch.
```

```
1. String: Jahrhundertereignis
2. String: Grundlagenvorlesung

String 1 ist größer als String 2.

Das Programm vergleicht zwei Strings lexikographisch.

1. String: Sortieren
2. String: SORTIEREN

String 1 ist größer als String 2.
```

5.2 Strukturen

Unter Struktur wird die Zusammenfassung verschiedener Einzeldaten (Komponenten) verschiedenen Typs zu einer Gesamtstruktur verstanden. Die Definition erfolgt in folgender Syntax:

Syntax (struct):

```
struct strukturname {
  typ1 komp_name_1;
  typ2 komp_name_2;
    ⋮
  typn komp_name_n;
} [variablenname1, variablenname2, ...];
```

oder

Syntax (struct):

```
typedef struct {
  typ1 komp_name_1;
  typ2 komp_name_2;
    ⋮
  typn komp_name_n;
} strukturname;
```

Um Strukturen in Programmen zu verwenden, müssen sie definiert und deklariert werden:

- Definition von Strukturen bedeutet das Anlegen einer Strukturbeschreibung.
- Deklaration von Strukturen bedeutet die Reservierung von Speicherplatz.

Wir wollen ein Datum in der Zusammensetzung Tag (**int**), Monat (Zeichenkette) und Jahr (**int**) verwenden. Es bieten sich diese Möglichkeiten an:

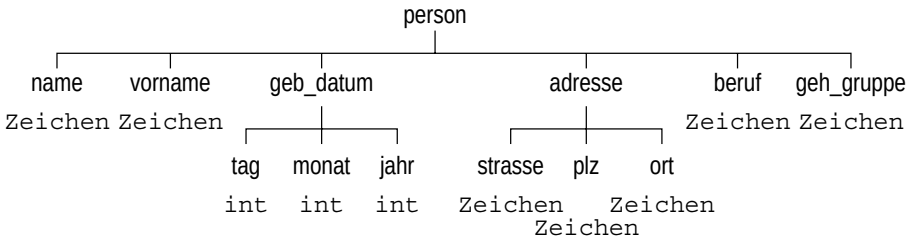


Abbildung 5.3: Darstellung der Struktur person

```
struct datum {
  int tag;
  char monat[10];
  int jahr;
};
struct datum termin;
```

```
struct datum {
  int tag;
  char monat[10];
  int jahr;
} termin;
```

```
typedef struct {
  int tag;
  char monat[10];
  int jahr;
} datum;
datum termin;
```

Natürlich kann man auch Strukturen aufbauen, die wiederum strukturierte Komponenten beinhalten. Solch eine Struktur kann man sich beispielsweise vorstellen, um den Personalbestand einer Firma zu verwalten (vgl. auch Abbildung 5.3):

```
struct person {
  char name[20];
  char vorname[20];
  struct datum geb_datum;
  struct adresse wohnort;
  char beruf[20];
  char geh_gruppe[5];
};

person studentin;
person angestellte;
```

Zu beachten ist, daß die Datentypen der beiden Komponenten datum und adresse vorher zu definieren sind.

5.2.1 Operationen auf Strukturvariablen

Der Zugriff auf Strukturvariablen kann ganzheitlich oder nur über Komponenten erfolgen. Soll Letzteres geschehen, so wird der *Punktoperator* verwendet:

studentin.name	greift auf den Namen der Variable Studentin zu.
studentin.geb_datum.jahr	liefert uns das Geburtsjahr der Studentin.

Zuweisungen erfolgen dann entsprechend dem Datentyp der Komponente:

```
strcpy(angestellte.adresse.strasse, "Reuter-Platz");
```

Aber auch komplette Strukturzuweisungen sind erlaubt. Wird eine Studentin zur Angestellten, so könnte man zunächst eine Anfangsbelegung mit

```
strcpy(angestellte, studentin);
```

vornehmen. Sollen Zuweisungen zwischen Strukturkomponenten vorgenommen werden, so ist in jedem Fall auf die Typverträglichkeit zu achten.

Adreß- und Größenoperationen

Vergleichsoperationen auf die Strukturvariable als Ganzes sind ausgeschlossen, nur auf Komponenten können solche Operationen ausgeführt werden. Man kann sich jedoch mit & die Adresse und mit **sizeof** die Größe der Strukturvariablen anzeigen lassen:

```
printf("%u%d", &studentin, sizeof(studentin));
```

Initialisierung

Natürlich kann man Strukturvariablen komponentenweise einlesen. Genau wie bei Feldern ist jedoch auch eine Anfangsbelegung möglich. Nichtbesetzte Komponenten werden mit 0 initialisiert. Beispiel:

```
person angestellte = {"Meier", "Renate", 01, 12, 1975,
                      "Reuter-Platz", "39106", "Magdeburg",
                      "Ingenieur", "IIa"};
```

Wollen wir eine Studentenkartei aufbauen, so bietet sich ein Feld mit strukturierten Feldelementen an. Die neu immatrikulierten Studenten an der Universität könnten dann in der Variable:

```
struct person neue_studenten[1000];
```

verwaltet werden.

Ergänzend sei angeführt, daß die Sprache C noch 2 spezielle Strukturtypen anbietet. Der Typ **union** enthält ebenfalls Komponenten, wobei diese sich den Speicherplatz teilen. Der Gesamtspeicherplatz entspricht dem der größten Komponente, z.B.:

```
union triple {
  float first;
  short second;
  char third;
} drei;
```

Die Variable drei belegt auf grund der **float**-Komponente maximal 4 Byte. Man muß sich natürlich im klaren darüber sein, daß die Komponenten in der Anwendung dann ständig überschrieben werden.

Oftmals werden mit einer Variablen nur die Zustände 0 oder 1 ausgedrückt, d.h. z.B. ein Mitarbeiter spricht Englisch (Zustand 1) oder nicht (Zustand 0). Dieser Sachverhalt könnte mit dieser Struktur(Bitfeld) beschrieben werden:

```
struct sprachen {
  unsigned int englisch:1;
  unsigned int russisch:1;
  unsigned int franz:1;
}
```

Definieren wir eine Strukturvariable mit:

```
struct sprachen mitarbeiter[2000];
```

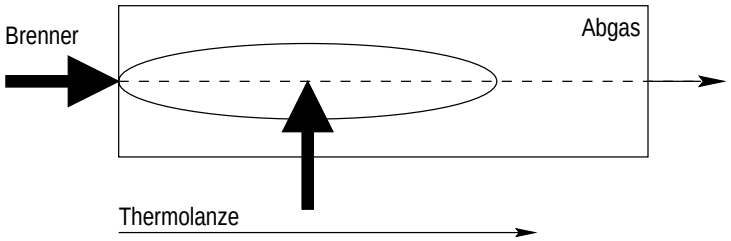
so können für den Mitarbeiter 400, der Englisch und Russisch spricht, diese Sprachkenntnisse festgehalten werden:

```
mitarbeiter[400].englisch=1;
mitarbeiter[400].russisch=1;
mitarbeiter[400].franz=0;
```

Der benutzte Typ ist ein Bitfeld. Hiermit wird Speicherplatz gespart.

Beispiel 5.6
Während eines technischen Versuches fällt eine Meßreihe für den Parameter Temperatur in einer Brennkammer an. Die Meßreihe setzt sich aus 100 Meßpunkten zusammen, die folgende Struktur aufweisen:

Nummer des Meßpunktes:	*1-100*
Name des Parameters:	*Temperatur*
Meßort:	*10-1000 cm in Abständen von 10 cm*
Zeitpunkt der Messung:	*3-300 min im Abstand von 3 min*
Meßwert:	*<1500 °C*



1. Programm

```
#include <stdio.h>

struct tabellenkopf {
  int nummer;
  char parameter[11];
  int ort;
  int zeit;
  float wert;
};

int main() {
  struct tabellenkopf messreihe[100];
  int i, n;
  float minimum, maximum, mittel;

  printf("Einlesen der Tabelle\n");
  printf(" Anzahl der Versuche n = ");
  scanf("%i", &n);

  for(i=0;i<=n-1;i++) {
    printf ("\nNummer\t\t");
    scanf ("%i", &messreihe [i].nummer);
    printf ("Parameter\t");
    scanf ("%s" , messreihe[i].parameter);
    printf ("Ort\t\t");
    scanf ("%i", &messreihe[i].ort);
    printf ("Zeit\t\t");
    scanf ("%i", &messreihe[i].zeit);
    printf ("Wert\t\t");
    scanf ("%f", &messreihe[i].wert);
  }
```

```
  printf("\nMeßreihenuntersuchung nach Maximum, Minimum ");
  printf("und Mittelwert\n");
  minimum=messreihe[0].wert;    /* Initialisierung */
  maximum=messreihe[0].wert;
  mittel=messreihe[0].wert;

  for(i=1;i<=n-1;i++) {
    if(messreihe[i].wert > maximum)
      maximum = messreihe[i].wert;
    if(messreihe[i].wert < minimum)
      minimum = messreihe[i].wert;
    mittel = mittel+messreihe[i].wert;
  }
  mittel=mittel/n;

  /* Ergebnisse darstellen */
  printf("\n\tErgebnistabelle\n");
  printf("\tNummer\tParameter\tOrt\tZeit\tWert\n\n");
  for(i=0;i<=n-1;i++) {
    printf ("\t%i\t%s\t\t%i\t%i\t%f\n",
            messreihe[i].nummer,
            messreihe[i].parameter,
            messreihe[i].ort,messreihe[i].zeit,
            messreihe[i].wert);
  }
  printf ("\nMaximum    =\t%f\n",maximum);
  printf ("Mittelwert =\t%f\n", mittel);
  printf ("Minimum    =\t%f\n",minimum);
  return 0;
}
```

2. Ergebnis

```
Messreihenuntersuchung nach Maximum, Minimum und Mittelwert

        Ergebnistabelle
        Nummer  Parameter       Ort     Zeit    Wert

        1       Temperatur      10      3       500.000000
        2       Temperatur      20      6       1000.000000
        3       Temperatur      30      9       1500.000000
Maximum    =    1500.000000
Mittelwert =    1000.000000
Minimum    =    500.000000
```

5.3 Selbstdefinierte Datentypen

Neben den durch die Sprache vorgegebenen Datentypen können mit Hilfe der Schlüsselworte **typedef** und **enum** eigene Typen definiert werden. Während jedoch die Anwendung von

typedef nur einen neuen Namen für einen bereits festliegende Bezeichner der bekannten Datentypen liefert, kann mit **enum** ein echter neuer Datentyp – der Aufzählungstyp kreiert werden.

Syntax (typedef):

```
typedef datentyp ersatzname_1[, ersatzname_2 ...];
```

Beispiel:

```
typedef struct {
  int  alter;
  char name[12];
  char beruf[10];
  char geschlecht;
} person;

person mitarbeiter[100];
```

Als Aliasname wurde person gewählt, der dann den Typ der Feldelemente in mitarbeiter bestimmt. Anders verhält es sich mit dem Aufzählungstyp.

Syntax (enum):

```
enum name_des_aufzählungstyps {
  name_1, name_2, ... , name_n
} aufzählungsvariable;
```

Beispiel:

```
enum erdteile {
  europa, amerika, asien, afrika, australien
} kontinent;
```

Auf die Variable kontinent wird jetzt praktisch über eine Ordnungsnummer zugegriffen, da der Compiler jedem Aufzählungselement, beginnend mit 0, eine Ordnungszahl zuweist. Rechnerintern wird eine solche Aufzählungsvariable wie eine **int**-Variable behandelt. Sie hat den gleichen Speicherbedarf und es können auch vergleichs- und arithmetische Operationen durchgeführt werden, z.B.:

```
for (kontinent=europa; kontinent<=australien; kontinent++) {
  printf("Gib Bevölkerungszahl ein\n");
  scanf ("%i", Zahl[kontinent]);
}
```

5.4 Speicherklassen

Bis auf den Zeigertyp sind uns jetzt alle Datentypen der Programmiersprache C bekannt. An dieser Stelle sollen einige Bemerkungen über die Art der Speicherung von C-Variablen gemacht werden. Einflußparameter sind der Definitionsort im Programm und die zugeordnete Speicherklasse. Zunächst unterscheiden wir in lokale und globale Variablen. Lokale Variablen werden in Funktionen definiert und sind auch nur dort gültig. Globale Variablen werden außerhalb von Funktionen, auch außerhalb main, definiert und besitzen ihre Gültigkeit im gesamten Programm ab dem Definitionsort. Jede Variable besitzt genau eine der folgenden Speicherklassen:

- **extern**,

- **static**,
- **auto**,
- **register**.

Während für lokale Variablen alle Speicherklassen möglich sind, kommen für globale Variablen nur die ersten beiden in Frage. Die Speicherklasse **auto** sichert die Speicherung der Variablen für den Zeitpunkt ab, wenn die betreffende Funktion gerade in Bearbeitung ist. Vor und nach dieser Bearbeitung ist der Speicherplatz wieder frei. **auto** ist Standard und wird in der Variablendefinition meist nicht mitgeschrieben. Die Speicherklasse **static** ist für lokale Variablen vorgesehen, um die Belegung des Speichers über die Lebensdauer der lokalen Funktion zu erhalten.

Beispiel 5.7
Überprüfen Sie an folgendem Beispiel die Wirkung der **static** *Speicherklassendefinition.*

```
#include <stdio.h>

int main() {
  int i;

  for(i=0;i<5;i++) {
    int a = 0;
    printf("Wert von (auto) während der %d. Ausführung des ", i+1);
    printf("Blocks:\t%d\n", a);
    a++;
  } /* Ergebnis für a ist 0,0,0,0,0 */

  printf("\n");

  /* im Gegensatz zu */
  for(i=0;i<5;i++) {
    static int a=0;
    printf("Wert von (static) während der %d.",i+1);
    printf("Ausführung des Blocks:\t%d\n", a);
    a++;
  } /* Ergebnis für a ist 0,1,2,3,4 */
  return 0;
}
```

Wenn die Zugriffszeit auf eine Variable verkürzt werden soll, so wird sie im Register, also im Speicherbereich des Prozessors gehalten. Anstelle von **auto** wird **register** gesetzt. Die Speicherklasse **extern** sichert den Zugriff globaler Variablen aus allen Programmteilen, die nach der Deklaration stehen. Eine Deklaration vor der main Funktion bekommt standardmäßig die Speicherklasse **extern** zugewiesen:

```
#include <stdio.h>

int e=1;

main() {
```

```
  ⋮
}
```

Baut man ein Programm modular aus verschiedenen Quelldateien auf, so ist es sinnvoll, globale Variablen extern anzukündigen.

Beispiel 5.8
Programm aus zwei Modulen [Wil95, S. 434].

1. **Programm** Folgende Anweisungen stehen in der Datei `extern.c`:

```c
#include <stdio.h>

/* Definition der externen Variablen e.
 * Gültigkeitsbereich: Datei extern.c und aufgrund der
 * dortigen Deklaration auch der Programm–Modul funcs.c
 */

int e = 1;

/* Deklaration der Funktionen aus extern.c */
void func1();
void func2();

int main() {
  printf("main:\tWert von e ist %d\n", e);
  func1();      /* Aufruf func1 */
  func2();      /* Aufruf func2 */
  return 0;
}
```

Folgende Anweisungen stehen in der Datei `funcs.c`:

```c
/* funcs.c: Funktionsmodul zu extern.c */

#include <stdio.h>

/* Globale Deklaration der Variablen e.
 * Die Variable ist ab dieser Deklaration
 * in funcs.c verwendbar
 */
extern int e;

void func1() {
  printf("func1:\tWert von e ist %d\n", e);
}

void func2() {
  printf("func2:\tWert von e ist %d\n", e);
}
```

2. **Ergebnis**

```
main:   Wert von e ist 1
func1:  Wert von e ist 1
func2:  Wert von e ist 1
```

	Speicherklasse	Gültigkeitsbereich	Lebensdauer
lokal	**auto**	Block	Laufzeit des Blocks
	register	Block	Laufzeit des Blocks
	static	Block	Laufzeit des Blocks
global	**extern**	Quelldatei mit Definition und je nach Deklaration alle Module des Programms	Laufzeit des Programms
	static	Quelldatei mit Definition	Laufzeit des Programms

Tabelle 5.2: Zusammenfassende Übersicht der Speicherklassen

Wird eine globale Variable mit **static** definiert, so gilt diese auch nur für den entsprechenden Block.

Ein C-Programm belegt im Arbeitspeicher des Rechners mehrere Teilbereiche. Das *Codesegment* enthält Programmcode in ausführbarer Form. Das *Datensegment* enthält Variablen der Speicherklassen **extern** und **static**. Der *Stack* (Keller) enthält Variablen der Speicherklassen **auto**. *Heap* (Halde) ist ein freier Speicher für dynamische Variablen.

6 Funktionen

6.1 Grundsätzliches

Problemlösungsprozesse sind durch Kreativität, Individualität und manchmal auch durch Intuition geprägt. Es versteht sich, daß komplexere Problemlösungen nicht an einem Stück und nicht mit einem Versuch geschaffen werden. Im Gegenteil, liegt zunächst eine grobe Vorstellung zur Lösung vor, so wird man einen Top-down Entwurf, d.h. vom Groben zum Detail, durchführen. Andererseits hat man vielleicht schon viele Detaillösungen in Bibliotheken abgelegt, die dann in einem Bottom-up Verfahren, d.h. vom Einzelnen zum Ganzen, eingebunden werden können. Die Kombination beider Ansätze trifft wahrscheinlich die Praxis. In dieser modulartigen Vorgehensweise spielen Funktionen eine entscheidende Rolle. Sie stellen Lösungen in abgeschlossenen Einheiten zur Verfügung, die zu jeder Zeit an jedem Ort in einem Programm oder einem Programmteil nutzbar sind. In einem C-Programm ist eine Funktion ein Programmteil, der aus einer oder mehreren Anweisungen besteht. Diese Funktionen können vordefiniert werden, indem sie in Bibliotheken liegen, um zu gegebener Zeit in das Programm eingelinkt zu werden (printf, scanf, clrscr, ...). Aber auch in dem individuellen Quellprogamm können Teile als wiederholbare Funktionen definiert und deklariert werden. Viele Sprachen unterscheiden dabei noch in Prozeduren und Funktionen (z. B. Pascal). Die Sprache C kennt nur Funktionen.

Eine Funktionsdefinition beinhaltet:

- Speicherklasse der Funktion,
- Ergebniswert der Funktion mit Angabe des Datentyps,
- Name der Funktion,
- Parameter, die an die Funktion übergeben werden mit Namen und Datentyp,
- lokale und externe Variablen, die die Funktion nutzt,
- andere Funktionen, die von der Funktion aufgerufen werden,
- die Anweisungen, die die Funktion ausführen soll.

6.2 Definition und Deklaration

Syntax (Funktionsdefinition):

```
[speicherklasse] [typ] name([d1 n1, ..., dn nn]) { /* Funktionskopf */
  [Definition lokaler Variablen]                   /* Funktionsrumpf */
  [Definition externer Variablen]
  [Deklaration weiterer Funktionen]
  [Anweisungen]
}
```

Bemerkung: In der älteren Form der Syntax werden die Parameter in der Funktion explizit angegeben. In eckigen Klammern angegebene Bestandteile sind optional. Der Funktionskopf enthält die sogenannte Schnittstelle der Funktion. Die Parameterliste versorgt die Funktion mit Daten, die in verarbeiteter Form mit der **return**-Anweisung typentsprechend zurückgegeben werden. Die Speicherklasse ist entweder **extern** (wobei diese Speicherklasse immer angenommen wird, wenn sie nicht ausdrücklich angeführt ist) oder **static**. Im letzteren Falle gilt die Funktion nur in dem Modul, wo sie definiert ist. Da sich extern verarbeitete Funktionswerte wie globale Variablen verhalten, dürfen Funktionen keine weiteren Funktionsdefinitionen enthalten.

Beispiel 6.1
Folgende Funktion führt eine Quadrierung aus und gibt den Rückgabewert unter dem Namen quadrat an die aufrufende Funktion (main oder eine andere Funktion) zurück.

```
double quadrat(double x) {
  return x*x;
}
```

Als Datentypen für Ergebniswerte sind einfache Datentypen (**char**, **short**, **int**, **long**, **float**, **double**, **long double**), Strukturen, Unions, sowie Zeiger auf beliebige, auch zusammengesetzte Datentypen erlaubt. Falls kein Ergebnistyp vereinbart ist, nimmt der Compiler **int** als Datentyp an.

Parameter beschreiben die Übergabe der Daten von der aufrufenden Funktion aus. In der Funktionsdefinition nennen wir diese Parameter formale Parameter. Beim Aufruf heißen dieselben dann aktuelle Parameter. In der Funktion quadrat ist x ein formaler Parameter. Beim Aufruf der Funktion in der Anweisung

```
Kreis = PI*quadrat(r);
```

ist die Variable r der aktuelle Parameter. Es gilt die Regel, daß formale und aktuelle Parameter in Anzahl, Typ und Reihenfolge übereinstimmen müssen. Werden an die Funktion keine Parameter übergeben, so steht in der Definition das Schlüsselwort `void` zwischen den runden Klammern, z.B.:

```
double quadrat(void);
```

Die **return**-Anweisung beendet die Funktion und gibt die Steuerung an den aufrufenden Programmteil zurück. Auch main kann einen Ergebniswert an das Betriebssystem zurückgeben. Eine Funktion ohne Ergebniswert wird ebenfalls mit **void** definiert:

```
void funk1();
```

Da Funktionen an verschiedenen Orten des Programms zu verschiedener Laufzeit ausgeführt werden sollen, ist es von Bedeutung, an welcher Stelle die Funktion deklariert, d.h. bekanntgemacht wird. Auf jeden Fall sollte man sichern, daß Funktionen dem Compiler vor ihrem ersten Aufruf erklärt werden. Ansonsten führen die Konvertierungsversuche des Compilers evtl. zu unerwarteten Ergebnissen. Funktionen können lokal, global, in verschiedenen

Modulen und in include-Dateien deklariert werden. Die Syntax einer Funktionsdeklaration unterscheidet sich vom Funktionskopf nur durch das abschließende Semikolon. Eine solche Deklaration wird auch als *Prototyp* der Funktion bezeichnet. Der Prototyp der Funktion **double** aus dem Beispiel lautet:

```
double quadrat(double x);
```

Beispiel 6.2
Berechnung der Fläche eines Kreises.

1. Programm

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

#define PI 3.14159265358979323846

int main() {
  double quadrat(double x); /* Prototyp für quadrat */
  double r, kreis;
  printf("Radius eingeben\n");
  scanf("%lf", &r);
  kreis = PI*quadrat(r);    /* Aufruf der Funktion */
  printf ("\nDie Fläche des Kreises beträgt: %f\n", kreis);
  return 0;
}

double quadrat(double x) {
  return x*x;
}
```

6.2.1 Lokale Funktionsdeklaration

Im obigen Beispiel haben wir die Funktion `quadrat` lokal in `main` deklariert. Damit ist sie auch nur dort lokal verfügbar. Man kann Funktionen aber auch in anderen Funktionen deklarieren und somit bestimmte Verbindungen gezielt aufbauen, z.B.:

```
int main() {
  Deklaration func1;
  ⋮
  Aufruf func1;
}

func1() {
  Deklaration func2;
  ⋮
  Aufruf func2;
}

func2() {
  ⋮
}
```

Sollen jedoch die Funktionen func1 und func2 in allen nachfolgend aufrufenden Funktionen verfügbar sein, so muß ihre Deklaration global erfolgen:

```
#include <...>
#define ...

Deklaration func1;
Deklaration func2;

int main() {
  ⋮
}

func1() {
  ⋮
}

func2() {
  ⋮
}
```

6.3 Parameterübergabe an Funktionen

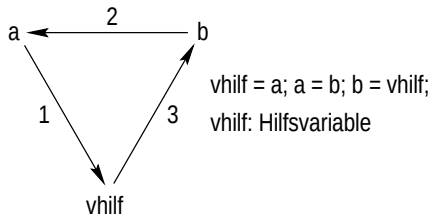
Ohne Speicherklassenangabe sind alle Funktionen global definiert. Alle innerhalb des Funktionsblocks definierten Variablen sind lokal. Funktionen können einfache und/oder zusammengesetzte Variablen und/oder Funktionen als Eingabeparameter enthalten. Die Übergabe erfolgt entweder als *Wertekopie (call by value)* oder als *Adreßzuweisung (call by reference)*.

6.3.1 Parameterübergabe durch call by value

Einfache Variablen als Parameter werden mit dem Prinzip *call by value* übergeben. Dabei wird von den Parametern eine Kopie angelegt, auf die die Funktion dann lesend und schreibend zugreifen kann. Der Wert des Originals wird dabei aber nicht verändert. Das hat den Vorteil, daß die Parameter nicht verändert werden können, aber auch den Nachteil, daß die Rückgabe des Ergebnisses nur über die **return**-Anweisung erfolgen kann. Bei der Verwendung von Strukturen als Parameter hat man je nach Zielstellung die Wahl der Übergabe als unveränderliche Kopie oder als Adresse der Struktur.

Beispiel 6.3
Variablentausch mit Funktion durch call by value *[Het93, S. 22]. Wir wollen die Wirkungsweise der Parameterübergabe am Beispiel des Algorithmus des Vertauschens zweier Variablen erklären, d.h., zwei Variablen werden mit Werten belegt. Anschließend soll die Funktion den Tausch vornehmen und die Ergebnisse in der main-Funktion ausgegeben.*

1. Algorithmus



2. Programm

```
#include <stdio.h>

void ztausch(int, int);  /* Prototyp für ztausch */

int main() {
  int zahl1, zahl2;

  zahl1=10;
  zahl2=20;
  printf("Ausgangswerte: %2i\t%2i\n", zahl1, zahl2);
  ztausch(zahl1, zahl2); /* Funktionsaufruf */
  printf("Ergebniswerte: %2i\t%2i\n", zahl1, zahl2);
  return 0;
}

void ztausch(int a, int b) {
  int vhilf;
  vhilf = a;
  a = b;
  b = vhilf;
}
```

3. Ergebnis

```
Ausgangswerte: 10      20
Ergebniswerte: 10      20
```

Aus dem Ergebnis kann man vermuten, daß zwar ein Tausch vorgenommen wurde, das Ergebnis aber der main-Funktion nicht übermittelt wurde. Der Grund ist die Übergabe der Parameter als Wert. Dadurch arbeitet die Funktion nur mit Kopien der Originalwerte und nicht mit den in der main-Funktion verwendeten. Nach der Rückkehr gehen die auf den Kopien durch den Tausch erfolgten Änderungen verloren. Dieses Problem läßt sich auch nicht durch die Verwendung von Rückgabewerten lösen, da immer nur ein Wert zurückgegeben werden kann. Im Beispielfalle müßten aber zwei Zahlen zurückgegeben werden. Diesen Mangel kann u.a. auf folgende Arten behoben werden:

1. Vereinbarung der Variablen als globale Variablen,
2. Verwendung von Strukturen,
3. Übergabe der Variablen nach dem Prinzip *call by reference*.

Beispiel 6.4
Variablentausch mit Funktion unter Verwendung globaler Variablen [Het93, S. 24].

1. Programm

```
#include <stdio.h>

void ztausch(); /* Prototyp für ztausch */
int zahl1, zahl2;

int main() {
  zahl1=10;
  zahl2=20;
  printf("Ausgangswerte: %2i\t%2i\n", zahl1, zahl2);
  ztausch();    /* Funktionsaufruf */
  printf("Ergebniswerte: %2i\t%2i\n", zahl1, zahl2);
  return 0;
}

void ztausch() {
  int vhilf;
  vhilf = zahl1;
  zahl1 = zahl2;
  zahl2 = vhilf;
}
```

2. Ergebnis

```
Ausgangswerte: 10       20
Ergebniswerte: 20       10
```

6.3.2 Parameterübergabe durch call by reference

Bei dieser Art der Übergabe werden keine Werte sondern Adressen der Datenobjekte, mit denen man direkt arbeiten will, übermittelt. Es wird somit keine Kopie angelegt und die Datenobjekte können auch schreibend verändert werden. Diese Methode kann bei einfachen Datentypen, Feldern, Strukturen, Vereinigungen und Funktionen angewandt werden.

Einfache Variablen als Parameter

Wir wollen zunächst die Übergabe der Adressen einfacher Variablen besprechen. Dazu ist es notwendig, einen Vorgriff auf das nächste Kapitel Zeiger zu tun. Eine Variable, die die Adresse einer **int**-Variablen a verwaltet, wird mit

```
int *a;
```

deklariert. Der Stern (∗) bedeutet Zeiger und könnte so interpretiert, daß die Zeigervariable a eine Adresse enthält, unter der der Wert der Variablen ∗a gespeichert wird.

Beispiel 6.5
Variablentausch mittels Funktion und Parameterübergabe mit call by reference *[Het93, S. 24].*

1. Programm

```
#include <stdio.h>

void ztausch(int*, int*); /* Prototyp für ztausch */

int main() {
  int zahl1, zahl2;

  zahl1=10;
  zahl2=20;
  printf("Ausgangswerte: %2i\t%2i\n", zahl1, zahl2);
  /* Funktionsaufruf mit Übergabe der Adressen */
  ztausch(&zahl1, &zahl2);
  printf("Ergebniswerte: %2i\t%2i\n", zahl1, zahl2);
  return 0;
}

void ztausch(int* a, int* b) {
  int vhilf;
  vhilf = *a;
  *a = *b;
  *b = vhilf;
}
```

2. Ergebnis

```
Ausgangswerte: 10       20
Ergebniswerte: 20       10
```

Arrays als Parameter

Felder werden in Funktionen als Formalparameter generell als Zeiger vereinbart. Somit gibt es zwei alternative Schreibweisen:

```
datentyp *name;
```

oder

```
datentyp name[ ];
```

Beispiel 6.6
Sortieren eines Feldes. Wir wollen ein Feld mit Zufallszahlen des Typs **short** *füllen, anschließend in aufsteigender Reihenfolge sortieren und ausgeben.*

1. **Algorithmus** Der Algorithmus besteht in der Suche des jeweils kleinsten Elementes und dem anschließenden Tausch an seine endgültige Position. Dieser Algorithmus arbeitet mit zwei Schleifen. Die äußere durchläuft alle Elemente des zu sortierenden Feldes. In

der inneren Schleife wird dann in den folgenden Elementen ein kleineres gesucht. Nach Abschluß der Suche wird dann das kleinste Element nach „vorn“ kopiert. Mit jedem Durchlauf der äußeren Schleife gelangt somit ein Element an seine endgültige Position, nach dem 1. Durchlauf das mit dem Index 0, im 2. Durchlauf das mit dem Index 1, usw. Dieser Algorithmus wird als *Selectionsort* bezeichner, da jeweils das kleinste Element *selektiert* und dann an seinen Platz getauscht wird. Der Tausch selbst erfolgt nach dem bekannten Algorithmus. Die folgende Tabelle verdeutlicht diesen Algorithmus.

Feldelement	Vergleich mit arr[0]					Vergleich mit arr[1]				Vergleich mit arr[2]		
arr[k]	Vergleich				Tausch	Vergleich			Tausch	Vergleich		Tausch
arr[0]	**7**	**7**	**7**	**7**	1	1	1	1	1	1	1	1
arr[1]	9	**9**	9	9	9	**9**	**9**	**9**	3	3	3	3
arr[2]	3	3	**3**	3	3	3	**3**	3	9	**9**	**9**	7
arr[3]	1	1	1	**1**	7	7	7	**7**	7	7	**7**	9
arr[min]	7	7	3	1		9	3	3		9	7	

2. Programm

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

void ssort(int *arr, int elements);

int main() {
  int n[10], i;

  printf("Das Programm hat die folgenden Zufallszahlen ausgewählt:\n");
  srand(time(NULL));
  for (i = 0; i < 10; i++) { /* Zufallszahlen einlesen */
    n[i] = rand() % 1000;
    printf("%4d ", n[i]);
  }

  ssort(n, 10);              /* Feld sortieren: call by reference */

  printf("\n\nDie Zahlen in sortierter Reihenfolge:\n");
  for (i = 0; i < 10; i++)   /* Array ausgeben */
    printf("%4d ", n[i]);
  printf("\n");
  return 0;
}

/* Sortierfunktion */
void ssort(int arr[], int elements) {
  void swapint(int *x, int *y);
  int i, k, min;

  for (i = 0; i < elements-1; i++) {
    min = i;
    for (k = i+1; k < elements; k++) /* finde kleinstes Element */
      if(arr[min] > arr[k])
```

```
        min = k;
      swapint(&arr[i], &arr[min]);      /* Tausch */
    }
  }

  /* Tauschfunktion */
  void swapint(int *x, int *y) {
    int buffer;
    buffer = *x;
    *x = *y;
    *y = buffer;
  }
```

3. Ergebnis

```
Das Programm hat die folgenden Zufallszahlen ausgewählt:
 332  200  789  863  507  713  629  627  912  921

Die Zahlen in sortierter Reihenfolge:
 200  332  507  627  629  713  789  863  912  921
```

Wir sehen, dadurch daß die Ausgabe der Zufallszahlen in main vereinbart wurde, muß die Übergabe von ssort erfolgreich gewesen sein.

Auch mehrdimensionale Felder können als *call by reference* Parameter übergeben werden. Bei der Vereinbarung des entsprechenden Formalparameters müssen außer der ersten Dimension alle weiteren explizit angegeben werden, da diese zur Berechnung der Speicherposition benötigt werden, z.B.:

```
void fuellenmatrix(int m[][4][4]);
```

Eine Möglichkeit, mehrdimensionale Felder unbekannter Dimension an Funktionen zu übergeben, bieten Zeiger (vgl. Kapitel 7). Darauf soll aber im weiteren nicht eingegangen werden.

Strukturen als Parameter

Wie bereits angeführt können Strukturen by value oder by reference übergeben werden. Die erstere Variante ist zeitaufwendiger und verliert somit an Bedeutung. Es soll als Beispiel die Funktion der Ausgabe einer Struktur für eine Buchverwaltung dienen.

```
void writestruktur(struct buch *b); /* Deklaration by reference */
void writestruktur(struct buch b);  /* Deklaration by value     */
writestruktur (&informatikbuch);    /* Aufruf über Adresse      */
writestruktur (informatikbuch);     /* Aufruf über Datenobjekt  */
```

Funktionen als Parameter

In klassischen Programmiersprachen werden Unterprogramme in der Regel auf drei verschiedene Arten genutzt:

- durch Aufruf aus einem Hauptprogramm heraus,
- durch Aufruf aus einem Unterprogramm heraus,
- durch Aufruf eines Unterprogrammes als Parameter eines anderen Unterprogrammes.

Die ersten beiden Fälle kennt die Programmiersprache C durch Aufruf aus main oder aus func heraus. Die dritte Variante verlangt gewisse Vorkehrungen. Wir haben kennengelernt, daß Funktionen in Ausdrücken oder in Anweisungen Anwendung finden. Nun liegt es nahe, daß Funktionswerte auch als Parameter wiederum in Funktionen eingehen sollen. Dies geht deshalb, weil auch Funktionen eine Adresse haben, unter welcher sie im Speicher untergebracht sind. Mit einer solchen Adresse, die vom Compiler festgelegt wird, sind dann wieder die entsprechenden Operationen möglich.

Syntax (Definition von Funktionszeigern):

```
datentyp (*zeigername) ([d1 n1,d2 n2, ..., dn nn]);
```

Die Angabe datentyp qualifiziert den Rückgabewert der Funktion, die Klammerangaben die Parameterliste. Ein Zeiger auf ein Funktion, die die größere von 2 Zahlen zurückgibt könnte beispielsweise folgendermaßen definiert werden:

```
int (*vergleich) (int zahl1, int zahl2);
```

Beispiel 6.7
Funktion als Parameter einer anderen Funktion [Het93, S. 29].

1. Programm

```
#include <stdio.h>

int funk1(int);
int funk2(int);

/* Deklaration mit Funktion als Parameter */
int berech(int, int (*)(int));

int main() {
  int w = 2;
  printf("Funktion1 benutzt: %i\n", berech(w, funk1));
  printf("Funktion2 benutzt: %i\n", berech(w, funk2));
  return 0;
}

int berech(int wert, int (*funkp)(int)) {
  return 2*wert+(*funkp)(wert) ; /* Verarbeitung der Funktionsadresse */
}

int funk1(int x) {
  return x*x+10;
}

int funk2(int x) {
  return 5*x+50;
}
```

2. Ergebnis

```
Funktion1 benutzt: 18
Funktion2 benutzt: 64
```

Rekursive Funktionen

Eine weitere Anwendung von Funktionen besteht in einem Selbstaufruf. Man spricht in diesem Fall von einem *rekursiven Funktionsaufruf*. Der Algorithmus einer rekursiven Nutzung einer Funktion muß neben dem Funktionsaufruf auch eine Abbruchbedingung für die Rekursion besitzen. Wir nehmen das Standardbeispiel der rekursiven Berechnung der Fakultät einer Zahl.

Beispiel 6.8
Rekursive Fakultätsberechnung [Het93].

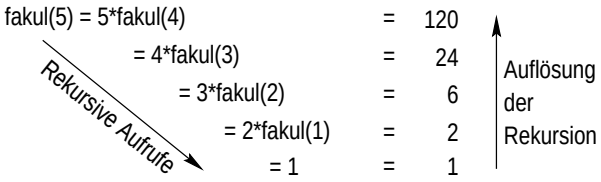
1. Programm

```
#include <stdio.h>

long fakul(int zahl);

int main() {
  int ogr;

  printf("\nFakultaetsberechnung für: ");
  scanf("%d", &ogr);
  printf("\n\n  %d! = %ld\n", ogr, fakul(ogr)); /* Funktionsaufruf */
  return 0;
}

long fakul(int zahl) {
  long ergeb;
  if(zahl>1)
     ergeb = zahl*fakul(zahl-1);
  else
    ergeb = 1;

  return ergeb;
}
```

2. Ergebnis

```
Fakultaetsberechnung fuer: 5

  5! = 120
```

In folgender Abbildung ist dargestellt, wie die Berechnung vor sich geht.



Beispiel 6.9
Es ist ein Programm zu schreiben, das die FIBONACCI-Zahlen bis zu einem bestimmten Grenzwert ausschreibt. Lösung durch Addition und Umspeicherung.

1. **Algorithmus** Eine Zahl der FIBONACCI-Folge ergibt sich aus der Summe seiner beiden Vorgänger: 1 1 2 3 5 8 13 21 ...

2. **Programm**

```
#include <stdio.h>

int main() {
  long grenze, fibo;
  long wert1 = 1;
  long wert2 = 1;

  printf("Grenze eingeben: ");
  scanf("%ld", &grenze);

  printf("1\t1\t"); /* Startzahlen */
  do {
    fibo = wert1 + wert2;
    printf("%ld\t", fibo);
    wert1 = wert2;
    wert2 = fibo;
  } while(fibo < grenze);
  printf("\n");
  return 0;
}
```

3. **Ergebnis**

```
Grenze eingeben: 10000
1       1       2       3       5       8       13      21      34
55      89      144     233     377     610     987     1597    2584
4181    6765    10946
```

Beispiel 6.10
Berechnung der FIBONACCI-Zahlen mittels Rekursion.

1. **Programm**

```
#include <stdio.h>

int fibo(int);
```

```
int main() {
  long grenze, wert, n;
  printf("Grenze eingeben: ");
  scanf("%li", &grenze);
  wert = 1;
  n = 0;
  do {
    wert = fibo(n++);
    printf("%li\t", wert);
  } while(wert < grenze);
  printf("\n");
  return 0;
}

int fibo(int n) {
  int ergebnis;
  if(n < 2)
    ergebnis = 1;
  else
    ergebnis = fibo(n-2) + fibo(n-1);
  return ergebnis;
}
```

2. Ergebnis

```
Grenze eingeben: 10000
1       1       2       3       5       8       13      21      34
55      89      144     233     377     610     987     1597    2584
4181    6765    10946
```

7 Das Zeigerkonzept

Über Adressen kann man in der Programmiersprache C direkt auf den Speicherplatz einer Variablen zugreifen. Das Mittel dazu sind *Zeiger* oder auch *Pointer*. Zeiger enthalten somit die Anfangsadressen von Variablen, d.h., sie verweisen oder zeigen darauf (vgl. Abbildung 7.1).

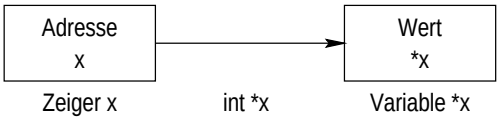


Abbildung 7.1: Skizze zum Zeigerkonzept

In C kann man mit Zeigern:

- direkt mit Adressen rechnen,
- man kann Funktionen mit mehreren Ergebnisparametern organisieren
- und man kann Felder als Zeiger auf das erste Feldelement verstehen.
- Darüber hinaus kann man mit Zeigern dynamische Datenstrukturen aufbauen.

7.1 Definition von Zeigervariablen

Zeiger können auf Datenobjekte beliebigen Typs verweisen. Die Syntax einer Zeigervariable lautet wie folgt:

```
typ_des_datenobjektes* name_der_zeigervariablen;
|<--- Zeigertyp ---->| |<-- Variablenname -->|
```

Beispiele:

```
int* zi; char* zc; short* zs; float* zf;
```

Sehen wir uns die erste Deklaration an. Sie besagt, daß die Variable zi eine Adresse aufnehmen kann, hinter welcher sich ein integer-Datenobjekt *zi befindet. Eine Anweisung

```
printf("%d %X", *zi, zi);
```

gibt eine ganze Dezimalzahl für den Wert des Datenobjektes *zi und eine ganze hexadezimale Zahl für die Adresse zi aus.

7.2 Operationen auf Zeigern

Operationen mit Zeigern sind grundsätzlich zweierlei Art:

1. Zugriff mittels Dereferenzierungsoperator $*$ und Zeiger auf Datenobjekte,
2. Manipulation von Zeigervariablen selbst über die sogenannte Zeigerarithmetik.

Zur Zeigerarithmethik gehören:

Addition einer Konstante n: Bewirkt die Erhöhung der Adresse um n mal der Bytezahl des Typs des Datenobjektes (zs = zs+5;).

Inkrementierung: Zeigt die gleiche Wirkung wie Addition von 1 (zs++;).

Subtraktion einer Konstanten n: Verschiebt den Zeiger um das n-fache der dem Typ entsprechende Anzahl von Speichereinheiten zurück (zs = zs−2;).

Dekrementierung: Bewirkt das Gegenteil der Inkrementierung (zs−−;).

Vergleich: Zeiger können untereinander auf Gleichheit (==) und Ungleichheit (<, >) verglichen werden.

Wertzuweisung der Konstante NULL: Bewirkt ein Zeigen ins Leere (zs = NULL;).

7.3 Anwendung von Zeigern

Wir wollen Zeiger in Verbindung mit verschiedenen Datenobjekten bringen.

7.3.1 Felder und Zeiger

Diese Verbindung trägt zwei Aspekte. Zum einen können Zeiger auf den Feldanfang oder auch auf Feldelemente gesetzt werden, zum anderen kann man Zeiger auch zu Zeigerfeldern zusammenfassen. Oftmals ist es angebracht, Vektoren mittels Zeigern zu durchmustern. Dabei kann man wie folgt vorgehen:

```
int main() {
  int vektor[100];     /* Deklaration eines Vektors  */
  int *zeiger, n;      /* Zeigervereinbarung         */
  ⋮
  zeiger=vektor;       /* Initialisierung von zeiger */
  for(n=0; n<100; n++)
  vektor[n] = n;

  /* oder */

  for(n=0; n<100; n++)
  *zeiger++ = n;
  ⋮
  return 0;
}
```

Die beiden Zählschleifen sind äquivalent und füllen das Feld vektor[100] mit Integer-Werten. Im ersten Fall muß man beachten, daß zunächst die Adresse erhöht wird und anschließend das Datenobjekt mit n gefüllt wird.

7.3.2 Zeichenketten und Zeiger

Zeichenketten sind Vektoren, die Zeichen als Elemente enthalten. Wir wissen bereits, daß wir auf eine Zeichenkette elementweise z.B. mit zk[10] oder als Ganzes mit zk zugreifen können. Im letzten Fall beinhaltet zk einen Zeiger auf das Datenobjekt zk[ ]. Diesen Umstand kann man nutzen, um beispielsweise durch Zuweisung von Zeigern dem Datenobjekt zk[ ] einen neuen Inhalt zu geben, ohne die Funktion strcpy für das Kopieren zu verwenden.

Beispiel 7.1
Verarbeitung von Zeichenketten. In Abbildung 7.2 sind die Operationen verdeutlicht.

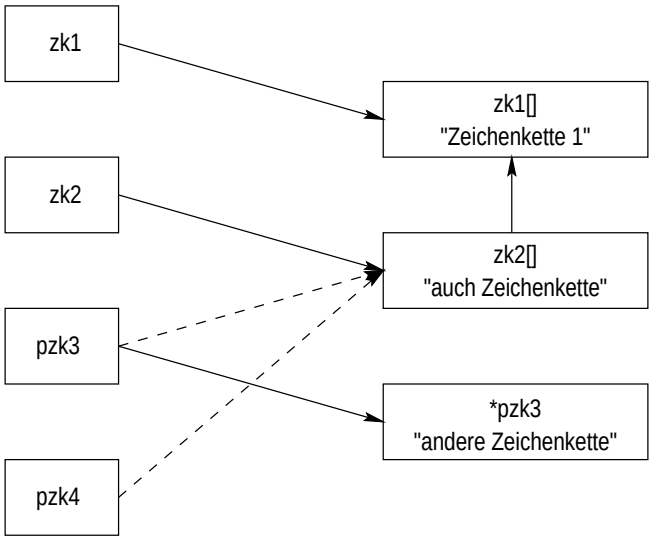


Abbildung 7.2: Grafische Erklärung des Kopierens von Zeichenketten mit strcpy-Funktion und Zeigerzuweisung

```
char zk1[] = "Zeichenkette1    ";
char zk2[] = "auch Zeichenkette";
char *pzk3 = "andere Zeichenkette";
char *pzk4;
/* Kopieren von zk2[] in  zk1[] */
strcpy(zk1, zk2);  /* Überschreiben der Zeichenketten */
/* oder */
pzk4=zk2;  /* Initialisieren des Zeigers pzk4 mit der Adresse von zk2 */
```

Beispiel 7.2
Zeiger und Zeichenketten [Het93].

1. Programm

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

int main() {
  int i;
  char zk1[18] = "Zeichenkette1";
  char zk2[18] = "auch Zeichenkette";
  char *pzk3 = "andere Zeichenkette";
  char *pzk4;

  /* zk1 = zk2;  Zuweisung nicht möglich -> über strcpy realisieren */
  strcpy(zk1, zk2); /* Syntax: char *strcpy(char *str1, char *str2) */

  printf("Ausgabe 1: Zeichenkette 1: %s ab %p\n", zk1, zk1);

  pzk4 = zk2;       /* Adressen werden zugewiesen */
  pzk3 = pzk4;

  printf("Ausgabe 2: Zeichenkette 3: %s ab %p\n", pzk3, pzk3);
  printf("Ausgabe 3: Zeichenkette 4: %s ab %p\n", pzk4, pzk4);

  for(i=0; *pzk3; i++)
    printf("%c\n", *pzk3++);

  return 0;
}
```

2. Ergebnis

```
Ausgabe 1: Zeichenkette 1: auch Zeichenkette ab 0xbffff454
Ausgabe 2: Zeichenkette 3: auch Zeichenkette ab 0xbffff440
Ausgabe 3: Zeichenkette 4: auch Zeichenkette ab 0xbffff440
a
u
c
h

Z
e
i
c
h
e
n
k
e
t
t
e
```

Auch zweidimensionale **char**-Felder können alternativ über eine Vereinbarung als Matrix mit konstanter Zeilenlänge oder eine Vereinbarung als Zeigervektor gehandelt werden. Wir werden sehen, daß sich beide Varianten in der erforderlichen Speicherlänge unterscheiden.

Beispiel 7.3
char-*Feld über ein 2-dim. Feld und Zeigervektor [Het93, Teil 2,S. 12]. Es soll ein Namensfeld mit 6 Zeilen und 14 Spalten mit:*

1. *2-dim.* **char**-*Feld:*
```
static char feld[6][14] = {"Gerd", "Egon", "Juergen", "Karl",
                           "Hans-Dieter", "Hans-Guenther" };
```

2. *Zeigervektor:*
```
char *feld[ ] = {"Gerd", ... };
```

vereinbart und anschließend auf dem Bildschirm angezeigt werden.

1. Programm – static
```
#include <stdio.h>

int main() {
  int i, j;
  /* Vereinbarung static */
  char feld[6][14] = {"Gerd", "Egon", "Juergen", "Karl",
                      "Hans-Dieter", "Hans-Guenther"};

  for(i = 0; i < 6; i++){
    j = 0;
    while(feld[i][j] != '\0') {
      printf("%c", feld[i][j]);
      j++;
    }
    printf("\n");
  }
  return 0;
}
```

2. Programm – Zeiger
```
#include <stdio.h>

int main() {
  char *feld[ ] = {"Gerd", "Egon", "Juergen", "Karl", "Hans-Dieter",
                   "Hans-Guenther"};
  int i, j;

  for(i = 0; i < 6; i++){
    j=0;
    while(*(feld[i]+j) != '\0') {
      printf("%c", *(feld[i]+j));
      j++;
    }
    printf("\n");
```

```
    }
    return 0;
}
```

3. Ergebnis

```
Gerd
Egon
Juergen
Karl
Hans-Dieter
Hans-Guenther
```

7.3.3 Dynamische Arrays

Wir haben soeben gesehen, wie man Speicherplatz sparen kann. Besonders bei großen mehrdimensionalen Feldern kann der Speicherplatzbedarf von Bedeutung sein. Bei der Erklärung des Datentyps array haben wir feststellen müssen, daß die Größe des Feldes von vornherein durch Angabe der Feldgrenzen bestimmt wird. Oftmals ist die Größe des Feldes aber vom Algorithmus abhängig, so daß eine Bereitstellung von Feldspeicherplatz während der Laufzeit wünschenswert wäre. Es müssen demzufolge *dynamische Felder* aufbaubar sein.

Zur Arbeit mit dynamischen Feldern werden Bibliotheksfunktionen (in `stdlib.h`) zur Verfügung gestellt. Mittels der Funktion malloc kann Speicher auf dem *Heap (Halde)* reserviert werden.

Syntax (malloc):

```
zeiger = malloc(groesse_in_bytes);
```

zeiger

ist eine Zeigervariable, die die Anfangsadresse des Blockes angibt, der durch malloc reserviert wird.

```
double *darray;          /* Zeiger auf Feld mit Elementen vom Typ double */
darray = malloc(400);    /* reserviert 400 Bytes, darray enthält die     */
                         /* Anfangsadresse des Blockes                   */
```

Nun kann es passieren, daß der gewünschte Speicherplatz nicht ausreicht. In diesem Fall wird die Adresse NULL auf zeiger zurückgegeben und das Programm bricht ab, wenn nicht eine geeignete Fehlerbehandlung eingebaut wird, z.B.:

```
int i;
double *darray;

darray=malloc(400);
if(darray == NULL)
  printf("Speicherbereich kann nicht allokiert werden.");
else
  for(i=0; i<50; i++)  /* Füllen des Feldes mit den Werten 10 - 500 */
    darray[i] = (double)(10*(i+1));
```

Da die Speicherplatzbeanspruchung compilerabhängig sein kann, ist es angebracht, den Compiler selbst über den Speicherplatzbedarf entscheiden zu lassen. Wollen wir 100 Feldelemente unterbringen, so schreiben wir:

```
darray = malloc(100*sizeof(double));
```

Bei bekannter Elementeanzahl und Objektgröße wird die calloc-Funktion genutzt.

Syntax (calloc):

```
calloc(anzahl, groesse);
```

Die Anweisungen

```
double *darray;
darray = (double*)calloc(50, sizeof(double));
```

reservieren einen Block mit 50*8 Bytes. Die explizite Typangabe (**double***) für den Zeiger vermeidet Irritationen bei unterschiedlicher Verarbeitung der Adressen durch die verschiedenen Compiler. Weiterhin stehen die Funktionen realloc zur Veränderung der Größe des Speicherblockes sowie free zur Freigabe der belegten Speicherplätze zur Verfügung.

Syntax (realloc):

```
realloc(zeiger, groesse);
```

Syntax (free):

```
free(zeiger);
```

7.3.4 Zeiger und Strukturen

Auch auf Datenobjekte strukturierter Art kann man mittels Zeigertechnik zugreifen. Deklariert man beispielsweise die Struktur

```
struct article {
  char name[20];
  int num;
};
```

so kann man mit

```
struct article *zx, x;
```

zwei Variablen anlegen, wobei der Zeiger zx auf die Variable *zx verweist. Setzt man zx = &x, so sind die Werte von x und *zx identisch. Für den Zugriff auf die Komponenten der Struktur gibt es zwei alternative Vorgehensweisen:

```
(*zeiger).strukturkomponente
(*zx).name
```

oder

```
zeiger->strukturkomponente
zx->name
```

Genauso wie mit Feldern können auch dynamische Strukturarrays aufgebaut werden.

Beispiel 7.4
Verwaltung eines Warenbestandes [Wil95, S. 566 ff.]. Die Variable articles speichert beliebig viele Datensätze bestehend aus Artikelbezeichnung und Artikelbestand in einem dynamischen Strukturarray und gibt auf Wunsch eine Liste der gespeicherten Werte aus.

1. Programm

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

/* Datensatzstruktur für einen Artikel */
struct article {
  char name[21];          /* Artikelbezeichnung */
  long num;               /* Bestand */
};

void header();

int main() {
  /* Zeiger auf struct article. Mit NULL initialsiert für den ersten
   * Aufruf von realloc
   */
  struct article *zlist = NULL;

  /* sichert die Adresse des Speicherblocks falls realloc NULL an
   * zlist liefert
   */
  struct article *backup;

  char buffer[81];        /* zur Kontrolle der Artikelbezeichnung */
  int i;                  /* Datensatzzähler */
  int k, rep;             /* Kontrollvariablen */
  int artSize;            /* Speicher für einen Artikel */

  printf("ARTIKELLISTE ERSTELLEN\n\n");
  i = 0;
  do {
    /* Blockadresse sichern, falls zlist von realloc NULL erhält */
    backup = zlist;

    /* Speicher allokieren */
    artSize = sizeof(struct article);
    zlist = (struct article *) realloc(zlist, (i+1)*artSize);
    if(zlist == NULL) {
      printf("\n\nNicht genügend Speicherplatz verfügbar.");
      /* für die Ausgabe: i++ sonst wegen des vorzeitigen
       * Schleifenabbruchs um 1 zu niedrig
       */
      i++;
      break;
    }

    /* Datensätze eingeben */
    printf("%d. Datensatz\n", i+1);
    printf("Artikelbezeichnung (max. 20 Zeichen, Ende mit \"0\"): ");
    gets(buffer);
    while(strlen(buffer) > 20) { /* Artikelbezeichnung kontrollieren */
      printf("\nBezeichnung zu lang.");
      printf("\nArtikelbezeichnung (max. 20 Zeichen, Ende mit \"0\"): ");
      gets(buffer);
```

```
    }

    /* Korrekte Bezeichnung in die Liste eintragen */
    strcpy(zlist[i].name, buffer);
    /* falls Artikelbezeichnung != "0" */
    if(strcmp(zlist[i].name, "0")) {
      printf("Artikelbestand: ");
      scanf("%ld", &zlist[i].num);
      getchar();          /* Return lesen */
    }
  } while(strcmp(zlist[i++].name, "0"));

  /* Datensätze ausgeben. Mindestens einen Datensatz eingegeben. Wert
   * von i nach der Schleife um 1 höher als Anzahl der Datensätze.
   */
  if(i > 1) {
    printf("\nAnzahl der eingegebenen Datensätze: %d", i-1);
    printf("\nListe der Datensätze ausgeben? (j/n): ");
    rep = getchar();
    if(rep == 'j') {
      header();           /* Kopfzeile ausgeben */
      if(zlist == NULL) /* falls Fehler bei Allokation */
        zlist = backup; /* Zeigerkopie zur Ausgabe benutzen */
      for(k = 0;k < i-1;k++) {
        printf("%d\t\t%-20s\t\t%ld\n",
               k+1, zlist[k].name, zlist[k].num);
      }                   /* Ende for */
    }                     /* Ende if rep */
    free(zlist);          /* Allokierten Speicher freigeben */
  }                       /* Ende if i > 1 */
  return 0;
}

/* Funktion für Kopfzeile bei der Ausgabe */
void header() {
  int j;
  printf("\nNr.\t\tArtikelbezeichnung\t\tBestand\n");
  for(j = 0 ; j < 55 ; j++)
    printf("_");
  printf("\n");
}
```

2. Ergebnis

```
ARTIKELLISTE ERSTELLEN

1. Datensatz
Artikelbezeichnung (max. 20 Zeichen, Ende mit "0"): Zentraleinheit
Artikelbestand: 100
2. Datensatz
Artikelbezeichnung (max. 20 Zeichen, Ende mit "0"): Monitor
Artikelbestand: 100
3. Datensatz
Artikelbezeichnung (max. 20 Zeichen, Ende mit "0"): Tastatur
```

```
Artikelbestand: 100
4. Datensatz
Artikelbezeichnung (max. 20 Zeichen, Ende mit "0"): Drucker
Artikelbestand: 100
5. Datensatz
Artikelbezeichnung (max. 20 Zeichen, Ende mit "0"): 0

Anzahl der eingegebenen Datensätze: 4
Liste der Datensätze ausgeben? (j/n): j

Nr.             Artikelbezeichnung              Bestand
-----------------------------------------------------
1               Zentraleinheit                  100
2               Monitor                         100
3               Tastatur                        100
4               Drucker                         100
```

Das vorgestellte Programm enthält nur bisher vorgestellte Konstruktionen der Programmiersprache C. Interessant ist der Aspekt der elementweisen Bereitstellung von Speicherplatz für die anzulegende Liste (siehe do/while-Schleife). Dabei wird diese Liste zlist [i] nicht von Anfang an mit ihren Feldgrenzen festgelegt, sondern nur ein Zeiger zlist auf diese dynamisch aufzubauende Liste vereinbart. Anschließend wird in Abhängigkeit einer abzuprüfenden Bedingung das nächste Element abgefordert oder der Abbruch organisiert.

7.3.5 Einfach verkettete Listen

Das im vorherigen Abschnitt vorgestellte Beispiel kann man hinsichtlich der Speicherplatznutzung noch optimieren. Durch den Abruf von Speicherplatz zur Laufzeit ist in dem Programm für das Feld ∗ zlist ein zusammenhängender Block entstanden, wobei ein Element auf das nächste unmittelbar folgt (Abbildung 7.3).

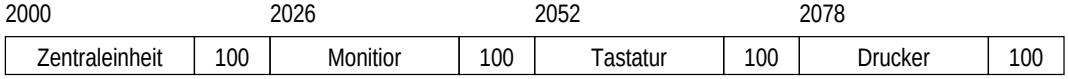


Abbildung 7.3: Speicherung eines Strukturarrays

Nun muß im Speicher dieser zusammenhängender Speicherplatz nicht immer verfügbar sein, so daß irgendwann die Adresse NULL zurückgegeben wird und das Programm beendet werden müßte. Eine Alternative wäre die Möglichkeit des „Zusammensuchens“ von Speicherplatz auf der Halde, wobei die Elemente dann verstreut liegen könnten. Dafür müssen jedoch Vorkehrungen getroffen werden, um die semantische Zusammengehörigkeit der Struktur nicht zu verlieren. Die geeigneten Strukturen dafür sind *Listen*.

Eine solche Liste hat einen Kopf, der nur die Adresse auf das erste Element enthält, und eine Endemarkierung, indem der Zeiger des letzten Elementes auf NULL zeigt. Wie sieht nun eine solche Datenstruktur aus, die diese Informationen trägt? Erweitern wir das vorhergehende Beispiel. Wir definieren zunächst einen Typ listenelement:

```
typedef struct article2 {
  char name[21];
  long num;
  /* Zeiger auf das nächste Listenelement, wobei der Typ der
   * gleiche ist, in welchem der Zeiger next definiert wurde.
   */
  struct article2 *next;
} listenelement;
```

Nun benötigen wir Variablen, die die Listenstruktur aufbauen lassen. Dazu gehören ein Listenanfang basis, ein aktuelles Listenelement zak und eine Varible zum Kopieren von Adressen bkup:

```
  listenelement *basis;     /* Zeiger auf den Anfang der Liste */
  listenelement *zak;       /* aktuelles Zeigerelement */
  listenelement *bkup;      /* Zeiger auf eine Kopie */
```

In Abbildung 7.4 ist ein Ausschnitt aus einer verketten Liste dargestellt.

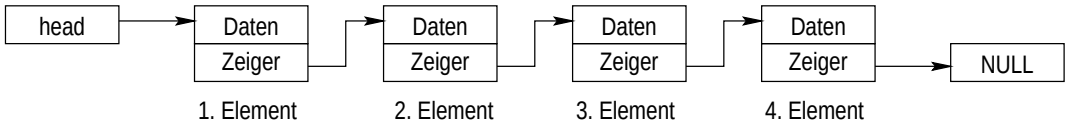


Abbildung 7.4: Schema einer einfach verketteten Liste

Will man eine rückwärtsverkettete Liste aufbauen, die dadurch gekennzeichnet ist, daß das zuletzt eingegebene Datum am Anfang der Liste steht, könnte man folgendes Konstrukt benutzen:

```
Wurzel=NULL;

for (i=1; i<n; i++) {
  zlist=(listenelement*) malloc(sizeof(listenelement));
  scanf("%s", zlist.name);
  scanf("%ld", &zlist.num);
  zlist.next=Wurzel;
  Wurzel=zlist;
}
```

Beispiel 7.5

Einfach verkettete Liste [Wil95, S. 573]. Das Programm speichert eine beliebige Anzahl von Datensätzen als einfach verkettete Liste und gibt diese auf Wunsch aus.

1. Programm

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

```
#include <string.h>

typedef struct article2 {
  char name[21];          /* Artikelbezeichnung */
  long num;               /* Bestand */
  struct article2 *next; /* Zeiger auf nächstes Listenelement */
} listelement;            /* neuer Typname für struct article2 */

void header();

int main() {
  listelement *basis;     /* Zeiger auf den Anfang der Liste */
  listelement *zak;       /* Zeiger auf aktuelles Element */
  listelement *bkup;      /* für Zeigerkopien */
  char buffer [81];       /* zur Eingabekontrolle */
  int i;                  /* Datensatzzähler */
  int k;                  /* Kontrollvariable */
  int rep;                /* Kontrollvariable */

  basis = (listelement *) malloc(sizeof(listelement));
  if(basis == NULL) {
    printf("\n\nKein Speicherplatz verfügbar.");
    exit(1);
  }

  i = 0;
  zak = bkup = basis;
  printf("ARTIKELLISTE ERSTELLEN\n\n");

  /* Datensätze eingeben */
  do {
    printf("%d. Datensatz\n", i+1);
    printf("Artikelbezeichnung (max. 20 Zeichen, Ende mit \"0\"): ");
    gets(buffer);
    /* Artikelbezeichnung kontrollieren */
    while(strlen(buffer) > 20) {
      printf("\nBezeichnung zu lang.");
      printf("\nArtikelbezeichnung (max. 20 Zeichen, Ende mit \"0\"): ");
      gets(buffer);
    }

    /* Korrekte Bezeichnung in die Liste eintragen */
    strcpy(zak->name, buffer);

    /* falls Artikelbezeichnung != "0" */
    if(strcmp(zak->name, "0")) {
      printf("Artikelbestand: ");
      scanf("%ld", &zak->num);
      getchar();          /* Return lesen */

      /* Platz für nächstes Listenelement allokieren */
      zak->next = (listelement *)malloc(sizeof(listelement));
      if(zak->next == NULL) {
        printf("\n\nKein weiterer Speicherplatz verfügbar.");
```

```
        i++;              /* Datensatzzähler aktualisieren */
        break;            /* Schleife verlassen */
      }

      /* Wegen der nächsten Anweisung Adresse des aktuellen
       * Datensatzes sichern für die drittnächste Anweisung und die
       * Überprüfung der Schleifenbedingung
       */
      bkup = zak;

      /* Adresse des nächsten zu bearbeitenden Listenelementes im
       * Zeiger auf das aktuelle Listenelement speichern
       */
      zak = zak->next;
      i++;
    } else {              /* wenn Eingabeende */
      /* NULL im Zeiger des letzten Datensatzes speichern. */
      bkup->next = NULL;
      /* Damit in der Schleifenbedingung der unmittelbar zuvor
       * eingegebene Name überprüft wird
       */
      bkup = zak;
    }
  } while(strcmp(bkup->name, "0"));

  /* Datensätze ausgeben */
  if(i > 0) {             /* Mindestens 1 Datensatz eingegeben */
    printf("\n\nAnzahl der eingegebenen Datensätze: %d", i);
    printf("\nListe der Datensätze ausgeben? (j/n): ");
    if((rep = getchar()) == 'j') {
      header();           /* Kopfzeile Bildschirmseite ausgeben */
      for(zak = basis, k = 1;zak != NULL;zak = zak->next, k++) {
        printf("%d\t\t%-20s\t\t%ld\n", k, zak->name, zak->num);
      }                   /* Ende for    */
    }                     /* Ende if rep */

    /* Speicher freigeben */
    for(zak = basis;zak != NULL;zak = bkup) {
      /* nächste Adresse v o r der Freigabe des aktuellen
       * Listenelements sichern, das diese Adresse enthält. Ohne diese
       * Sicherung wäre diese Adresse verloren und der Zugriff auf die
       * Liste nicht mehr möglich.
       */
      bkup = zak->next;
      free(zak);
    }
  }                       /* Ende if i > 0 */
  return 0;
}

/* Funktion für Kopfzeile bei der Ausgabe */
void header() {
  int j;
  printf("Nr.\t\tArtikelbezeichnung\t\tBestand\n");
```

```
    for(j = 0;j < 55;j++)
      printf("_" );
    printf("\n");
}
```

2. Ergebnis

```
ARTIKELLISTE ERSTELLEN

1. Datensatz
Artikelbezeichnung (max. 20 Zeichen, Ende mit "0"): Fahrrad
Artikelbestand: 10
2. Datensatz
Artikelbezeichnung (max. 20 Zeichen, Ende mit "0"): PKW
Artikelbestand: 18
3. Datensatz
Artikelbezeichnung (max. 20 Zeichen, Ende mit "0"): LKW
Artikelbestand: 12
4. Datensatz
Artikelbezeichnung (max. 20 Zeichen, Ende mit "0"): 0


Anzahl der eingegebenen Datensätze: 3
Liste der Datensätze ausgeben? (j/n): j
Nr.             Artikelbezeichnung              Bestand
-----------------------------------------------------
1               Fahrrad                         10
2               PKW                             18
3               LKW                             12
```

7.3.6 Doppelt verkettete Liste

Für das Bearbeiten von Listen ist es oftmals angebracht, daß jedes Datenobjekt seinen „Nachfolger“ und auch „Vorgänger“ kennt. Somit entstehen doppelt verkettete Listen (Abbildung 7.5).

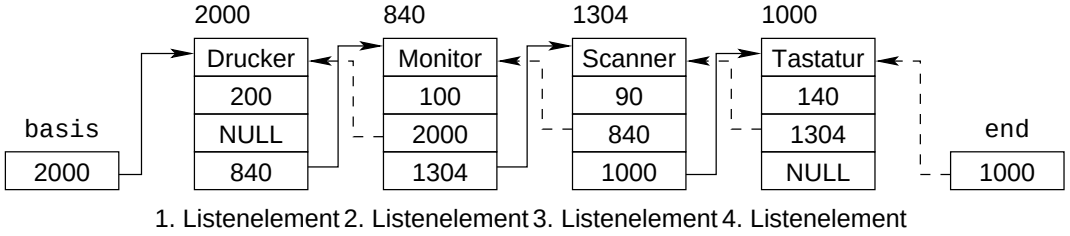


Abbildung 7.5: Doppelt verkettete Liste

Um eine solche Struktur aufzubauen, muß die Typedefinition von `listelement` um einen Zeiger erweitert werden:

```
typedef struct article2 {
  char name[21];
  long num;
  struct article2 *pre;   /* Zeiger sorgt für Verweis auf Vorgänger */
  struct article2 *next;  /* Zeiger verweist auf Nachfolger         */
} listenelement;
```

Zum Aufbau einer doppelt verketteten Liste benötigt man einen Anfangszeiger basis, einen Zeiger für ein neu einzufügendes Listenelement new, einen Zeiger auf das zuletzt einsortierte Listenelement lastin und ein Listenende end:

```
listenelement *basis;
listenelement *end;
listenelement *new;
listenelement *lastin;
```

Für das Anlegen und Bearbeiten von doppelt verketteten Listen müssen spezielle Algorithmen genutzt werden (vgl. [Wil95, S. 584 ff.].

7.3.7 Weitere Anwendungsfälle für Zeiger

Weitere Strukturen unter Nutzung der Zeigertechnik sind:

Zeigerfeld: Speichert eine festgelegte Anzahl von Adressen auf Datenobjekte eines Typs, z.B.:

```
int *zarray[100];
```

Dieses Feld speichert 100 Adressen, die sämtlichst auf Datenobjekte des Typs **int** verweisen.

Zeiger auf Zeiger: Genauso wie Zeiger auf beliebige Datenobjekte verweisen können, ist natürlich der Verweis auf einen Speicherplatz möglich, der wiederum nur eine Adresse auf ein Datenobjekt verwaltet, z.B.:

```
int a=20;       /* Integer-Variable                    */
int za=&a;      /* Zeigervariable, mit Adresse von a   */
int **zza=&za;  /* Zeigervariable, mit Adresse von za  */
```

Mit solchen Konstrukten können z.B. dynamische Zeigerfelder aufgebaut werden.

8 Dateiverwaltung

In Programmen werden Informationen (Daten) verarbeitet, die dann in bestimmter Präsentation (Text, Grafik, Bild) ausgewertet werden. Meistens sollen diese Daten über einen gewissen Zeitraum permanent aufbewahrt bleiben, um sie dann zu gegebener Zeit wiederzuverwenden. So führt ein Unternehmen z.B. eine Kartei seiner Angestellten, Kunden oder Lieferanten. Oder es werden irgendwelche Dokumente der Fertigungsvorbereitung (Zeichnungen, Stücklisten, Arbeitspläne, ...) aufgehoben. Diese Informationen werden auf bestimmten Speichermedien (Festplatte, Diskette, CD, Magnetband, ...) als Dateien gespeichert. Die Verwaltung dieser Dateien erfolgt zum einen über Betriebssystemroutinen, um sie dann den entsprechenden Anwendungen zur Verfügung zu stellen. Das Zusammenspiel zwischen Speichermedium und Anwendung soll Gegenstand dieses Kapitels sein.

In C ist eine Datei eine kontinuierliche Folge von Zeichen oder Bytes, wobei jedes Zeichen mit 0 beginnend und n-1 endend eine Positionsnummer bekommt. Man bezeichnet diese Datei auch als Datenstrom (byte stream). Obwohl dieser Datenstrom grundsätzlich eine unstrukturierte Abfolge von Bytes enthält, bleibt es dem Anwendungsprogrammierer überlassen, „seine“ Dateistruktur und „sein“ Dateiformat zu entwickeln. In der Programmiersprache C kann man auf zwei Ebenen mit Dateien operieren:

Auf der unteren (low level) Ebene: Auf dieser Ebene werden die sogenannten elementaren Dateizugriffe organisiert. Hierbei wird unmittelbar auf die entsprechenden Routinen (system calls) für Dateioperationen des Betriebssystems zugegriffen. Diese sind nicht Bestandteil des ANSI-Standards.

Auf der oberen (high level) Ebene: Die Dateiarbeit auf dieser Ebene wird mit komplexeren Funktionen realisiert, die in der Standard-Bibliothek (`stdio.h`) verwaltet werden. Man nennt diese Dateioperationen nichtelementar.

8.1 Textdateien

Wird es notwendig, aus einem Programm heraus Daten von einem permanenten Speicher abzurufen oder auch hineinzuschreiben, so geschieht das nicht direkt zwischen dem vom Programm belegten Teil des Arbeitspeichers und der Platte (oder Diskette oder CD), sondern immer über einen Pufferbereich im Arbeitsspeicher. Dieser Puffer nimmt eine größere Menge von Daten auf, damit nicht jedes Datenobjekt separat zwischengespeichert werden muß. Für dieses Wechselspiel ist ein strukturierter Typ **FILE** festgelegt:

```
typedef struct {
  char *buffer;    /* Zeiger für die Adresse des Dateipuffers  */
```

```
  char *ptr;        /* Zeiger auf das nächste Zeichen im Puffer */
  int cnt;          /* Anzahl der Zeichen im Puffer             */
  int flags;        /* Bits mit Angaben zum Dateistatus         */
  int fd;           /* Deskriptor (Kennzahl der Datei)          */
} FILE;
```

In der Datei `stdio.h` ist ein Array aus solchen FILE-Strukturen deklariert. Im Anwendungsprogramm ist quasi zur Aufnahme einer logischen Verbindung zur Bibliothek ein Zeiger auf eine Strukturvariable vom Typ FILE zu verwenden:

```
FILE *fz;    /* Zeiger auf FILE-Strukturvariable */
```

Wird nun eine Datei zur Bearbeitung geöffnet, so sucht die dafür zuständige Funktion einen Platz in dem oben erwähnten Feld. Die Anfangsadresse wird dem Zeiger fz mitgeteilt. Alle Zugriffe erfolgen nun über diesen Zeiger. In Abbildung 8.1 ist der Zugriff auf eine Datei schematisch dargestellt (nach [Wil95, S. 741]).

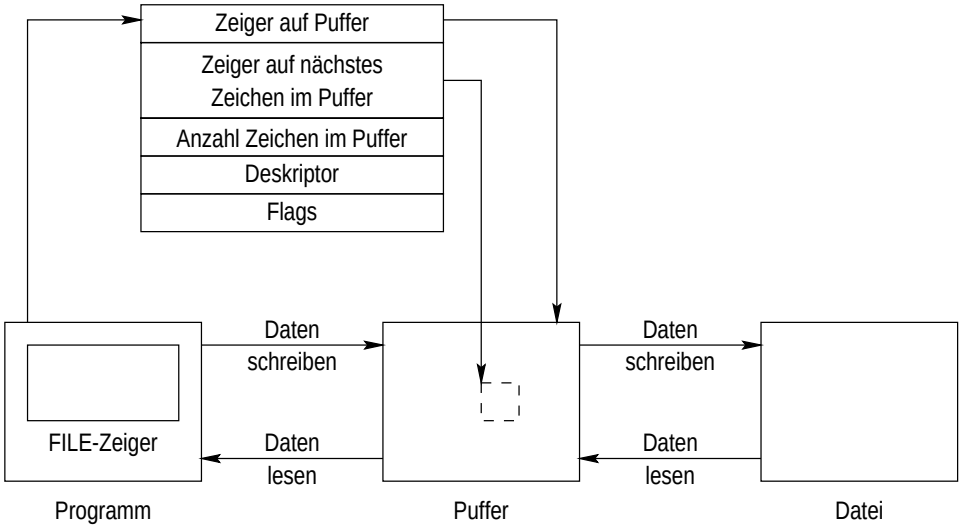


Abbildung 8.1: High-level-Zugriff auf eine Datei

8.1.1 Öffnen von Dateien

Mit der Funktion fopen verschafft man sich mit Hilfe des Betriebssystems Zugang zur gewünschten Datei. Diese Funktion hat den Prototypen:

Syntax (fopen):

```
FILE *fopen(char *dateiname, char *zugriffsmodus);
```

Dabei ist dateiname ein Zeiger auf eine Zeichenkette mit dem Namen der betreffenden Datei und zugriffsmodus ein Zeiger auf eine Zeichenkette, die angibt, welche Operationen nach der Öffnung mit der Datei ausgeführt werden können. Als Ergebnis liefert fopen einen Zeiger auf eine Variable des Datentyps **FILE**. Die möglichen Zugriffsmodi sind in Tabelle 8.1 zusammengefaßt. Diese Zugriffsmodi können noch um das Zeichen b erweitert werden. Der Grund liegt darin, daß die unterschiedlichen Betriebssysteme (z.B. UNIX oder DOS) auf

der physischen Ebene (quasi auf der Speicherebene) unterschiedliche Dateidarstellungen zulassen (Text- und Binärdateien). Der Parameter b kennzeichnet eine Binärdatei.

Zugriffsmodus	Ziel der Operation
„r“	Öffnen zum Lesen, fopen==NULL wenn Datei nicht vorhanden
„w“	Öffnen zum Schreiben, Datei wird erzeugt, bereits existierende Datei wird überschrieben und geht damit verloren
„a“	Anfügen am Dateiende, wenn Datei nicht vorhanden, wird sie erzeugt
„r+“	Öffnen zum Lesen und Schreiben, fopen==NULL, wenn Datei nicht vorhanden
„w+“	Öffnen zum Schreiben und Lesen, Datei wird erzeugt, bereits existierende Datei wird überschrieben
„a+“	Öffnen zum Lesen und Anfügen, noch nicht existierende Datei wird erzeugt

Tabelle 8.1: Zugriffsmodi für fopen

Eine Datei mit Namen Kunde.dat, die im aktuellen Verzeichnis liegt, kann mit folgenden Anweisungen zum Lesen geöffnet werden:

```
FILE *datei;                      /* FILE-Zeiger definieren */
datei=fopen("KUNDE.dat", "r");    /* Datei öffnen           */
```

Kann diese Datei nicht gefunden oder aus einem anderen Grunde nicht geöffnet werden, so wird NULL zurückgegeben. Dies kann im Programm beispielsweise genutzt werden, um eine Nachricht auszugeben:

```
if (datei==NULL)
  printf("Fehler: Datei konnte nicht geöffnet werden!");
```

8.1.2 Schließen von Dateien

Das Schließen von Dateien ist aus zwei Gründen notwendig:

1. Es wird die Verbindung zur Datei gelöst, indem der zugehörige **FILE**-Zeiger freigegeben wird.
2. Die Daten des Dateipuffers werden restlos in die betreffende Datei übertragen.

Die Bibliotheksfunktion zum Schließen hat den Prototyp:

Syntax (fclose):

```
int fclose(FILE *dateizeiger);
```

und besitzt somit als Parameter einen Zeiger auf den Datentyp **FILE**. Der Funktionsaufruf:

```
fclose(fz);
```

schließt die Datei, auf welche fz zeigt. Die Funktion fcloseall schließt alle im Programm geöffneten Dateien:

Syntax (fcloseall):

```
int fcloseall();
```

8.1.3 Lese- und Schreiboperationen mit Dateien

Das Lesen und Schreiben auf Dateien wird analog zur Eingabe über Tastatur bzw. zur Ausgabe über Drucker durch Funktionen unterstützt. An welcher Stelle der Datei gelesen oder geschrieben wird, gibt ein spezieller Positionszeiger (seek pointer) an. Er enthält die gegenwärtige Position des Lese-/Schreibkopfes (vgl. Abbildung 8.2). Bei jeder Operation verändert sich die Position des seek pointers. Wir werden später kennenlernen, daß man durch Positionieren dieses Zeigers nicht nur sequentiell (in Folge) sondern auch direkt zugreifen kann.

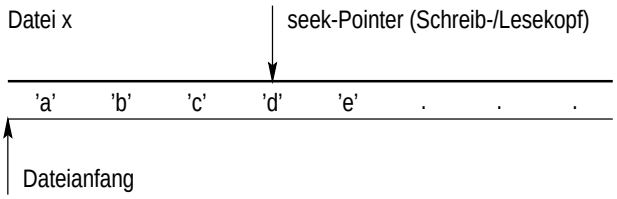


Abbildung 8.2: Wirkung von seek pointer

Zeichenweises Lesen und Schreiben

Mit den Funktionen fputc und fgetc kann man einzelne Zeichen in eine Datei schreiben bzw. lesen:

Syntax (fputc):

```
int fputc(int zeichen, FILE *dateizeiger);
```

Diese Ausgabefunktion überträgt ein Zeichen in die durch den Zeiger dateizeiger gekennzeichnete Datei. Dabei wird der Datentyp **int** vorher in **unsigned char** umgewandelt, z.B. schreibt:

```
putc('A', fz);
```

das Zeichen A in die mit fz verknüpfte Datei an die aktuelle Stelle des Lese-/Schreibkopfes. Als Rückgabewert liefert fputc das geschriebene Zeichen, im Falle eines Fehlers EOF (end of file).

Das Lesen aus einer Datei erfolgt mit der Funktion fgetc:

```
int fgetc (FILE *dateizeiger);
```

Diese Eingabefunktion holt ein Zeichen aus der Datei. Der Rückgabewert EOF zeigt das Dateiende oder einen Fehler an, z.B.:

```
int c;
/* liest ein Zeichen aus der Datei und speichert es in Variable c */
c=fgetc(fz);
```

Mit der Schleife

```
c=fgetc(fz);
while(c != EOF) {
  putchar(c);
  c=fgetc(fz);
}
```

kann der ganze Dateiinhalt ausgegeben werden.

Nun weiß man bei einem Abbruch der Schleife nicht unbedingt den Grund, ob ein Fehler beim Lesen aufgetreten ist, oder ob einfach das Ende der Datei erreicht wurde. Dazu wird die Funktion feof genutzt:

Syntax (feof):

```
int feof (FILE *dateizeiger);
```

Diese Funktion prüft EOF ab und liefert den Rückgabewert verschieden von 0 bei erfolgreichem Erreichen des Dateiendes anderenfalls 0 beim Auftreten eines Fehlers. In einem Programm kann demnach die nachfolgende Anweisungsfolge Klarheit über den Zugriff auf eine Datei schaffen:

```
c=fgetc(fz);
while(c != EOF) {   /* Dateiinhalt ausgeben */
  putchar(c);
  if(feof(fz))      /* Dateiende?           */
    printf("Dateiende\n");
  else              /* Nein                 */
    printf("Fehler beim Dateizugriff\n");
  c=fgetc(fz);
}
```

Anstelle von fgetc und fputc können auch die Makros getc und putc benutzt werden.

Beispiel 8.1

Eingabe eines Textes über Tastatur und schreiben in eine Datei [Wil95, S. 751]. Das Programm schreibt einen über die Tastatur eingegebenen Text beliebiger Länge in eine Datei und zeigt deren Inhalt auf Wunsch an. Für die Ein- und Ausgabeoperationen werden die Funktionen fgetc und fputc verwendet.

1. Programm

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

#define END 64          /* das Zeichen @ in Dezimalcode */

int main() {
  FILE *fz;             /* Zeiger auf FILE-Struktur */
  char filename[81];    /* Dateiname */
  int rep1, rep2;       /* Kontrollvariablen */
  int c;                /* Zeichenpuffer */

  printf("Text zum Speichern eingeben.\n");
  printf("In welche Datei soll geschrieben werden?\n");
  do {
    printf("Dateiname (mit Pfad): ");
    gets(filename);
    fz = fopen(filename, "r");
    if(fz != NULL) {
      printf("\nDatei existiert bereits. überschreiben? (j/n): ");
      rep1 = toupper(getchar());
      getchar();        /* Return lesen */
```

```
      if(rep1 != 'J') {
        fclose(fz);
      }
    } else {          /* Datei existiert noch nicht */
      break;          /* Eingabeschleife verlassen */
    }
  } while(rep1 != 'J');

  if(fz != NULL)      /* Datei soll Überschrieben werden */
    fclose(fz);       /* daher: schließen */

  /* und zum Schreiben öffnen */
  fz = fopen(filename, "w");
  if(fz == NULL) {
    printf("\nDatei kann nicht beschrieben werden.\n");
    exit(1);
  }
  printf("Eingabetext (Ende mit <@> + <RETURN>):\n");
  c = getchar();
  while(c != END) {   /* Text in Datei schreiben */
    fputc(c, fz);
    c = getchar();
  }
  getchar();          /* Return lesen */
  fclose(fz);

  /* Dateiinhalt anzeigen */
  printf("\nDatei anzeigen? (j/n): ");
  rep2 = toupper(getchar());
  if(rep2 == 'J') {
    fz = fopen(filename, "r");
    if(fz == NULL) {
      printf("\nFehler beim Öffnen der Datei.\n");
      exit(2);
    }
    printf("Inhalt der Datei %s:\n", filename);
    c = fgetc(fz);
    while(c != EOF) { /* Zeichen aus Datei lesen */
      putchar(c);     /* und ausgeben */
      c = fgetc(fz);
    }                 /* Ende while */
    fclose(fz);
  }                   /* Ende if rep2 == 'J' */
  return 0;
}
```

2. Ergebnis

a) Neue Datei anlegen:

```
Text zum Speichern eingeben.
In welche Datei soll geschrieben werden?
Dateiname (mit Pfad): dateineu.txt
Eingabetext (Ende mit <@> + <RETURN>):
```

```
Vorlesung Dateiarbeit
neue Datei anlegen
dateineu.txt
@

Datei anzeigen? (j/n): j
Inhalt der Datei dateineu.txt:
Vorlesung Dateiarbeit
neue Datei anlegen
dateineu.txt
```

b) Datei existiert:

```
Text zum Speichern eingeben.
In welche Datei soll geschrieben werden?
Dateiname (mit Pfad): dateineu.txt

Datei existiert bereits. überschreiben? (j/n): j
Eingabetext (Ende mit <@> + <RETURN>):
Vorlesung Dateiarbeit
Fallbeispiel: überschreiben einer bereits existierenden Datei
dateineu.txt
@

Datei anzeigen? (j/n): j
Inhalt der Datei dateineu.txt:
Vorlesung Dateiarbeit
Fallbeispiel: überschreiben einer bereits existierenden Datei
dateineu.txt
```

Lesen und Schreiben von Zeichenketten

Analog zu den Funktionen gets zum Einlesen über Tastatur und puts zur Ausgabe auf dem Bildschirm kann man mit fgets eine Zeichenkette aus einer Datei lesen. Die Funktion hat folgenden Prototyp:

Syntax (fgets):

```
char* fgets(char *pufferzeiger, int anzahl, FILE *dateizeiger);
```

Als Parameter erwartet die Funktion einen Zeiger auf die Puffervariable, die Anzahl der zu lesenden Zeichen und den Dateizeiger. In allen Fällen schließt fgets die eingelesene Zeichenkette mit \0 ab. Die Wirkung des Programmausschnittes

```
FILE *fz;
char zeichenkette[81];

fz = fopen("A:\\versuch.doc", "r");
fgets(zeichenkette, 81, fz);
```

ist das Einlesen von maximal 80 Zeichen aus einer Datei mit Namen `versuch.doc` in die Variable zeichenkette einschließlich dem abschließenden Nullzeichen.

Mit fputs wird eine Zeichenkette in eine Datei geschrieben. Der Prototyp lautet:

Syntax (fputs):

```
int fputs(char *pufferzeiger, FILE *dateizeiger);
```

Um eine Zeichenkette an die Datei `versuch.doc` anzuhängen kann folgender Code verwendet werden:

```
FILE *fz;
char zeichenkette[ ] = "Das ist ein Beispiel.";

fz = fopen("A:\\versuch.doc", "a");
fputs(zeichenkette, fz);
```

Formatiertes Lesen und Schreiben

Oftmals haben die vom Nutzer angelegten Dateien schon eine bestimmte Struktur. Nehmen wir einmal an, wir wollen eine Kundendatei anlegen. Für die Speicherung eines Kunden wird folgende Struktur verwendet:

```
struct kunde {
  long nr;          /* Kundennummer     */
  char name[31];    /* Name des Kunden */
  long plz;         /* Postleitzahl     */
  char ort[31];     /* Wohnort          */
  float umsatz;     /* Umsatz           */
};

struct kunde k[100];
```

Mit den Funktionen fprintf und fscanf können formatierte Dateien angelegt und gelesen werden. Die Prototypen lauten:

Syntax (fprintf und fscanf):

```
int fprintf(FILE *dateizeiger, char *formatstring, arg1, ...);
int fscanf(FILE *dateizeiger, char *formatstring, arg1, ...);
```

Das Öffnen der Datei und das Schreiben der Datensätze in die Datei und das Lesen der Datensätze aus der Datei erfolgt anschließend mit folgenden Anweisungen:

```
FILE *fz;
int i, n;
fz = fopen("kunden.dat", "w+");
if(fz == NULL)
  fputs("\nDatei konnte nicht geöffnet werden:", stderr);
else
  for (i=0; i<n; i++)
    fprintf(fz, "%ld %s %ld %s %.2f", k[i].nr, k[i].name, k[i].plz,
            k[i].ort, k[i].umsatz);
rewind(fz);
while(fscanf(fz, "%ld %30s %ld %30s %f",
             &k[i].nr, k[i].name, &k[i].plz, k[i].ort,
             &k[i].umsatz) != EOF)
  printf("%ld %s %ld %s %.2f", k[i].nr, k[i].name, k[i].plz, k[i].ort,
         k[i].umsatz);
```

Man kann schlußfolgern, daß fscanf(stdin , ...) und scanf() sowie fprintf (stdout , ...) und printf () äquivalent sind.

Es muß an dieser Stelle erwähnt werden, daß es neben dem Zugriff auf Dateien in Form von Zeichen, Zeichenketten oder formatierten Datenobjekten noch die Möglichkeit gibt, blockweise, d.h. mit einer bestimmten Anzahl von Bytes, zuzugreifen. Die Funktionen dazu sind fread und fwrite.

Direktzugriff

Der bisherige Zugriff auf Dateien war sequentiell. Das bedeutet, man bewegt sich beim Lesen oder Schreiben immer von oben nach unten. Wünschenswert ist es jedoch, an Datenobjekte heranzukommen, die an einer bestimmten Stelle der Datei stehen. Dazu muß erst einmal der Schreib-/Lesekopf auf diese Position gebracht werden. Anschließend erfolgt der Zugriff. Diese Art der Bearbeitung nennt man Direktzugriff (random access) oder auch wahlfreien Zugriff.

Die Funktion fseek setzt den Dateipositionszeiger an die gewünschte Stelle. Der Prototyp lautet:

Syntax (fseek):

```
int fseek(FILE *dateizeiger, long offset, int basis);
```

Der Parameter offset beinhaltet dabei die Anzahl der Bytes, um welche der Positionszeiger relativ zu basis verschoben werden soll. Die möglichen Werte für Basis sind in Tabelle 8.2 zusammengefaßt. Ist fseek erfolgreich durchgeführt, gibt die Funktion den Wert 0 zurück ansonsten einen Wert ungleich 0. Die Funktion rewind stellt den Positionszeiger immer an den Anfang der Datei. Der Prototyp lautet:

Syntax (rewind):

```
int rewind(FILE *dateizeiger);
```

Mit der Funktion ftell kann die aktuelle Position nach einem Zugriff bestimmt werden. Der Prototyp lautet:

Syntax (ftell):

```
int ftell(FILE *dateizeiger)
```

Wert von basis	symbolische Konstante	Bedeutung
0	seek_set	Dateianfang
1	seek_cur	Aktuelle Position in der Datei
2	seek_end	Dateiende

Tabelle 8.2: Konstanten für basis-Parameter der fseek-Funktion

Beispiel 8.2
Anwendung von fseek [BS95]. Datei `DATNEU` *durchmustern und ausgeben.*

1. Programm

```
#include <stdio.h>

int main() {
```

```
  int ch, n;
  long offset, last;
  char datei[80];
  FILE *in_file;
  printf("Datei: ");
  gets(datei);
  printf("\n");
  in_file = fopen(datei, "r");
  fseek(in_file, 0L, 2);        /* Position Ende der Datei */

  /* Sichert Pos. des letzten Zeichens */
  last = ftell(in_file);
  printf("%i \n", last);
  for(offset = 0; offset <= last; ++offset) {
    fseek(in_file, offset, 0); /* Pos. zum nächsten Zeichen */
    ch = getc(in_file);         /* Zeichen holen */
    switch(ch) {
    case '\n' :
      printf("LF:\n");
      break;
    case EOF:
      printf("EOF:\n");
      break;
    default:
      printf("%c:", ch);
      break;
    }
  }
  fclose(in_file);
  return 0;
}
```

2. Ergebnis

```
Datei: dateineu.txt

54
V:o:r:l:e:s:u:n:g: :D:a:t:e:i:a:r:b:e:i:t:LF:
n:e:u:e: :D:a:t:e:i: :a:n:l:e:g:e:n:LF:
d:a:t:e:i:n:e:u:.:t:x:t:LF:
EOF:
```

8.2 Standardgerätedateien und vordefinierte FILE-Zeiger

Genauso wie die Ein- und Ausgabe von permanenten Speichern über das Dateikonzept wird mit der Ein- bzw. Ausgabe anderer „Geräte“ verfahren. Zu Beginn eines C-Programmes

werden bis zu 5 Gerätedateien geöffnet, die in der Bibliothek `stdio.h` mit einem **FILE**-Zeiger verbunden sind (vgl. Tabelle 8.3).[1]

Zeigerkonstante	Bedeutung	Standardeinstellung
stdin	Standardeingabe	meist Tastatur
stdout	Standardausgabe	meist Bildschirm
stderr	Standardfehlerausgabe	meist auch über Bildschirm
stdaux	Standardzusatz	Zusatzgeräte, wie Plotter, Tablett
stdprn	Standarddruck	Drucker

Tabelle 8.3: In `stdio.h` definierte **FILE**-Zeiger und deren Bedeutung

Mit Hilfe dieser Gerätezeiger kann man nun Ein- und Ausgaben umleiten. Wollen wir beispielsweise einen über die Tastatur eingegebenen Datenstrom auf den Drucker umleiten, so könnte dies über folgende Schleife erfolgen:

```
while((c=fgetc(stdin)) != EOF)
  fputc(c, stdprn);
```

Beispiel 8.3
Umleiten einer Dateiausgabe auf den Drucker [Wil95, S. 758]. Das Programm printfl druckt eine Datei, deren Name als Kommandoparameter übergeben wird.

1. Programm

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

int main(int argc, char **argv) {
  FILE *fz;
  int c;

  if(argc != 2) {   /* Nur ein Parameter erlaubt */
    printf("\nSyntax: printfl dateiname\n");
    exit(1);
  }

  fz = fopen(argv[1], "r");
  if(fz == NULL) {
    printf("\nDatei %s konnte nicht geöffnet werden.", argv[1]);
    exit(2);
  }

  c = fgetc(fz);
  while(c != EOF) { /* Datei drucken */
    fputc(c, stdprn);
    c = fgetc(fz);
  }
  return 0;
}
```

[1] stdaux und stdprn gehören nicht zum ANSI-Standard und stehen nicht unter allen Compilern (auch nicht dem gcc) zur Verfügung.

2. Ergebnis

```
(siehe Druckbild)
```

8.3 Binärdateien

Die bisher beschriebenen Dateien können ohne Schwierigkeiten mit einem Texteditor bearbeitet werden, da sie als Textdatei angelegt wurden. Nun kann man die Variablen aber auch direkt als Folge der Bits abspeichern, die den entsprechenden Datentyp abbilden. Diese Art der Darstellung nennt man Binärdatei. Bei der Arbeit mit Binärdateien ist zu beachten, daß die Zugriffsmodi um ein b ergänzt werden, z.B. „rb“ für das Öffnen einer Binärdatei zum Lesen. Es gibt zwei Hauptfunktionen für Binärdateien. Die eine heißt fwrite und schreibt in Binärdateien, die andere ist fread und liest Binärdateien. Der Prototyp von fwrite ist:

Syntax (fwrite):

```
int fwrite(adresse, groesse, anzahl, fhd);
```

Dabei ist adresse die Adresse der zu speichernden Variablen oder eines Feldes, groesse ist die Größe der Variablen oder eines Feldelementes, anzahl ist die Anzahl der Feldelemente, im Falle einer Variablen gleich 1 und fhd ist ein Zeiger auf die **FILE**-Struktur. Die Funktion fwrite gibt die Anzahl der komplett zurückgegebenen Elemente zurück. Der Aufruf von fread besitzt eine ähnliche Syntax:

Syntax (fread):

```
int fread(adresse,groesse,anzahl,fhd);
```

Zur Bestimmung der Bytezahl benutzen wir die Funktion **sizeof**.

Beispiel 8.4
Schreiben und Lesen von Binärdateien [Wil96]

1. Programm

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

#define DATNAME "test.txt"
#define MAXVORNAME 25
#define MAXNACHNAME 20

void schreiben(void) {
  FILE *fhd;
  char vname[MAXVORNAME], nname[MAXNACHNAME];
  unsigned int alter, groesse;

  fhd=fopen(DATNAME,"wb");
  if(!fhd) {
    printf("Datei konnte nicht erzeugt werden!\n\n");
  } else {
```

```
    printf("Vorname (Max. %i Zeichen):  ", MAXVORNAME-1);
    scanf("%s", vname);
    printf("Nachname (Max. %i Zeichen): ", MAXNACHNAME-1);
    scanf("%s", nname);
    printf("Alter in Jahren:            ");
    scanf("%u", &alter);
    printf("Groesse in Zentimetern:     ");
    scanf("%u", &groesse);

    fwrite(vname, sizeof(vname), 1, fhd);
    fwrite(nname, sizeof(char), MAXNACHNAME, fhd);
    fwrite(&alter, sizeof(alter), 1, fhd);
    fwrite(&groesse, sizeof(unsigned int), 1, fhd);

    fclose(fhd);
    printf("\nEingabe beendet!\n");
  }
}

void lesen(void) {
  FILE *fhd;
  char vname[MAXVORNAME], nname[MAXNACHNAME];
  unsigned int alter, groesse;

  fhd=fopen(DATNAME, "rb");
  if(!fhd) {
    printf("Datei konnte nicht erzeugt werden!\n\n");
  } else {
    fread(vname, sizeof(vname), 1, fhd);
    fread(nname, sizeof(nname), 1, fhd);
    fread(&alter, sizeof(alter), 1, fhd);
    fread(&groesse, sizeof(groesse), 1, fhd);
    fclose(fhd);

    printf("Vorname:         %s\n", vname);
    printf("Nachname:        %s\n", nname);
    printf("Alter in Jahren: %u\n", alter);
    printf("Groesse in cm:   %u\n", groesse);
  }
}

int main() {
  int input;

  printf("Soll die Datei 1=gelesen oder 2=beschrieben werden? ");
  scanf("%i",&input);

  if(input==1)
    lesen();
  else if(input==2)
    schreiben();
  else
    printf("\nFalsche Eingabe!\n\n");
  return 0;
```

```
}
```

2. Ergebnis

```
Soll die Datei 1=gelesen oder 2=beschrieben werden? 2
Vorname (Max. 24 Zeichen):  Gerd
Nachname (Max. 19 Zeichen): Lange
Alter in Jahren:            53
Groesse in Zentimetern:     163

Eingabe beendet!


Soll die Datei 1=gelesen oder 2=beschrieben werden? 1
Vorname:         Gerd
Nachname:        Lange
Alter in Jahren: 53
Groesse in cm:   163
```

9 Objektorientierte Programmierung mit C++

9.1 Einführung

Objektorientierte Sprachen sind der nächste wichtige Schritt in der Entwicklung der Programmierung. Dieser Schritt fällt aus dem Rahmen einer traditionellen Weiterentwicklung heraus. Das Paradigma des objektorientierten Ansatzes liegt in der *Vereinigung von Daten und Algorithmen zur Bearbeitung der Daten*. Damit entstehen *abstrakte Datentypen*, die zur Beschreibung der Objekte genutzt werden. Gleichermaßen ist es möglich, verschiedene Objekte in Beziehungen zueinander zu bringen, um dann das Zusammenwirken über Steueralgorithmen zu organisieren.

Oftmals ist die Situation so, daß sich das zu realisierende Modell ändert. Bedeutet das auch, daß in jedem Fall das Programm auch umgeschrieben werden muß? Die Antwort ist „nein“. Es ist ratsam, einen Prototyp zu entwickeln, um den Entwurf zu unterstützen. Das ist durch die Verwendung „gewöhnlicher“ Datentypen nicht zu erreichen, sondern erfordert abstrakte Datentypen.

Der Vorteil der objektorientierten Sprachen ist, daß sie den Abstand zwischen dem Modell und dem Prototyp reduzieren. Ein Vertreter dieser Sprachen ist C++. Andere Sprachen dieser Familie sind z.B. Smalltalk, Java, Turbo-Pascal, usw. Die Programmiersprache C++ [Str98] wurde im Jahre 1980 von Bjarne Stroustrup in den AT&T Laboratorien entwickelt. Ihren Namen bekam sie 1983. C++ hat sich aus der Programmiersprache C heraus entwickelt und umfaßt Weiterentwicklungen auf drei wichtigen Gebieten:

1. Definition und Einsatz abstrakter Datentypen,
2. Objektorientierter Entwurf und Programmierung,
3. Viele kleine Verbesserungen gegenüber vorhandenen C-Konstrukten unter Beibehaltung der Einfachheit des Ausdrucks und des Laufzeitverhaltens von C.

9.2 Allgemeine Eigenschaften objektorientierter Sprachen

Jede objektorientierte Sprache unterstützt die folgenden Merkmale:

1. Abstrakte Datentypen,

2. Vererbung,

3. Polymorphismus (Vielgestaltigkeit).

9.2.1 Abstrakte Datentypen

In C++ kann der Programmierer seine eigenen Datentypen bearbeiten. Diese benutzerdefinierten Datentypen werden als Klassen bezeichnet. Eine Klasse besteht aus einer Zusammenfassung von definierten Datenkomponenten, die verschiedene Datentypen (Eigenschaften) und eine Reihe von Operationen (Funktionen oder auch Methoden) aufweisen können, die auf diese Daten angewendet werden können. Üblicherweise wird eine Klasse dazu verwendet, um einen neuen Datentyp einzuführen. Die Variablen, die zu einer bestimmten Klasse gehören, nennt man Objekte. Die Sprache C++ unterstützt einen lokalen Bereich in der Klasse und der Zutritt zu ihren Komponenten ist stark beschränkt. Der Programmierer steuert diesen Zutritt über Schnittstellen, die einzig Methoden darstellen. Ihr Aufruf erfolgt über den sogenannten Botschaftenaustausch. In der Sprache C++ erreicht man durch die Klassen ein höheres Niveau der Abstraktion.

9.2.2 Vererbung

Eine zweite wichtige Anwendung des Klassenmechanismus betrifft die Definition von abgeleiteten Datentypen. Dieser Mechanismus läßt die Möglichkeit zu, daß eine Klasse die Komponenten einer anderen Klasse, also Eigenschaften und Methoden, erbt und gleichzeitig neue Komponenten hinzugefügt werden können. Man sagt, daß die neue Klasse abgeleitet sei. Die abgeleitete Klasse kann wiederum weitervererbt werden. Dieser Mechanismus definiert eine Hierarchie zwischen den Klassen. Wenn eine Klasse nur Komponenten von einer anderen Klasse erbt, spricht man von einfacher Vererbung (vgl. Abbildung 9.1). Wenn eine Klasse Komponenten von mehreren Klassen erbt, bezeichnet man diese Vererbung als mehrfache Vererbung (vgl. Abbildung 9.2). Die mehrfache Vererbung definiert eine Beziehung zwischen unabhängigen Klassen.

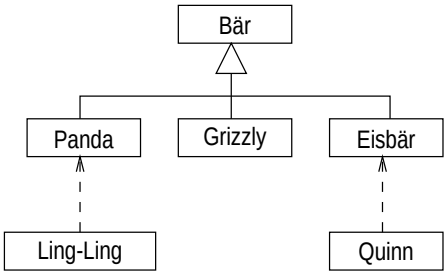


Abbildung 9.1: Einfache Vererbung

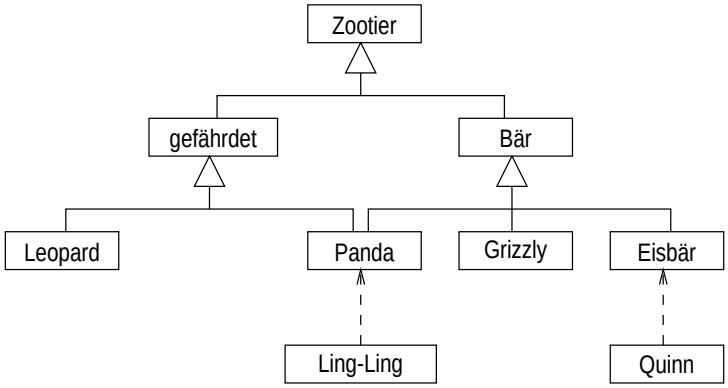


Abbildung 9.2: Mehrfache Vererbung

9.2.3 Polymorphismus (Vielgestaltigkeit)

Diese Eigenschaft bietet u.a. die Möglichkeit für mehrfache Anwendung von Methoden gleichen Namens über die Klassenhierarchie. Das bedeutet, daß eine Methode mit unterschiedlichem Programmcode verbunden werden kann. Einfacher ausgedrückt: Was eine Botschaft tut, hängt davon ab, an welches Objekt man sie richtet. So hat z.B. eine Botschaft „Zeige Objekt auf dem Bildschirm“ bei den Objektklassen Punkt und Kreis grundsätzlich die gleiche Bedeutung. Die Ausführung (Implementierung) ist jedoch sehr verschieden. Es ist üblich, die Zuordnung der Methoden eines Programmes auf der Stufe der Kompilation zu bestimmen. In objektorientierten Sprachen kann diese Zuordnung zur Laufzeit innerhalb der Vererbungshierarchie erfolgen. Man nennt diese Form der Bindung *späte Bildung (late binding)*.

9.3 Änderungen und Erweiterungen in C++

Eingabe und Ausgabe von Daten gehören nicht zur Sprache C++, werden aber durch eine Bibliothek unterstützt, die in C++ geschrieben wurde und als iostream-Bibliothek bezeichnet wird. Die Eingabe von der Tastatur des Anwenders, die als Standardeingabe bezeichnet wird, ist mit dem vordefinierten Objekt iostream cin verbunden (entspricht stdin in C). Die Ausgabe auf dem Terminal, die als Standardausgabe bezeichnet wird, ist mit dem vordefinierten Objekt iostream cout (entspricht stdout in C) verbunden. Das Objekt iostream cerr ist das Analogon zu stderr. Der Ausgabeoperator << wird zur Umleitung eines Wertes zur Standardausgabe verwendet, z.B.:

```
cout << "Die Summe von 7 + 3";
cout << 7 + 3;
cout << "\n";
```

Die Zeichenfolge \n stellt das Zeichen für einen Zeilenvorschub (newline) dar. Bei der Ausgabe spielt das Zeilenvorschubzeichen dieselbe Rolle, wie bei der Sprache C. Anstelle des Zeilenvorschubzeichens kann man auch den standardmäßig definierten Operator endl verwenden, endl fügt einen Zeilenvorschub in den Ausgabestrom ein und gibt den Pufferinhalt sofort aus. Statt

```
cout << "\n";
```

schreibt man dann

```
cout << endl;
```

Aufeinanderfolgende Ausgabeoperatoren lassen sich zu einem Befehl zusammenfassen, z.B:

```
cout << "Die Summe von 7 + 3" << 7 + 3 << endl;
```

Jeder der nachfolgenden Ausgabeoperatoren wird noch einmal auf cout angewendet. Damit die Lesbarkeit erhalten bleibt, läßt sich eine Ausgabeanweisung auf mehrere Zeilen verteilt. Die folgenden drei Zeilen stellen eine einzige Ausgabeanweisung dar:

```
cout << "Die Summe von" << v1 << "+"
     << v2 << " = "
     << v1 + v2 << endl;
```

In ähnlicher Weise wird der Eingabeoperator >> zum Einlesen eines Wertes von der Standardeingabe verwendet. Das folgende Programm implementiert zum Beispiel einen einfachen Algorithmus, um zwei Werte einzulesen und den größeren von beiden zu ermitteln und daraufhin auszugeben.

Beispiel 9.1
Ein- und Ausgabe über cin und cout

1. Programm

```
#include <iostream.h>

void Nimm2(int&, int& );
int Max (int, int);
void Max_schreiben (int);

int main() {
  int Wert1, Wert2;
  Nimm2 (Wert1, Wert2 );
  int Max_Wert = Max (Wert1, Wert2);
  Max_schreiben (Max_Wert);
  return 0;
}

void Nimm2(int& v1, int& v2) { /* Eingabe */
  cout << "Bitte geben Sie zwei Zahlen ein: ";
  cin >> v1 >> v2;
}

int Max(int v1, int v2) {       /* Maximumsuche */
  if(v1 > v2)
    return v1;
  else
    return v2;
}

void Max_schreiben(int Wert) { /* Ausgabe */
  cout << Wert << " ist der grössere Wert." << endl;
}
```

2. Ergebnis

```
Bitte geben Sie zwei Zahlen ein: 17 125
125 ist der größere Wert.
```

3. **Anmerkungen** Zu diesem Programm sind folgende Anmerkungen erforderlich. Die drei Funktionsbeschreibungen stehen vor der Definition von main(). Die erste Aufstellung, die auch als Vordeklaration bezeichnet wird, informiert das Programm darüber, daß diese Funktionen existieren und die Definitionen an einer anderen Stelle im Programm stehen - entweder nachfolgend in derselben Datei oder in einer eigenen Datei. Eine Funktion muß in einem Programm immer zuerst deklariert werden, bevor sie aufgerufen werden kann. Dies läßt sich mit einer Vorausdeklaration sicherstellen. Wert1 und Wert2 werden als symbolische Variablen bezeichnet. Variablen müssen ebenfalls bekannt gemacht werden, bevor sie verwendet werden dürfen. v1 und v2, die als formale Parameter bezeichnet werden, bilden die Parameterliste von Nimm2() und Max(). Wert ist der einzige Parameter für Max_schreiben(). Bei der Compilierung und Ausführung liefert das Programm (nach Eingabe der Werte 17 und 125) obige Ausgabe. Die beiden vom Anwender eingegebenen Werte 17 und 125 sind durch ein Leerzeichen getrennt. Leerzeichen, Tabulatoren und Zeilenvorschubzeichen für eine neue Zeile werden als Whitespace-Zeichen zusammengefaßt. Der Eingabeoperator >> berücksichtigt nicht alle eventuell vorhandenen Whitespace-Zeichen.

Ein dritter vordefinierter Ausgabestrom iostream cerr liefert eine Fehlermeldung. cerr wird zur Information des Anwenders in Ausnahmesituationen verwendet. Zum Beispiel weist der folgende Programmabschnitt auf eine Division durch Null hin:

```
if(v2 == 0)
  cerr << "\n Fehler: Division durch Null versucht";
else
  cout << v1 / v2;
```

Beispiel 9.2

Das folgende Programm liest solange jeweils ein Zeichen von der Tastatur ein, bis das Zeichen „“ eingelesen wird. Es zählt die Anzahl der Zeichen und Zeilen, die eingelesen wurden. Die Ausgabe hat folgenden Aufbau:*

Zeilenanzahl Zeichenanzahl

1. **Programm**

```
#include <iostream.h>

int main() {
  char Z;
  int Zeilenanzahl=0, Zeichenanzahl=0;

  Z='0';
  while(Z != '*') {
    cin.get(Z);
    switch(Z) {
    case '\t':
```

```
      case ' ':
        break;
      case '\n':
        ++Zeilenanzahl;
        break;
      default:
        ++Zeichenanzahl;
        break;
      }
    }
    cout << Zeilenanzahl << "  " << Zeichenanzahl << "\n";
    return 0;
  }
```

2. Ergebnis

```
a
b


c
*
5 4
```

3. Anmerkungen get() ist eine iostream-Funktion, die ein einzelnes Zeichen einliest und es als Parameter übergibt, in diesem Fall Z. Bei jedem gelesenen Zeichen für eine neue Zeile wird der Wert von Zeilenanzahl um eins erhöht. Zeichenanzahl wird jedesmal um eins erhöht, wenn ein anderes Zeichen als ein Leerzeichen, Tabulator oder Zeilenvorschubzeichen eingelesen wird.

Weiterhin hält C++ für die Definition einer Konstante das Terminalwort **const** bereit, z.B.:

```
const int x = 5;
```

Funktionsaufrufe kosten Zeit, deshalb besteht in C++ die Möglichkeit eine Funktion *inline* zu definieren:

```
inline void printstr(char* s) {
  printf("%s", s);
}
```

Die Angabe von **inline** ist ein Hinweis an den Compiler den Funktionsaufruf nach Möglichkeit *inline* zu generieren, anstatt den Code einmal anzulegen und diesen über den Funktionsaufrufmechanismus abzuarbeiten.

9.4 Klassen

Mit Hilfe der Klassen kann sich der Programmierer in C++ seine eigenen Datentypen definieren. Eine Klasse in C++ erklärt sich durch vier Eigenschaften:

1. Die Klasse beinhaltet eine Menge von Datenkomponenten, mit denen die Daten der Objekte dargestellt werden. Dies können null oder mehrere Datenkomponenten eines beliebiges Typs sein.

2. Eine Klasse beinhaltet eine Zusammenstellung von Methoden. Diese stellen die möglichen Operationen dar, die auf Objekten dieser Klasse ausgeführt werden können. Eine Klasse kann null oder mehr Methoden enthalten.

3. In der Klasse gibt es verschiedene Zugriffsebenen. Die Komponenten einer Klasse können als **private**, **protected** und **public** definiert werden. Diese Abstufungen regeln während des Programmablaufs den Zugriff auf die Komponenten.

4. Jede Klasse hat eine Definitionsbezeichnung, die bei einer benutzerdefinierten Klasse als Typangabe verwendet werden kann.

Syntax (class):

```
class typname {
  // Zugriffsebene der globalen Variablen und Methoden
public:
  // Deklaration von Variablen und Methoden
  // Zugriffsebene der lokalen Variablen und Methoden
private:
  // Deklaration von Variablen und Methoden
  // Zugriffsebene der versteckten, lokalen Variablen und Methoden
protected:
  // Deklaration von Variablen und Methoden
};
```

Der Kopf der Klassenbezeichnung besteht aus dem Schlüsselwort **class** und aus dem Namen für den Typ der Klasse. Der Name wird nach den bekannten Regeln für Bezeichner gebildet. Der Körper der Klasse befindet sich zwischen den geschweiften Klammern. Er ist in drei Zugriffsebenen gegliedert, deren Namen **public**, **private** und **protected** sind. Die Deklaration der Klasse endet mit „;“. Innerhalb der Zugriffsebenen werden die Komponenten in bekannter Weise wie in der Sprache C deklariert.

Die Methoden zur Bearbeitung der Variablen des Objekts sind innerhalb des Klassenkörpers definiert oder in dem Klassenkörper werden nur ihren Prototypen gezeigt. Im zweiten Fall werden die Methoden außerhalb des Klassenkörpers definiert. Methoden der Klasse unterscheiden sich von normalen Funktionen durch folgende Eigenschaften:

- Sie haben freien Zugriff auf **public**- und **private**-Komponenten der Klasse, während normale Funktionen nur auf die **public**-Elemente der Klasse Zugriff haben. Die **private**-Variablen können nur durch die Methoden der Klasse verändert werden. Diesen Mechanismus nennt man *Informationsschutz (information hiding)*. Dadurch erfährt die interne Darstellung eines Klassenobjekts eine *Kapselung*. Damit wird der Anwender davor bewahrt, diese Darstellung versehentlich zu verändern. Genauso wichtig ist, daß der interne Zustand des Klassenobjektes vor einer zufälligen Programmänderung geschützt wird. Nur wenige Funktionen dürfen in die Objektvariablen schreiben. Wenn es zu einem Fehler kommt, beschränkt sich der Bereich, der durchgesucht werden muß, auf diese Funktionen. Methoden, die den Zugriff auf die ansonsten privaten Variablen unterstützen, sind ausschließlich klasseneigene, also *Memberfunktionen*.

- Methoden sind innerhalb des Bezugsrahmens ihrer Klasse definiert, während sich normale Funktionen im Bezugsrahmen des Programms befinden.

Die Definition einer Memberfunktion außerhalb des Klassenvereinbarung hat die folgende Syntax:

Syntax (Memberfunktion):

```
typ_zurueck klassen_name::methoden_name(parameterliste) {
  // Körper der Methode
}
```

Die einzelnen Komponenten haben dabei folgende Bedeutung:

typ_zurueck:	Typ des Rückgabewertes der Methode
klassen_name:	Name der Klasse, zu der die Methode gehört
methoden_name:	Name der Methode
parameterliste:	Liste der formalen Parameter in der Methode.

Bei dieser Definition gibt es eine Besonderheit. Man muß den *Bezugsoperator (scope operator)* „::“ verwenden, um zu zeigen, daß die entsprechende Methode zur gegebenen Klasse gehört. Der Zugriff auf die Variablen, die sich im **public**-Bereich befinden, erfolgt über den Variablennamen, gefolgt von dem „.“-Operator und dem Methodenkopf.

Beispiel 9.3
Es ist eine Klasse zu definieren, durch die folgende Aufgabe gelöst werden kann: Speichern der Namen und Anschriften von Telefonteilnehmern, der Dauer der Gespräche und der Telefonkosten. Um diese Situation durchspielen zu können, nutzen wir die Klasse phone*. Die Daten für die Teilnehmer werden in der* **private***-Variablen gespeichert. Vier Methoden werden definiert. Die erste liefert die Anfangswerte der* **private***-Variablen und die nächsten drei verwirklichen den Zugriff darauf.*

1. Programm

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

#define LIM 80

class phone {
private:
  unsigned long key;
  double sum;
  char name[LIM];
  char address[LIM];
  char date[LIM];
public:
  void init(unsigned long k, double s, char* na, char* ad, char* da);
  unsigned long phone_num() {return key;}
  double phone_tax() {return sum;}
  void phone_out();
};

void phone::init(unsigned long k, double s, char* na, char* ad,
```

```
                    char* da) {
  key = k;
  sum = s;
  strcpy(name, na);
  strcpy(address, ad);
  strcpy(date, da);
}

void phone::phone_out() {
  cout << "Nummer: " << key << " Summe: " << sum << "\n";
  cout << "Name:       " << name << "\n";
  cout << "Anschrift: " << address << "\n";
  cout << "Datum:      " << date << "\n";
}

int main() {
  unsigned long k;
  double s;
  char na[LIM], ad[LIM], da[LIM];

  // Prototyp der Funktion initial
  void initial(unsigned long*, double*, char*, char*, char*);
  phone First, Second;

  initial(&k, &s, na, ad, da);
  // Zugriff auf die Komponentenfunktion init() der Klasse First
  First.init(k, s, na, ad, da);

  initial(&k, &s, na, ad, da);
  // Zugriff auf die Komponentenfunktion init() der  Klasse Second
  Second.init(k, s, na, ad, da);
  cout << "\n";

  // Zugriff auf die Komponentenfunktion phone_out() der Klasse First
  First.phone_out();
  cout << "Erste Telefonnummer: " << First.phone_num() << "\n";
  cout << "Erste Summe:         " << First.phone_tax() << "\n";
  cout << "\n";

  // Zugriff auf die Komponentenfunktion phone_out() der Klasse Second
  Second.phone_out();
  cout << "Zweite Telefonnummer: " << Second.phone_num() << "\n";
  cout << "Zweite Summe:         " << Second.phone_tax() << "\n";
  return 0;
}

void initial(unsigned long* i, double* f, char* p, char* q, char* s) {
  cout << "Telefonnummer: ";
  cin >> *i;
  cout << "Summe:         ";
  cin >> *f;
  cout << "Name:          ";
  cin >> p;
  cout << "Anschrift:     ";
```

```
  cin >> q;
  cout << "Datum:          ";
  cin >> s;
}
```

2. Ergebnis

```
Telefonnummer: 11
Summe:         120
Name:          Paul
Anschrift:     Magdeburg
Datum:         25.03.1998
Telefonnummer: 12
Summe:         200
Name:          Nikolov
Anschrift:     Sofia
Datum:         25.02.1998

Nummer: 11 Summe: 120
Name:      Paul
Anschrift: Magdeburg
Datum:     25.03.1998
Erste Telefonnummer: 11
Erste Summe:         120

Nummer: 12 Summe: 200
Name:      Nikolov
Anschrift: Sofia
Datum:     25.02.1998
Zweite Telefonnummer: 12
Zweite Summe:         200
```

3. Anmerkungen Zwei der Methoden, phone_number und phone_tax, sind im Klassenkörper definiert. Die Definition der zwei anderen erfolgt außerhalb. Zur Ein- und Ausgabe von Daten werden die Standardfunktionen scanf, printf, sowie die Methoden und Operatoren von cin und cout genutzt. In der Funktion main ist der Prototyp der Methode initial verwendet. Man muß die Includedatei `iostream.h` im Programmkopf importieren, da dort cin und cout definiert sind.

Beispiel 9.4
Es ist ein Programm zu schreiben, das die arithmetischen Operationen „+“ (Addition), „“ (Multiplikation) und die Berechnung des Betrages komplexer Zahlen durchführen kann.*

Zuerst werden wir eine Klasse deklarieren, durch die die komplexen Zahlen definiert werden können. Weiterhin werden Methoden für folgenden Operationen benötigt: input *für die Übernahme der Anfangswerte von komplexen Zahlen durch die Methode* add, mult *und* modul *für die Addition, Multiplikation und Berechnung des Betrages* accs_real *und* accs_imag *für Zugriff zu den realen und imaginären Teil der komplexen Zahlen.*

1. Mathematisches Modell

$$z = a + ib$$

$$z_1 + z_2 = (a_1 + a_2) + i(b_1 + b_2)$$

$$z_1 * z_2 = (a_1 * a_2 - b_1 * b_2) + i(a_1 * b_2 + a_2 * b_1)$$

$$|z| = \sqrt{a^2 + b^2}$$

2. Programm

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

class compl {
private:
  double real, imag;
public:
  void input(double, double);
  compl add(compl, compl);
  compl mult(compl, compl);
  double modul(compl);
  double accs_real() {return real;}
  double accs_imag() {return imag;}
};

void compl::input(double a, double b) {
  real = a;
  imag = b;
}

compl compl::add(compl x, compl y) {
  compl z;
  z.real = x.real + y.real;
  z.imag = x.imag + y.imag;
  return z;
}

compl compl::mult(compl x, compl y) {
  compl z;
  z.real = x.real * y.real - x.imag * y.imag;
  z.imag = x.real * y.imag + y.real * x.imag;
  return z;
}

double compl::modul(compl x) {
  return sqrt(x.real*x.real + x.imag*x.imag);
}

void initial(double *p, double *q) {
  cout << "Realer Teil = ";
  cin >> *p;
  cout << "Imaginaerer Teil = ";
  cin >> *q;
  cout << "t = a+ib = " << *p << "  +i" << *q << "\n";
}

int main() {
```

```
  double a, b;
  compl u, v, t;

  initial(&a, &b);
  u.input(a,b);
  initial(&a, &b);
  v.input(a,b);

  cout << "\nErgebnis der Addition \n";
  t = t.add(u,v);
  cout << "(x+y)_real = " << t.accs_real();
  cout << "   (x+y)_imag = " << t.accs_imag() << "\n";

  cout << "\nErgebnis der Multiplikation \n";
  t = t.mult(u,v);
  cout << "(x*y)_real = " << t.accs_real();
  cout << "   (x*y)_imag = " << t.accs_imag() << "\n";
  cout << "Betrag = " << t.modul(u) << "\n";
  return 0;
}
```

3. Ergebnis

```
Realer Teil = 1
Imaginaerer Teil = 2
t = a+ib = 1  +i2
Realer Teil = 1
Imaginaerer Teil = 2
t = a+ib = 1  +i2

Ergebnis der Addition
(x+y)_real = 2   (x+y)_imag = 4

Ergebnis der Multiplikation
(x*y)_real = -3   (x*y)_imag = 4
Betrag = 2.23607
```

4. Anmerkungen Die zu bearbeitenden komplexen Zahlen werden über Tastatur mit der Funktion initial eingegeben. Die Variablen a und b sind für die realen und imaginären Teile der Zahlen reserviert. Die Methode input übernimmt die Werte der **private**-Variablen von den komplexen Objekten u und v. Die Methoden berechnen den realen und imaginären Teil der Zahl und das Ergebnis ist wiederum eine komplexe Zahl.

9.5 Bezugsrahmen eines Programms

Es wurde bereits erwähnt, daß jeder Bezeichner in einem Programm eindeutig sein muß. Das bedeutet nicht, daß ein Bezeichner in einem Programm nur einmal verwendet werden darf. Ein Name kann auch wiederholt verwendet werden, solange der Zusammenhang

für die einzelnen Verwendungsformen eindeutig ist. Um Eindeutigkeit der verschiedenen Programmeinheiten in dem Programm zu erreichen, muß man die folgenden Prinzipen beachten:

- Funktionen sind Programmeinheiten, die nur in einem übergeordneten Bezugsrahmen definiert werden können. Jede Funktion hat aber auch einen lokalen Bezugsrahmen, nämlich ihren Körper.
- Die Klassen können in dem lokalen oder in dem globalen Bereich definiert werden. Die Klasse hat wiederum selbst einen eigenen lokalen Bezugsrahmen, ihren Körper.
- Der lokale Bezugsrahmen wird durch den entsprechenden Programmblock definiert. Der Block wird durch die beiden Klammern „{“ und „}“ abgegrenzt.
- Man kann die lokalen Bezugsrahmen einbinden. Auf dem höchsten Niveau ist der globale Bereich der Quelle.
- Bei der Bestimmung der Wirkungsbereiche der Namen muß man mit dem lokalen Bezugsrahmen beginnen, wo der Name zum Erstenmal auftaucht. Der Wirkungsbereich beginnt mit seiner Definition und endet dort, wo der lokale Bezugsrahmen endet.

Bei der Bestimmung der Wirkungsbereiche der Programmeinheiten sind folgende Regeln anwendbar:

- Die Objekte der lokalen Klassen sind nur gültig in dem Körper der Funktion, wo die entsprechende Klasse deklariert ist.
- Die Methoden der lokalen Klasse müssen im Klassenkörper bezeichnet werden.
- Konflikte zwischen bei gleichen Namen werden dadurch entschieden, daß die Variable im innersten Bereich die höchste Priorität hat.

Beispielsweise sei die folgende Situation gegeben:

```
void fun(int glob, int b) {
  int k = glob;

  class Local {
  private:
    int glob; // Lokaler Name hat höhere Priorität
  public:
    // Es wird die private-Variable glob verwendet.
  int method() {return glob;};
  };
}
```

9.6 Initialisierung von Klassen

9.6.1 Initialisierung von Klassenobjekten mittels Konstruktoren

Ein Klassenobjekt wird durch die Anfangsbelegung seiner Datenkomponenten initialisiert. Vorausgesetzt, daß alle Komponenten **public** sind, kann das Objekt mit einer durch Kommata abgetrennte Liste von Werten in Klammern initialisiert werden:

```
class Wort {
public:
      int Haeufigkeit;
      char *String;
};

// explizite Initialisierung der Elemente des Objektes
Wort Suche = (0, "Rosenknospe");
```

Allgemein unterstützt C++ die automatische Initialisierung von Klassenobjekten. Eine spezielle Methode, der *Konstruktor*, wird jedesmal implizit vom Compiler aufgerufen, wenn ein Klassenobjekt definiert oder mit Hilfe des Operators **new** reserviert wird. Der Konstruktor ist eine benutzerdefinierte Initialisierungsfunktion, die den Namen der Definitionsbezeichnung der Klasse trägt. Eine Klasse darf mehr als einen Konstruktor haben, aber der Unterschied zwischen den Konstruktoren liegt in ihren formalen Parametern. Den Konstruktor, der keine formale Parameters hat, nennt man parameterlos. Die Deklaration eines Konstruktors erfolgt nach folgender Syntax:

Syntax (Konstruktordeklaration):

```
klassen_name::klassen_name(parameterliste) {
  Anweisungen;
}
```

Dabei ist klassen_name der Name der Klasse und parameterliste die Bezeichnung der formalen Parameter. Die Definition eines Konstruktors ist nicht komplizierter als die einer Methode. Der Konstruktor für Wort lautet zum Beispiel:

```
Wort::Wort(const char *Str, int i) {
  String = new char[strlen(Str) + 1];
  assert(String != 0);
  strcpy(String, Str);
  Haeufigkeit = i;
}
```

Das Makro assert(**int** expression) dient zum Einfügen von Testpunkten in ein Programm. Ist der Ausdruck expression gleich Null bei Ausführung von assert, wird eine Fehlermeldung auf stderr ausgegeben:

`Assertion failed: `*`expression`*`, file `*`filename`*`, line `*`nnn`*

Anschließend wird das Programm beendet.

Beispiel 9.5
Anwendung von Konstruktoren.

1. Programm

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

class Wort {
private:
  int Haeufigkeit;
  char *String;
public:
```

```
  Wort(const char*, int i = 0);  // Konstruktordeklaration
  void Write() {
    cout << "string = " << String << " Haeufigkeit = " << Haeufigkeit;
    cout << "\n";
  }
};

Wort::Wort(const char *Str, int i) {
  String = new char[strlen(Str) + 1];
  assert(String != 0);
  strcpy(String, Str);
  Haeufigkeit = i;
}

Wort Suche = Wort("Rosenknopse");  // globale Objekte
Wort *Ant_Zeiger = new Wort("Schlitten", 1);

int main() {
  Wort Film = Wort("Citizen_Kane", 0);
  Wort Regisseur = "Orson Wells";
  Suche.Write();
  Ant_Zeiger->Write();
  Film.Write();
  Regisseur.Write();
  return 0;
}
```

2. Ergebnis

```
string = Rosenknopse Haeufigkeit = 0  string = Schlitten
Haeufigkeit = 1
string = Citizen_Kane Haeufigkeit = 0  string = Orson Wells
Haeufigkeit = 0
```

3. Anmerkungen Der Konstruktor kann keinen Rückgabetyp haben oder explizit einen Wert zurückgeben. Ansonsten ist die Definition eines Konstruktors mit einer normalen Methode identisch. In diesem Fall erfordert der Konstruktor für Wort ein Argument des Typs **const char** *. Ein zweites Argument mit dem Typ `int` kann optional angegeben werden. Obige Beispiele zeigen, wie ein Wort-Objekt mit diesem Konstruktor definiert werden könnte.

9.6.2 Destruktoren

C++ unterstützt einen den Konstruktoren entsprechenden Mechanismus zum automatischen „Aufräumen“ von Klassenobjekten. Eine spezielle, vom Benutzer definierte Methode, als Destruktor bezeichnet, wird jedesmal aufgerufen, wenn ein Objekt den Bezugsrahmen verläßt oder der Operator `delete` auf einen Zeiger einer Klasse angewendet wird. Wenn

eine Referenz eines Klassenobjektes aus dem Bezugsrahmen fällt, wird kein Destruktor aufgerufen. Dies liegt daran, daß eine Referenz im Grunde genommen als ein anderer Name für ein bereits definiertes Objekt verwendet wird und selbst eigentlich kein Klassenobjekt darstellt. Man könnte den Destruktor für Wort so schreiben:

```
class Wort {
public:
  ~Wort();     // Destruktor von Wort
};

Wort::~Wort() {
  delete string;
}
```

Eine Methode wird als Destruktor einer Klasse gekennzeichnet, indem man vor die Definitionsbezeichnung der Klasse eine Tilde „~“ schreibt. Ein Destruktor kann niemals ein Argument haben. Jede Klasse darf nur einen Destruktor haben. Es darf kein Rückgabetyp angegeben oder ein Ergebnis zurückgeliefert werden. Ein Konstruktor darf niemals Speicherplatz reservieren, sondern dient zum Initialisierung des Speicherplatzes, der für ein Klassenobjekt reserviert wurde. Genauso wird auch durch einen Destruktor kein Platz freigegeben. Statt dessen beendet der Destruktor die Lebensdauer des Klassenobjektes. Ein Destruktor wird bei der Anwendung von Zeigern auf Klassenobjekte, die den Bezugsrahmen verlassen, nicht automatisch aufgerufen. Vielmehr muß der Programmierer den **delete**-Operator explizit anwenden.

Beispiel 9.6
Komplexe Zahlen mit Initialisierungsteil. Das Beispiel soll mit Konstruktoren und Destruktoren komplettiert werden. Der Konstruktor liefert die Anfangswerte der **private***-Variablen und der Destruktor annulliert diese Werte am Ende des Programms.*

1. Programm

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

class compl {
  // Private Komponenten der Klasse
private:
  double real, imag;
public:
  // Erster Konstruktor. Initialisierung durch formale Parameter
  compl(double, double);
  // Zweiter Konstruktor. Initialisierung durch Tastatur
  compl(char* p);
  // Methode für Addition der komplexen Zahlen
  compl add(compl, compl);
  // Methode für Multiplikation der Komplexzahlen
  compl mult(compl, compl);
  // Methode Berechnung des Betrages
  double modul(compl);
  // Methode  - Zugriff auf den Realteil
  double accs_real() { return real; }
```

```
  // Methode - Zugriff auf den imaginären Teil
  double accs_imag() { return imag; }
  // Destruktor - annuliert die Werte der komplexen Zahl
  ~compl();
};

// Definition der Konstruktoren

compl::compl(double a, double b) {
  real = a;
  imag = b;
}

compl::compl(char* p) {
  double a;
  cout << p << "\n";
  cout << "realer Teil der Komplexzahl:     ";
  cin >> a;
  real = a;
  cout << "imaginärer Teil der Komplexzahl: ";
  cin >> a;
  imag = a;
  cout << "t = a +ib=" << real << " + i" << imag << "\n";
}

// Definition der Methoden

compl compl::add(compl x, compl y) {
  compl z (0, 0); // Anwendung des 1. Konstruktors
  z.real = x.real + y.real;
  z.imag = x.imag + y.imag;
  return z;       // Rückgabe der Komplexzahl z
}

compl compl::mult(compl x, compl y) {
  compl z (0, 0); // Anwendung des 1.Konstruktors
  z.real = x.real * y.real - x.imag * y.imag;
  z.imag = x.imag * y.real + x.real * y.imag;
  return z;       // Rückgabe der Komplexzahl z
}

double compl::modul(compl x) {
  return sqrt (x.real*x.real + x.imag*x.imag);
}

// Definition des Destruktors

compl::~compl() {
  cout << "Ausgang\n";
  cout << "t = a + ib=" << real << " + i " <<imag << "\n";
  real = imag = 0;
}

// Hauptprogramm Komplexzahlen
```

```
int main() {
  char *q = "\nAnwendung des 2. Konstruktors";
  // Initialisierung durch den ersten Konstruktor von z
  compl zero (0, 0);
  // Initialisierung durch einfache Zuweisung
  compl t = zero;
  // Initialisierung von u und v durch den ersten Konstruktor
  compl u(1, 2), v(10, 12);

  cout << "u = " << u.accs_real() << "+ i" << u.accs_imag() << "\n";
  cout << "v = " << v.accs_real() << "+ i" << v.accs_imag() << "\n";
  cout << "\nErgebnis nach der Addition ist : \n";

  // Addition der Zahlen u und v
  t = t.add (u, v);
  cout << "(x+y)_real = " << t.accs_real();
  cout << "  (x+y)_imag = " << t.accs_imag() << "\n";

  // Initialisierung durch den zweiten Konstruktor
  compl a(q), b(q);

  // Multiplikation der Zahlen a und b
  cout << "Ergebnis nach der Multiplikation ist: \n";
  t = t.mult (a, b);
  cout << "(x*y)_real = " << t.accs_real();
  cout << "  (x*y)_imag = " << t.accs_imag() << "\n";
  cout << "\nBetrag = " << t.modul(u) << "\n";
  return 0;
}
```

2. Ergebnis

```
u = 1+ i2
v = 10+ i12

Ergebnis nach der Addition ist :
Ausgang
t = a + ib=11 + i 14
Ausgang
t = a + ib=1 + i 2
Ausgang
t = a + ib=10 + i 12
Ausgang
t = a + ib=11 + i 14
(x+y)_real = 11  (x+y)_imag = 14

Anwendung des 2. Konstruktors
realer Teil der Komplexzahl:     2
imaginärer Teil der Komplexzahl: 2
t = a +ib=2 + i2

Anwendung des 2. Konstruktors
realer Teil der Komplexzahl:     1
```

```
imaginärer Teil der Komplexzahl: 2
t = a +ib=1 + i2
Ergebnis nach der Multiplikation ist:
Ausgang
t = a + ib=-2 + i 6
Ausgang
t = a + ib=2 + i 2
Ausgang
t = a + ib=1 + i 2
Ausgang
t = a + ib=-2 + i 6
(x*y)_real = -2  (x*y)_imag = 6
Ausgang
t = a + ib=1 + i 2

Betrag = 2.23607
Ausgang
t = a + ib=1 + i 2
Ausgang
t = a + ib=2 + i 2
Ausgang
t = a + ib=10 + i 12
Ausgang
t = a + ib=1 + i 2
Ausgang
t = a + ib=-2 + i 6
Ausgang
t = a + ib=0 + i 0
```

3. **Anmerkungen** Jedes Objekt muß durch einen der beiden Konstruktoren initialisiert werden. Das betrifft auch das Objekt `z`. Es ist durch eine einfache Zuweisung initialisiert. Man nutzt den Parameter `q`, um die Aktivierung des zweiten Konstruktors zu erreichen. Wenn man den Anweisungen des Destruktors folgt, zeigt sich, daß die Objekte in umgekehrter Reihenfolge freigegeben werden.

9.7 Der implizite Zeiger this

Üblicherweise wird auf das Objekt über seinen Namen und auf die Memberfunktionen durch Anfügen der Methoden mittels „.“ oder „−>“ Operator zugegriffen. Ein expliziter Zugriff auf das Objekt ist ebenfalls unter Verwendung von **this** als Zeiger auf das aktuelle Objekt möglich. Der **this**-Zeiger enthält die Adresse des Klassenobjektes, über das die Komponentenfunktion aufgerufen wird. Wir wollen unser Beispiel über die komplexen Zahlen so umarbeiten, daß die Anwendung des Zeigers **this** deutlich wird.

Beispiel 9.7
Komplexzahlen unter Verwendung von **this**.

1. Programm

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

class compl {
  // Private- Elemente der Klasse
private:
  double real, imag;
public:
  // Erster Konstruktor. Initialisierung durch formalen Parameter
  compl(double, double);
  // Methode für Addition der Komplexzahlen
  compl add(compl, compl);
  // Methode für Multiplikation der Komplexzahlen}
  compl mult(compl, compl);
  // Methode Berechnung des Betrages
  double modul();
  // Methode - Zugriff auf den realen Teil
  double accs_real() { return real; }
  // Methode - Zugriff auf den imaginären Teil
  double accs_imag() { return imag; }
};

// Definition der Konstruktoren

compl::compl(double a, double b) {
  real = a;
  imag = b;
}

// Definition der Methoden

compl compl::add(compl x, compl y) {
  real = x.real + y.real;
  imag = x.imag + y.imag;
  return *this; // Rückgabe der komplexen Zahl
}

compl compl::mult(compl x, compl y) {
  real = x.real * y.real - x.imag*y.imag;
  imag = x.real * y.imag  + y.real*x.imag;
  return *this; // Rückgabe der komplexen Zahl
}

double compl::modul() {
  return sqrt(real*real + imag*imag);
}

// Hauptprogramm Komplexzahlen
int main() {
  // Initialisierung durch den ersten Konstruktor von z
  compl zero(0, 0);
  // Initialisierung durch einfache Zuweisung
```

```
  compl t = zero;
  // Initialisierung durch den ersten Konstruktor von u und v
  compl u(1, 2), v(10, 12);

  cout << "u = " << u.accs_real() << "+ i" << u.accs_imag() << "\n";
  cout << "v = " << v.accs_real() << "+ i" << v.accs_imag() << "\n";
  cout << "Ergebnis nach der Addition ist : \n";

  // Addition der Zahlen u und v
  t = t.add(u, v);
  cout << "(x+y)_real = " << t.accs_real();
  cout << "   (x+y)_imag = " << t.accs_imag() << "\n";

  // Multiplikation der Zahlen u und v
  cout << "Ergebnis nach der Multiplikation ist: \n";
  t = t.mult(u, v);
  cout << "(x*y)_real = " << t.accs_real();
  cout << "   (x*y)_imag = " << t.accs_imag() << "\n";
  cout << "\nBetrag = " << t.add(u, v).modul() << "\n";
  return 0;
}
```

2. Ergebnis

```
u = 1+ i2
v = 10+ i12
Ergebnis nach der Addition ist :
(x+y)_real = 11   (x+y)_imag = 14
Ergebnis nach der Multiplikation ist:
(x*y)_real = -14   (x*y)_imag = 32

Betrag = 17.8045
```

3. Anmerkungen Die Methoden add und mult übergeben nicht die Zeiger **this** zurück. Das macht das Objekt t, dessen Methoden add und mult sind. Dieses Objekt ist von Typ compl. Das aktivierte Objekt in den Methoden ist t. Für das zurückgegebene Ergebnis wird ein temporäres Objekt hergestellt, in welches die neuen Werte geschrieben werden. Diese Werte werden danach durch einfache Zuweisung den Komponenten von t übergeben.

9.8 Vererbung

9.8.1 Einfache Vererbung

Die objektorientierte Programmierung erweitert die abstrakten Datentypen um das Konzept der Typ-/Untertyp-Beziehungen zwischen einzelnen Klassen. Dies wird mit einem Mechanismus erreicht, der als *Vererbung (inheritance)* bezeichnet wird. Statt gemeinsame Eigenschaften einer Klasse (Ober-, Super- oder Eltern-Klasse) immer neu zu implementieren,

kann eine Klasse (Unter-, Sub- oder Kind-Klasse) Datenkomponenten und Komponentenfunktionen anderer Klassen erben. In C++ ist die Vererbung in Form einer Klassenableitung implementiert. Bei der einfachen Vererbung erbt die abgeleitete Klasse nur von einer Oberklasse. Jede beliebige Klasse kann auch Oberklasse für eine andere sein. Die Unterklasse kann in ihrem Körper neue, folgende Deklarationen enthalten:

- eigene Komponenten,
- modifizierte Methoden der Oberklasse,
- neue Methoden.

Es ist sehr wichtig zu wissen, daß die Unterklasse nicht die Daten der Oberklasse, sondern das Modell erbt. Die Deklaration einer Sub-Klasse hat folgende Syntax:

Syntax (Vererbung):

```
class NameUnterklasse : zugriffsart NameOberklasse {
  // Körper der Unterklasse
};
```

Die einzelnen Komponenten haben dabei folgende Bedeutung:

NameUnterklasse:	Name der abgeleiteten Klasse
zugriffsart:	eines der Schlüsselwörter **public**, **protected** oder **private**. Dieses Argument bestimmt die Zugriffsart auf die vererbten Komponenten
NameOberklasse:	Name einer schon bezeichneten Oberklasse
Körper der Unterklasse:	Definition der abgeleiteten Klasse

Um eine vererbte Klasse abzubilden, sind folgende Erweiterungen der Klassensyntax nötig:

- Der Kopf einer Klasse muß so verändert werden, daß eine Ableitungsliste mit den vererbten Klassen angegeben werden kann. Der Doppelpunkt hinter NameUnterklasse gibt an, daß diese Klasse von einer oder mehreren Klassen angeleitet ist. Das Schlüsselwort, z.B. **public**, gibt an, daß NameUnterklasse eine Klasse ist, deren Komponenten frei zugänglich sind. Ableitungen können entweder **public** oder **private** erfolgen. Die Angabe betrifft die Verfügbarkeit der vererbten Komponenten in der abgeleiteten Klasse.
- Weiterhin gibt es die Zugriffsart **protected**. Eine **protected**-Klassenkomponente verhält sich in einer abgeleiteten Klasse wie eine **public**-Komponente.

Beispiel 9.8
Definition einer Klasse Base und Ableitung einer weiteren Klasse Inherit.

```
class Base { // Oberklasse
private:
  int k;
  char *ptr;
public:
  int method();
};
```

```
class Inherit : public Base {
double f;
};
```

9.8.2 Mehrfache Vererbung

Wenn eine abgeleitete Klasse mehr als eine Basisklasse erbt, dann kann diese Situation nicht durch den Mechanismus der einfachen Vererbung modelliert werden. Wir wollen die Vererbungshierarchie in Abbildung 9.3(a) betrachten. Dort sind zwei unabhängige Zweige A-B und A-C dargestellt. Wenn man diese unabhängigen Zweige durch die einfache Vererbung verbinden muß, muß noch eine Klasse D am Anfang eingeführt werden (Abbildung 9.3(b)). Die Vererbung würde effektiver sein, wenn die mehrfache Vererbung benutzt wird Abbildung 9.3(c). Die Deklaration der Klassen erfolgt folgendermaßen:

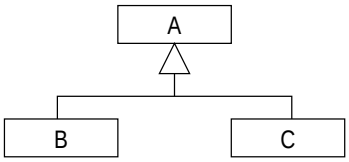
```
class A {public: int k; void fun() ... };
class B: virtual public A {public: int k; void fun() ... };
class C: virtual public A { ... };
class D: public B, public C { ... };
```

In diesem Fall wird die Klasse A durch zwei Zweige A-C und A-B vererbt. Um in den beiden Zweigen die gleiche Schablone von Klasse A zu nutzen, ist sie als **virtual** angemeldet. Die Komponentenfunktion der Klasse D kann die vererbte **public**-Komponenten der Klasse A direkt anwenden. Wenn die Klasse nicht als **virtual** bezeichnet wäre, müßte die Vererbung wie in Abbildung 9.3(d) aussehen. Man nutzt zwei verschiedene Kopien des Objektes A.

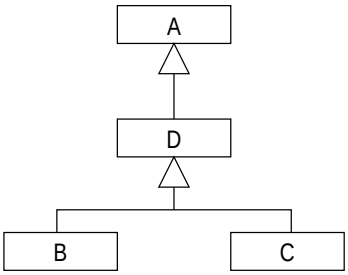
Beispiel 9.9
Es soll eine Vererbungshierarchie aufgebaut werden, in der die Klasse theta die Klassen beta und alpha erbt. Die Konstruktoren geben entschprechende Mitteilungen auf dem Schirm aus. Es werden zwei Vererbungszweige (alpha−theta und beta−theta) realisiert.

1. Programm

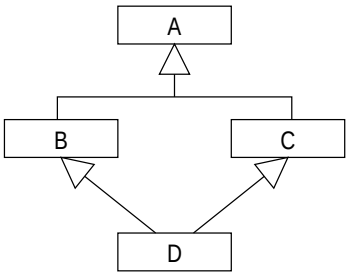
```
#include <iostream.h>

class alpha {
  int n;
  float x;
public:
  // Konstruktor der Klasse alpha
  alpha(int p=2) {
      n = p;
      x = 1;
      cout << "Konstructor alpha " << n << " " << x << "\n";
  }
};

class beta {
  int n;
  float y;
public:
```



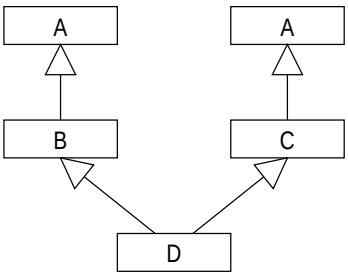
(a) Einfache Vererbung



(b) Einfache Vererbung und Verbindung der unabhängigen Zweige



(c) Mehrfache Vererbung mit virtuellem Objekt A



(d) Mehrfache Vererbung mit zwei Kopien von A

Abbildung 9.3: Einfache vs. Mehrfache Vererbung

```
  // Konstruktor der Klasse beta
    beta(float v=0.0) {
      n = 1;
      y = v;
      cout << "Konstructor beta " << n << " " << y << "\n";
    }
};

class teta : public alpha, public beta {
  // Private Variablen der Klasse teta
  int n;
  int p;
public:
  /* Konstruktor der Klasse teta und Initialisierung der beiden von
   * den Klassen alpha und beta gerbten Ableitungen. Übergabe der
   * Parameter für Initialisierung der privaten und vererbten
   * Ableitungen
   */
  teta(int n1=1, int n2=3, int n3=3, float v=0.0) : alpha(n1), beta(v) {
    n = n3;
    p = n1 + n2;
    cout << "Konstruktor teta " << n << " " << p << "\n";
  }
};

int main() {
  teta c1;
  teta c2(10, 11, 12, 5.0);
  return 0;
}
```

2. Ergebnis

```
Konstruktor alpha 1 1
Konstruktor beta 1 0
Konstruktor theta 3 4
Konstruktor alpha 10 1
Konstruktor beta 1 5
Konstruktor theta 12 21
```

Beispiel 9.10

Eine Vererbungshierarchie soll so wirken, daß die Klasse delta *die Klasse* alpha *über die Zweige (vgl. auch Abbildung 9.3(d))* alpha−beta−delta *und* alpha−theta−delta *erbt. Die Konstruktoren der Klassen geben entschprechende Mitteilungen auf dem Schirm aus.*

1. Programm

```
#include <iostream.h>

class alpha {
  int na;
  float x;
public:
```

```
  // Konstruktor der Klasse alpha
  alpha(int nn=1) {
    na = nn;
    cout << "Konstruktor alpha " << na << "\n";
  }
};

class beta : public alpha {
  float xb;
public:
  beta(float xx=0.0) {
    xb = xx;
    cout << "Konstruktor beta " << xb << "\n";
  }
};

class teta : public alpha {
  int nc;
public:
  teta(int nn=2) : alpha(2*nn+1) {
    nc = nn;
    cout << "Konstruktor teta " << nn << "\n";
  }
};

class delta : public beta, public teta {
  int nd;
public:
  // Initialisierung der geerbten und privaten Ableitung von delta
  delta(int n1, int n2, float x) : teta(n1), beta(x) {
    nd = n2;
    cout << "Konstruktor delta " << nd << "\n";
  }
};

int main() {
  delta d(10, 20, 5.0);
  return 0;
}
```

2. Ergebnis

```
Konstruktor alpha 1
Konstruktor beta 5
Konstruktor alpha 21
Konstruktor theta 10
Konstruktor delta 20
```

3. Anmerkung Die Klasse alpha ist nicht als **virtual** deklariert, deshalb unterstützt das Programm zwei verschiedene Ableitungen der Klasse alpha.

10 Grundlagen der Computergrafik

10.1 Einführung

Computergrafik beinhaltet die Erzeugung und Manipulation grafischer Bilder mit Hilfe eines Computers. Numerische Daten durch Bilder zu interpretieren, hat dazu beigetragen, dem Benutzer die Information klarer und verständlicher zu präsentieren, z.B. sind große Datenmengen in Tortengrafiken, Balkendiagrammen und Kurven darstellbar. Dabei folgt man der Erfahrung KONFUZIUS (551-478 v. Christi Geburt): „Ein Bild sagt mehr als tausend Worte!“

Bis 1980 waren Grafikmöglichkeiten noch stark begrenzt, erst die PC- und Workstation-Welt hat zum Aufschwung in der Grafiknutzung geführt. Die Ursache sind billigere Schaltkreise und Prozessoren, trotzdem sind anspruchsvolle Grafikgeräte noch teuer. Für die Darstellung eines einzigen Bildes auf einem Bildschirm mit einer Auflösung von 1024x1024 Pixeln im Echtfarbenmodus (3 Byte je Bildpunkt) wird ein sogenanntes Pixmap angelegt, das 1024x1024x3 Byte oder 3 MByte Speicherplatz im Bildwiederholspeicher benötigt. Für die Ausführung anspruchsvoller Grafikanwendungen werden neben der entsprechenden Speicherkapazität auch hohe Rechenleistungen gefordert. Heutige Hochleistungsworkstations aber auch viele PC erfüllen diese Voraussetzungen.

In der Computergrafik spielt die Ausgabe des Bildes auf dem Bildschirm eine entscheidende Rolle. Meistens bedient man sich dabei der Rastertechnik. Das bedeutet, daß der Bildschirm mit einem Gitternetz überzogen wird, wobei jeder Schnittpunkt als Punkt darstellbar ist und einzeln angesteuert werden kann. Man nennt ein solches Element ein *Pixel* (von Picture Element abgeleitet).

Die Qualität der erzeugten Bilder hängt vom Grad der Auflösung des Bildschirmes ab. Ab 1000 mal 1000 Pixeln empfindet man angenehme Bilder, jedoch ist damit eine hohe Videofrequenz verbunden, d.h., schnelle Bildwiederholspeicher und Ansteuerung sind notwendig.

Schwerpunkte der Grafikarbeit sind die Schaffung von Grundfunktionen für die Darstellung grafischer Objekte (Punkte, Linien, Kurven, Füllfunktionen) sowie Funktionen zur Manipulation dieser Objekte in der Ebene oder im Raum (Verschieben, Zoomen, Drehen). Die Ausführung derartiger Grundfunktionen erfolgt zunehmend durch leistungsfähige Grafikprozessoren. Diese implementieren die Funktionen in Hardware und können somit die CPU von aufwendigen Berechnungen für die Grafik befreien. Für den Zugriff auf die Hardwarefunktionen werden von den Herstellern entsprechende Treiber und Bibliotheken bereitgestellt. Standards, die solche Funktionen definieren, sind u.a. OpenGL und ActiveX.

Im folgenden sollen zunächst einige Anwendungsgebiete der Computergraphik vorge-

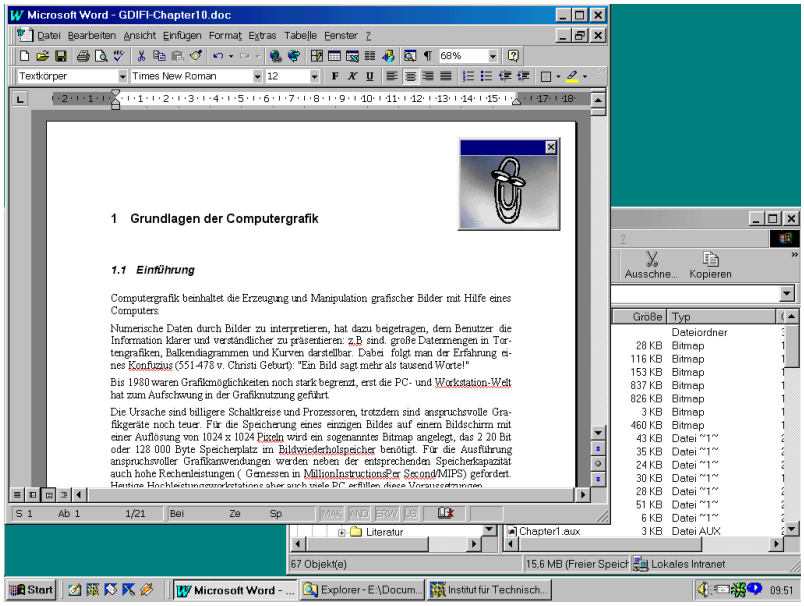


Abbildung 10.1: Benutzeroberfläche von Windows 98

stellt werden. Daran anschließend erfolgt die Vorstellung einiger grundlegender Algorithmen der Computergraphik sowie eine Einführung in OpenGL.

10.1.1 Benutzerschnittstellen

Fast alle Anwendungen auf PC und Workstation treten dem Nutzer mit einer grafischen Benutzeroberfläche entgegen. Arbeiten Sie unter Windows, dann kommt Ihnen der Schnappschuß auf der Abbildung 10.1 sicherlich bekannt vor.

10.1.2 Interaktive Darstellung von Daten

Eine weitverbreitete Anwendung der Computergrafik ist die zwei- oder dreidimensionale Darstellung der Graphen von mathematischen, physikalischen und ökonomischen Funktionen, Histogrammen, Balken- und Tortendiagrammen (siehe Abbildung 10.2), Zeitplänen, Produktionstabellen, usw. Diese Darstellungen erleichtern die Erklärung komplexer Zusammenänge und Entscheidungen.

10.1.3 Kartographie

Mittels Computergrafik können aus Meßdaten sowohl exakte, als auch schematisierte Darstellungen von Naturphänomenen erstellt werden. Beispiele für eine solche computergestützte Visualisierung sind Wetterkarten (Abbildung 10.3), geographische Karten und Reliefkarten.

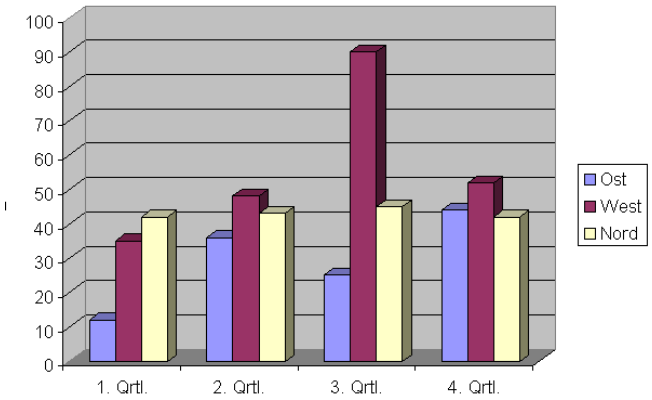


Abbildung 10.2: Beispiel einer grafischen Darstellung statistischer Werte

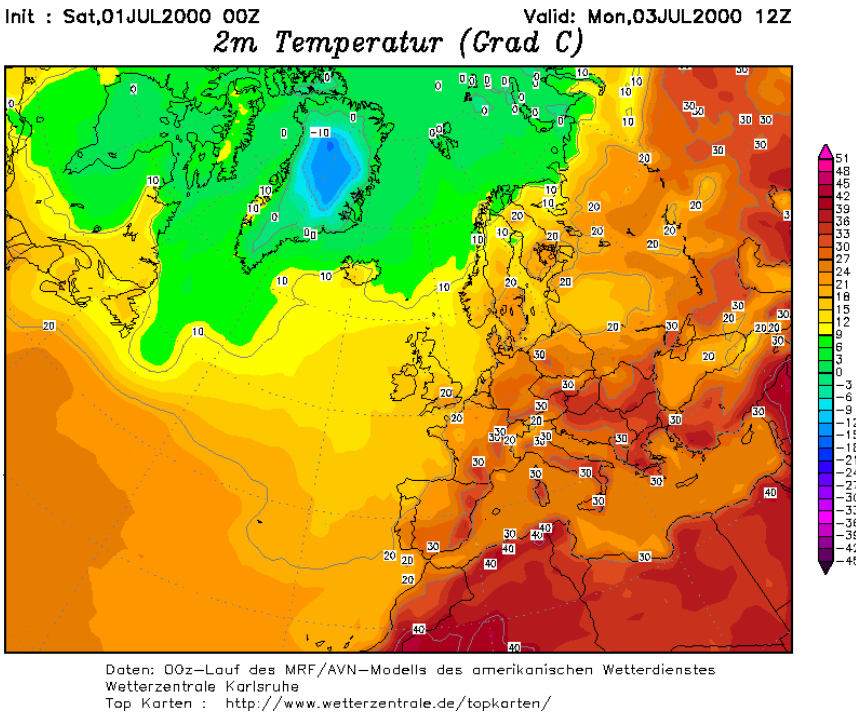


Abbildung 10.3: Darstellung der Ergebnisse einer Wettersimulation

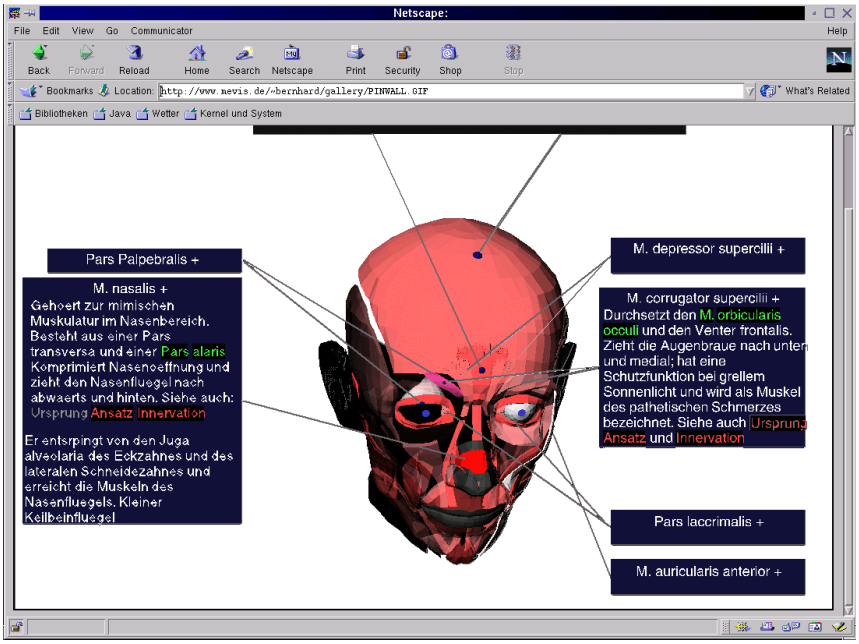


Abbildung 10.4: Interaktives, medizinisches Lehrmaterial

10.1.4 Medizin

Die Computergrafik hat viele Bereiche der Medizin erobert. Am bekanntesten sind die Anwendungen in der Diagnose (z.B. Computertomographie). Aber auch die Herstellung interaktiver grafischer Lernhilfen (Abbildung 10.4) in der Medizin erfahren einen großen Aufschwung.

10.1.5 Computerunterstützter Entwurf und Design

Unter Computer Aided Design (CAD) versteht man den Einsatz interaktiver Grafik für den Entwurf von Komponenten und Systemen für die verschiedensten Produkte (Maschinen, Schiffe, Flugzeuge, aber auch Bauten jeglicher Art). Im Mittelpunkt steht dabei die Darstellung geometrischer Modelle in zwei- und dreidimensionaler Form oder oftmals auch nur die Erstellung von Zeichnungen (Abbildung 10.5).

10.1.6 Multimedia

In Verbindung mit Text, Ton, Video wird Grafik integrierter Bestandteil vieler modernen Computeranwendungen. Das beste Beispiel aus der jüngeren Entwicklung ist das Internet und insbesondere des WWW (World Wide Web) mit seinem unerschöpflichen Angebot an Informationen (Abbildung 10.6).

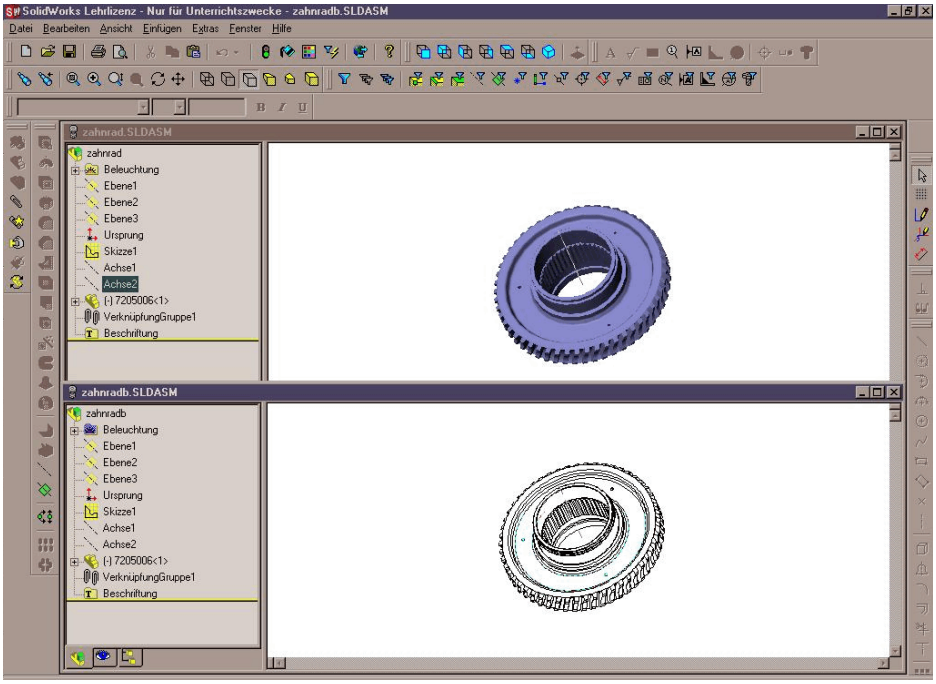


Abbildung 10.5: CAD-Darstellung

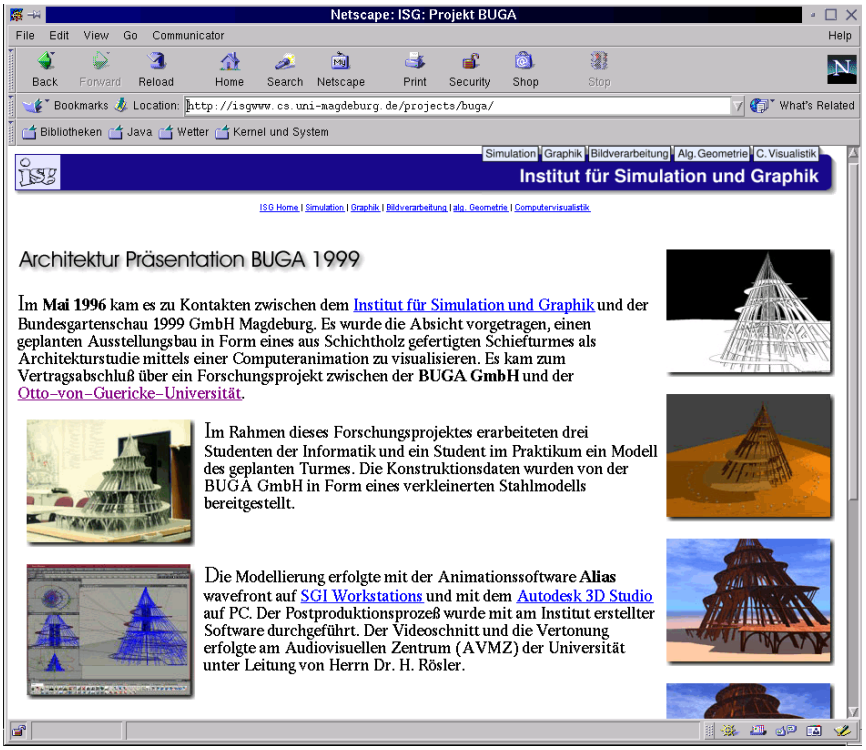


Abbildung 10.6: Anwendung von Multimedia im Word Wide Web

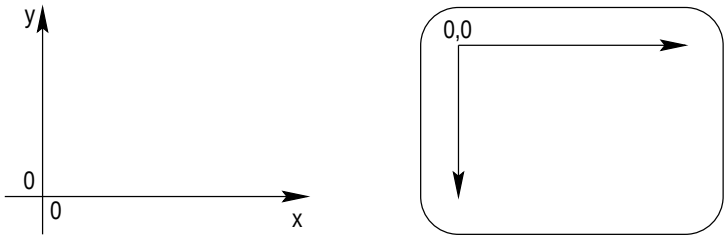


Abbildung 10.7: Koordinatensysteme

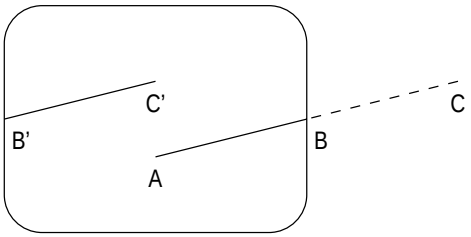


Abbildung 10.8: Überzeichnen eines Bildschirms

10.2 Zeichnen elementarer Figuren

Zur Ausgabe auf einen Rasterbildschirm müssen die Punkte, die das Bild ausmachen, im Bildpuffer definiert werden Dazu sind Punkt-Ausgabe-Algorithmen notwendig. Der Bildschirm entspricht dem auf den Kopf gestellten 1. Quadranten des uns bekannten Koordinatensystems (vgl. Abbildung 10.7). Zeichnet man über den Bildschirm hinaus, bricht das Programm ab oder es entsteht der „Wraparound"-Effekt (vgl. Abbildung 10.8).

10.2.1 Punkte

Das grundlegende Element, das bei der Darstellung von grafischen Figuren in der Computergrafik verwendet wird, ist das Pixel. Ein Pixel ist ein Rasterpunkt, der durch zwei Integer-Werte gekennzeichnet ist. Optional kann ein weiterer Parameter für die Farbzuweisung angegeben werden. Aus der maximalen Anzahl von Pixeln für Länge und Breite ergibt sich ein Pixelverhältnis, das bei der Präsentation berücksichtgt werden muß. Sonst kann es bespielsweise dazu kommen, daß aus ein Quadrat als Rechteck dargestellt wird. Die Umrechnung auf das vorliegende Verhältnis wird bereits durch die Grafiksoftware selbst vorgenommen.

Alle Grafikfunktionen lassen sich auf das setzen von Pixeln durch eine PlotPoint(x,y) Funktion zurückführen. Dabei wird nur ein Pixel aktiviert. In verschiedenen Programmiersprachen stehen unterschiedliche Implementierungen dieser Funktion zur Verfügung, z.B.:

BASIC:	PSET(x,y,F)
PASCAL:	PSET(x,y)
TURBO-PASCAL:	PutPixel(x,y)
C:	WritePixel (x, y, Farbe)

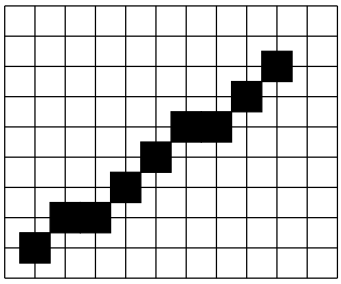


Abbildung 10.9: Gerade als Folge von Punkten

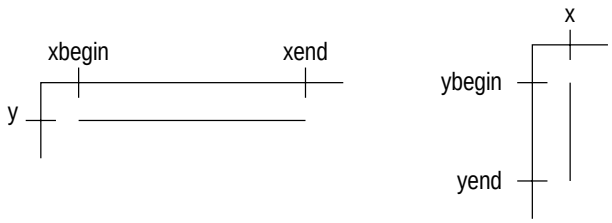


Abbildung 10.10: Erzeugung horizontaler und vertikaler Linien

10.2.2 Geraden

Geraden entstehen, wenn eine dichte Folge von benachbarten Pixeln erzeugt wird (vgl. Abbildung 10.9). Die Exaktheit der Geraden hängt dabei sehr stark von der Bildschirmauflösung ab. Da Pixel nur ganzzahlig angesprochen werden können, hat deren Berechnung nur für **int**-Werte einen Sinn. Außerdem sind **float**-Operationen sehr langsam. In der Grafik ist es aber unbedingt notwendig, schnelle Operationen zu programmieren.

Horizontale und vertikale Geraden

Die Erzeugung horizontaler und vertikaler Geraden zeichnet sich dadurch aus, daß jeweils eine der Koordinaten konstant bleibt, y bei vertikalen und x bei horizontalen Linien (vgl. Abbildung 10.10). Um die Linie zu erzeugen, genügt dann eine einfache **for**-Schleife, die jeweils von der Anfangs- zur Endkoordinate zählt:

```
/* Zeichnen einer horizontalen Linie */
for(x=xbegin; x<=xende; x++)
  WritePixel(x, y, Farbe);

/* Zeichnen einer vertikalen Linie */
for(y=ybegin; y<=yende; y++)
  WritePixel(x, y, Farbe);
```

Diagonale Geraden mit einem Anstieg von 45°

Diagonale Geraden können, wie horizontale und vertikale Linien, durch eine einfache **for**-Schleife erzeugt werden. Der Unterschied besteht darin, daß bei jedem Schleifendurchlauf

sowohl die x-, als auch die y-Koordinate um 1 erhöht werden:

```
for(x=xanfang, y=yanfang; x<xend; x++, y++)
  WritePixel(x,y,Farbe);
```

Diagonale Geraden mit beliebigem Anstieg

Beliebige Geradenanstiege sind problematischer, da jeweils die richtigen Pixel, die die Gerade annähernd abbilden, dargestellt werden müssen. Im folgenden sollen zwei Algorithmen zur Darstellung beliebiger Geraden vorgestellt werden.

Direkte Methode nach $y = mx + b$ Die Berechnung basiert auf der aus der Geometrie bekannten Geradengleichung. Da dieser Algorithmus die Koordinaten zunächst als **float**-Werte berechnet und erst vor der Darstellung in einen **int** konvertiert, ist die Berechnung verhältnismäßig langsam. Die Erzeugung der Linie erfolgt mittels folgender Funktion:

```
void line(int x0, int y0, int x1, int y1, int farbe) {
  int x;
  float dy, dx, y, m;
  dy=y1-y0;
  dx=x1-x0;
  m=dy/dx;
  y=y0;
  for(x=x0; x<=x1; x++) {
    WritePixel(x, (int)floor(y+0.5), farbe);
    y+=m
  }
}
```

Digital Differential Analyzer (DDA) Liefert gute Geraden. Für $|m| < 1$ und $xanf < xend$ erzeugen wir die Gerade, indem der x-Wert um jeweils eine Einheit bis zum Erreichen von $xend$ erhöht wird. Die Punkte werden nach folgenden Gleichungen berechnet:

$$x2 = x1 + 1 \quad \text{oder} \quad x2 - x1 = 1$$

und

$$y2 = y1 + m$$

Der auf diesen Formeln basierende Algorithmus ist zeitsparender, hat aber weiterhin den Nachteil, daß Operationen mit reellen Zahlen durchzuführen sind.

Beispiel 10.1
Zeichnen einer Geraden mit $xanf = 5.3$, $xend = 10.4$, $yanf = 1.0$ *und* $yend = 3.8$.

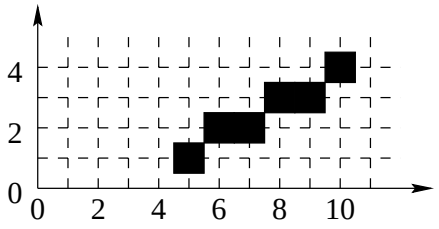
Für m ergibt sich:

$$m = \frac{3.8 - 1.0}{10.4 - 5.3} = 0.55 < 1.0$$

Damit ergeben sich folgende Werte für x und y:

x	Round(x)	y	Round(y)
5.3	5	1.0	1
6.3	6	1.55	2
7.3	7	2.1	2
8.3	8	2.65	3
9.3	9	3.2	3
10.3	10	3.75	4

Auf dem Bildschirm ergibt sich damit folgende Linie:



10.2.3 Kreise

Nutzung der Kreisgleichung

Die grundlegende Kreisgleichung lautet:

$$r^2 = (x - x_m)^2 + (y - y_m)^2$$

Daraus ergibt sich:

$$y = y_m \pm \sqrt{r^2 - (x - x_m)^2} \quad \text{mit} \quad x_m - r < x < x_m + r$$

Zur Darstellung auf dem Bildschirm ist abschließend wiederum eine Wandlung in ganzzahlige Werte notwendig. Dieses Verfahren ist einfach umzusetzen, aber auf Grund der komplexen Operationen, u.a. das häufige Quadrieren und Wurzelziehen, sehr rechenintensiv und daher langsam.

Parameterdarstellung zum inkrementellen Zeichnen

Die Parametergleichungen für den Kreis lauten:

$$x = x_m + r * \cos\alpha \quad \text{und} \quad y = y_m + r * \sin\alpha \quad \text{mit} \quad 0° \leq \alpha \leq 360°$$

$$x_1 = r * \cos\alpha \quad \text{und} \quad x_2 = r * \cos(\alpha + \delta) \quad \text{(vorher Mittelpunktlage hergestellt)}$$

$$y_1 = r * \sin\alpha \quad \text{und} \quad y_2 = r * \sin(\alpha + \delta)$$

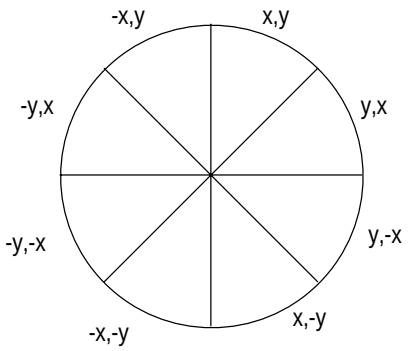


Abbildung 10.11: Symmetriedarstellung am Kreis

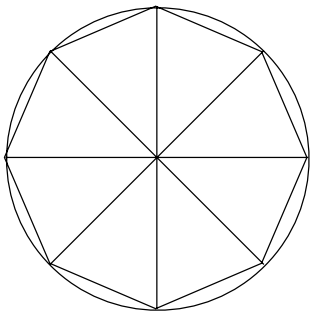


Abbildung 10.12: Approximation eines Kreises durch einen Polygonzug

Aus Additionstheorem folgt:

$$x_2 = x_1 * \cos\delta - y_1 * \sin\delta$$

$$y_2 = y_1 * \cos\delta + x_1 * \sin\delta$$

Mit $y_1 = 0$ und $x_1 = r$ sind für δ die Winkelfunktionen nur einmal zu berechnen und anschließend aufzuaddieren. Unter Ausnutzung von Symmetrieaspekten ist nur $\frac{1}{8}$ des Vollkreises zu berechnen (vgl. Abbildung 10.11).

Vieleck als Näherung eines Kreises

Kreise können durch viele Geradenabschnitte (Sehnen) angenähert werden (vgl. Abbildung 10.12). Die Anzahl der Abschnitte muß so groß sein, daß der Kreis erkennbar ist. Die Anzahl der Ecken soll proportional dem Umfang sein.

Bresenham-Algorithmus

Obwohl die vorgestellten Verfahren schon elegant sind, hat Bresenham ein noch schnelleres Verfahren entwickelt. Es basiert auf der direkten Pixelberechnung und benutzt die Kreissymmetrie. Der Algorithmus nutzt Integerberechnungen, so daß gegenüber den anderen Verfahren die Geschwindigkeit höher ist. Die Werte werden für $x = 0$ bis $x = \sqrt{\frac{r}{2}}$

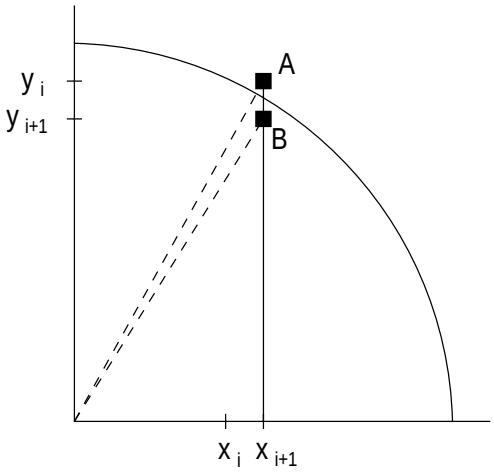


Abbildung 10.13: Bresenham-Algorithmus für Kreise

berechnet ($\frac{1}{8}$ des Kreises, anschließend dreimal spiegeln). Wenn $P(0, r)$ der Anfangspunkt ist und x um jeweils ein Pixel zunimmt, dann bleibt y gleich oder vermindert sich um ein Pixel. Wenn $P(x_i, y_i)$ ein Pixel auf dem Kreis darstellt, dann ist das nächste Pixel entweder $P(x_{i+1}, y_i)$ (A) oder $P(x_{i+1}, y_{i-1})$ (B) (vgl. Abbildung 10.13). Der Algorithmus muß nun entscheiden, ob A oder B verwendet werden. Ein Punkt liegt auf dem Kreis, wenn $d = 0$:

$$d_A = \sqrt{(x_i + 1)^2 + y_i^2} - r$$

$$d_B = \sqrt{(x_i + 1)^2 + (y_i - 1)^2} - r$$

$$S = d_A + d_B$$

$$S > 0 \implies \text{Pixel B}$$

$$S < 0 \implies \text{Pixel A}$$

10.3 Grafikgrundlagen

Für viele Anwendungen in der Praxis sind die bisher benutzten Koordinatensysteme zu beschränkt. Dazu kommt, daß die Darstellungsflächen und das Auflösungsvermögen des Bildschirms nicht ausreichen. Der Benutzer muß ein eigenes Koordinatensystem auswählen können und das Bild oder einen Ausschnitt des maßstäblich veränderten Bildes an jeder beliebigen Stelle des Bildschirms anzeigen lassen können. Hinzu kommen die verschiedenen Ansichten, z.B.:

- räumlich,
- Rißdarstellungen,
- innen/außen,

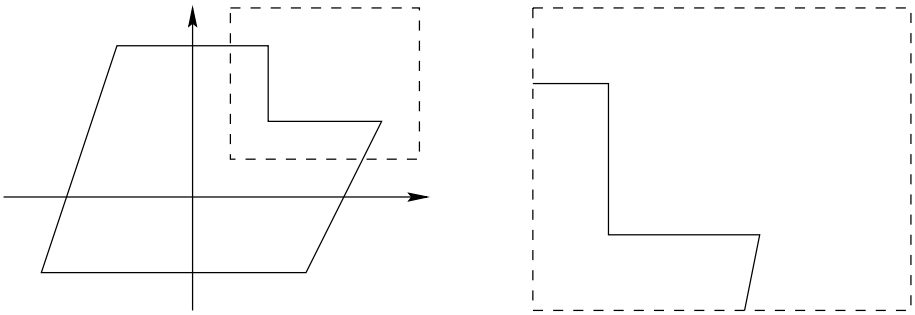


Abbildung 10.14: Darstellung des Fensters mit Objektausschnitt = viewport

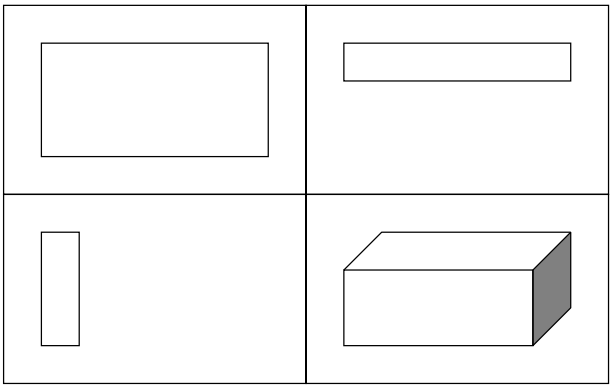


Abbildung 10.15: Zeichnung mit verschiedenen Ansichten

- Details(z.B. Verzahnung eines Getriebes).

Koordinatensysteme und Transformationen sind die wichtigsten Mittel zur flexiblen grafischen Arbeit. Auf dem Bildschirm ist das Koordinatensystem durch die Hardware vorgegeben. Weltkoordinatensystem:

- benutzerfreundliches Koordinatensystem,
- betriebssystemunabhängig,
- läßt alle Werte zu.

Durch Vergrößern, Verkleinern, Verschieben, Drehen und Verzerren sind die Bildschirmfenster beliebig auf das Objekt anwendbar (Abbildung 10.14). Damit können bequeme und informative Darstellungen erzeugt werden, ohne das Originalobjekt zu verändern. Verschiedene Fenster und Darstellungsbereiche ermöglichen eine Bildschirmanzeige mit mehreren Bildern (Abbildung 10.15).

Bei professionellen Lösungen wird mit einem normalisierten Gerätekoordinatensystem gearbeitet. Die Koordinaten bewegen sich in beiden Richtungen zwischen 0 und 1, damit werden die Grafikprogramme von der Auflösung des Bildschirms und der zugehörigen

Grafikkarte unabhängig. Das Betriebssystem hat einen Gerätetreiber, der die Normalkoordinaten in die konkreten Bildschirmkoordinaten umwandelt.

In der Vergangenheit wurden verschiedene Standards für Grafiksysteme entwickelt, z.B. das GKS (Graphic Kernel System) oder OpenGL (Abschnitt 10.5). Neben der Behandlung von Koordinaten sind folgende Probleme bei der grafischen Programmierung zu beachten:

- Clipping für Punkte, Geraden, Kreise, Polygone,
- Texte in Grafik.

10.4 Zweidimensionale geometrische Transformationen

Geometrische Transformationen sind Hilfen bei der Konstruktion und Modifikation eines Objektes. Folgende Operationen werden behandelt:

- Skalierung (Vergrößern, Verkleinern),
- Translation (Verschiebung),
- Rotation (Drehung),
- Scherung (Verzerrung),
- Spiegelung,
- Bewegung (Animation).

Basis für alle Operationen sind die Matrizenmultiplikation:

$$Vektor * Matrix$$

und die Einheitsmatrix

$$V = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Für die Berechnungen werden homogene Koordinaten verwendet. Der Vektor einer homogenen Koordinate enthält jeweils ein Element mehr als die Anzahl der Dimensionen des entsprechenden Punktes, d.h. 3 Elemente für einen Punkt im 2-dimensionalen Raum und vier Elemente für einen Punkt im 3-dimensionalen Raum. Das zusätzliche Element besitzt immer den Wert 1. Die homogene Koordinate eines Punktes $P(x, y)$ ist beispielsweise $P(x, y, 1)$. Durch dieses Vorgehen lassen sich die Berechnungen vereinfachen (siehe [Fol99]).

10.4.1 Translation

Unter Translation versteht man eine geradlinige Verschiebung eines Objektes bzw. seiner Definitionspunkte. Das Translationsergebnis wird allgemein nach folgender Gleichung berechnet:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ H & V & 1 \end{bmatrix}$$

Wobei H und V die Beträge sind, um die in horizontaler und vertikaler Richtung verschoben werden soll.

10.4.2 Rotation

Eine Rotation ist die Drehung eines Objektes um einen bestimmten Punkt. Sie wird üblicherweise durch den Rotationswinkel und das Rotationszentrum festgelegt. Liegt das Rotationszentrum nicht im Koordinatenursprung, dann muß vor und nach der eigentlichen Rotation jeweils eine Translation durchgeführt werden.

Die Rotation des Punktes $P(x, y)$ um den Winkel μ geschieht nach folgender Formel:

$$RK = MK * T$$

Wobei RK die Rotationskoordinaten, MK die Modellkoordinaten (Originalkoordinaten) und T die Rotationsmatrix darstellen. Die vollständige Formel lautet:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} * \begin{bmatrix} \cos\mu & \sin\mu & 0 \\ -\sin\mu & \cos\mu & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

10.4.3 Skalierung (Maßstabsveränderung)

Skalierung ist die Veränderung des Maßstabes eines Objektes um bestimmte Skalierungsfaktoren. Wenn $S_x > 1$ und $S_y > 1$, dann Vergrößerung des Objektes, wenn $S_x < 1$ und $S_y < 1$, dann Verkleinerung des Objektes. Wenn $S_x \neq S_y$, dann wird das Bild verzerrt. Für die Skalierung gilt die Gleichung:

$$SK = MK * T$$

bzw.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} * \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x * S_x \\ y * S_y \\ 1 \end{bmatrix}$$

10.4.4 Scherung, Verzerrung

Eine Scherungstransformation erzeugt die Verzerrung eines Objektes (Abbildung 10.16). Es gibt zwei Arten der Scherung (je nach Achsenrichtung):

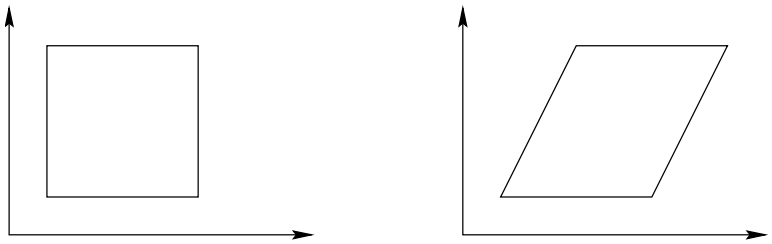


Abbildung 10.16: Scherung in x-Richtung

1. x-Scherung (SHX - Scherungsfaktor):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ SHX & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2. y-Scherung (SHY - Scherungsfaktor):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & SHY & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

10.4.5 Spiegelung

Objekte können an der x-Achse, der y-Achse oder an anderen Achsen gespiegelt werden. Es gibt zwei Arten der Spiegelung:

1. an der x-Achse:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2. an der y-Achse:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

10.4.6 Bewegung (Animation)

Animation ist die n-fache Reproduktion eines Objektes in mehreren Variationen. Die Realisierung erfolgt meist mit Translation, Rotation und Skalierung. Bei der Erzeugung von bewegten Objekten ist stets das Löschen des replizierten Objektes notwendig.

Die Animation vieler Objekte erfordert eine entsprechend hohe Rechenleistung, da für einen flüssigen Bewegungsablauf mindestens 25 Bilder je Sekunde berechnet werden sollten. Um das Flackern des Bildes während der Berechnung zu verhindern, wird mit einer sogenannten *Doppelpufferung* gearbeitet, d.h., es werden zwei oder mehr Kopien des Bildspeichers verwendet, von denen immer nur einer aktiv ist. Dessen Inhalt wird auf dem Bildschirm angezeigt. In den inaktiven Puffern werden die folgenden Bilder berechnet. Ein Bildwechsel erfolgt dann, indem der entsprechende Puffer aktiv wird.

10.5 Einführung in OpenGL

10.5.1 Vorbemerkungen

Bisher wurden die elementaren Algorithmen und mathematischen Grundlagen, die in der Computergrafik [FvDF+90] Anwendung finden, vorgestellt. Dabei wurde auch klar, dass diese Funktionen möglichst effizient implementiert sein müssen, damit eine schnelle Bildschirmausgabe möglich wird.

Wenn man diese Algorithmen komplett in Software realisiert, dann begrenzt sehr schnell die *von-Neumann*-Architektur die maximal erreichbare Geschwindigkeit. Aus diesem Grund hat sich die Grafikhardware in den letzten Jahren stürmisch weiterentwickelt. Einfache *Framebuffer*, die nur den Bildschirminhalt in einem Speicher abbilden, findet man nur noch selten. In den letzten 10 Jahren wurden immer mehr der elementaren Bildoperationen in Hardware realisiert. Nach einfachen *2D-Beschleunigern* haben sich in letzter Zeit *3D-Beschleuniger* durchgesetzt. Diese können die rechen- und bandbreitenintensiven Operationen für das Zeichnen der texturierten Polygone, das Schattieren und den Tiefentest ohne Zutun der CPU durchführen. Selbst die Transformation wird zunehmend in solch spezialisierter Hardware ausgeführt.

Bei der rasanten Hardware-Entwicklung wird klar, dass nicht jeder Programmierer sein Programm an solch unterschiedliche Umgebungen anpassen kann. Deshalb hat man diese Funktionen in Bibliotheken ausgelagert, die von den Hardwareherstellern optimiert werden. Für die Programmierung von grafischen Benutzeroberflächen haben sich unter den verschiedenen Betriebssystemen sehr unterschiedliche Bibliotheken durchgesetzt (z.B. Win32, Xlib, Motif, GTK, Qt, ...). Die Tabelle 10.1 stellt die verbreitetsten Bibliotheken vor, die heute für aufwendige Grafikprogrammierung genutzt werden.

Name	Entwickler	Betriebssysteme	Anwendung
DirectX	Microsoft	Windows	Multimedia, Spiele
OpenGL	SGI, ARB	Unix, Windows, BeOS, MacOS, ...	professionelle Visualisierung, Spiele
QuickTime3D	Apple	MacOS, Windows	Multimedia, Spiele

Tabelle 10.1: Die verbreitetsten Grafikbibliotheken

Im folgenden wird ein kleiner Überblick über OpenGL [Men97] gegeben. Diese Bibliothek eignet sich für die zwei- und dreidimensionale Darstellung von Punkten, Linien und Polygonen. Ein *Depth Buffer* erlaubt das Herausfiltern verdeckter Objekte. Das Aussehen der Objekte wird durch Farbgebung, Schattierung und Transparenz beeinflusst. Für eine realistische Darstellung können Texturen und Lichter verwendet werden. Ihre einfach portierbare und erweiterbare Struktur hat OpenGL in den letzten Jahren zum Standard für professionelle Computergrafik auf allen gängigen Betriebssystemen gemacht. Mittlerweile wird sie von einem Zusammenschluss von über 20 namhaften IT-Firmen, dem *Architecture Review Board (ARB)*, weiterentwickelt.

10.5.2 Die Bibliothek

Aufteilung der Funktionen

OpenGL [SAF97] ist so portabel, weil ihre Kern-Bibliothek nur die nötigsten Funktionen für die Grafikprogrammierung bereitstellt. Methoden für das Binden der Ausgabe an ein Fenster fehlen ganz, weil diese sehr vom Betriebssystem abhängig sind. Realisiert sind:

- Funktionen zum Zeichnen von Punkten, Linien, konvexe Polygone,
- Funktionen zum Ausführen von Transformationen,
- Elementare Pixeloperationen.

Alle Funktionen, die höhere Zeichenfunktionen (Freiform-Flächen, Zylinder, Kugeln) ermöglichen und andere nicht unbedingt notwendige Funktionen, die die Programmierung erleichtern, sind in eine eigene Bibliothek ausgelagert: Die *OpenGL Utility Library* (*GLU*) [CFH+97].

Das Binden der Ausgabe von OpenGL an ein Fenster des Window-Systems erledigen andere Bibliotheken. Für professionelle Projekte nutzt man dazu betriebssystemspezifische Bibliotheken, wie *WGL* für Windows und *GLX* für Unix. Für einfache Anwendungen ist das betriebssystemunabhängige *OpenGL Utility Toolkit* (*GLUT*) [Kil96a] besser geeignet, weil es mit wenigen Aufrufen dem Programmierer die meiste Arbeit abnimmt. Die Tabelle 10.2 zeigt die zugehörigen Include-Dateien für C.

Name	Beschreibung
`GL/gl.h`	Die OpenGL Kern-Bibliothek
`GL/glu.h`	Die OpenGL Utility Library
`GL/glut.h`	Das OpenGL Utility Toolkit

Tabelle 10.2: Die OpenGL-Include-Dateien

Datentypen, Funktionen und Konstanten

Da ANSI C die Größen der Basis-Datentypen nicht festlegt, definiert OpenGL eigene Datentypen, um die richtige Größe sicherzustellen. Diese beginnen immer mit dem Prefix GL und sind in der Tabelle 10.3 zusammengestellt. Explizite Typumwandlungen sind, wegen der Standard-Typkonvertierungen von C und den Headern, nur selten nötig.

Alle Funktionen von OpenGL folgen einem Namensschema. Ein Prefix kennzeichnet die Zugehörigkeit zu der Teilbibliothek, z.B.: gl, glu und glut. Danach folgt der eigentliche aussagekräftige Name. Jedes Teilwort beginnt mit einem Großbuchstaben. Gibt es die Funktion mit unterschiedlicher Parameteranzahl, so wird diese nach dem Namen angegeben. In diesem Fall wird als Suffix der Datentyp der Argumente, in Form von 1–2 Buchstaben, angegeben. Die Zuordnung zwischen Suffix und Datentyp erfolgt gemäß Tabelle 10.4. Falls man die Argumente als Feld übergeben kann, hängt man ein v an.

Typ	minimale Größe	Beschreibung
GLboolean	1	Wahrheitswert
GLbyte	8	vorzeichenbehaftete Ganzzahl
GLubyte	8	positive Ganzzahl
GLshort	16	vorzeichenbehaftete Ganzzahl
GLushort	16	positive Ganzzahl
GLint	32	vozeichenbehaftete Ganzzahl
GLuint	32	positive Ganzzahl
GLsizei	32	natürliche Zahl (>0)
GLenum	32	symbolische Ganzzahl
GLbitfield	32	Bit-Feld
GLfloat	32	Fließkommazahl
GLclampf	32	Fließkommazahl $\in [0, 1]$
GLdouble	64	Fließkommazahl
GLclampd	64	Fließkommazahl $\in [0, 1]$

Tabelle 10.3: Die OpenGL-Datentypen

Suffix-Buchstaben	zugehöriger OpenGL-Datentyp
b	GLbyte
ub	GLubyte
s	GLshort
us	GLushort
i	GLint
ui	GLuint
f	GLfloat
d	GLdouble

Tabelle 10.4: Zuordnung zwischen Suffix-Buchstaben und OpenGL-Datentypen

Die Konstanten der Bibliothek folgen dem selben Schema, nur sind bei ihnen alle Buchstaben groß geschrieben und die Teilworte werden durch einen Unterstrich voneinander getrennt. Die Tabelle 10.5 gibt einige Beispiele für diese systematische Namensgebung.

Beispiel	Art	Header	Argument
glColor4f	Funktion	`gl.h`	4 GLfloat-Variablen
glVertex3fv	Funktion	`gl.h`	3 GLfloat-Werte in einem Feld
gluOrtho2D	Funktion	`glu.h`	
glutInit	Funktion	`glut.h`	
GL_QUADS	Konstante	`gl.h`	
GLU_FILL	Konstante	`glu.h`	
GLUT_RGB	Konstante	`glut.h`	

Tabelle 10.5: Beispiele für die systematische Namensvergabe in OpenGL

Das Arbeitsprinzip

OpenGL arbeitet als eine *Zustandsmaschine* (*Statemachine*), das heißt, die Bibliothek befindet sich zu jedem Zeitpunkt in einem definierten Zustand, der durch Aufruf von Funktionen manipuliert werden kann. Jede Zustandsänderung wirkt immer global. Zum Beispiel setzt ein Aufruf von glColor3f (1.0,1.0,1.0) die Farbe auf weiß. Diese Farbe bleibt solange aktuell, bis ein erneuter Aufruf von glColor getätigt wird. Möchte man etwas zeichnen, so muss die Bibliothek in einen Zustand versetzt werden, der sie auf die Eckpunkte warten lässt. Ein Aufruf von glMatrixMode(matrix) wählt die zu bearbeitende Matrix aus.

Der aktuelle Zustand der Bibliothek kann auf je einem *Matrizen-* und *Attribut-Stack* gespeichert und später wiederhergestellt werden. Man kann sich also OpenGL als eine Maschine vorstellen, die durch viele Knöpfe (ihre Funktionen) bedient wird.

Mit der bisherigen Herangehensweise war der Programmablauf bereits beim Schreiben des Quelltextes festgelegt. Für eine grafische Oberfläche ist das nicht möglich. Das Programm muss auf verschiedene Ereignisse reagieren, von denen nicht bekannt ist, wann und in welcher Reihenfolge sie eintreten. Dazu zählen zum Beispiel:

- Ein Tastendruck,
- das Drücken einer Schaltfläche,
- das Verändern der Größe eines Fensters,
- das Schließen eines Fensters.

Eine Fensterumgebung wie Windows sammelt diese Ereignisse zentral und teilt sie den jeweiligen Programmen durch Nachrichten mit. Dazu rufen die Programme beim Eintreffen einer Nachricht einen sogenannten *Eventhandler* auf. Das ist eine Funktion, die vom Programmierer bei der Fensterbibliothek registriert wird. Diese enthält die Anweisungen für die Reaktion auf das Ereignis. Bei jedem Auftreten des Ereignisses wird von nun an

diese Funktion automatisch aufgerufen. Bei diesem Verarbeitungsmodell spricht man von *Ereignisorientierter Programmierung*.

Beim Entwurf eigener Programme ist darauf zu achten, dass jederzeit bekannt ist, wie die Ausgabe aussieht. Jede Veränderung der Größe oder Aufdecken des Fensters machen ein Neuzeichnen nötig. Will man die Ausgabe verändern, so muss dies in einem *Eventhandler* geschehen.

10.5.3 Die Praxis

Erste Schritte

Um ein OpenGL-Programm mit Hilfe von GLUT zu erstellen, muss der Programmierer stets folgende Aufgaben implementieren.

1. OpenGL und GLUT initialisieren.
2. Fenster anlegen.
3. Notwendige Initialisierungen durchführen:
   - Sichtbaren Ausschnitt des Koordinatensystems festlegen,
   - notwendige Berechnungen für die darzustellenden Körper durchführen,
   - Event-Handler registrieren.
4. GLUT-Hauptschleife starten und auf Ereignisse warten.

Derartige Programme haben dadurch stets einen ähnlichen Aufbau.

Anhand des ersten Beispiels sollen nun die einzelnen Schritte erläutert werden. Das Programm soll eine typische Aufgabe erfüllen: *Ausgabe einer Funktion in einem gegebenen Definitionsbereich*. Der zugehörige Quelltext ist ab Seite 179 abgedruckt.

Zuerst wird GLUT initialisiert. Dazu übergibt glutInit (...) mit Hilfe der Variablen argc und argv die Kommandozeilenargumente an GLUT.[1]

Im nächsten Schritt stellt glutInitDisplayMode (...) die benötigten Eigenschaften des Ausgabefensters ein. Benötigt wird ein einfaches Fenster (GLUT_SINGLE) mit einem RGB-Farbschema (GLUT_RGB). Die Anfangsfenstergröße wird mit glutInitWindowSize (...) auf 400×300 Pixel gesetzt. Danach kann das Fenster mit glutCreateWindow(name) erzeugt werden.

In OpenGL wird mit Weltkoordinaten gearbeitet, das heißt, der Programmierer kann in dem Koordinatensystem arbeiten, das günstig für ihn ist. OpenGL skaliert die Ausgabe immer so, dass sie in das Fenster passt.

Die Projektionsmatrix legt den sichtbaren Ausschnitt fest. Zur Bearbeitung wird diese mit glMatrixMode(GL_PROJECTION) ausgewählt. Die Funktion gluOrtho2D(...) erzeugt die

[1] Das Betriebssystem übergibt beim Start des Programmes die Anzahl der Argumente in argc. Deren Inhalt wird in dem String-Feld argv gespeichert. An Position 0 steht der Programmname, dann folgen die Argumente in der Reihenfolge, wie sie beim Start angegeben wurden.

passende Matrix für ein zweidimensionales kartesisches Koordinatensystem. Ihre Parameter legen nacheinander die *linke*, *rechte*, *untere* und *obere Grenze* fest.

Bevor nun mit glutMainLoop() die Hauptschleife von GLUT zur Ereignisverarbeitung gestartet werden kann, müssen die Eventhandler (siehe 10.5.2) registriert werden. Für unser Beispiel wird nur eine Funktion gebraucht, die das Neuzeichnen des Fensters übernimmt. Dies erledigt glutDisplayFunc(plotfunc). Die Funktion plotfunc () wird als Zeiger übergeben. Sie enthält alle Zeichenanweisungen.

glClear (GL_COLOR_BUFFER_BIT) löscht den Inhalt des Ausgabefensters.

OpenGL stellt für das Zeichnen einige geometrische Grundformen (*Primitive*) bereit. Die Abbildung 10.17 zeigt diese mit ihren Namen und der Reihenfolge, in der ihre Eckpunkte angegeben werden. Komplexere Formen, wie konkave Polygone oder Freiform-Flächen müssen durch viele kleine Dreiecke angenähert werden. Die GLU-Bibliothek stellt dazu Hilfsfunktionen bereit, die über diese Einführung hinausgehen. Alle diese *Primitive* bestehen aus einer Reihe von Eckpunkten (*Vertices*), die zwischen

```
glBegin(Objekt);
  ⋮
glEnd();
```

angegeben werden. Jeder dieser Eckpunkte hat eine Reihe von Eigenschaften:

Farbe wird mit glColor gesetzt:

```
glColor{3,4}{f,d,i,s,ub,us,ui}[v](...);
```

Dabei gibt man nacheinander die Intensitäten der *roten*, *grünen*, *blauen* und der optionalen *alpha-Komponente* an. *Alpha* bezeichnet die Transparenz. 0.0 ist komplett durchsichtig und 1.0 ist undurchsichtig. Üblicherweise wird die Intensität als Gleitkommazahl im Bereich zwischen 0.0 und 1.0 angegeben. Die aktuelle Farbe bleibt bis zum erneuten Aufruf von glColor bestehen und wird solange für alle weitere Eckpunkte genutzt.

Normalenvektor steht senkrecht auf einer Fläche. Er ist wichtig für Sichtbarkeitstests und die Beleuchtung von Flächen im 3D-Raum. Beides wird in unserem Beispiel nicht benötigt.

Texturkoordinate legt fest, wie eine *Textur* auf eine Fläche aufgetragen wird. Dies kann auf Grund der Komplexität nicht Bestandteil dieser Einführung sein.

Raumpunkt gibt die Position im Raum an. Gesetzt wird dieser mit der Funktion glVertex:

```
glVertex{2,3,4}{f,d,i,s}[v](...)
```

Ein Raumpunkt kann also mit 2–4 Koordinaten vom Typ **float**, **double**, **int** oder **short** an OpenGL übergeben werden.

Der Befehl glBegin(Objekt) versetzt OpenGL in den Zustand auf *Eckpunkte* zu warten, um das *Objekt* zu zeichnen. glEnd() beendet das Warten auf *Vertices*. Damit die Objekte tatsächlich im Fenster erscheinen, muss der Befehl glFlush () ausgeführt werden.

Beispiel 10.2
Ausgabe einer Funktion in einem gegebenen Definitionsbereich

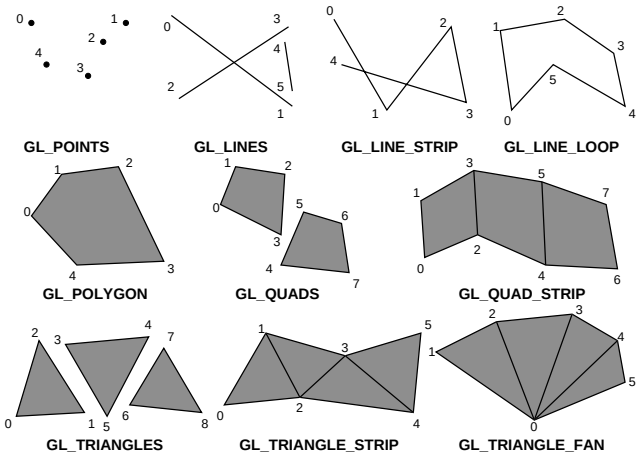


Abbildung 10.17: Die von OpenGL unterstützten geometrischen Grundformen

```
#include <stdio.h>
#include <math.h>
#include <GL/glut.h>

#define PI 3.14159265358979323846

const GLfloat Df[]={-2*PI, 2*PI}; /* Definitionsbereich */
const GLfloat Wf[]={-2, 2};       /* Wertebereich */
const GLfloat dx=0.1;             /* Schrittweite */
GLfloat f(GLfloat x);             /* Funktion f(x) */

void plotfunc();                  /* Zeichenfunktion */

int main(int argc, char **argv) {
  /* GLUT initialisieren */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
  /* Fenster anlegen */
  glutInitWindowSize(400, 300);
  glutCreateWindow("Funktionsplotter");
  /* Weltkoordinatensystem festlegen */
  glMatrixMode(GL_PROJECTION);
  gluOrtho2D(Df[0], Df[1], Wf[0], Wf[1]);
  /* Zeichenfunktion festlegen */
  glutDisplayFunc(plotfunc);
  /* GLUT-Hauptschleife aufrufen */
  glutMainLoop();
  return 0;
}

/* Funktion f(x) */
GLfloat f(GLfloat x) {
  return sin(x);
}
```

```
/* Zeichenfunktion */
void plotfunc() {
  GLfloat x, y;
  /* Fensterinhalt löschen */
  glClear(GL_COLOR_BUFFER_BIT);
  /* Koordinatenkreuz zeichnen */
  glBegin(GL_LINES);
    glColor3f(1, 1, 1);
    glVertex2f(Df[0], 0);
    glVertex2f(Df[1], 0);
    glVertex2f( 0, Wf[0]);
    glVertex2f( 0, Wf[1]);
  glEnd();
  /* f(x) zeichnen */
  glBegin(GL_LINE_STRIP);
    glColor3f(0.2, 0.2, 1.0);
    for (x=Df[0]; x<=Df[1]+dx; x+=dx) {
      y=f(x);
      glVertex2f(x, y);
    }
  glEnd();
  /* Zeichenfunktionen ausführen */
  glFlush();
}
```

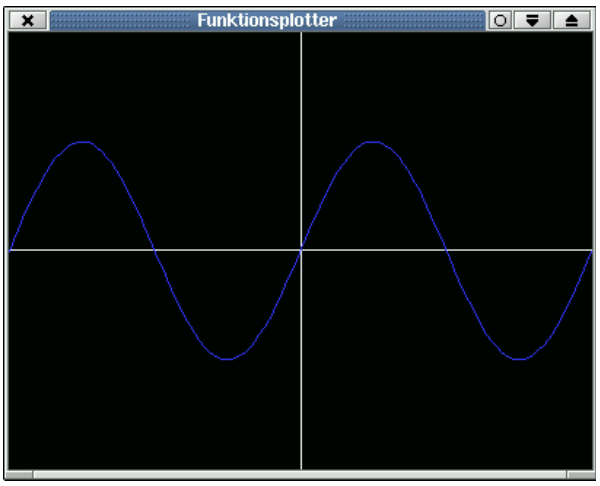


Abbildung 10.18: Ein einfacher Funktionsplotter

Reagieren auf Ereignisse

Das nächste Beispiel demonstriert die Behandlung von *Benutzereingaben* und *Textausgabe* in einem Grafikfenster. Das Programm zeichnet das *Haus vom Nikolaus* und der Nutzer

kann per Tastendruck die einzelnen Zeichenschritte durchlaufen. Zugehöriger Quelltext befindet sich ab Seite 182.

Die Koordinaten des Hauses sind in dem zweidimensionalen Feld haus [][2] gespeichert. Die Variable schritt speichert den Zeichenschritt, bis zu dem das Haus gezeichnet werden soll.

Die Funktion tastatur wird mit glutKeyboardFunc(tastatur) für die Verarbeitung von Tastaturereignissen angemeldet. Wird „+“ gedrückt, erhöht die Funktion den Wert von schritt. Bei einem „-“ erniedrigt sie ihn. Danach veranlasst glutPostRedisplay () das Neuzeichnen des Fensters.

Das Programm zeichnet eine erläuternde Zeichenkette in das Fenster. OpenGL und GLUT bieten nur Funktionen für die Ausgabe einzelner Zeichen. Aus diesem Grund übernimmt die Funktion drawstring (...) den wiederholten Aufruf von glutBitmapCharacter (...), bis der ganze String ausgegeben ist. Mit glRasterPos2f(x, y) legt man den Anfang für die Stringausgabe im Fenster fest.

Beispiel 10.3
Haus vom Nikolaus

```
#include <GL/glut.h>

/* Koordinaten des Nikolaushauses */
GLint haus[][2]={{1, 1}, {3, 3}, {1, 3}, {2, 4}, {3, 3},
                 {3, 1}, {1, 3}, {1, 1}, {3, 1}};
/* Anzahl der Zeichenschritte */
int schritt=9;

void zeichneHaus();                                  /* Haus zeichnen */
void tastatur(unsigned char key, int x, int y); /* Tastaturhandler */
void drawstring(char *s);                            /* Sring ausgeben */

int main(int argc, char **argv) {
  /* OpenGL und GLUT initialisieren */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
  /* Fenster anlegen */
  glutInitWindowSize(250, 350);
  glutCreateWindow("Haus vom Nikolaus");
  /* Ereignishandler registrieren */
  glutDisplayFunc(zeichneHaus);
  glutKeyboardFunc(tastatur);
  /* Weltkoordinatensystem festlegen */
  glMatrixMode(GL_PROJECTION);
  gluOrtho2D(0.75, 3.25, 0.75, 4.25);
  /* GLUT-Hauptschleife aufrufen */
  glutMainLoop();
  return 0;
}

/* Zeichenfunktion */
void zeichneHaus() {
  int s;
```

```
  glClear(GL_COLOR_BUFFER_BIT);
  /* Zeichenposition festlegen */
  glRasterPos2f(0.75, 0.75);
  /* String ausgeben */
  drawstring("+/-: Zeichenschritte");
  /* gewünschte Anzahl von Zeichenschritten ausführen */
  glBegin(GL_LINE_STRIP);
    for (s=0; s<schritt; s++)
      glVertex2iv(haus[s]);
  glEnd();
  /* Zeichenschritte ausführen */
  glFlush();
}

/* Tastaturhandler */
void tastatur(unsigned char key, int x, int y) {
  switch (key) {
  case '+':
    if (++schritt>9)
      schritt=1;
    break;
  case '-':
    if (--schritt<1)
      schritt=9;
    break;
  }
  /* Neuzeichnen veranlassen */
  glutPostRedisplay();
}

/* String im Fenster ausgeben */
void drawstring(char *s) {
  int i;
  for (i=0; s[i]!='\0'; i++)
    glutBitmapCharacter(GLUT_BITMAP_9_BY_15, s[i]);
}
```

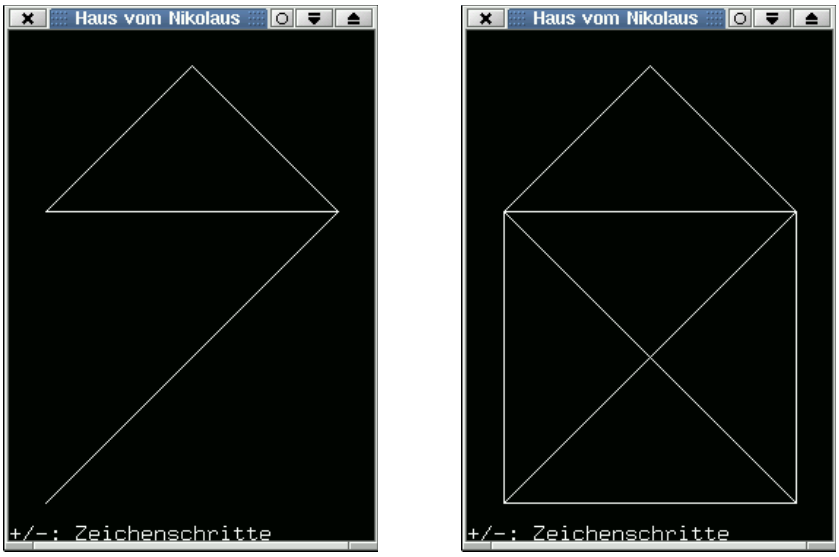
Transformation

Die ersten beiden Beispiele haben einfache Zeichenoperationen demonstriert. Nun soll gezeigt werden, wie man mit Transformationen die Ausgabe beeinflussen kann. Bevor ein mit glVertex erzeugter Eckpunkt auf dem Bildschirm erscheint, wird er nacheinander mit einer Reihe von Matrizen multipliziert, bis seine endgültige Position im Fenster feststeht. Der Programmierer ist für die ersten beiden Stufen zuständig:

Modelview-Matrix wandelt die *Objektkoordinaten* in *Weltkoordinaten* um.

Projection-Matrix wandelt die Weltkoordinaten in *Fenster-* oder *Augenkoordinaten* um.

Oft muss ein Objekt, zum Beispiel ein Zahnrad in einem CAD-Programm, an verschiedenen Stellen im Fenster gezeichnet werden. Dann ist es sinnvoll, das Objekt einmal im



(a) schritt =5 (b) schritt =9

Abbildung 10.19: Das Haus vom Nikolaus

Ursprung zu definieren, und es durch eine anschließende Transformation in seine endgültige Position zu verschieben. Diese Aufgabe übernimmt die *Modelview*-Matrix. Die *Projection*-Matrix legt den sichtbaren Ausschnitt der „Welt“ fest. Bisher haben wir nur die letzte Matrix genutzt. Ein Aufruf von glMatrixMode(matrix), entweder mit dem Argument GL_MODELVIEW oder GL_PROJECTION, legt die aktuell zu bearbeitende Matrix fest.

Mit glLoadIdentity () lädt man die *Einheitsmatrix* in die aktuelle Matrix. OpenGL bietet dann eine Reihe von Standardtransformationen um diese zu modifizieren:

- glTranslatef (x, y, z) fügt der aktuellen Matrix eine Translation in x-, y-, z-Richtung hinzu.
- glRotatef(deg, x, y, z) fügt eine Rotation um deg°, um die durch den Vektor $\vec{r} = \{x, y, z\}$ definierte Achse, hinzu.
- glScalef (x, y, z) fügt eine Skalierung in x-, y- und z-Richtung hinzu.

Diese drei Funktionen werden im ersten Beispiel dieses Abschnitts ab Seite 185 demonstriert. Mit glRectf (−1.5, −1.0, 1.5, 1.0) wird viermal das selbe Rechteck gezeichnet. Durch vorherige Veränderung der *Modelview*-Matrix erscheint es jedesmal an einer anderen Stelle des Fensters und ist zusätzlich verschoben, rotiert oder skaliert.

Eine Veränderung der Matrix muss auch wieder rückgängig gemacht werden können. Man kann deshalb jederzeit die aktuelle Matrix mit glPushMatrix() auf einem *Matrizen-Stack* sichern. Der alte Zustand kann dann mit glPopMatrix() wiederhergestellt werden.

Reichen die vordefinierten Transformationen nicht aus, so kann eine beliebige Matrix mittels glLoadMatrix(matrix) aus einem Feld geladen werden. OpenGL erwartet dabei immer

eine 4×4-Matrix in Form eines eindimensionalen Feldes. Die Tabelle 10.6 zeigt den Zusammenhang zwischen der mathematischen Notation und der Realisierung in C. Soll eine selbst erstellte Matrix mit der aktuellen multipliziert werden, so ruft man glMultMatrix(matrix) auf. Die Nutzung eigener Matrizen wird im zweiten Beispiel ab Seite 186 an Hand der *Scherung* demonstriert.

mathematische Notation	Realisierung in C

$\mathbf{M} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$	`GLfloat M[]={ a11, a12, a13, a14,`
	`a21, a22, a23, a24,`
	`a31, a32, a33, a34,`
	`a41, a42, a43, a44 };`

Tabelle 10.6: Definition einer Matrix für OpenGL

Beispiel 10.4
Standardtransformationen in OpenGL: Translation, Rotation und Skalierung.

```
#include <GL/glut.h>

void zeichne(); /* Zeichenfunktion */

int main(int argc, char **argv) {
  /* GLUT initialisieren */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
  /* Fenster anlegen */
  glutInitWindowSize(400, 300);
  glutCreateWindow("Standardtransformationen");
  /* Weltkoordinatensystem festlegen */
  glMatrixMode(GL_PROJECTION);
  gluOrtho2D(-8, 8, -6, 6);
  glMatrixMode(GL_MODELVIEW);
  /* Zeichenfunktion festlegen */
  glutDisplayFunc(zeichne);
  /* GLUT-Hauptschleife aufrufen */
  glutMainLoop();
  return 0;
}

/* Zeichenfunktion */
void zeichne() {
  /* Bildschirm loeschen */
  glClear(GL_COLOR_BUFFER_BIT);
  /* Trennlinien zeichnen */
  glBegin(GL_LINES);
    glColor3f(1, 1, 1);
    glVertex2f(-8, 0);
    glVertex2f( 8, 0);
    glVertex2f( 0, -6);
    glVertex2f( 0, 6);
```

```
  glEnd();
  /* Farbe der Rechtecke */
  glColor3f( 1, 1, 0);
  /* Original zeichnen */
  glPushMatrix();
    glTranslatef(-4, 3, 0);
    glRectf(-1.5, -1.0, 1.5, 1.0);
  glPopMatrix();
  glPushMatrix();
    glTranslatef( 4, 3, 0);
    /* Translation um 1 in x- und y-Richtung */
    glTranslatef( 1, 1, 0);
    glRectf(-1.5, -1.0, 1.5, 1.0);
  glPopMatrix();
  glPushMatrix();
    glTranslatef(-4, -3, 0);
    /* Rotation um 45° z-Achse */
    glRotatef(45, 0, 0, 1);
    glRectf(-1.5, -1.0, 1.5, 1.0);
  glPopMatrix();
  glPushMatrix();
    glTranslatef( 4, -3, 0);
    /* Skalierung um den Faktor 2 in x-Richtung */
    glScalef( 2, 1, 1);
    glRectf(-1.5, -1.0, 1.5, 1.0);
  glPopMatrix();
  /* Zeichenschritte ausführen */
  glFlush();
}
```

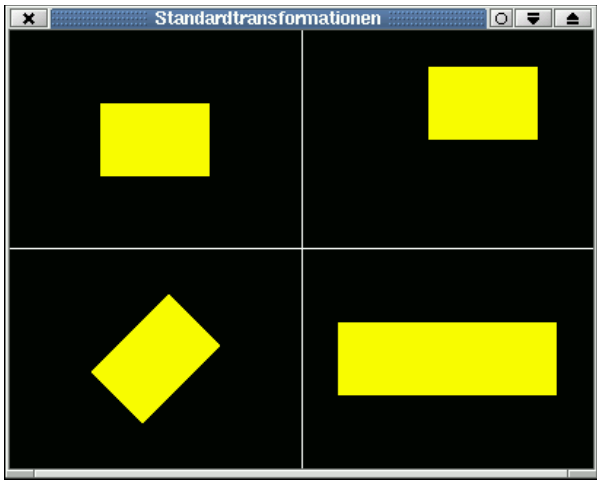


Abbildung 10.20: Die in OpenGL eingebauten Transformationen

Beispiel 10.5
Beliebige Transformationen in OpenGL am Beispiel der Scherung.

```
#include <GL/glut.h>

/* Scherungsfaktoren in x- und y-Richtung */
#define SHX 0.25
#define SHY 0.25

/* Scherungs-Matrizen */
const GLfloat SCHERUNG_X[]={  1, 0, 0, 0,
                            SHX, 1, 0, 0,
                              0, 0, 1, 0,
                              0, 0, 0, 1 };

const GLfloat SCHERUNG_Y[]={  1,SHY, 0, 0,
                              0,  1, 0, 0,
                              0,  0, 1, 0,
                              0,  0, 0, 1 };

void zeichne(); /* Zeichenfunktion */

int main(int argc, char **argv) {
  /* GLUT initialisieren */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
  /* Fenster anlegen */
  glutInitWindowSize(400, 300);
  glutCreateWindow("Scherungsdemo");
  /* Weltkoordinatensystem festlegen */
  glMatrixMode(GL_PROJECTION);
  gluOrtho2D(-8, 8, -6, 6);
  glMatrixMode(GL_MODELVIEW);
  /* Zeichenfunktion festlegen */
  glutDisplayFunc(zeichne);
  /* GLUT-Hauptschleife aufrufen */
  glutMainLoop();
  return 0;
}

/* Zeichenfunktion */
void zeichne() {
  /* Bildschirm löschen */
  glClear(GL_COLOR_BUFFER_BIT);
  /* Trennlinien zeichnen */
  glBegin(GL_LINES);
    glColor3f(1, 1, 1);
    glVertex2f(-8, 0);
    glVertex2f( 8, 0);
    glVertex2f( 0, -6);
    glVertex2f( 0, 6);
  glEnd();
  /* Farbe der Rechtecke */
  glColor3f(1, 1, 0);
  /* Original zeichnen */
  glPushMatrix();
    glTranslatef(-4, 3, 0);
```

```
    glRectf(-1.5, -1.0, 1.5, 1.0);
  glPopMatrix();
  /* Scherung nach x zeichnen */
  glPushMatrix();
    glTranslatef( 4, 3, 0);
    glMultMatrixf(SCHERUNG_X);
    glRectf(-1.5, -1.0, 1.5, 1.0);
  glPopMatrix();
  /* Scherung nach y zeichnen */
  glPushMatrix();
    glTranslatef(-4, -3, 0);
    glMultMatrixf(SCHERUNG_Y);
    glRectf(-1.5, -1.0, 1.5, 1.0);
  glPopMatrix();
  /* Scherung nach x und y zeichnen */
  glPushMatrix();
    glTranslatef( 4, -3, 0);
    glMultMatrixf(SCHERUNG_X);
    glMultMatrixf(SCHERUNG_Y);
    glRectf(-1.5, -1.0, 1.5, 1.0);
  glPopMatrix();
  /* Zeichenschritte ausführen */
  glFlush();
}
```

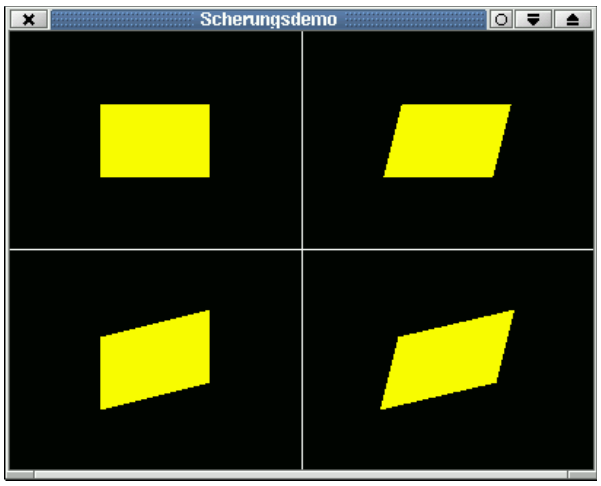


Abbildung 10.21: Beliebige Transformationen mit OpenGL am Beispiel der Scherung

Ausblick auf 3D-Grafik und Animation

Zum Abschluss dieser Einführung soll gezeigt werden, wie statt eines statischen zweidimensionalen Bildes eine dreidimensionale Animation programmiert werden kann. Das *RGB-Farbmodell* soll mit einem *rotierenden Würfel* veranschaulicht werden. Ab Seite 189 steht der Quelltext.

Damit die Ausgabe nicht flimmert, muss das neue Bild im Hintergrund aufgebaut werden. Diese Technik nennt man *Doppelpufferung*. Mit glutSwapBuffers() wird zwischen beiden Bildern gewechselt.

Verdeckte Flächen erkennt OpenGL mit Hilfe eines *Tiefenpuffers*. Der Tiefentest wird mit glEnable(GL_DEPTH_TEST) aktiviert. Mit glShadeModel(GL_SMOOTH) wird festgelegt, dass die Farben weich ineinander übergehen sollen.

Um das Zeichnen von Objekten weiter zu beschleunigen, werden OpenGL-Anweisungen, die zwischen glNewList(list , GL_COMPILE) und glEndList() stehen, in einer sogenannten *Display-Liste* zusammengefasst. Aufwendige Berechnungen können so vor dem Zeichnen durchgeführt und gespeichert werden. Damit stehen sie beim eigentlichen Zeichnen durch den Aufruf von glCallList (list) sofort zur Verfügung.

Für eine Animation müssen die Objekte in kleinen Schritten mindestens fünfundzwanzigmal in der Sekunde bewegt werden. Die Funktion animiere() wird mit glutIdleFunc (...) registriert. Eine *Idle-Funktion* wird immer dann aufgerufen, wenn das Programm nichts weiter zu tun hat, also fertig mit dem Zeichnen ist. Somit ist sie der ideale Platz um die Welt schrittweise zu verändern – sie zu animieren.

Für eine realistische Darstellung ist die Perspektive wichtig. Die Funktion gluLookAt (...) definiert den Blickwinkel, aus dem die „Welt“ gesehen wird. Dafür erwartet sie nacheinander die Koordinaten des Standpunktes der Kamera, den Punkt auf den sie schaut und einen Vektor, der nach oben weist, als Parameter. Die Funktion gluPerspective (...) richtet die Perspektive ein: Den Öffnungswinkel der Kamera, das Seitenverhältnis des Bildes und der Tiefenbereich (minimale und maximale Tiefe) des sichtbaren Bildes.

Das Buch [Kil96b] bietet eine gute Einführung in die OpenGL-Programmierung mit GLUT.

Beispiel 10.6
Veranschaulichung des RGB-Farbmodells mit einem rotierenden Würfel.

```
#include <GL/glut.h>

/* Rotationswinkel um die jeweiligen Achsen */
GLfloat rotx=0, roty=0, rotz=0;

void init();      /* Initialisierung */
void wuerfel();   /* Würfel zeichnen */
void zeichnen();  /* Zeichenfunktion */
void animiere();  /* Würfel animieren */

int main(int argc, char **argv) {
  /* OpenGL und GLUT initialisieren */
  glutInit(&argc, argv);
  /* Benötigt werden ein RGB-Bild mit Tiefenpuffer und Doppelpufferung
   * zur Vermeidung von Bildschirmflackern
   */
  glutInitDisplayMode(GLUT_RGB|GLUT_DEPTH|GLUT_DOUBLE);
  /* Fenster öffnen */
  glutInitWindowSize(400, 400);
  glutCreateWindow("Farbwürfel");
  /* Ausgabe vorbereiten */
```

```
  init();
  /* Zeichenfunktion festlegen */
  glutDisplayFunc(zeichnen);
  /* Animationsfunktion festlegen */
  glutIdleFunc(animiere);
  glutMainLoop();
  return 0;
}

/* Initialisierung */
void init() {
  /* Tiefentest einschalten */
  glEnable(GL_DEPTH_TEST);
  /* weiche Farbverläufe einschalten */
  glShadeModel(GL_SMOOTH);
  /* Blickwinkel einrichten */
  glMatrixMode(GL_PROJECTION);
  gluPerspective(40, 1, 1, 10);
  gluLookAt(3, 0, 0, 0, 0, 0, 0, 1, 0);
  glMatrixMode(GL_MODELVIEW);
  /* Würfel erzeugen und in Liste abspeichern */
  glNewList(1, GL_COMPILE);
  glTranslatef(-0.5, -0.5, -0.5);
  wuerfel();
  glEndList();
}

/* erzeugt einen Farbwürfel */
void wuerfel() {
  glBegin(GL_QUADS);
    /* Boden */
    glColor3f(0, 0, 0);
    glVertex3i(0, 0, 0);
    glColor3f(0, 1, 0);
    glVertex3i(0, 1, 0);
    glColor3f(1, 1, 0);
    glVertex3i(1, 1, 0);
    glColor3f(1, 0, 0);
    glVertex3i(1, 0, 0);
    /* Deckel */
    glColor3f(0, 0, 1);
    glVertex3i(0, 0, 1);
    glColor3f(1, 0, 1);
    glVertex3i(1, 0, 1);
    glColor3f(1, 1, 1);
    glVertex3i(1, 1, 1);
    glColor3f(0, 1, 1);
    glVertex3i(0, 1, 1);
  glEnd();
  glBegin(GL_QUAD_STRIP);
    /* Mantel */
    glColor3f(0, 0, 0);
    glVertex3i(0, 0, 0);
    glColor3f(0, 0, 1);
```

```
    glVertex3i(0, 0, 1);
    glColor3f(1, 0, 0);
    glVertex3i(1, 0, 0);
    glColor3f(1, 0, 1);
    glVertex3i(1, 0, 1);
    glColor3f(1, 1, 0);
    glVertex3i(1, 1, 0);
    glColor3f(1, 1, 1);
    glVertex3i(1, 1, 1);
    glColor3f(0, 1, 0);
    glVertex3i(0, 1, 0);
    glColor3f(0, 1, 1);
    glVertex3i(0, 1, 1);
    glColor3f(0, 0, 0);
    glVertex3i(0, 0, 0);
    glColor3f(0, 0, 1);
    glVertex3i(0, 0, 1);
  glEnd();
}

/* Zeichenfunktion */
void zeichnen() {
  /* sowohl Farb- als auch Tiefenpuffer löschen */
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
  /* Würfel aufrufen */
  glCallList(1);
  /* neues Bild zeigen */
  glutSwapBuffers();
}

/* Würfel animieren */
void animiere() {
  /* Rotationswinkel erhöhen */
  rotx+=1;
  roty+=2;
  rotz+=3;
  /* Neue Transformationsmatrix erzeugen */
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glRotatef(rotx, 1, 0, 0);
  glRotatef(roty, 0, 1, 0);
  glRotatef(rotz, 0, 0, 1);
  /* Neuzeichnen veranlassen */
  glutPostRedisplay();
}
```

10.5.4 Begriffe

Doppelpuffer wird oft bei Animationen eingesetzt. Dabei erfolgt das Neuzeichnen in einem getrennte Puffer, der erst nach der Fertigstellung mit dem aktuell sichtbaren Puffer vertauscht wird. So wird ein Flackern der Animation vermieden.

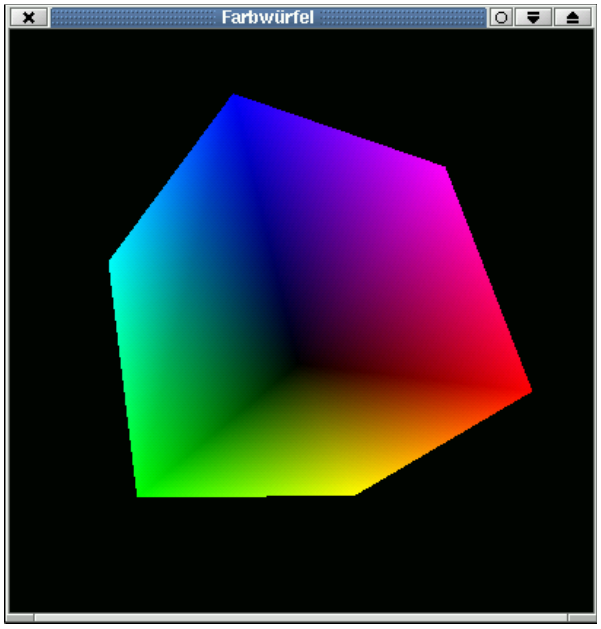


Abbildung 10.22: Der rotierende Farbwürfel

Eventhandler wird von GLUT aufgerufen, wenn ein bestimmtes Ereignis (z.B. notwendiges Neuzeichnen des Fensters, Tastendruck, Mausklick, abgelaufene Zeitspanne ...) eintritt, damit das Programm Gelegenheit zum Reagieren hat.

GLU Die *OpenGL Utility Library* [CFH+97] erleichtert das Erstellen von OpenGL-Programmen. Sie bietet Funktionen für das Einrichten des sichtbaren Ausschnittes, der gewünschten Perspektive, die Ausgabe von Freiform-Flächen und weitere fortgeschrittene Aufgaben.

GLUT Das *OpenGL Utility Toolkit* [Kil96a] ist eine plattformübergreifende Fensterbibliothek, die sich vor allem für die Erstellung von kleinen bis mittleren Programmen eignet. Sie wird in [Kil96b] für die Einführung OpenGL-Programmierung genutzt.

Idle-Funktion wird von einem Programm immer dann aufgerufen, wenn keine anderen Aufgaben anstehen.

Normalenvektor steht senkrecht auf einer Fläche. Man nutzt ihn in der Computergrafik zur Berechnung der Beleuchtung einer Fläche und deren Sichtbarkeit.

OpenGL Die *Open Graphics Library* [SAF97] ist eine Plattformübergreifende Grafikbibliothek, die sich als Industriestandard etabliert hat.

Stack (ein *Stapelspeicher*) speichert eine beliebige Anzahl von Elementen. Man kann im Gegensatz zu einem Feld nur auf das zuletzt gespeicherte Element direkt zugreifen. Wird dieses vom Stack entfernt (*pop*), bekommt man Zugriff auf das Element, dass direkt davor auf den Stack geschoben (*push*) wurde.

Textur beschreibt die Struktur (Farbe und Beschaffenheit) einer Oberfläche.

Texturkoordinate legt fest, wie die Textur auf eine Fläche „aufgetragen“ wird.

Tiefenpuffer gibt an, wie weit jeder Pixel des Bildes vom Betrachter entfernt ist. Er enthält somit Tiefeninformationen. Die Verwendung eines Tiefenpuffers ist eine oft eingesetzte Methode für die Bestimmung sichtbarer Flächen in der 3D-Programmierung.

10.6 Zusammenfassung

Die Grafik ist heute ein abgesichertes Lehrgebiet der Informatik. Es gibt bereits Grafikbibliotheken und Standardsoftware zum Zeichnen, 2D-Zeichnungserstellung, 3D-Modellieren, Simulation von Bewegungen (Animation), für Schnitte, Oberflächeneffekte, verdeckte Kanten usw. . . In diesem Kapitel ging es um das prinzipielle Verständnis der einzelnen Transformationen und um die Art ihrer Realisierung. Genauere Ausführungen zur Theorie der Computergrafik findet man zum Beispiel in [FvDF$^+$90].

11 Datenbanksysteme

11.1 Begriffliche Erklärungen

Zusammenfassende Bemerkungen zum Programmieren:

- Sinn und Zweck der Programmierung ist Übernahme eines Eingabedatenstromes und die Verarbeitung desselben durch entsprechende Algorithmen. Anschließend wird der veränderte Datenstrom ausgegeben.

- Variablen sind Datenträger. Typvereinbarungen erklären ihren Wertebereich. Variablen- und Typenvereinbarungen als Einheit bilden das rechnerinterne Datenmodell (RIM).

- Daten können im Hauptspeicher oder außerhalb (Diskette, Platte, Band, CD) gehalten werden. Letzteres funktioniert über das File-Konzept.

- File-System ist eine Sammlung von Daten.

- Der Zugriff auf Files erfolgt in Form von:

  - Lesen (Read),
  - Schreiben (Write),
  - Aktualisieren/Ändern (Update),
  - Löschen (Erase),
  - Kopieren (Copy),
  - Sortieren (Sort).

- Die Zunahme numerischer und nichtnumerischer Daten verlangt einen Verwaltungsapparat, der über die Fähigkeiten eines Filesystems hinaus geht. Damit wird die Forderung nach einem Datenbanksystem begründet.

- Ein Datenbanksystem ist eine Sammlung von Datenstrukturen, Daten einschließlich der dafür notwendigen typischen Zugriffsalgorithmen.

11.1.1 System, Kommunikationssystem, Informationssystem

Eine Menge von Elementen, die auf irgendeine Weise in Beziehung stehen, wird als *System* bezeichnet. Tauschen diese Elemente gegenseitig Mitteilungen aus, so spricht man von einem *Kommunikationssystem*. Ist dieser Austausch von Mitteilungen (Kommunikationsprozess) formalisiert, sind also feste Regeln vorhanden, nach denen diese Kommunikationsprozesse ablaufen, so nennt man diese formalisierten Kommunikationssysteme *Informationssysteme*.

Voraussetzung für ein Informationssystem ist die Organisierung der Vorgänge, die in diesem System ablaufen sollen und die Formalisierung der Mitteilungen, d.h. Nachrichten. Bestimmte Kommunikations- und Verarbeitungsvorgänge lassen sich dabei soweit formalisieren, daß sie in Programmiersprachen beschrieben und als Computerprogramme in einem Rechner abgewickelt werden können. Elemente dieses Informationssystems sind Benutzer (Mensch), Prozessor (Maschine) und Programme (Funktionen).

11.1.2 Rechnerunterstütztes Informationssystem

In betrieblichen Anwendungen finden Informationssysteme ihren Ausdruck als Beschreibungssystem des Objektsystems Betrieb oder von Teilen desselben. Folgende wesentlichen Komponenten bilden ein rechnerunterstütztes Informationssystem:

- Systemeingabe/Systemausgabe,
- Betriebssystem,
- Datenverwaltungssystem/Datenbank,
- Anwendungssystem,
- Übersetzer (Compiler), Dienstprogramme.

Die einzelnen Komponenten haben folgende Funktionen (Abbildung 11.1):

Datenbank (DB): Sammlung aller Datenbestände verschiedener Strukturen

Datenverwaltungssystem (DBVS): Programme der Bereitstellung, Pflege und des Aufbaus der Daten (gegebenenfalls Sprachübersetzer für Anfragesprachen)

Datenbanksystem (DBS = DB + DBVS): Speichert und verwaltet betriebliche Daten.

Anwendungssysteme (AWS): Beschreiben bestimmte betriebliche Abläufe.

11.2 Begründung des Datenbankkonzeptes

11.2.1 Nachteile von Dateisystemen

Der Wert eines betrieblichen Informationssystems wird in erster Linie von der Genauigkeit und Aktualität seiner Daten bestimmt. Historisch gesehen wurde zunächst die Speicherung und Verwaltung dieser Daten auf unabhängigen Dateien durchgeführt (Abbildung 11.2). Dieses Dateikonzept beinhaltet schwerwiegende Nachteile:

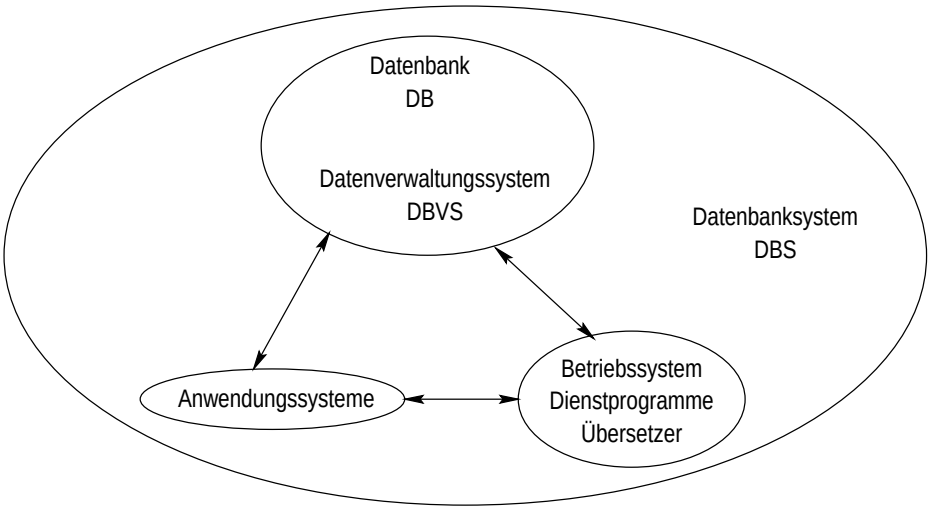


Abbildung 11.1: Rechnerunterstütztes Informationssystem

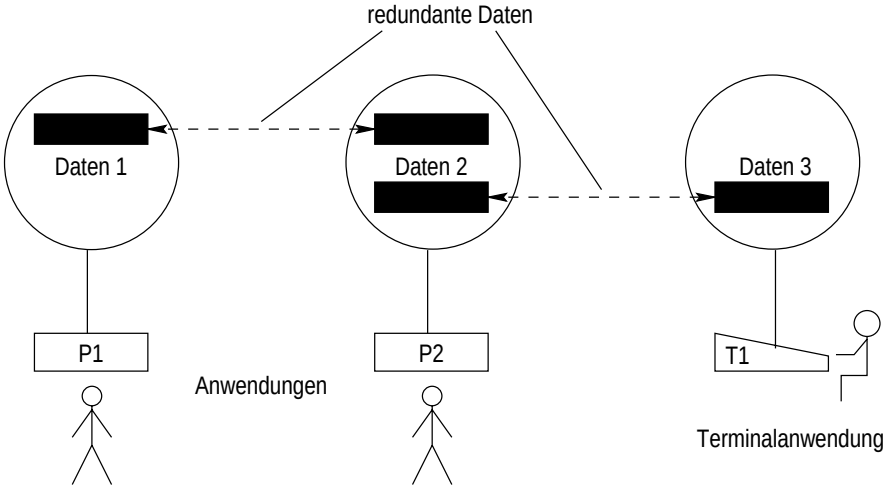


Abbildung 11.2: Notwendige Kommunikation für Änderungen bei konventioneller Dateiverarbeitung

1. Wiederholte Speicherung gleicher Daten führt zu einer hohen Redundanz.
2. Zeitgerechte Änderung von Daten wird durch Mehrfachkopien verhindert.
3. Gleiche Funktionen wie Speicher- und Datenverwaltung, Änderungsdienst, Wiederauffinden (Retrieval) und Vorkehrungen für die Datensicherheit sind in allen Anwendungsprogrammen wiederholt zu realisieren.
4. Daten- und Speicherstrukturen sind auf spezielle Anwendungen zugeschnitten.
5. Die Strukturen spiegeln sich in den Anwendungsprogrammen wider. Geringe Änderungen in Speicher- und Datenstruktur verlangen umfangreiche Programmänderungen.
6. Kontrolle der Richtigkeit und Qualität der verarbeiteten Daten erfolgt im Anwendungsprogramm (somit von einzelnen Programmierern bestimmt).
7. Diese isolierten Anwendungen sind starr. Isolierte Anwendungen mit Hilfe des Dateikonzeptes stellen ein untaugliches Konzept dar, auf dynamisch wechselnden Informationsbedarf in einem betrieblichen Informationssystem zu reagieren.

11.2.2 Anforderungen an ein Datenbanksystem

Wenn wir die Datenbank als eine Sammlung gespeicherter operationaler Daten, die vom Anwendungssystem eines Betriebes benötigt werden, ansehen, dann zielt das Datenbankkonzept durch Integrations- und Standardisierungsmaßnahmen auf eine zentralisierte Datenverwaltung und eine einheitliche und widerspruchsfreie Integritätskontrolle aller Daten für die Gesamtheit aller Anwendungen ab. Damit soll wirkungsvoll und kosteneffektiv einem sich ändernden Informationsbedarf Rechnung getragen werden. Voraussetzung dafür ist die Lösung folgender Probleme:

- Vermeidung der Redundanz der Daten in Hinblick auf alle Verarbeitungsprozesse,
- Einführung einer zentralen Kontrolle über die Datenintegrität,
- Bereitstellung von Sprachen zur leichteren Handhabung der Daten,
- Erhöhung der Unabhängigkeit der Anwenderprogramme von den Daten.

Als Anforderungen für die Schaffung von Datenbanksystemen kann man zusammenfassen:

- zentralisierte Kontrolle für operationale Daten (Redundanzfreiheit),
- zentrale Kontrolle der Datenintegrität (bei Datenspeicherung und -verarbeitung),
- leichte Handhabbarkeit der Daten (einfache Datenmodelle, leicht erlernbare Sprachen),
- Hoher Grad der Datenunabhängigkeit (Wartungseffizienz) als Maß für die Isolation zwischen Anwendungsprogramm und Daten).

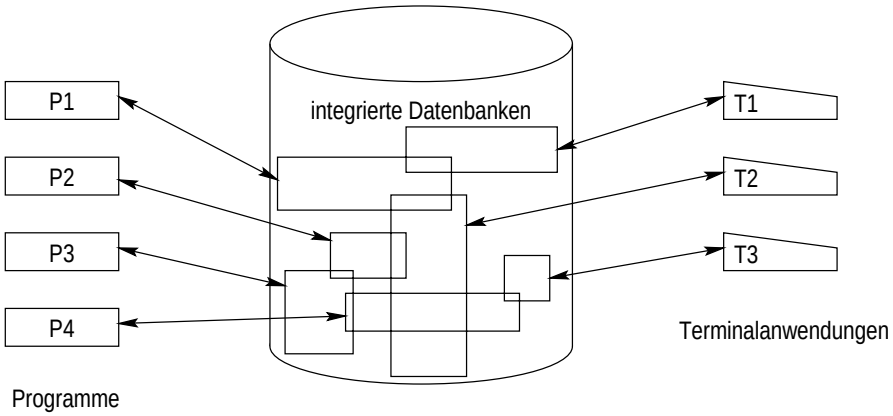


Abbildung 11.3: Zugriff auf eine integrierte Datenbank

Zielstellung neuer DBS ist eine erhöhte Datenunabhängigkeit. Ein *Datenbanksystem* ist durch folgende Eigenschaften charakterisiert:

1. Daten und Datenstrukturen stehen im Vordergrund.
2. Es werden große Datenmengen behandelt.
3. Es sind komplexe Operationen implementiert, die für die Verarbeitung dieser Daten typisch sind.
4. Der prozedurale Aspekt der Datenverarbeitung tritt zurück.
5. Dafür wird zusätzlich eine Adressierung von Objekten durch ihren Inhalt vorgenommen.

11.3 Entwicklungsetappen auf dem Gebiet der Datenbanksysteme

Die verschiedenen Entwicklungsetappen sind im wesentlichen durch das Streben nach erhöhter Datenunabhängigkeit gekennzeichnet. Damit wurde das Wissen des Benutzers, z.B.:

- wie auf die Daten zurückgegriffen wird,
- welche Zugriffspfade vorhanden sind,
- welche Implementierungstechniken benutzt wurden,

schrittweise verringert. Diese Aufgaben werden dem Datenbankadministrator übertragen, der sich um die Einrichtung der DB zu kümmern hat.

0. Generation (1956): Der Programmierer schrieb seine Ein- und Ausgaberoutinen selbst und speicherte seine Daten direkt als Speicherzuordnungsstrukturen auf einem externen Speichermedium.

1.Generation (1960): Die Aufgaben der 0. Generation wurden durch Zugriffsmethoden des Betriebssystems übernommen.

2. Generation (1965): Einführung von Dateiverwaltungssystemen und deskriptiver Sprachen zur beschränkten Auswertung

3. Generation (1969): hierarchische, relationale und Netzwerkdatenbanksysteme, Einführung von Sprachen zur Datenbeschreibung (Data Definition Language (DDL)) sowie zur Handhabung durch den Programmierer (Data Manipulation Language (DML))

4. Generation (1975): Der Benutzer kennt nur die logische Beschreibung der Datenstruktur. Eine deskriptive Datenmanipulationssprache mit hohem Auswahlvermögen wird bereitgestellt.

5. Generation (ab 1990): Entwicklung objektorientierter, multimedialer, verteilter Datenbanksysteme. Die Entwicklung von Datenbanksystemen hat einen hohen Stand erreicht, gilt aber als keineswegs abgeschlossen. Das Zusammenführen von DBS verschiedenster Konzepte ist z.B. eine dringende Forschungsaufgabe.

11.4 Architektur von Datenbanksystemen

11.4.1 Aufgabenstellungen

Für den Zugriff auf Daten stehen unter anderem die in den Abbildungen 11.4(a) und 11.4(b) dargestellten Ansätze zur Verfügung. Diese lassen sich u.a. folgendermaßen realisieren:

- Dateiarbeit: Nutzung des Dateikonzeptes in prozeduralen Sprachen (z.B. File-Konzept in Pascal oder C),
- Datenbankkonzept.

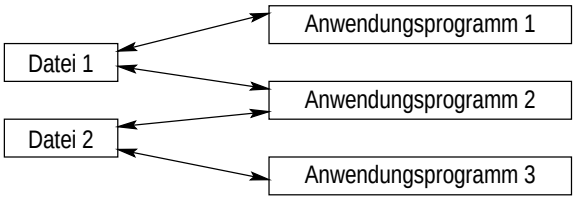
11.4.2 Beschreibungsmodelle für Datenbanksysteme

„Durch ein Datenbanksystem soll ein anwendungsorientierter Ausschnitt der realen Welt, den wir als *Miniwelt* bezeichnen wollen, modellhaft nachgebildet werden, um Vorgänge der realen Welt, beispielsweise bestimmte Abläufe in einer betrieblichen Umgebung, als Transaktionen in der Datenbank als Modellwelt nachvollziehen zu können“. Diese Nachbildung muß in verschiedenen Schritten vollzogen werden, die in Schalen darstellbar sind (Abbildung 11.5).

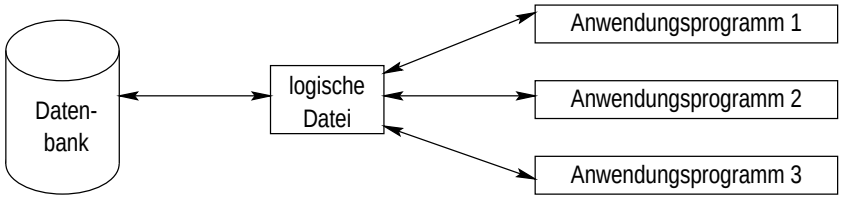
Für die Beschreibung der Datenstrukturen gibt es mehrere Vorschläge. Wir wollen uns hier auf den Vorschlag des amerikanischen Normenausschusses ANSI/SPARC (American National Standard Institute, Standard Planning And Requirement Commitee - 1975) konzentrieren. Dieser sieht einen dreischichtigen Aufbau der Datenbeschreibung vor:

1. Externes Schema (Modell)

- Beschreibt Daten, so wie sie der Nutzer zu sehen wünscht.



(a) Zugriff auf Datei



(b) Zugriff auf Datenbank

Abbildung 11.4: Gegenüberstellung von Dateiarbeit und Datenbankarbeit

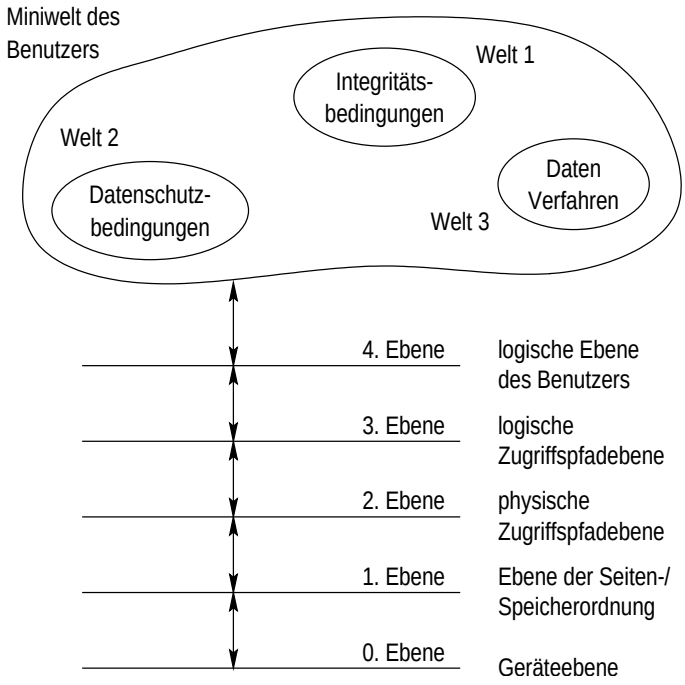


Abbildung 11.5: Ebenendarstellung zur Datenbankarbeit

A-Nr.	Bezeichnung	Farbe	Größe	Menge	L-Nr.	Name	Adresse
A1	Hosen	rot	38	150	L1	Schmidt	Magdeburg
A1	Hosen	rot	38	250	L2	Maier	Dresden
A2	Jacken	grün	42	350	L1	Schmidt	Magdeburg
A2	Jacken	grün	42	450	L2	Maier	Dresden
A2	Jacken	grün	42	550	L3	Becker	Dresden
A3	Pullis	blau	42	650	L1	Schmidt	Magdeburg
A3	Pullis	rot	40	750	L3	Becker	Dresden

Abbildung 11.6: Beispieltabelle „Bestelldaten“

- Externes Modell muß auf das logische Gesamtmodell rückführbar sein.
- Stellt die Programmierschnittstelle dar.

2. Konzeptionelles Schema (Modell)

- Enthält logische Gesamtsicht der Daten.
- Wird durch die Data Definition Language (DDL) realisiert.
- Datenelemente (Entities), die Beziehungen zwischen den Daten (Relationships) sowie die Attribute und Wertevorräte (Domäne) sind erforderlich.

3. Internes Schema (Modell)

- Wird durch die Storage Structure Language (SSL) beschrieben.
- Klärt die Speicherung der Daten und die Zugriffsmöglichkeiten.

11.5 Überblick der wichtigsten Datenmodelle

11.5.1 Einleitung

Aus der Darstellung des Beschreibungsmodells für Datenbanksysteme wurde deutlich, daß das Datenmodell (oder konzeptionelles Modell) das Herzstück jedes DBS ist. Die Auswahl eines geeigneten Datenmodells ist deshalb für den Benutzer der wichtigste Schritt.

Ein Datenmodell beschreibt die Sicht des Anwenders oder Anwendungsprogramms auf die Daten und legt zugleich die möglichen Operationen auf diese Daten fest. Das Datenmodell bestimmt damit die praktische Nützlichkeit des DBS. Das Datenmodell sollte nur den für den Benutzer relevanten Informationsgehalt darstellen. Solche logischen Datenstrukturen, die von ihrem Zugriff auf ihre physische Darstellung abstrahieren, verlangen eine höhere nichtprozedurale Sprache, eine deskriptive Sprache. Bei einer solchen Sprache wird angegeben, welches Ziel erreicht werden soll, d.h., *was* zu tun ist, nicht jedoch der Weg, *wie* dies zu tun ist.

Im folgenden sollen 3 Klassen von Datenmodellen am Beispiel der Bestelldaten (Abbildung 11.6) eines Textilgroßhändlers vorgestellt werden.

A1	Hosen	rot	38

L1	Schmidt	Magdeburg	150
L2	Maier	Dresden	250

A2	Jacken	grün	42

L1	Schmidt	Magdeburg	350
L2	Maier	Dresden	450
L3	Becker	Dresden	550

A3	Pullis	blau	42

L1	Schmidt	Magdeburg	650

A4	Pullis	rot	40

L3	Becker	Dresden	75

Abbildung 11.7: Beispiel im Hierarchischen Datenbankmodell

11.5.2 Hierarchisches Datenbankmodell

Das hierarchische Datenmodell ist das kommerziell erfolgreichste Datenmodell der ersten Generation. Es wurde 1969 von IBM eingeführt. Dieses Modell weist einen unsymmetrischen Aufbau aus, da der Zugang zu den Datenelementen unterschiedliche Suchwege beinhalten kann. In Abbildung 11.7 ist das Beispiel aus Abbildung 11.6 im hierarchischen Datenmodell dargestellt.

11.5.3 Netzwerk-Datenbankmodell

Hier liegt eine Zeigerkettenverarbeitung vor (Abbildung 11.8).

linke Zeigerkette: Zusammenhang Bestellungen-Artikel
rechte Zeigerkette: Zusammenhang Bestellungen-Lieferant

11.5.4 Relationales Modell

In dem Beispiel beschreibt jede Zeile einen Artikel sowie seine Bestellmenge bei einen bestimmten Lieferanten. Derselbe Artikel kann in verschiedenen Mengen bei mehreren Lieferanten bestellt sein. Dafür stehen mehrere Tabellenzeilen. Jede Zeile ist eindeutig durch die Kombination Artikelnummer und Lieferantennummer gekennzeichnet, z.B. A1 und L1. Daher bildet diese Kombination den *Primärschlüssel*. Die Anordnung aller Datenelemente erfolgt in zweidimensionalen Tabellen, wobei jede Zeile alle Zeilenelemente in eine Beziehung setzt. Eine Tabelle wird im relationalen Datenmodell als *Relation* bezeichnet. Begriffe und Regeln im relationalen Datenbankmodell:

- Tabelle = Relation,
- Zeile einer Tabelle = Satz oder Tupel,

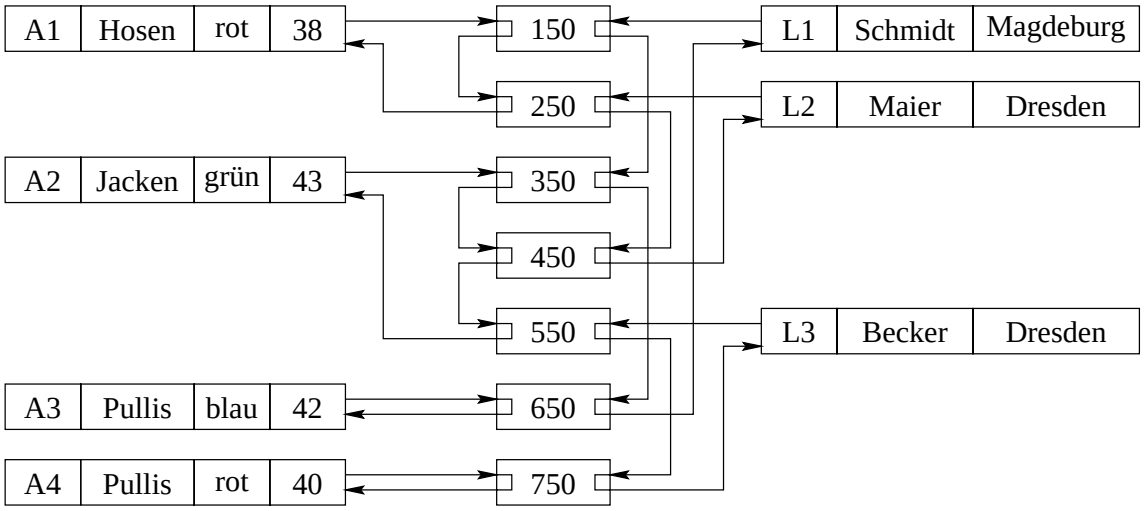


Abbildung 11.8: Beispiel im Netzwerk-Datenbankmodell

- Spalte einer Tabelle = Attribut,
- jedes Element der Tabelle = Datenfeld,
- jedes Attribut hat Wertebereich = Domäne.
- Alle Datensätze sind gleich lang, d.h., die Anzahl der Attribute ist bei allen Sätzen gleich.
- Jeder Datensatz kommt nur ein einziges Mal in der Tabelle vor, die Reihenfolge der Sätze ist jedoch beliebig.
- Der Wert jedes Attributs kommt aus einem bestimmten Bereich.
- Durch Kombination einzelner Spalten (Attribute) und Zeilen (Datensätze) können neue Relationen gebildet werden.

Die Tabelle ist zunächst Ausgangsbasis für ein relationales Datenbankmodell. Es treten noch Wiederholungen (Redundanzen) auf (z.B. A2/Jacken/grün/42, oder L1/Schmidt/Magdeburg). Eine Änderung eines Datenelementes müßte unter Umständen mehrfach ausgeführt werden, womit umfangreiche Suchvorgänge durch die Datenbank verbunden wären. Um dies zu vermeiden werden Relationen *normalisiert*, wodurch eine Entfernung der Redundanzen erreicht wird. In Abbildung 11.9 sind die Abhängigkeiten der Tabelle des Beispiels dargestellt und in den Abbildungen 11.10(a) bis 11.10(c) die normalisierten Relationen.

11.5.5 Einschätzende Bemerkungen zu den Datenbankmodellen

Das Relationale Modell stellt Informationen in Form von normalisierten Relationen ausschließlich durch den Inhalt der Daten dar. Dieses Modell kennt keine physischen Verbindungen. Dadurch kann als Datenbanksprache eine deskriptive Sprache mit hohem Auswahlvermögen genutzt werden.

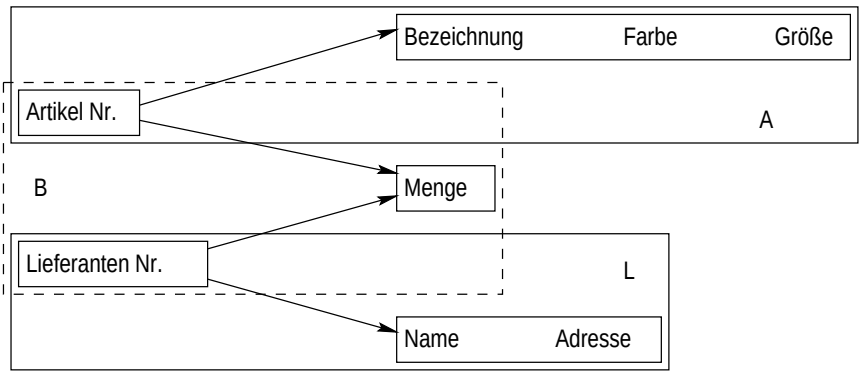


Abbildung 11.9: Abhängigkeitsstruktur der Datenelemente

A-Nr.	Bezeichnung	Farbe	Größe
A1	Hosen	rot	38
A2	Jacken	grün	42
A3	Pullis	blau	42
A4	Pullis	rot	40

(a) Artikel-Relation (A)

L-Nr.	Name	Adresse
L1	Schmidt	Magdeburg
L2	Maier	Dresden
L3	Becker	Dresden

(b) Lieferanten-Relation (L)

A-Nr.	L-Nr.	Menge
A1	L1	150
A1	L2	250
A2	L1	350
A2	L2	450
A2	L3	550
A3	L1	650
A4	L3	750

(c) Bestellungs-Relation (B)

Abbildung 11.11: Normalisierte Relationen des Beispiels

- Vorteile:
  - Einfachheit der Datenstrukturen,
  - strenge theoretische Grundlage,
  - Datenstrukturen und Operationen sind nicht an Zugriffspfade und Speicherstrukturen gebunden,
  - hoher Grad der Datenunabhängigkeit,
  - als symmetrisches Datenmodell kennt es keine bevorzugten Zugriffs- und Auswertungsrichtungen.
- Nachteil:
  - beschränkte Leistungsfähigkeit der deskriptiven Sprache.

Beim hierarchischen Modell und beim Netzwerkmodell erfolgt die Darstellung von Informationen durch Verbindungen von Datensätzen und durch ihre Anordnung in Speicherstrukturen.

- Vorteile:
  - Durch die Verbindung der Sicht auf die Daten mit den Aspekten des Zugriffs kann eine Leistungssteigerung möglich sein (Anwendung prozeduraler Sprachen).
- Nachteil:
  - Datenstrukturierung entspricht nicht immer den realen Gegebenheiten, Erhöhung der Datenabhängigkeit

11.6 Entwurf einer Datenbank

Soll ein Diskursbereich mit Hilfe eines Datenbanksystems abgebildet werden, so muß dieser Bereich zunächst einmal modelliert werden. Dabei müssen die Phasen der Informationsgewinnung (Hier werden Informationen über den Diskursbereich gesammelt und aufbereitet), der semantischen Modellierung (Die Informationen werden mit geeigneten Methoden dargestellt.), der konzeptuellen Modellierung (Das semantische Modell wird in ein Datenbankmodell überführt.) und schließlich der Implementierung (Das konzeptionelle Modell wird in die Datenbank überführt.) durchlaufen werden (Abbildung 11.12).

Eine herausragende Rolle spielt dabei die semantische Modellierung. Das Ergebnis der semantischen Modellierung sind abstrakte Objekte, die die reale Welt unter bestimmter Anwendungssicht abbilden. Zur Vorbereitung der Nutzung einer relationalen Datenbank wird häufig das Entity-Relationship- Modell (ERM) benutzt. Dafür wird eine bestimmte Notation benutzt (Abbildung 11.13). Ein Beispiel ist in Abbildung 11.14.

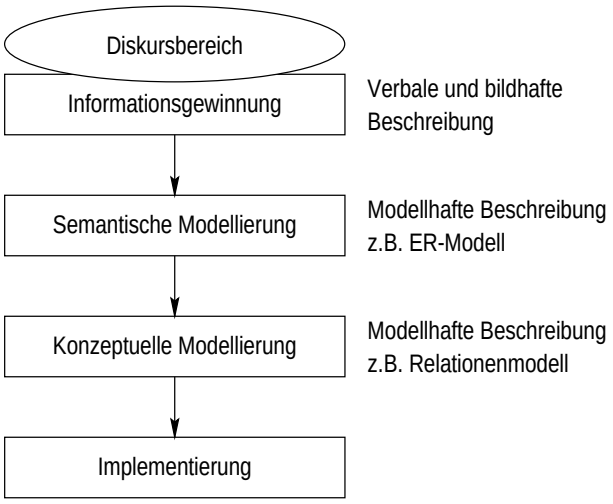


Abbildung 11.12: Teilprozesse der Modellierung eines Diskursbereiches

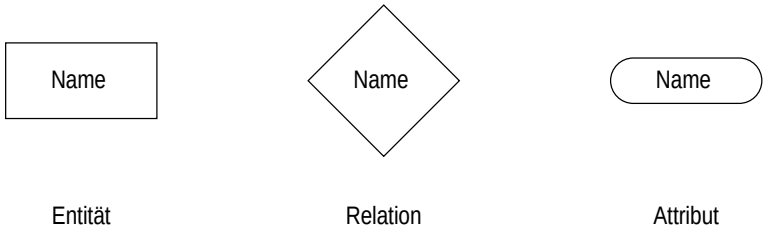


Abbildung 11.13: Notation für ER-Modelle

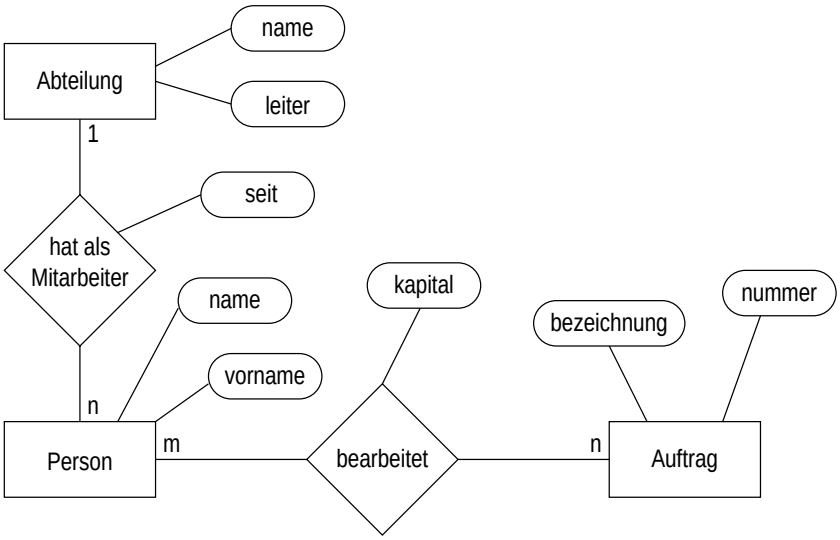


Abbildung 11.14: Beispiel eines ER-Modells

11.7 Die Datenbankanfragesprache SQL

Wie bereits angedeutet, können die unter 11.6 ausgewiesenen Modellierungstechniken benutzt werden, um ein relationales Datenbanksystem aufzubauen. Relationen als Mengen von Tupeln (Zeile einer Tabelle) repräsentieren dabei die Entitäts- und Beziehungstypen des Diskursbereiches. Das Speichern und Wiederauffinden solcher strukturierter Informationen ist eine wesentliche Aufgabe eines Datenbanksystems. Kommerziell vertriebene Datenbanksysteme wie Oracle, Ingres, Informix, Sybase, Access, Paradox (alle relational) stellen dafür eine Vielzahl von Schnittstellen zur Verfügung. Eine der wichtigsten ist die Datenbanksprache SQL (Structured Query Language). Bereits 1986 wurde diese Sprache standardisiert und wird heute deshalb auch als Standard Query Language bezeichnet. Mathematische Grundlagen für diese Sprachen sind die Relationenalgebra bzw. das Relationenkalkül. Im Rahmen dieser Vorlesung kann darauf nicht eingegangen werden.

Die Datenbanksprache SQL besteht aus 3 Bestandteilen:

- der Anfragesprache (Query Language, QL) zur Formulierung von Informationsanforderungen,
- der Manipulationssprache (Data Manipulation Language, DML) zur Speicherung und Veränderung von Informationen,
- der Beschreibungssprache (Data Description Language, DDL) zur Definition von Informationsstrukturen und Zugriffsrechten.

11.7.1 Anfragesprache

Das wichtigste Konstrukt dieser Sprache ist die **SELECT**-Anweisung.

Syntax (SELECT):

SELECT–Anweisung::=
SELECT <select–list>
FROM <table–reference>
WHERE <search–condition>

Beispielsweise sind die Nummern der Mitarbeiter, die am Projekt 111 mit mehr als 100 Stunden beteiligt sind, herauszufinden:

```
SELECT MNR FROM MitPro WHERE PNR=111 AND StdAnz>100
```

Aus dieser Anfrage geht schon hervor, daß die Syntax der Anfrage wie in prozeduralen Sprachen auch durch Bedingungen, Schachtelungen usw. ergänzt werden kann.

11.7.2 Datenmanipulationssprache

Manipulationsanweisungen dienen der Veränderung von Relationeninhalten. SQL kennt dafür 3 Operationen:

- INSERT zum Hinzufügen von Tupeln,
- **DELETE** zum Löschen von Tupeln,

- UPDATE zum Ändern von Tupeln.

Ein Tupel mit den Attributwerten FNR=4500 und Fname='Wohnungsverwaltung' soll in die Relation Firma eingefügt werden.

```
INSERT INTO Firma (FNR, Fname) VALUES (4500, 'Wohnungsverwaltung')
```

11.7.3 Datendefinitionssprache

Wird ein Ausschnitt der realen Welt in einer Datenbank abgebildet, so erfolgt die Speicherung der Strukturen über Metadaten, die in einem Data Dictionary gehalten werden. Auf das Data Dictionary hat im allgemeinen nur der Datenbankadministrator Zugriff. Auf die Metadaten kann ebenso mit den Anweisungen der Anfrage- und Manipulationssprachen zugegriffen werden. Zur DDL gehören u.a. die Anweisungen **CREATE TABLE** zum Anlegen einer Tabelle.Eine neue Tabelle Tabelle Mitarbeiter kann beispielsweise mit folgender Anweisung angelegt werden:

```
CREATE TABLE Mitarbeiter(
  MNR INTEGER NOT NULL,
  Name CHARACTER(20) NOT NULL,
  Vorname CHARACTER(20),
  Gebtag DATE,
  Gehalt INTEGER CHECK (Gehalt>= 2000 AND Gehalt<= 10000)
)
```

Mit der Anweisung INSERT kann diese Tabelle anschließend gefüllt werden. Weitere Anweisungen der DDL sind:

- **CREATE** VIEW zum Anlegen einer Sicht (View),
- GRANT und **REVOKE** zum Setzen von Rechten auf Tabellen.

Will man aus einer Anwendung heraus auf Datenbanken zugreifen, so benötigt man eine Schnittstelle, die SQL-Anweisungen verarbeiten kann. Diese Funktion übernehmen Programmiersprachen. Man spricht in diesem Zusammenhang von eingebettetem SQL (siehe hierzu [Hor95]).

12 Softwaretechnologie

12.1 Aufgabe, Begriffsbestimmung

Softwareentwicklungsprozesse (SEP) sind komplexe Arbeitsprozesse mit schöpferischem Charakter. Eine rationelle Gestaltung dieses SEP gewinnt an Bedeutung, weil die Arbeitsproduktivität bei der Softwareentwicklung der schnellen Entwicklung der Hardware nicht standhält. Softwareentwicklung verlangt Spezialisten der Informatik und der Anwendungsdisziplinen. Deshalb beschäftigen wir uns mit diesem Thema. Voraussetzung ist, daß alle Beteiligten

- Kenntnisse über Inhalt, Ablauf, Arbeitsteilung, Methoden und Mittel der Softwareentwicklung haben,
- die Fähigkeit besitzen, am kollektiven Softwareentwicklungsprozeß teilzunehmen.

Zunächst sollen einige Begriffe, die im Zusammenhang mit der Softwareentwicklung von Bedeutung sind genannt werden.

Software: Programm mit zugehörigen Dokumentationen.

Basissoftware: Dient zum Entwickeln, Warten und Nutzen von Anwendersoftware, z.B.: Betriebssysteme, DBMS.

Anwendersoftware: Löst eine Klasse fachspezifischer Anwenderaufgaben, z.B CAD, PPS, Office-Anwendungen.

Softwareprodukt: Stellt eine Menge zusammengehöriger Softwarekomponenten dar.

Softwareentwicklungsprozeß: Beinhaltet den Prozeß der systematischen Herstellung eines Softwareproduktes.

Softwaretechnologie: Zweig der Informatik, der sich mit den Gesetzmäßigkeiten der produktionstechnischen Vorgänge im Prozeß der Herstellung, Nutzung und Wartung von Software befaßt.

Gegenstand der Softwaretechnologie sind:

- Theorien, Prinzipien, Methoden,
- Normen, Standards, gesetzliche Regelungen,
- Hilfs- und Arbeitsmittel,
- Leitungs- und Organisationsformen der Arbeit in Programmentwicklergruppen.

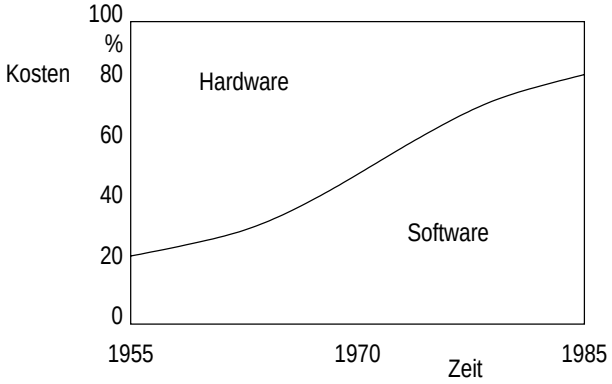


Abbildung 12.1: Ökonomischer Aspekt der Softwareproduktion

12.2 Software als Produkt

Software enthält einen hohen Anteil lebendiger Arbeit, der sowohl für das Entwickeln als auch das Warten aufgewendet wird. Von anderen Produkten unterscheidet es sich weiterhin dadurch, daß für seine vielfache Herstellung (Kopien) relativ wenig Fertigungsaufwand benötigt wird. Software ist ein technisches Produkt, dessen Gebrauchswert vorrangig von der investierten geistig-kreativen, lebendigen Arbeit bestimmt wird.

In Abbildung 12.1 ist das Verhältnis der Kosten für die Entwicklung von Hardware und Software gegenübergestellt. Ursachen für den in Abbildung 12.1gezeigten Trend sind:

- fallendes Preis-Leistungsverhältnis bei Hardware (Faktor 1000-10000 seit 1955)
- geringfügige Zunahme der AP bei Softwareentwicklung (Faktor 2-5)
- „Lebensdauer“ von teilweise mehreren Jahrzehnten bei Softwareprodukten führt zu kostenaufwendigen Wartungszyklen.

Im Zusammenhang mit den Kosten für ein Softwareprodukt müssen auch die Qualitätsanforderungen genannt werden. Qualitätsmerkmale sind:

Korrektheit: Beschreibt den Grad der Übereinstimmung zwischen programmtechnischer Lösung, Dokumentation und Programm.

Flexibilität: Bezieht sich auf Nachnutzbarkeit und Wartbarkeit.

Verständlichkeit: Bewertet den Grad des Einarbeitungsaufwandes für sachgerechte Nutzung.

Stabilität: Beinhaltet den Grad der Sicherheit des Programms bei Eingabe- und Bedienfehlern, bei Fehlern der Gerätetechnik und des Betriebssystems.

Effizienz: Drückt das Verhältnis des Aufwands bei der Nutzung einer Software zur Art, Anzahl und zum Schwierigkeitsgrad der damit zu lösenden Aufgabe aus.

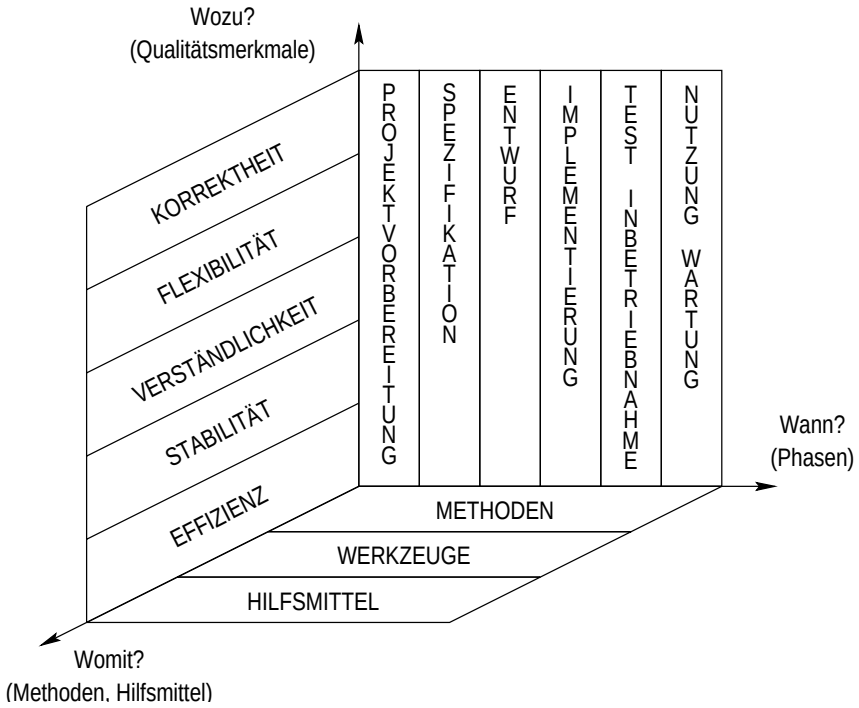


Abbildung 12.2: Den Softwareentwicklungsprozess beeinflussende Faktoren

12.3 Softwareentwicklungsprozeß, Softwarelebenszyklus

Das Entwickeln von Anwenderprogrammen beinhaltet das Programmieren im Kleinen - der Softwareentwicklungsprozeß das Programmieren im Großen.

Die Softwareentwicklung ist durch Fähigkeiten geprägt, die bestimmt werden durch (Abbildung 12.2):

- die Phasen der Entwicklung,
- die benutzten Methoden und Hilfsmittel,
- die hervorzubringenden Qualitätsmerkmale.

Nachdem wir eine kurze Definition des Begriffes Softwareentwicklungsprozeß kennengelernt haben, wollen wir uns mit den Phasen des Entwicklungsprozesses vertraut machen. Dieser Zyklus beinhaltet die Grundphasen (Abbildung 12.3):

- Aufgabenstellung,
- Entwicklung,
- Nutzung,
- Wartung,

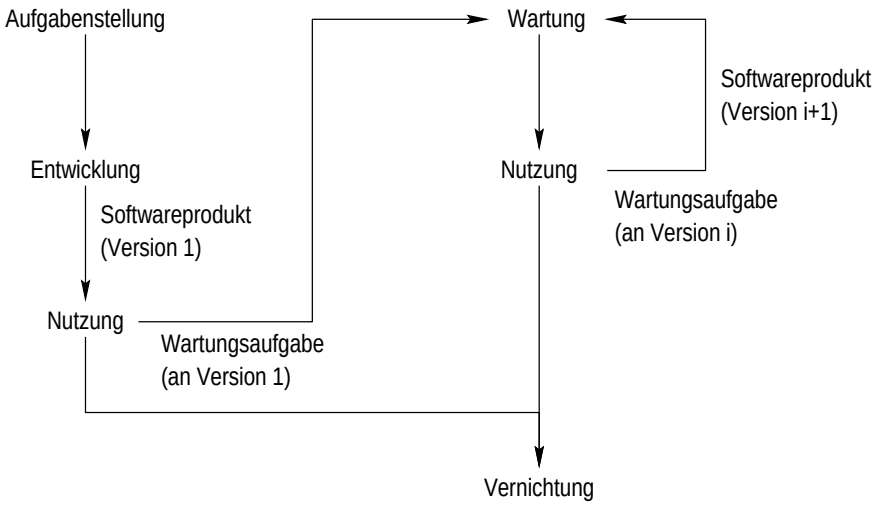


Abbildung 12.3: Softwarelebenszyklus

- Vernichtung.

Es ist festzustellen, daß die Wartung meist zu einer qualitativen Weiterentwicklung des Softwareproduktes führt. Damit unterscheidet sich ein Softwareprodukt wesentlich von anderen Fertigungsprodukten. Erst ein Wartungsaufwand, der die Kosten einer Neuentwicklung übersteigt (Abbildung 12.4) bzw. die fehlende Relevanz der ursprünglichen Aufgabenstellung führen zur Vernichtung des Softwareproduktes. Genauso wie in jedem anderen gesellschaftlichen Arbeitsprozeß wirken im Softwareentwicklungsprozeß Arbeitskräfte und Arbeitsmittel (Softwarewerkzeuge, -systeme u.a.) auf den Arbeitsgegenstand (Softwareprodukt) ein (Abbildung 12.5). Dabei müssen ökonomische, organisatorische, arbeitsrechtliche, technisch-technologische und andere Bedingungen beachtet werden.
Der gesamte Softwareentwicklungsprozeß ist unterteilbar in:

Softwareproduktentwicklung (SPE): Hier werden die Prozesse der Transformation vom Zustand Z_0 in Z_n vorgenommen.

Softwaremanagement (SME): Es erfolgt die Planung der Arbeitskräfte, des arbeitsorganisatorischen und zeitlichen Ablaufs.

Softwarequalitätssicherung (SQS): Diese beinhaltet Zuordnung und Kontrolle der Qualitätsanforderungen.

Technisch-technologische Sicherung (TTS): Sichert Arbeitsmittel, Darstellungsmittel, Softwarezustände, Speicherung.

12.4 Phasenmodell der Softwareentwicklung

Ein dynamisches Phasenmodell für die Softwareentwicklung besteht aus einer Menge von Phasen und Gesetzmäßigkeiten zur Kopplung dieser Phasen. Jede Phase transformiert

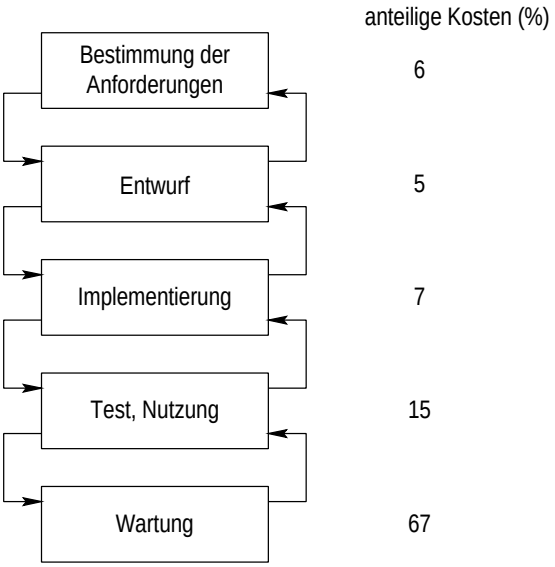


Abbildung 12.4: Kostenverteilung im Softwarelebenszyklus (in Veränderung begriffen)

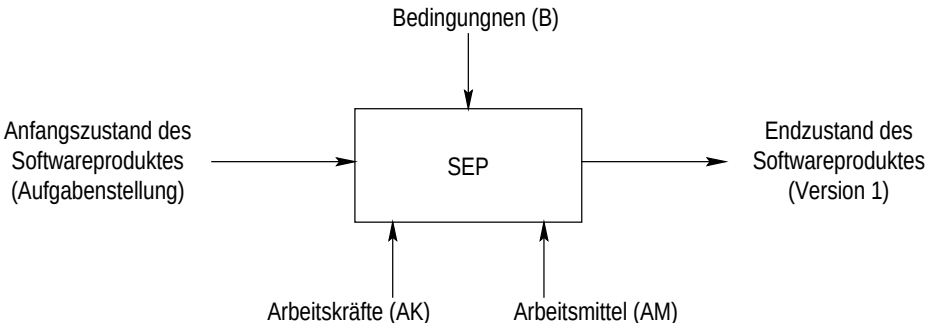


Abbildung 12.5: Wirkungen am SEP

Hauptphase	Phase	Teilphase
Softwarekonstruktion	Analysieren	
	Spezifizieren	
	Entwerfen	fachlich-log. Entwerfen
		programmtechn. Entwerfen
Softwarefertigung	Implementieren	
	Testen	
	Fertigstellen	Bereitstellen
		Erproben

Tabelle 12.1: Phasen des Softwareentwicklungsprozesses

Eingangs- in Ausgangsgrößen. Ein- und Ausgangsgrößen sind Zustände des Softwareproduktes. Die Gesetzmäßigkeiten zur Phasenkopplung sind:

- Prinzipien,
- Bedingungen,
- Regeln

für die Konstruktion von konkreten Phasenstrukturen. Zwei Hauptphasen des Modells sind:

- Softwarekonstruktion als Entwurf des Softwareprodukts,
- Softwarefertigung als Realisierung auf Computern.

In Tabelle 12.1 ist die Aufgliederung dieser Phasen dargestellt. Im folgenden soll auf die wesentlichen Phasen des Softwareentwurfes eingegangen werden.

12.4.1 Analysieren

Ist-Zustand und Umgebung für die Entwicklung des Softwareprodukts werden analysiert (Ist-Analyse) und in Zusammenarbeit von Auftraggeber und Entwickler wird das Soll-Konzept formuliert. Die Grundlage für die Analyse ist eine *verbale Aufgabenstellung*, das Ergebnis eine *präzisierte Aufgabenstellung und Darstellung des Softwareproduktes in einer ein-/ausgangsorientierten Form (formalisierte Aufgabenstellung).*

12.4.2 Spezifizieren

Spezifizieren ist das Ausarbeiten der Anwenderanforderungen, einschließlich Schnittstellen zur Umgebung (funktionelle Anforderungen, Anforderungen an Datenkommunikation, Inhalt und Form der Ein- und Ausgangsgrößen, Qualitätsanforderungen). Die Grundlage für die Spezialisierung bildet die während der Analyse erarbeitete *präzisierte Aufgabenstellung und Darstellung des Softwareproduktes in einer ein-/ausgangorientierten Form.* Das Resultat des Spezifizierens ist ein *Systemkonzept.*

12.4.3 Entwerfen

Erarbeitung von Funktionsumfang, Struktur, Bestandteile, Schnittstellen, Datenstrukturen, Testmittel, -verfahren und -daten basierend auf dem zuvor entwickelten *Systemkonzept*. Das Resultat ist ein *Entwurf des Softwareprodukts*. Der Entwurf gliedert sich in zwei Teilphasen:

1. fachlich-logisches Entwerfen Basierend auf dem in der Phase des Spezifizierens entwickelten *Systemkonzept* wird in diesem Arbeitsschritt eine *Funktionsstruktur* entwickelt, die logisch sinnvolle Funktionen, deren Eingangs- und Ausgangsinformationen, sowie die logischen Schnittstellen und Datenstrukturen enthalten soll.

2. programmtechnisches Entwerfen Der programmtechnische Entwurf beinhaltet die hierarchische Struktur von Programmen und Modulen einschließlich der Schnittstellen, programmtechnischen Datenstrukturen und der Ablaufstrukturen. Die Basis bilden die *Funktionsstruktur, sowie die Hardware und die Basissoftware*. Das Ergibnis ist der *programmtechnischer Entwurf*.

12.4.4 Implementieren

Das Implementieren stellt sich als Kodieren und Herstellen eines lauffähigen Programms sowie die Beseitigung syntaktischer Fehler dar. Es wird aus dem *programmtechnischer Entwurf* ein *ablauffähiges Programm* erstellt.

12.4.5 Testen

Durch planmäßiges Testen ist zu prüfen, ob die spezifizierten Anforderungen erfüllt wurden. Unterschiede sind auszuweisen und zu dokumentieren. Die Grundlage bilden das *ablauffähiges Programm*, die *Testdaten*, sowie die *Testzielstellung*. Das Ergebnis sind das *getestetes Programm*, sowie die *Testergebnisse*

12.4.6 Fertigstellen

Programme und Dokumentation sind in der vorgeschriebenen oder vereinbarten Form bereitzustellen und im Anwendungsbereich zu erproben sowie die Anwendbarkeit nachzuweisen. Dabei wird aus dem *getestetes Programm* ein *Softwareprodukt in der ersten Version* erstellt.

1. Bereitstellen Software ist für die Erprobung und Auslieferung vorzubereiten.

2. Erproben Mittels Erprobungsplan ist Software insgesamt unter Betriebsbedingungen zu testen. Ein Erprobungsbericht ist zu erstellen.

Phase	Ergebnis
Projektvorbereitung (Analysieren)	Idee für ein Softwareprodukt (globale Ziele) Planung des Softwareproduktes (Auftraggeber, Bearbeiter, Aufwand, Termine) Entscheidung über Durchführung der Entwicklung
Spezifikation	Dokumentation der Solleigenschaften des künftigen Softwareproduktes
Entwurf	Dokumentation der Problemlösung (logische Gliederung der Funktionen und Daten, Abläufe u.a., Lösungsideen unabhängig von der verwendeten Programmiersprache)
Kodierung (Implementieren)	Quelltexte in einer Programmiersprache (Umsetzung der Problemlösung in übersetzbare Anweisungen für einen Zielcomputer einschließlich dokumentierender Kommentare)
Test	Fehlerbeseitigung durch Vergleich von Soll- und Isteigenschaften des Softwareproduktes einschließlich Behebung der Differenzen
Inbetriebnahme (Fertigstellen)	Überführung in stabile Dauernutzung (Nutzereinweisung; Vorbereitung der rechentechnischen Umgebung; Probebetrieb)
Nutzung	Nutzanwendung des freigegebenen Softwareproduktes
Wartung	Ändern (Anpassung an neue Bedingungen, Mängelbeseitigung)

Tabelle 12.2: Phasen der Softwareentwicklung und -nutzung

12.4.7 Zusammenfassung

In Tabelle 12.2 sind die wesentlichen Punkte der verschiedenen Phasen des Softwareentwicklungsprozesses zusammengefaßt.

12.5 Methoden und Hilfsmittel der Softwareentwicklung

Die Methoden des Softwareentwicklers sind vielfältig und differenziert (Tabelle 12.3). Sie sind das Ergebnis einer wissenschaftlichen Durchdringung des SEP als auch reicher Erfahrungen in der Erstellung von Softwareprodukten. Im folgenden wollen wir uns nur mit einigen Aspekten vertraut machen, die uns in der Lehrveranstaltung bereits begleitet haben.

12.5.1 Projektbegleitendes Dokumentieren und Verwalten

Warum wird dieser Aspekt an den Anfang gestellt? Die bisherige Arbeitsweise war die Erstellung einer Dokumentation als geschlossene Phase im Anschluß an den Programmtest. Die mögliche Folge ist, daß sich nach einem Jahr normaler Nutzung und Wartung mit begleitender Änderung des Quelltextes eine Schere zwischen Dokumentation und Quelltext, die bis zur Nutzungsunfähigkeit des Programms führen kann, auftut. Deshalb sollte man folgende Prinzipien beherzigen:

Methoden	Wann anzuwenden? (Phase)	Wozu anzuwenden? (Qualitätsmerkmale)
Projektbegleitendes Dokumentieren und Verwalten	alle Phasen	flexibel
Hierarchisches Gliedern	Projektvorbereitung, Spezifikation, Entwurf	entwicklerverständlich
Modularprogrammierung	Entwurf, Kodierung, Test, Wartung	
Strukturierte Programmierung	Entwurf, Kodierung	
Softwareergonomie	Spezifikation, Entwurf	nutzerverständlich, nutzereffizient
Tolerieren von Nutzungsfehlern	Spezifikation, Entwurf, Kodierung	stabil
Schreiben verständlicher Quelltexte	Kodierung	entwicklerverständlich, flexibel
Systematisches Testen	Testen, Wartung	korrekt
Wartungsmethoden	Wartung	effizient, korrekt
Mehrfachnutzung	Spezifikation, Entwurf	flexibel, korrekt

Tabelle 12.3: Methoden des Softwareentwicklers

projektbegleitendes Dokumentieren Jede Phase muß mit spezifischen schriftlichen Ergebnissen abschließen.

rechnerunterstütztes Dokumentieren Es sollte eine kontinuierliche Erfassung der Dokumentationsteile auf elektronischen Datenträgern erfolgen.

Schaffung selbstdokumentierenden Quelltextes Bezeichner von Konstanten, Funktionen, Variablen nebst erforderlichen Kommentaren sollen den Quelltext lesbar gestalten.

projektbegleitendes Verwalten der Entwicklungsergebnisse Die Ordnung zwischen Teilergebnissen eines Softwareproduktes muß aufrechterhalten werden.

Klassischer, überholter Stil des Programmierens:

Man will ein Programm zum Laufen und nicht zum Leben bringen. Späterer Nutzer kommt zu kurz!

Deshalb sollten als Dokumentation im Quelltext vorhanden sein:

1. *Titelzeile:* Modulname, Parameterliste,
2. *organisatorische Angaben:* Version, Autor, Projekt, Datum,
3. *Modulschnittstelle:* Leistung, Eingaben, Ausgaben, Besonderheiten zur Anwendung,
4. *Variablenliste:* Bedeutung aller lokalen Datenelemente,
5. *Datendeklaration:* Vereinbarung von Typ, Dimension, Struktur; Anfangswerte verwendeter Datenelemente,
6. *kommentierter Programmcode:* Ausführbare Anweisungen (Ablaufsteuerung, Wertzuweisung, Ein- und Ausgaben u.a. gemischt mit Kommentaren zur Problemlösung (Algorithmus, rechentechnische Details, problemseitige Hintergrundinformationen).

Kommentare erhöhen *nicht* den Speicherbedarf für das lauffähige Programm. Einige Regeln erhöhen die Verständlichkeit des Quelltextes, z.B. selbsterläuternde Variablennamen, Einrückungen, systematische Markenvergabe u.a. Gliederung der Aufgabe in Unterprogramme macht Quelltext überschaubar.

12.5.2 Hierarchisches Gliedern

Die „Vogelperspektive“ vermittelt Problemübersicht. Man verfolgt das Prinzip des *Schrittweisen Verfeinerns* von Gesamtproblemen zur Basis. Dieses Vorgehen wird auch als *Top-Down-Entwurf* bezeichnet. Die „Froschperspektive“ setzt bereits Kenntnisse über Details voraus. In diesem Fall wird von *Bottom-Up-Entwurf* gesprochen.

Wie erfolgt ein solcher „Höhenflug“ bei der Softwareentwicklung? Am Anfang steht die Faktensammlung. Es schließt sich der Entwurf einer hierarchischen Gliederung des Problemfeldes als Baumstruktur an: Faktenordnung (grafisch oder numerisch). Zwei Möglichkeiten der Strukturierung zeigt Abbildung 12.7. Die Gliederung beinhaltet dabei auszuführende

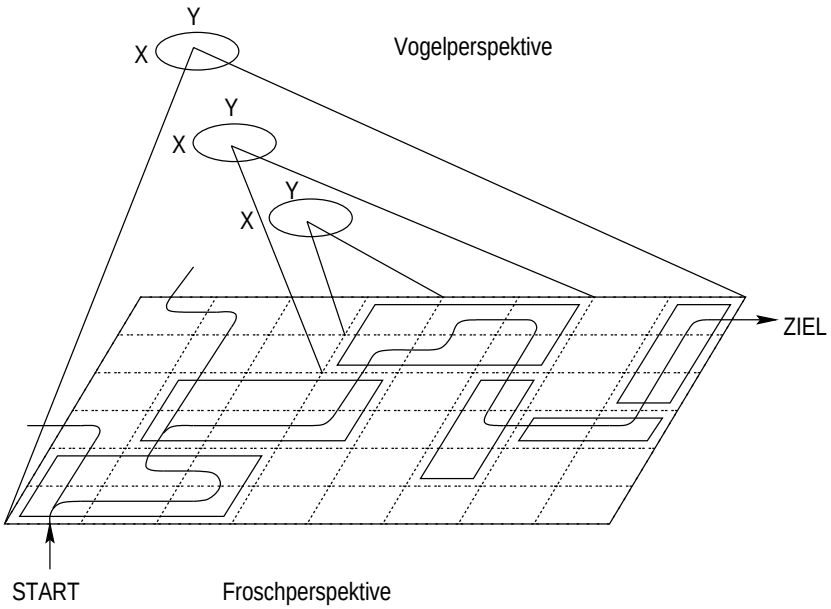


Abbildung 12.6: Problemlandschaft mit Überblick

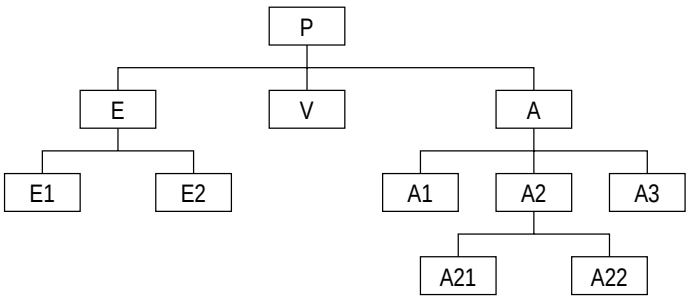


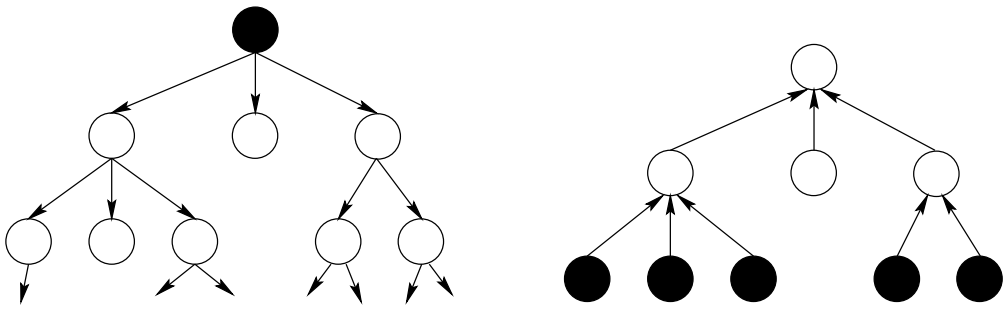
Abbildung 12.7: Gliederung des Problemfeldes

Funktionen und zu verarbeitende Daten. Das spätere Anwendungsgebiet bestimmt die Herangehensweise bei der Gliederung. Liegt der Schwerpunkt der Anwendung auf der Veränderung von wenigen Daten durch komplizierte Algorithmen (z.B. wissenschaftlich-technische Berechnungen erfolgt eine *funktionsorientierte Gliederung*. Liegt der Schwerpunkt andererseits auf der Verwaltung großer Datenbestände (Speichern, Übertragen, Aktualisieren) bietet sich eine *datenorientierte Gliederung* an.

Es wurde bereits erwähnt, daß es prinzipiell zwei Vorgehensweisen zur Bearbeitung der Struktur der Lösung gibt, den Top-Down- und den Bottom-Up-Ansatz. Diese sollen im folgenden genauer betrachtet werden.

Top-down-Entwurf

Bei diesem Ansatz wird von einer generellen, groben Lösung ausgegangen, die schrittweise verfeinert wird, bis sie alle Anforderungen des Problems erfüllt. Abbildung 12.8(a) veran-



(a) Top-Down-Entwurf (b) Bottom-Up-Entwurf

Abbildung 12.8: Entwurfsstrategien

schaulicht diese Vorgehensweise.

Bottom-up-Entwurf

In diesem Fall wird von bekannten Teillösungen für bestimmte Teilprobleme, z.B. Standardfunktionen für Dateiverarbeitung, Grafik, Standardberechnung, etc. ausgegangen. Diese werden dann in geeigneter Weise zusammengefügt, so daß sie das gegebene Problem lösen. Abbildung 12.8(b) veranschaulicht diese Vorgehensweise.

12.5.3 Modularprogrammierung

Beinhaltet die Zusammensetzung eines Softwareproduktes aus vielen sorgfältig voneinander abgegrenzten Bausteinen (Modulen). Ein Modul ist u.a. gekennzeichnet durch abgegrenzte Funktionen, einfache Datenschnittstellen und Geheimhaltung der Wirkungsweise. Die Schnittstelle eines Moduls muß mindestens zwei Angaben enthalten:

Leistung: Hier wird beschrieben, welchen Hauptzweck das Modul erfüllt. Außerdem ist zu spezifizieren, welche Spezialfälle bei seiner Anwendung auftreten, bzw. wie es zu Handhaben ist und welche Fehlerbehandlung durch den Modul erfolgt.

Eingaben und Ausgaben: Ein- und Ausgabeparameter, Ein- und Ausgaben über periphere Geräte (z.B. Dialogeingaben, Druckerausgaben, Datenübertragungen, Abfrage eines Schalters), Lesen o. Schreiben aus modulexternen Datenbereichen.

Die Schnittstellen-Beschreibung eines Moduls ist als sogenannte EVA-Tabelle bekannt (EVA - Eingabe Verarbeitung Ausgabe). Ist die Funktionsgliederung in Baumstruktur durchgeführt worden und wird jedem Element des Baumes eine EVA-Tabelle zugeordnet, so praktizieren Sie damit die HIPO-Technik (Hirarchy plus Input/Process/Output). Die HIPO-Technik basiert auf der „schrittweisen Verfeinerung“ und nimmt die zu verarbeitenden Daten als einen Ausgangspunkt des Entwurfs (Ausgabedaten als Funktion der Eingabedaten).

Der Nutzen der Modularprogrammierung besteht u.a. in der:

- arbeitsteiligen Softwareentwicklung mit vertretbarem Kommunikationsaufwand,
- Übersicht durch Abgrenzung, die zu höherer Produktivität des Programmierers führt,
- leichteren Wartbarkeit, da überschaubarer bzw. besser zugänglich für Änderungen,
- erhöhten Mehrfachnutzung (insbesondere bei problemunabhängigen Modulen).

12.5.4 Strukturierte Programmierung

Die beiden vorgestellten Methoden hierarchisches Gliedern und Modularprogrammierung helfen Ordnung und Durchschaubarkeit in eine große Programmieraufgabe zu bringen. Weiterhin ist es aber erforderlich, die Beziehungen zwischen den Elementen Eingabe, Verarbeitung und Ausgabe innerhalb eines Moduls (oder Programms) zu ordnen, d.h. der Steuerfluß (Programmablauf) muß nach bestimmten Vorschriften erfolgen. Dieser Aufgabe stellt sich die Strukturierte Programmierung als Methode zur Schaffung übersichtlicher Programmablaufstrukturen (Steuerflußdisziplin). Eine wesentliche Forderung der strukturierten Programmierung ist die Vermeidung von „undisziplinierten“ Sprüngen (z.B. **goto**), da sonst „Spaghettiprogramme“ entstehen, die sehr unübersichtlich sind (Abbildung 12.9).

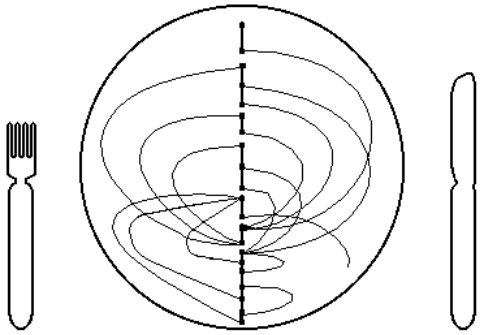


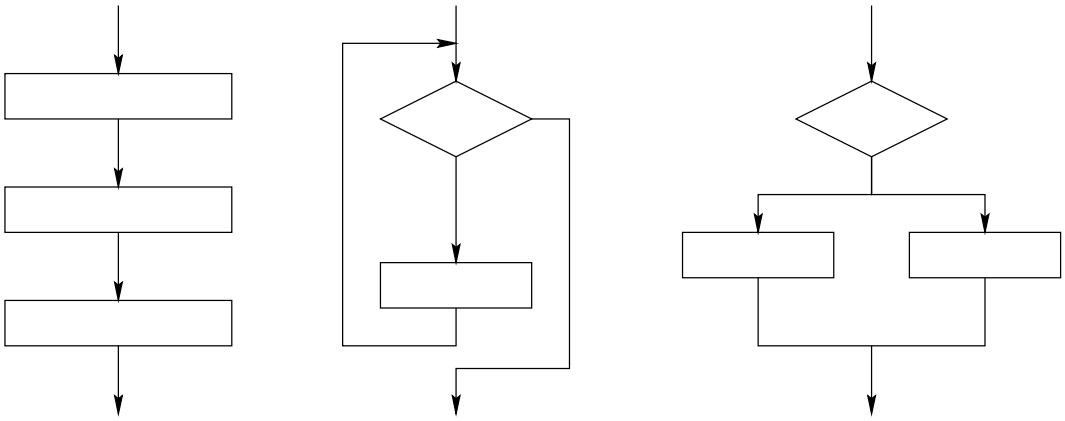
Abbildung 12.9: Spaghettiprogramm

Programme sollten also einer bestimmten Steuerdisziplin unterliegen. Das bedeutet die Beschränkung der Abläufe auf die drei Grundstrukturen (Abbildung 12.10):

- Sequenz,
- Schleife,
- Fallunterscheidungen.

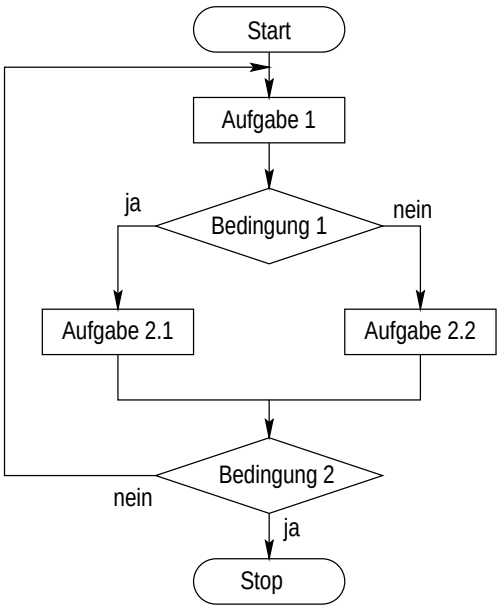
Für die Veranschaulichung von Algorithmen stehen verschiedene Darstellungsmittel zu Verfügung:

- Flußdiagramm,
- Struktogramm,

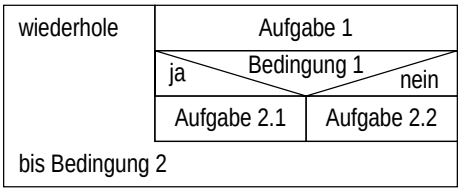


(a) Sequenz (b) Schleife (c) Alternative

Abbildung 12.10: Grundkonstrukte der strukturierten Programmierung



(a) Programmablaufplan



(b) Struktogramm

```
repeat
  Aufgabe 1;
  if Bedingung 1 then
    Aufgabe 2.1;
  else
    Aufgabe 2.2;
until Bedingung 2;
```

(c) Pseudocode

Abbildung 12.11: Darstellungsmethoden für Algorithmen

- Pseudocode,
- Programmiersprache.

Drei dieser Möglichkeiten sind Beispielhaft in Abbildung 12.11 dargestellt.

Mittel	Ausprägung
theoret. Grundlagen	Wirkprinzipien, Modelle
Darstellungsmittel	Diagramme, Tabellen, Sprachen
Organisationshilfen	Vorschriften, Checklisten
Hardware-Werkzeuge	Computer, Ein-/Ausgabemedien
Software-Werkzeuge	Sprachübersetzerprogramme

Tabelle 12.4: Hilfsmittel zur Softwareentwicklung

12.6 Softwarewerkzeuge, Hilfsmittel

Softwarewerkzeuge (Softwaretools) sind programmierte Hilfsmittel des Softwareentwicklers. Nach dem Einsatzgebiet gibt es Werkzeuge zur:

- Textverarbeitung (Editor),
- Sprachübersetzung (Compiler),
- Dokumentationsaufbereitung,
- Testunterstützung,
- Ergebnisverwaltung,
- Projektleitung.

Weitere sind in Tabelle 12.4 aufgelistet.

12.7 Softwareergonomie, Qualitätsmerkmale

Ergonomie ist die Lehre von den Leistungsmöglichkeiten des Menschen mit dem Ziel, die Arbeit dem Menschen anzupassen. *Softwareergonomie* ist, Software so entwickeln, daß sie auf die spezifischen Stärken und Schwächen des Nutzers abgestimmt ist. Besondere Bedeutung hat dabei die Nutzerschnittstelle. Sie bietet alle Eigenschaften des Computersystems bezogen auf das Softwareprodukt an, die der Nutzer von außen wahrnehmen kann, z.B.:

- Steuerung des Programmablaufs,
- Gestaltung der Dateneingaben,
- Gestaltung der Ergebnisausgaben,
- Zeitverhalten des Gesamtsystems.

Die Frage nach dem „Wozu“? innerhalb des SEP läßt sich durch die Übersicht in Tabelle 12.5 beantworten.

Merkmal	Bedeutung
korrekt	Übereinstimmung zwischen Spezifikation (Solleigenschaften), Programmverhalten Dokumentation (Produktbeschreibung)
flexibel	Wartbar, d.h. nachträglich änderbar zwecks Anpassung an neue Nutzerforderungen, Anpassung an geänderte Hard- oder Software, Korrektur von Entwicklungsfehlern, Optimierung der Effizienz. Nachnutzbar mit möglichst geringem Aufwand als Ganzes oder in Teilen, sowie selbständig oder kombiniert.
verständlich	Nutzerverständlich mit geringem Aufwand für Einarbeitung in die sachgerechte Nutzung und Abarbeitung. Entwicklerverständlich mit geringem Aufwand zum Verständnis der inneren Struktur und Funktionsweise.
stabil	sinnvolles Verhalten des Softwareproduktes bei Eingabefehlern, Rechenfehlern und Folgefehlern aus Hardware- oder Systemsoftwarefehlern
effizient	Rechentechnisch effizient, d.h. geringe Inanspruchnahme von Speicherplatz, Rechenzeit, peripheren Geräten und Leistungen des Betriebssystems. Nutzereffizient, d.h. Durchführung und Nachbereitung der Abarbeitung

Tabelle 12.5: Qualitätsmerkmale eines Softwareproduktes

12.8 Weitere Konzepte der Softwareentwicklung

Es versteht sich, daß im Rahmen dieses Kapitels keine ausgiebigen Betrachtungen zum gesamtheitlichen Prozeß der Softwareentwicklung angestellt werden können. Dafür sind Spezialvorlesungen gedacht, die sich u.a. mit:

- objektorientierter Softwareentwicklung,
- Softwarevorgehensmodellen,
- Softwaremetriken,
- Softwaremanagement

auseinandersetzen. Dieses Kapitel soll das Rüstzug liefern, um einen kleinen Schritt vom Programmieren im Kleinen zum Programmieren im Großen gehen zu können.

13 Zusammenfassung

Mit diesem Skript haben die Autoren versucht, dem künftigen Anwender der Rechentechnik aufzuzeigen, daß zur Lösung der eigenen spezifischen Aufgabenstellungen eine Reihe von Methoden und Techniken erforderlich sind. Eine zentrale Rolle spielt hierbei der Algorithmus. Gelingt es einen plausiblen Algorithmus für eine Aufgabe zu finden, so kann dann mit Hilfe einer Programmierumgebung ein lauffähiges Programm entwickelt werden. Den Weg dahin weisen die Methoden und Werkzeuge der Softwaretechnologie.

A Einführung in die Benutzung des `gcc`

Die *GNU Compiler Collection* ist eine Sammlung von freien Compilern für die unterschiedlichsten Programmiersprachen. Neben *C* und seinen objektorientierten Erweiterungen *C++* und *Objective-C* gibt es auch Unterstützung für Pascal, Fortran, ADA und andere weniger bekannte Sprachen. Dieses Compiler-System wird im Rahmen des GNU-Projekts entwickelt und steht deshalb unter der *GPL*. Diese Lizenz garantiert die freie, uneingeschränkte Nutzung und Anpassung dieser Programme, die für jedermann, auch im Quellcode, zugänglich sein müssen. Nur die Lizenz der Programme darf nicht verändert werden. Dies ermöglichte die Portierung der GCC, die ursprünglich nur auf Unix-Rechnern verfügbar war, auf die unterschiedlichsten Betriebssysteme und Rechnerarchitekturen. So gibt es auch Anpassungen für MS DOS (`DJGPP`) und MS Windows (`MingW32` und `Cygwin`).

Die einzelnen Programme der GCC werden mit Hilfe vieler Optionen über die Kommandozeile gesteuert. Zum Übersetzen von einfachen Programmen genügt es, aber einige wenige Optionen zu kennen. Wer weitergehende und umfassendere Hilfe benötigt, der lese in der Dokumentation nach. Da der Compiler ursprünglich aus der Unix-Welt stammt, werden für den Aufruf die Konventionen dieses Betriebssystems verwendet. Alle Optionen werden nicht mit einem Schrägstrich („/“) eingeleitet, sondern mit einem Minuszeichen `-[Option]`. Optionen, deren Name länger als ein Buchstabe ist, beginnen mit zwei Minuszeichen `--[Option]`. Groß- und Kleinschreibung sind zwingend zu beachten, da sonst die Optionen falsch oder gar nicht ausgeführt werden. Selbiges gilt auch für alle Dateinamen. *Optionsparameter* werden direkt hinter den Optionsbuchstaben angegeben. Ist der Optionsname *länger* als ein Buchstabe, dann trennt ein *Gleichheitszeichen* den Parameter vom Namen.

So benötigt man nur noch einen beliebigen Texteditor, um die Quelltexte zur erstellen. Wer es hingegen etwas komfortabler haben will, für den gibt es mittlerweile eine Reihe von integrierten Entwicklungsumgebungen, die den Aufruf des gcc hinter einer graphischen Benutzeroberfläche verbergen. Sie bieten eine Integration von Editor, Compiler, Debugger und Dokumentation unter einer einheitlichen Oberfläche. Beispiele für solche freien IDEs sind `DEV-C++`, `VIDE` (beide Windows) und `RHIDE` (DOS). Diese brauchen sich, in Hinsicht auf Funktion und Komfort, vor ihrer kommerziellen Konkurrenz, wie `Visual C++` oder `C++-Builder` nicht verstecken, und bieten oft einen leichteren Einstieg, da die kommerziellen Produkte auf sehr große Projekte ausgerichtet sind. `Dev-C++` ist seit WS2000/2001 die empfohlene Entwicklungsumgebung für die Lehrveranstaltung „Grundlagen der Informatik für Ingenieure“. Trotzdem sollte man sich mit dem Aufruf des `gcc` einmal auseinandersetzen, da die Komplexität eines Compilers nicht vollständig hinter einer grafischen Fassade versteckt werden kann.

A.1 Übersetzung von C-Programmen

Nehmen wir an, das Programm heißt `prog.c` und soll in `prog.exe` übersetzt werden, dann genügt folgender Befehl in der DOS-Shell:

```
C:\> gcc -o prog.exe prog.c
```

`gcc` ist der C-Compiler. Die Option `-o` gefolgt von `prog.exe` gibt den Namen des zu erzeugenden Programms an. Sonst heißt das erzeugte Programm `a.exe`. Die Datei `prog.c` enthält den C-Quellcode. Besteht das Programm aus mehreren Dateien, z.B.: `prog1.c` und `prog2.c`, dann werden sie durch Leerzeichen getrennt angegeben:

```
C:\> gcc -o prog.exe prog1.c prog2.c
```

Nun gibt es noch eine Vielfalt von anderen Optionen. Hier sind nur die Wichtigsten aufgelistet. Diese kann man in (*fast*) beliebiger Reihenfolge an den ursprünglichen Befehl anhängen:

`-o [program.exe]` Setzt den *Namen* des zu erzeugenden Programms: z.B.:

```
C:\> gcc -o progr.exe prog.c
```

`-c` Compiliert den Quellcode, *ohne* ihn zu *linken* und damit ein ausführbares Programm zu erzeugen. Es entstehen *Objekt-Dateien* mit der Endung „`.o`“. Schalter wie `-o` und `-l` machen deshalb keinen Sinn. Um aus den Objektdateien ein ausführbares Programm zu erhalten, muss man wie beim normalen Übersetzen, direkt aus C-Dateien, vorgehen. Der einzige Unterschied besteht darin, dass man jetzt die Objektdateien angibt. C- und Objektdateien können beim Aufruf des Compilers beliebig gemischt werden. Diese Option macht beim Übersetzen größerer Programme Sinn. Man spart Zeit, indem man nur noch die gerade geänderten Quelldateien übersetzt. Das Linken der Objektdateien geht viel schneller, als eine komplette Neu-Übersetzung. Ein Beispiel:

```
C:\> gcc -c prog.c

C:\> gcc -c prog1.c

C:\> gcc -c prog2.c

C:\> gcc -o prog.exe prog.o prog1.o prog2.o
```

Zum Übersetzen größerer Programme wird dann meist `make` eingesetzt. Siehe dazu auch die Programmdokumentation.

`-O oder -O2` Schaltet verschiedene *Optimierungen* ein. Einer von beiden Schaltern genügt. Bei der Verwendung zusammen mit `-g` kann es zu Problemen kommen (siehe unten).

`-g` Beim Compilieren werden zusätzliche *Debugging-Informationen* erzeugt und in das ausführbare Programm eingefügt. Dies ermöglicht den Einsatz eines Debuggers, wie `gdb` oder `RHIDE`, zur Fehlersuche. Diese Option vergrößert die ausführbare Datei leicht um ein Vielfaches! Bei der Benutzung

mit `-O` oder `-O2` kann es passieren, dass manche Variablen und Schleifen nicht im Debugger angezeigt werden, da sie bei der Optimierung entfernt wurden!

`-Wall` Schaltet *alle Fehlermeldungen und Warnhinweise* beim Übersetzen ein. Das hilft bei der Fehlersuche, da sonst der Compiler nur für die Übersetzung kritische Fehler meldet. Ist die Option eingeschaltet, werden auch Fehler gemeldet, die sonst erst bei der Ausführung auftreten. Es ist deshalb immer ratsam diese Option einzuschalten. Allerdings kann diese Option auch keine Wunder vollbringen. Typische Fehler, die der Compiler nicht entdecken kann sind z.B:

- Hinausschreiben über Feldgrenzen in Schleifen,
- Schreiben in nichtreservierte Speicherbereiche bei der Verwendung von Zeiger-Arithmetik,
- Verwendung nichtinitialisierter Variablen,
- Semantische Fehler.

`-l[libname]` Mit dieser Option werden zusätzliche *Funktionsbibliotheken* eingebunden. Gleich hinter `-l` wird, ohne trennendes Leerzeichen, der `[libname]` angegeben. Der `[libname]` wird entweder in der Dokumentation angegeben, oder man muss einen Blick in das `lib\`-Verzeichnis werfen: Dort findet man viele Dateien, die mit `lib` anfangen und die Endung `.a` haben. Der Teil des Dateinamens dazwischen ist der `[libname]`! Diese Dateien enthalten, thematisch sortiert, den Maschinencode für die verschiedenen Bibliotheksfunktionen. Um zu sehen, welche Funktionen in einer Bibliotheksdatei enthalten sind, gibt es das Programm `nm`. Der Aufruf erfolgt mit:

```
C:\> nm libgcc.a | more
```

Für einfache Programme ist es normalerweise nicht nötig, zusätzliche Bibliotheken anzugeben. Falls doch, so informiert einen die Fehlermeldung `undefined reference` darüber. Mancher Compiler bindet zum Beispiel standardmäßig nicht die mathematischen Funktionen, wie `sqrt` ein. Das manuelle Einbinden der Mathematik-Bibliothek mit `-lm` schafft Abhilfe. Auch beim Übersetzen von OpenGL-Programmen müssen zusätzliche Bibliotheken eingebunden werden (siehe Abschnitt A.3).

A.2 Übersetzung von C++-Programmen

Will man C++-Programme übersetzen, heißt der Compiler `g++`. Argumente und Optionen haben die selbe Bedeutung wie beim `gcc`. Hier ein Beispielaufruf:

```
C:\> g++ -o prog.exe prog1.cpp prog2.cpp -O2 -Wall -g -lm
```

Da in DOS `g++` ein ungültiger Dateiname ist, heißt der C++-Compiler in DJGPP `gxx`.

A.3 Übersetzung von OpenGL-Programmen

A.3.1 Voraussetzungen für die OpenGL-Entwicklung unter Windows

Für das Übersetzen von OpenGL-Programmen müssen die notwendigen Bibliotheken und Include-Dateien bereitgestellt werden.

Seit Windows 95 wird OpenGL von Microsoft unterstützt. Die meisten Compiler unter Windows enthalten die notwendigen Include-Dateien für die Kern- (`GL/gl.h`) und die Utility-Bibliothek (`GL/glu.h`). Die GLUT-Bibliothek (`GL/glut.h`) ist meist nicht vorhanden. Diese drei Dateien müssen im Include-Verzeichnis des Compilers vorhanden sein. Die Bibliotheken, die den Maschinencode für die Ausführung der Funktionen enthalten, haben unter Windows die Endung `dll` und befinden sich im Verzeichnis `c:\windows\system\`. Für OpenGL benötigt dort die Dateien `opengl32.dll`, `glu32.dll` und `glut32.dll`.

Die zur Vorlesung bereitgestellte Compiler-CD enthält diese notwendigen Dateien mit Installationsanweisungen. Diese Dateien sind auch im Internet verfügbar. Die OpenGL-Homepage `http://www.opengl.org/` bietet die OpenGL- und GLU-Bibliothek für eine Reihe von Betriebssystemen und Compiler an. Sie bietet auch eine umfassende Dokumentation zu diesem Thema. Unter `http://www.pobox.com/~ndr/glut.html` findet man alles rund um GLUT.

A.3.2 Aufruf des Compilers

Zum Übersetzen von OpenGL-Programmen müssen zusätzliche Optionen angegeben werden. Der Compiler muss eine Windows-Anwendung erzeugen und die OpenGL-Bibliotheken einbinden. Mit der Option `-mwindows` erzeugt der Compiler ein Windows-Programm. Die notwendigen Bibliotheken werden mit dem oben beschriebenen Schalter `-l` angegeben. Unter Windows heißen die Bibliotheken `opengl32`, `glu32` und `glut32`.

Die folgende Zeile übersetzt z.B. das Nikolaus-Programm aus Abschnitt 10.5.3.

```
C:\> gcc -o nikolaus.exe nikolaus.c -mwindows -lopengl32 -lglu32 -lglut32
```

Um ein OpenGL-Programm unter einem anderen Betriebssystem übersetzen und ausführen zu können, müssen die nötigen Bibliotheken bereitgestellt werden.

B Adressen zu weiterführenden Seiten im Internet

Das Internet bietet eine Fülle von weiterführenden Dokumentationen, Tutorials und Software. In folgender Tabelle sind einige Startpunkte zusammengetragen, die für weitergehende Recherchen hilfreich sein können. Diese Adressen können natürlich nur einen kleinen Ausschnitt der Fundgrube Internet darstellen. Auch sind sie nicht mehr als ein Schnappschuss, so dass es leicht möglich ist, dass einige dieser Adressen nicht mehr aktuell sind oder einige der Angebote vollständig eingestellt wurden. Oft bieten Internet-Kataloge, wie *Yahoo* oder *Web.de*, mit übersichtlich, in Kategorien sortierten und kommentierten Link-Listen schnellen Zugriff auf Seiten zum gewünschten Thema. Wird man auch dort nicht fündig, so bietet sich die Nutzung einer der vielen Suchmaschinen, wie *MetaGer*, *Altavista*, *HotBot* oder *Go2Net*, an. Allerdings sind deren Ergebnisse nicht redaktionell betreut, so dass man selber die Spreu vom Weizen trennen muss.

C/C++ Programmierung	
C/C++ Programming Resources	`http://www.cprogramming.com/`
C++ Ecke	`http://www.c-plusplus.de/`
Programming Tutorials	`http://www.gustavo.net/programming/`
The Dinkum C++ Library Reference	`http://www.dinkumware.com/refcpp.html`
Grafikprogrammierung	
OpenGL	`http://www.opengl.org/`
GLUT	`http://reality.sgi.com/mjk/glut3/glut3.html`
GLUT for Win32	`http://www.pobox.com/~ndr/glut.html`
Internetseiten mit Software zum Thema Programmierung	
GNU Win32 Projekte	`http://www.nanotech.wisc.edu/~khan/software/gnu-win32/`
Colin Peters GNU WIN32 Homepage	`http://www.geocities.com/Tokyo/Towers/6162/`
Developer City	`http://www.thefreecountry.com/developercity/`
Freie Software	`http://sourceware.cygnus.com/`
GNU-Projekt	`http://www.gnu.org/gnu/`
Free Software Foundation	`http://www.gnu.org/fsf/`
Programming stuff	`http://www.site-eerie.ema.fr/~merle/Prog.html`

Freie Compiler	
GCC	http://gcc.gnu.org/
Mingw32	http://www.mingw.org/
DJGPP	http://www.delorie.com/
LCC-Win32	http://www.cs.virginia.edu/~lcc-win32/
Editoren	
UltraEdit	http://www.ultraedit.com/
Foldmaster	http://www.foldmaster.de/
XEmacs	http://www.xemacs.org/
Integrierte Entwicklungsumgebungen	
Dev-C++	http://www.bloodshed.net/
VIDE	http://www.objectcentral.com/
JFE	http://cs-alb-pc3.massey.ac.nz/
Tools	
Acrobat Reader	http://www.adobe.com/
GSview	http://www.cs.wisc.edu/~ghost/
WinZIP	http://www.winzip.com/

Literaturverzeichnis

[BS95] Bronson, Gary J. and Howard Silver: *C for Engineers and Scientists: an Introduction to Programming with ANSI C*. Thomson Learning, 2 edition, 1995.

[CFH+97] Chin, Norman, Chris Frazier, Paul Ho, Zicheng Liu, Kevin P. Smith, and Jon Leech (Editor): *The OpenGL Graphics System Utility Library (Version 1.3)*. Silicon Graphics, Inc., 1992–1997.

[Dum00] Dumke, Reiner: *Software-Engineering: Eine Einführung für Informatiker und Ingenieure: Systeme, Erfahrungen, Methoden, Tools*. Vieweg-Lehrbuch. Vieweg, Braunschweig, 2000.

[Fol99] Foley, James D.: *Computer Graphics: Principles and Practice* . Addison-Wesley Systems Programming Series. Addison Wesley Longman, Inc., Reading, Massachusetts, 1999.

[FvDF+90] Foley, James D., Andries van Dam, Steven K. Feiner, John F. Hughes, Richard L. Phillips, and Peter S. Gordon (Editor): *Introduction to Computer Graphics*. Addison-Wesley Publishing Company, Inc., 1994, 1990.

[GL90] Goldschlager, Les und Andrew Lister: *Informatik: eine moderne Einführung*. Hanser-Studienbücher. Carl Hanser Verlag, München, Wien, 2. Auflage, 1990.

[Het93] Hetze, Bernd: *Programmieren in C: Einführung in die Sprache; Übungen am PC*. Studienliteratur Informatik. VMS, Verl. Modernes Studieren, Hamburg, 1. Auflage, 1993. zweibändiges Werk.

[Hor95] Horn, Christian (Herausgeber): *Lehr- und Übungsbuch Informatik*. Carl Hanser Verlag, München, Wien, 1. Auflage, 1995.

[HS99] Heuer, Andreas und Gunter Saake: *Datenbanken: Konzepte und Sprachen*. Informatik Lehrbuch-Reihe. MITP-Verlag, Bonn, 2. Nachgedr. Auflage, 1999.

[Job87] Jobst, Eberhard (Herausgeber): *Mikroelektronik und künstliche Intelligenz*. Akademieverlag, Berlin, 1. Auflage, 1987.

[Kil96a] Kilgard, Mark J.: *The OpenGL Utility Toolkit (GLUT) – Programming Interface (API Version 3)*. Silicon Graphics, Inc., 1996.

[Kil96b] Kilgard, Mark J.: *OpenGL^{TM} – Programming for the X Window System*. Addison-Wesley, first edition, 1996.

[KPZ95] Kopacek, Peter, Robert Probst und Martin Zauner: *Informatik für Maschinenbauer*. Springer-Lehrbuch-Technik. Springer-Verlag Inc., Berlin, Heidelberg, 1. Auflage, 1995.

[KR90] Kernighan, Brian W. und Dennis M. Ritchie: *Programmieren in C: mit dem C-reference-Manual in deutscher Sprache*. PC professionell. Carl Hanser Verlag, München, Wien, 2. Auflage, 1990.

[Lor94a] Lorenz, Peter: *Computergrafik 1.* Vorlesungsmanuskript, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, August 1994.

[Lor94b] Lorenz, Peter: *Computergrafik 2.* Vorlesungsmanuskript, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, August 1994.

[Men97] Menge, Reinald: *Offener Jurassic Park*. SE – Software Entwicklung, Februar 1997.

[Rec94] Rechenberg, Peter: *Was ist Informatik? – Eine Allgemeinverständliche Einführung*. Carl Hanser Verlag, München, Wien, 2., bearb. und erw. Auflage, 1994.

[Reg91] Regionales Rechenzentrum für Niedersachsen/Universität Hannover (RRZN): *Die Programmiersprache C – Ein Nachschlagewerk*, 4., unveränderte Auflage, März 1991.

[Reg93] Regionales Rechenzentrum für Niedersachsen/Universität Hannover (RRZN): *Die Programmiersprache C++ – Ein Nachschlagewerk*, ? Auflage, ? 1993.

[RL99] Rembold, Ulrich und Paul Levi: *Einführung in die Informatik für Naturwissenschaftler und Ingenieure*. Carl Hanser Verlag, München, Wien, 3., vollst. überarb. und erw. Auflage, 1999. Lehrbuch.

[SAF97] Segal, Mark, Kurt Akeley, and Chris Frazier (Editor): *The OpenGL Graphics System – A Specification (Version 1.1)*. Silicon Graphics, Inc., 1992–1997.

[Sch92] Schulz, Arno: *Software-Entwurf: Methoden und Werkzeug*. 3., verb. Oldenbourg„ München, 1992.

[Sed92] Sedgewick, Robert: *Algorithmen in C++*. Addison-Wesley Verlag, ein Imprint der Pearson Education Company Deutschland GmbH, München, 1. Auflage, 1992.

[SH87] Schefe, Peter und Michael Hussmann: *Informatik – eine konstruktive Einführung: LISP, PROLOG und andere Konzepte der Programmierung*. Nummer 48 in *Reihe Informatik*. BI-Wissenscahftsverlag, Mannheim, 2. überarb. Auflage, 1987.

[SH99] Saake, Gunter und Andreas Heuer: *Datenbanken: Implementierungstechniken*. MITP-Verlag, Bonn, 1999.

[SST97] Saake, Gunter, Ingo Schmitt und Can Türker: *Objektdatenbanken: Konzepte, Sprachen, Architekturen*. Informatik Lehrbuch-Reihe. International Thomson Publishing, Bonn, 1997.

[Str98] Stroustrup, Bjarne: *Die C++-Programmiersprache*. Addison-Wesley Verlag, ein Imprint der Pearson Education Company Deutschland GmbH, München, 3., aktualisierte und erw. Auflage, 1998.

[Wer95] Werner, Dieter (Herausgeber): *Taschenbuch der Informatik*. Fachbuchverlag, Leipzig, 2., völlig neu bearb. Auflage, 1995.

[Wil95] Willms, Gerhard: *Das C-Grundlagen-Buch: der Grundstein für erfolgreiches Programmieren mit C*. Data-Becker, Düsseldorf, 2., überarb. Auflage, 1995.

[Wil96] Willms, André: *C-Programmierung: Programmiersprache, Programmiertechnik, Datenorganisation*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1996.

[Wil99] Willms, André: *C++-Programmierung: Programmiersprache, Programmiertechnik, Datenorganisation* . Addison Wesley Longman, Inc., Reading, Massachusetts, 4 Auflage, 1999.