Grundlagen der Softwaretechnik

Teil 1 Einführung

Leitfaden zur Vorlesung

Univ.-Prof. Dipl.-Ing. A. Reinhardt
Produktionssysteme
Inst. f. Produktionstechnik und Logistik
Universität Kassel
Tel. +49 561 8042693, Fr. Treskow
www.fps.maschinenbau.uni-kassel.de

WS 2003/04

Grundlagen der Softwaretechnik Teil 1

Inhaltsverzeichnis

1 Vorwort 1
2 Datenverarbeitung und Erkenntnisprozess 2
2.1 Schnelle Zahlenverarbeitung............................ 2
2.2 Objekt-Subjekt-Modell-Relation......................... 3
2.2.1 Objekt-Subjekt-Beziehung 4
2.2.2 Subjekt-Modell-Beziehung 4
2.2.3 Modell-Objekt-Beziehung 4
2.3 Weltsicht in der Softwaretechnik....................... 5
3 Manuelle Lösung einer Aufgabe 5
3.1 Der Förster und sein Gehilfe........................... 5
3.2 Lösungsansätze... 5
3.2.1 Verbaler Ansatz 5
3.2.2 Formalisierter Ansatz 7
4 Schritte der Programmentwicklung 9
4.1 Entwerfen.. 9
4.2 Codieren.. 12
4.3 Übersetzen.. 13
4.4 Binden.. 14
4.5 Laden und Ausführen................................... 15
4.6 Testen.. 16
5 Struktur eines Rechnersystems 17
5.1 Eine Rechnerkonfiguration............................. 18
5.2 Betriebssysteme....................................... 19
5.3 Dokumentation... 20
6 Übungsaufgaben 20
6.1 C/C++-Quellcode HELLO................................. 21
6.2 FORTRAN-Quellcode HELLO............................... 21
6.3 MinMax in C/C++....................................... 22
6.4 MinMax in FORTRAN..................................... 23
6.5 Rechnernetzwerk....................................... 24
6.6 Dateistruktur unter UNIX............................. 255
7 Anhang 26

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

1 Vorwort

Das vorliegende Skript dient als Leitfaden für die Lehrveranstaltung "Grundlagen der Softwaretechnik" im Fachbereich Maschinenbau der Universität Kassel.

Die Lehrveranstaltung richtet sich an Studenten der ersten Semester der Universität und setzt keine EDV-Kenntnisse voraus. Ebensowenig wird ein tiefgreifender theoretischer oder technischer Sachverstand vorausgesetzt. Anhand exemplarischer Aufgaben aus dem "Alltagsleben" soll eine Technik der Softwareentwicklung erarbeitet werden. Die Lehrveranstaltung ist damit auch ein Angebot an Alle, die sich mit der Materie Software und Computer befassen wollen.

Der Rechner soll als Werkzeug zur Lösung der Aufgaben erarbeitet und eingesetzt werden. Zur formalen Beschreibung von Lösungen wird die Programmiersprache C und C++ als aktueller Standard eingesetzt. Die Geschichte der Programmiersprachen wird aufgezeigt. (http://www.fps.maschinenbau.uni-kassel.de dann Forschung und Publikationen).

Das Skript darf nicht als Lehrbuch missverstanden werden. Für die Einarbeitung in die Softwaretechnik sind Bücher notwendig. Wichtigste Voraussetzung ist jedoch das Arbeiten im Dialog an einem Rechner.

Mit dieser Lehrveranstaltung sind zwei hochgesteckte Ziele verbunden. Die Teilnehmer sollen ein fundiertes Wissen und die handwerklichen Fähigkeiten erlangen, um mit dem Instrument Rechner elegante Lösungen zu erarbeiten bzw. Software anwenden zu können.

Sie sollen weiter ein Minimum an Sensibilität im Umgang mit dieser Technik erlangen und erkennen, dass jedes Programm ein Modell realer Zusammenhänge darstellt und als Mittel zur Steuerung oder ganz allgemein Beeinflussung dieser Zusammenhänge eingesetzt wird.

Viele Menschen, insbesondere junge, tragen heute täglich einen Rechner mit sich, ohne es zu wissen. Oder, wer weiß schon wie ein Handy funktioniert? Im Rahmen der Lehrveranstaltung werden wir auch dies erfahren.

Kassel, Oktober 2003

A.Reinhardt

2 Datenverarbeitung und Erkenntnisprozess

2.1 Schnelle Zahlenverarbeitung

Mit dem Begriff “Informationstechnik (IT)“, "Datenverarbeitung" oder "Elektronische Datenverarbeitung", abgekürzt EDV, beschreiben wir landläufig Vorgänge, in denen wir Zahlenreihen beliebiger Länge sehr schnell und hocheffizient mit einem Automaten, genannt Rechner oder neuhochdeutsch Computer, verarbeiten. Ziel dieser Arbeit ist es immer, aggregierte Aussagen zur Charakteristik dieser Zahlen in statistischer Form oder auch individuelle Zahlen aus der Menge dieser Zahlen zu bestimmen.

Nehmen wir an, wir verarbeiten Zahlen aus der Einwohnermeldekartei. Auf einer Karteikarte stehen Eigenschaften individueller Einwohner wie Adresse, Geburtsdatum, Größe und weitere Merkmale wie Schuhgröße und Augenabstand als typische Eigenschaften.

Nehmen wir weiter an, wir wollen alle Schuhgrößen mit den Werten 45 ermitteln.

Wenn die Eigenschaften der Einwohner auf Kartei nur existieren, dann werden wir die Schuhgrößen manuell ermitteln. Sind die Eigenschaften jedoch auf "Datenträger" gespeichert, dann können wir diese durch einen Computer mit unserem Programm ermitteln lassen.

Die Verfahrensschritte sind für beide Speichermedien gleich. Nur die Verarbeitungszeit ist unterschiedlich.

Gehen wir davon aus, dass bei manueller Ausfertigung der Zugriff und die Verarbeitung einer Karteikarte 36 sec dauert, so muss für eine Stadt mit 100000 Einwohnern eine Verarbeitungszeit von ca. 100 Stunden angesetzt werden. Sind die Daten auf Magnetplatte gespeichert, dann dauert die Verarbeitung etwa 17 Minuten, bei einer Zugriffs- und Verarbeitungszeit von 10 msec.

Das Verarbeiten von Zahlen oder Daten mit dem Computer bringt also erhebliche zeitliche Vorteile. Da die Kosten für solche Auswertungen sehr gering sind, könnte man beliebige Fragestellungen auf der Basis vorhandener Datensammlungen beantworten.

Die Frage könnte nun entstehen nach dem Warum solcher Auswertungen. Warum interessiert uns die Schuhgröße, das Gewicht oder der Augenabstand der Einwohner einer Stadt? Was passiert eigentlich mit dem individuellen Einwohner, wenn wir ihn mit unseren Programmen bearbeiten? Oder, welche Datenspuren hinterlässt ein Handy-Besitzer in der Landschaft und wer könnte damit was anfangen?

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

2.2 Objekt-Subjekt-Modell-Relation

In der Philosophie werden Modelle diskutiert, die die Struktur und den Prozess unserer Erkenntnisgewinnung beschreiben. In einem dieser Modelle (Abb. 1) wird die Struktur beschrieben durch:

- den Objektbereich mit den Gegenständen oder den Objekten, die wir betrachten
- das Subjekt, das sind wir, die Erkenntnissuchenden bzw. die, die sich mit dem Objektbereich beschäftigen und diesen beeinflussen wollen
- das Modellmedium, das uns als Hilfsmittel für unser Wollen, Denken und Tun zur Verfügung steht

Der Prozess in dieser Struktur wird durch Teilprozesse beschrieben, die in drei Beziehungen ablaufen.

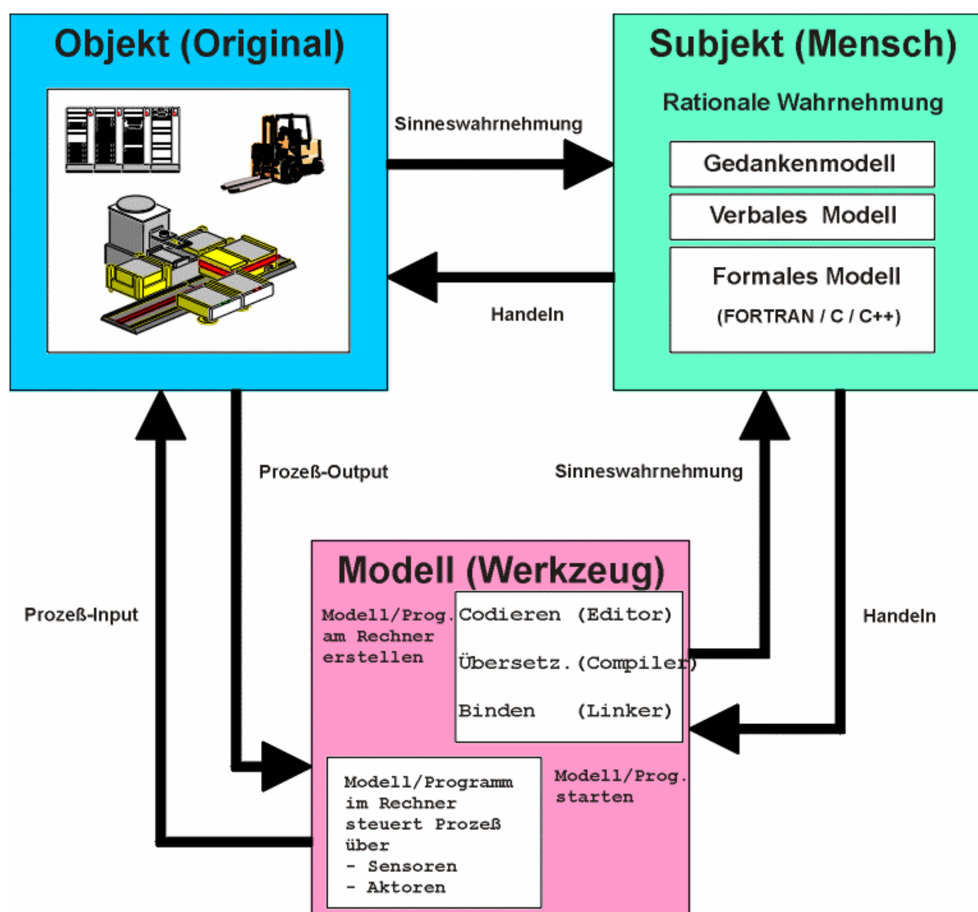


Abbildung 1 Datenverarbeitung und Erkenntnisprozess

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

2.2.1 Objekt-Subjekt-Beziehung

- Sinneswahrnehmung
- rationale Wahrnehmung
- Entwicklung von Gedankenmodellen
- Handeln

2.2.2 Subjekt-Modell-Beziehung

- verbale Formulierung des Gedankenmodells
- formalisierte Beschreibung, z.B. mit Fachsprachen und Zeichnung
- formale Beschreibung und Umsetzung im Computer
  (vgl. Verifizierung,/Validierung in VDI Richtlinie 3633)
- Experiment und Erkenntnisgewinnung am Computermodell

2.2.3 Modell-Objekt-Beziehung

- Sollwerte ausgeben an den Objektbereich über Aktoren
- Istwerte erfassen mittels Sensoren und dem Subjektbereich mitteilen
- Auf Soll-Ist-Abweichung mit modellierten Entscheidungsregeln handelnd eingreifen oder Handlungsanweisung vom Subjekt einholen

2.3 Weltsicht in der Softwaretechnik

Es gibt zwei extreme Verfahren, mit denen wir den Objektbereich sehen, erkennen, modellieren und beeinflussen können. Sie sind phänomen- oder strukturorientiert. Abhängig von der Zielsetzung oder Aufgabenstellung, dem Modellmedium, unserer Sozialisation und Ausbildung werden wir das eine oder andere Verfahren mehr oder weniger anwenden.

3 Manuelle Lösung einer Aufgabe

3.1 Der Förster und sein Gehilfe

Nach einem Sturm sind in einem Wald Hunderte von Bäumen entwurzelt und gefallen. Der Förster benötigt Geld, um den Wald wieder aufzuforsten. Die Bäume werden entastet, und die Stämme liegen wie sie gefallen sind. Der Förster und sein Gehilfe haben nun die Aufgabe, die Baumstämme zu vermessen. Sie sollen über die Menge der Baumstämme quantifizierende Aussagen für den Verkauf machen. Sie entscheiden, die Werte für die kürzeste, längste und mittlere Baumlänge zu ermitteln. Der Förster will High-tech einführen und deshalb die quantifizierenden Aussagen durch einen Computer ermitteln. Er formuliert sein erstes Programm und stößt dabei auf das Problem der Initialisierung als grundlegendes Problem der Softwaretechnik.

3.2 Lösungsansätze

3.2.1 Verbaler Ansatz

Der Förster steht vor seinen Baumstämmen und weiß, er muss das, was er sieht, abbilden. Er kneift die Augen zu, um aufgabenbezogen den Horizont zu beschränken (Abstraktion). Der Gehilfe wird die Länge der einzelnen Baumstämme messen und ihm eine Reihe von Zahlen zurufen. Am Ende der Reihe ruft der Gehilfe "nichts mehr da" oder "Null Länge". Der Förster weiß dann sofort, was die größte, die kleinste und mittlere Zahl bzw. Länge eines Baumstammes war. Der Förster hat keinen portablen PC mit. Er weiß aber, wenn er sich fünf Größen merkt oder auf eine Tafel schreibt, dann kann er am Schluss das Ergebnis nennen.

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

Er merkt sich folgende Größen:

- aktuelle Länge
- Anzahl der Längen
- Summe der Längen
- kleinste Länge
- größte Länge

Der Förster weiß, wenn er eine Vorgehensweise aufschreibt und als richtig erkannt hat, kann er diese Beschreibung des Vorgehens oder Verfahrens delegieren an den Gehilfen. Er versucht deshalb, in mehreren Ansätzen das richtige Lösungsverfahren zu ermitteln. Der Förster markiert fünf Felder auf seiner Tafel, in die er die zu merkenden Größen eintragen will. Er schreibt in alle fünf Felder eine "Null" als Ausgangswerte (Initialisierung) und entwickelt ein Verfahren in einzelnen Schritten:

(1) Felder auf "Null" setzen

(2) Die aktuelle Länge (erste Länge) messen und sie in das Feld "aktuelle Länge" schreiben

(3) Das Feld "Anzahl der Längen" um "eins" erhöhen

(4) Das Feld "Summe der Längen" um den Wert in "aktuelle Länge" erhöhen

(5) Wenn der Wert in "aktuelle Länge" kleiner ist als der Wert in "kleinste Länge", dann den Wert aus "aktuelle Länge" in das Feld "kleinste Länge" setzen

(6) Wenn der Wert in "aktuelle Länge" größer ist als der Wert in "größte Länge", dann den Wert aus "aktuelle Länge" in das Feld "größte Länge" setzen

(7) Nächsten Wert abrufen und in "aktuelle Länge" setzen

(8) "Anzahl der Längen" um "eins" erhöhen

(9) "Summe der Längen" um den Wert in "aktuelle Längen" erhöhen

(10) Wenn der Wert in "aktuelle Länge" kleiner als der Wert in "kleinste Länge" ist, dann die "aktuelle Länge" in "kleinste Länge" setzen.

An dieser Stelle erkennt der Förster, dass sich die Verfahrensschritte (3) bis (7) immer wiederholen.

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

3.2.2 Formalisierter Ansatz

Der Förster überdenkt seine Aufgabenstellung und entwickelt mehrere Lösungsstufen, in denen er zunächst die Grobstruktur in Schritten entwickelt.

Grobstruktur

(1) Die Eigenschaften der Baumstämme benennen, die aufgabenbezogen zu einer Lösung beitragen und auf der Tafel Felder benennen, in denen die Eigenschaften festgehalten werden sollen.

(2) Den Ablauf in Schritten beschreiben, wie sich die Werte in den bezeichneten Tafelfeldern ändern müssen, um den Mess- und Verarbeitungsvorgang abzubilden.

Feinstruktur

(1) Vereinbarung

(1.1) Feld für "aktuelle Länge"

(1.2) Feld für "Anzahl der Längen"

(1.3) Feld für "Summe der Längen"

(1.4) Feld für "kleinste Länge"

(1.5) Feld für "größte Länge"

(2) Ablauf

(2.1) Initialisierung der Felder

(2.1.1) "aktuelle Länge" auf "Null" setzen

(2.1.2) "Anzahl der Längen" auf "Null" setzen

(2.1.3) "Summe der Längen" auf "Null" setzen

(2.1.4) "kleinste Länge" auf "Null" setzen

(2.1.5) "größte Länge" auf "Null" setzen

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

(2.2) Dialog

(2.2.1) Gehilfe, liefere die Baumlängen oder melde "Null", wenn nichts mehr da ist

(2.2.2) Den ersten Wert abrufen und in das Feld "aktuelle Länge" setzen

(3) Verarbeiten der Längen

(3.1) Solange der Wert in "aktuelle Länge" größer "Null" ist, tue folgendes:

1. "Anzahl der Längen" um "eins" erhöhen

2. "Summe der Längen" um den Wert in "aktuelle Länge" erhöhen

3. Wenn der Wert in "aktuelle Länge" kleiner ist als der Wert in "kleinste Länge", dann übertrage den Wert aus "aktuelle Länge" in "kleinste Länge"

4. Wenn der Wert in "aktuelle Länge" größer ist als der Wert in "größte Länge", dann übertrage den Wert aus Aaktuelle Länge≅ in "größte Länge"

5. Rufe den nächsten Wert ab und übertrage ihn in das Feld "aktuelle Länge"

(4) Ergebnis an den Vertrieb melden

(4.1) Den Wert aus dem Feld "Anzahl der Längen" melden

(4.2) Den Wert aus dem Feld "kleinste Länge" melden

(4.3) Den Wert aus dem Feld "größte Länge" melden

(4.4) Den Wert aus der Division der Werte aus ASumme der Längen≅ und "Anzahl der Längen" als mittlere Länge melden

Der Förster weiß, daß in den Ergebnissen Fehler sind. Er überprüft anhand einfacher Zahlen das Programm (Verifizierung) und erkennt, daß ihm bei der Initialisierung ein Fehler unterlaufen ist.

Die Initialisierung ist eine der Hauptquellen für Fehler in der Software. Sie ist abhängig von der Aufgabe und dem Lösungsverfahren.

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

4 Schritte der Programmentwicklung

Die Programmentwicklung ist ein Prozess mit dem Ziel, eine gegebene Aufgabe durch einen Automaten in der Form eines Rechners lösen zu lassen.

Ein Rechner kann

* Zustände speichern und

* gespeicherte Zustände verknüpfen.

Die Lösung einer Aufgabe muß deshalb schrittweise strukturiert und so weit verfeinert werden, bis sie im simplen Formalismus eines Rechners beschreibbar ist.

Durch die Entwicklung problemorientierter Programmiersprachen werden Teile dieses Prozesses durch Rechnerprogramme unterstützt oder automatisch durchgeführt.

Das Entwerfen eines Programmes bzw. die Entwicklung einer formalisierten Lösung bleibt als wesentlicher Schritt dem Programmentwickler vorbehalten.

4.1 Entwerfen

Der systematische Entwurf von Programmen oder allgemeiner ausgedrückt von Software steht heute im Mittelpunkt der Diskussion in der Praxis und in der Forschung.

Mit dem Begriff "Software Engineering" wird eine Fachdisziplin beschrieben, die sich mit der Entwicklung von

* Entwurfsmethoden

* Entwurfswerkzeugen

* Programmiersprachen

befasst.

Entwurfsmethoden geben Regeln für eine bestimmte Verfahrensweise der Problem-strukturierung vor. Sie werden durch entsprechende Werkzeuge unterstützt.

Heute gibt es jedoch weder eine abgeschlossene Theorie noch eine allgemein gültige Methode des Programmentwurfs.
Es haben sich bestimmte Techniken eingebürgert, die mehr oder minder gut zum Programmentwurf geeignet sind.

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

Baumstruktur

Eine Methode besteht in einer baumorientierten Lösungsbeschreibung. Eine Lösung wird schrittweise zergliedert und verfeinert.

Eine Aufgabe, d. h. eine offene Lösung besteht danach aus einem

Stamm

Die Lösung wird in einem ersten Schritt zerlegt und besteht aus

Ästen

Die Elemente der ersten Zerlegung werden verfeinert. Sie bestehen dann aus

Zweigen

Eine weitere Verfeinerung führt zu

Blättern

Die Blätter sind die tiefste Ebene der Verfeinerung. Sie können im Formalismus einer Programmiersprache beschrieben werden.

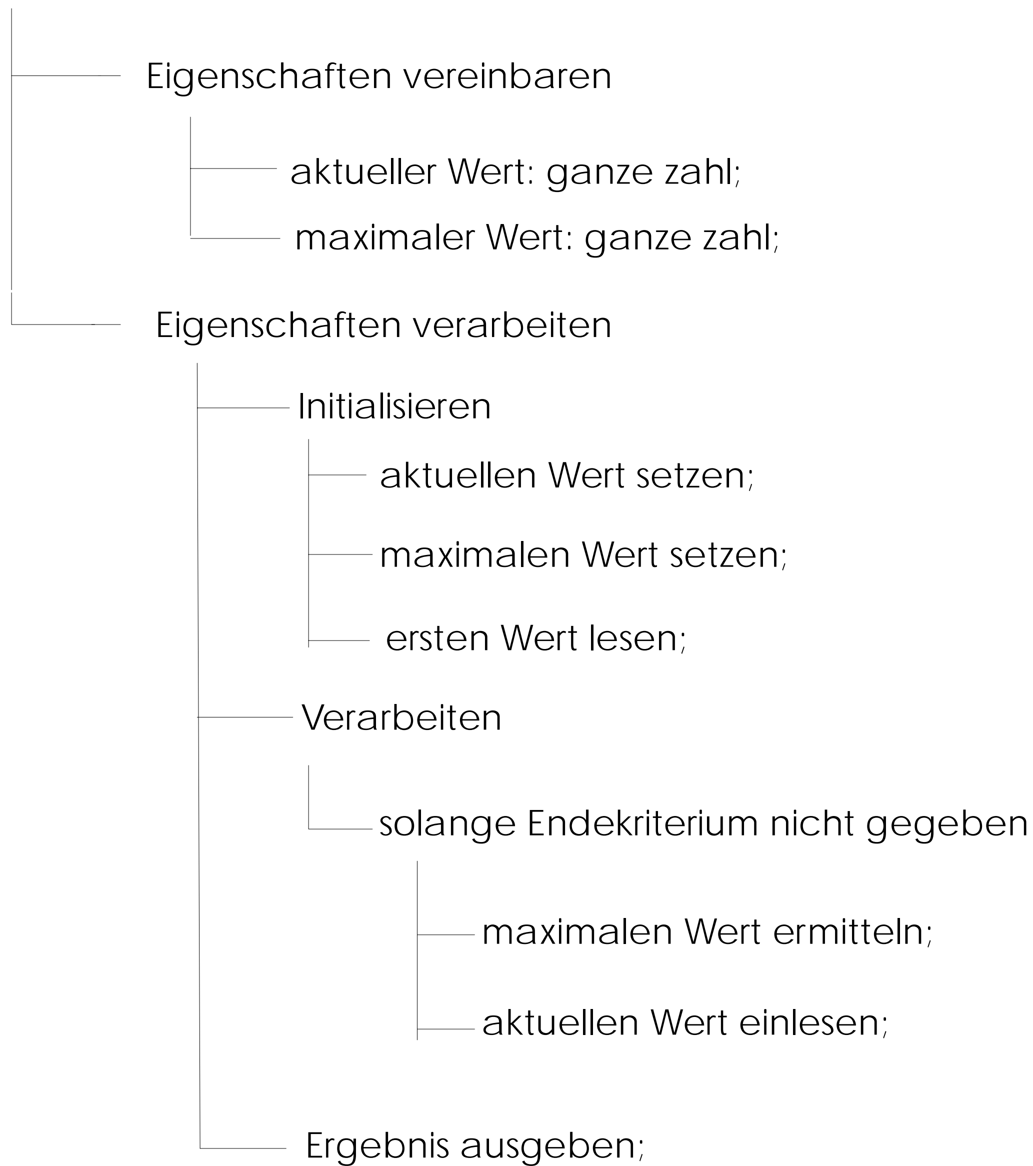
Die Gesamtheit der Blätter ergibt das Programm als formale Beschreibung der Lösung.

In diesem Prozeß der schrittweisen Verfeinerung kann ein Zweig wieder als Ast betrachtet und damit weiter verfeinert werden. Die Lösungsstruktur kann auf diese Weise beliebig tief verfeinert werden.

An einem Beispiel soll diese Vorgehensweise weiter erläutert und geübt werden. In Kap.6 finden Sie Programme in FORTRAN und C als Lösungen.

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

Aufgabenlösung als Baumstruktur



4.2 Codieren

In der frühen Phase der Datenverarbeitung wurden Programme auf Lochkarten oder Lochstreifen codiert und in dieser Form zur Verarbeitung in den Rechner eingegeben. Heute gibt es auf jedem Rechner ein Dienstprogramm, das die Programmcodierung im Dialog am Rechner ermöglicht.

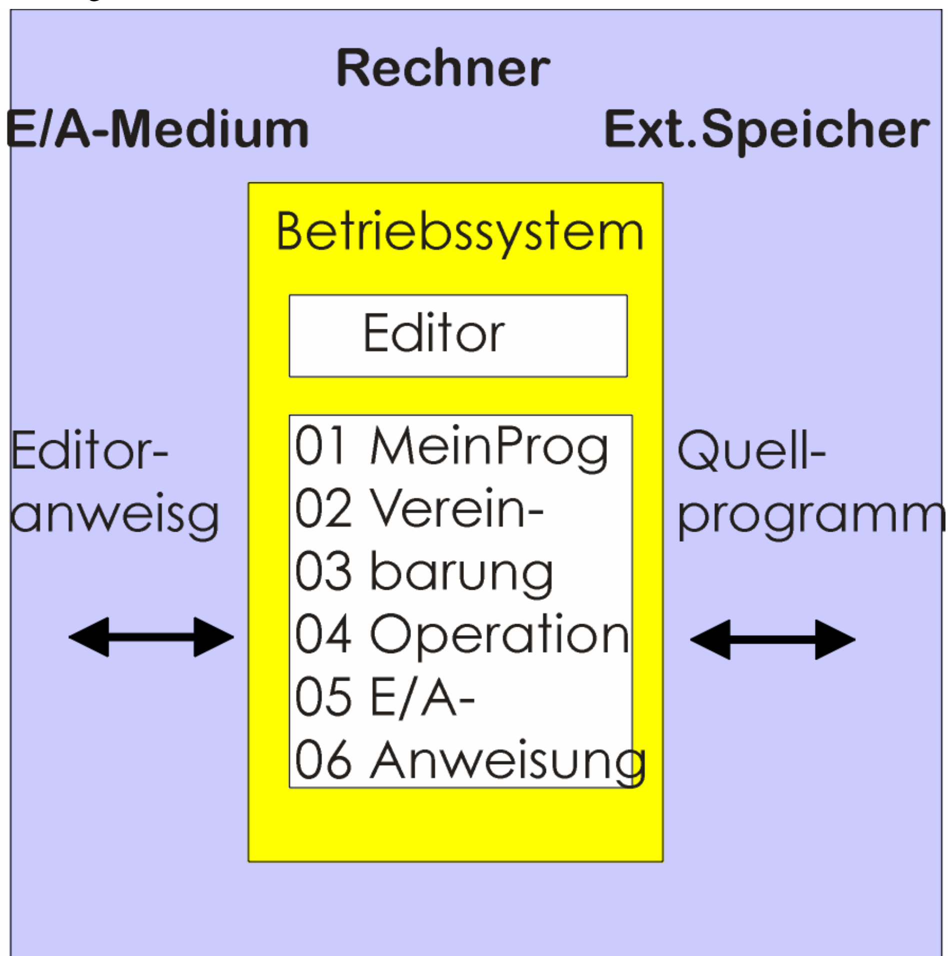


Abbildung 2 Codieren

Dienstprogramme dieser Art werden Editoren genannt. Sie sind Textverarbeitungsprogramme, die folgende Funktionen enthalten:

* Eröffnen oder Öffnen von Dateien auf externen Speichermedien
* Eingabe von Text
* Löschen von Zeilen und Zeichen
* Einfügen von Zeilen und Zeichen
* Suchen von Zeichenketten
* Ersetzen von Zeichenketten
* Schließen bzw. Ablegen von Dateien

Ein Editor verarbeitet ein Programm als Text, der aus einzelnen Zeichen besteht.

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

4.3 Übersetzen

Ein abgelochter oder auf einem externen Speichermedium eines Rechners abgelegter Programmtext wird Quellprogramm genannt und kann in einer beliebigen Sprache formuliert sein.

Das Quellprogramm wird mit einem Sprachübersetzer (Compiler) in den Formalismus des jeweiligen Rechners (Maschinencode) übertragen. Der erzeugte Code wird Objektprogramm genannt.

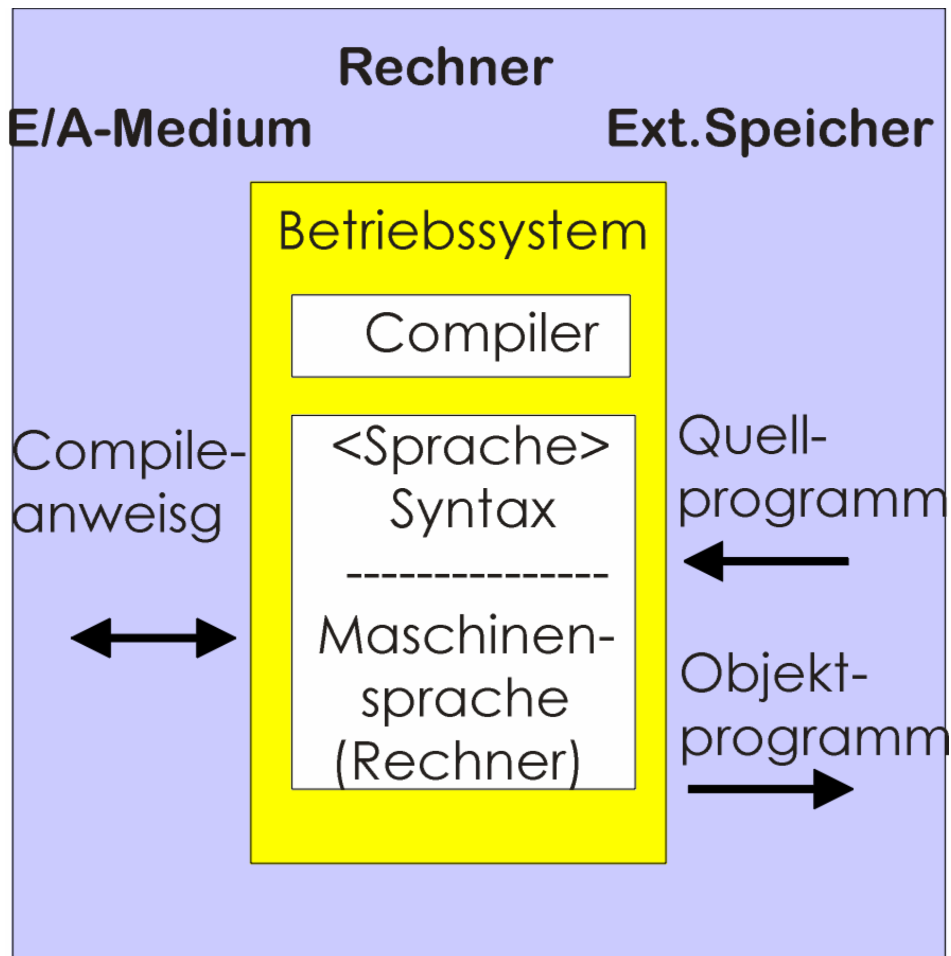


Abbildung 3 Übersetzen

Übersetzen eines Programmes:

Der Compiler überprüft das Quellprogramm auf die Konvention der Sprache

* Schlüsselworte
* Grammatik (Syntax)

und überträgt die einzelnen Anweisungen der Programmiersprache in einer weiteren Stufe der Verfeinerung in die Befehle der Maschinensprache. In einem Übersetzungsprotokoll (Listing) wird das Quellprogramm mit Fehlermeldungen und zusätzlicher Information erzeugt.

4.4 Binden

Zu einem Compiler und zum Betriebssystem eines Rechners gehören Bibliotheken, in denen Funktionen wie Ein/Ausgabeoperationen, mathematische Funktionen etc. als übersetzte Programmsegmente bereits gesammelt sind.

In das Objektprogramm werden diese Funktionen eingebunden und damit lauffähige (executable) Programme erstellt.

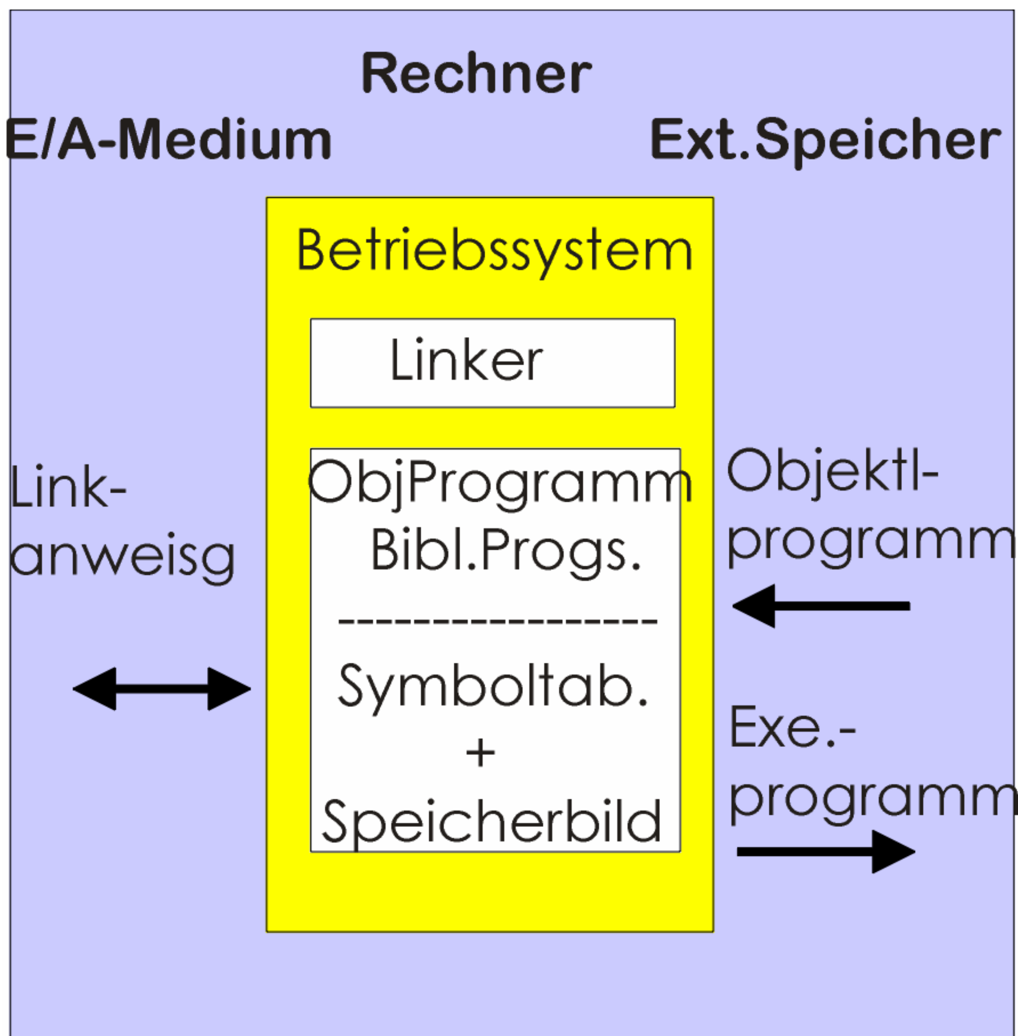


Abbildung 4 Binden

Der Vorgang des Bindens wird wiederum durch ein Programm (Binder, Linker) ausgeführt.

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

4.5 Laden und Ausführen

Ein ausführbares Programm, das also editiert, übersetzt und gebunden ist, kann in den Arbeitsspeicher des Rechners geladen und gestartet werden.

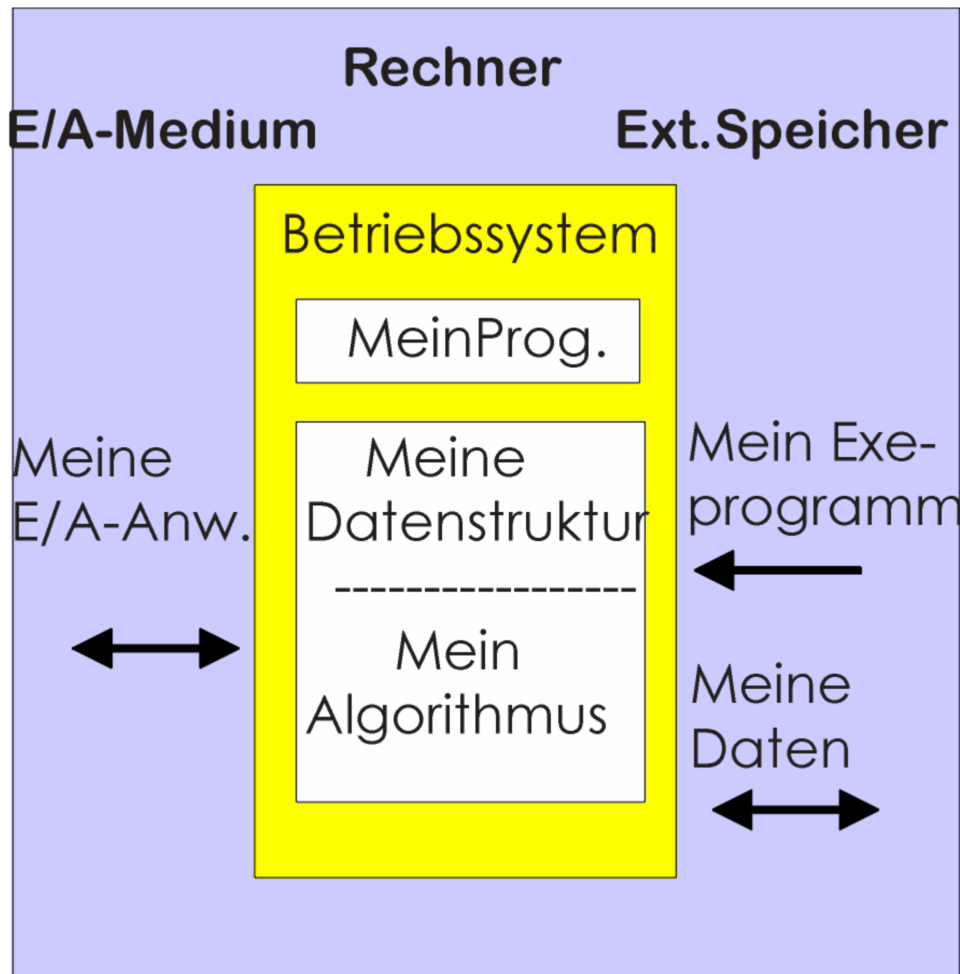


Abbildung 5 Laden und ausführen (execute)

Nach dem Start des geladenen Programmes werden die Anweisungen des Programmerstellers vom Rechner abgearbeitet.

Daten werden dann so eingelesen, verarbeitet und ausgegeben, wie der Entwerfer den Ablauf in seinem Programm vorgeschrieben hat.

Ein Rechner ist ein Automat, der nichts falsch macht. Nur Programme können logisch falsch aufgebaut sein.

ADie Rechner, die verstehen einen so schlecht.≅

Ein Programmentwerfer am 29.10.79

4.6 Testen

Ein Programm ist in der Regel fehlerhaft. Ein Übersetzer erkennt Verstöße gegen die Syntax der Sprache. Logische Fehler in der Programmstruktur werden nicht erkannt. Sie treten in Erscheinung durch Programmabbruch bei der Ausführung oder durch falsche Ergebnisse.

Auf den meisten Rechnern stehen heute Testhilfen (Debugger, Trace) zur Verfügung, mit denen der Programmablauf in Einzelschritten verfolgt werden kann. Der Zugriff auf Variable bzw. Datenspeicher kann damit nach jedem Anweisungsschritt überprüft oder automatisch ausgegeben werden.

Stehen solche Testhilfen nicht zur Verfügung, so muß der Ablauf durch programmierte Testausdrucke dargestellt werden.

Die Plausibilität von Programmierergebnissen muss durch Testdaten mit bekannten Ergebnissen und durch Grenzwertdaten überprüft werden.

Beispiel: In den Übungsprogrammen

- Werte mit bekannten Ergebnissen eingeben
- Extremwerte am Anfang oder Ende eingeben
- Als erste Zahl eine Null eingeben

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

5 Struktur eines Rechnersystems

Ein Rechnersystem besteht aus Hardware, Software und Dokumentation.

Die Rechnerhardware besteht aus Rechnerkern und Peripherie.

Der Rechnerkern besteht aus der zentralen Recheneinheit (CPU=Central Processor Unit) und dem Arbeitsspeicher (RAM=Random Access Memory).

Die Peripherie besteht aus Bildschirm, Tastatur und Maus für den Benutzerdialog und aus externen Speichereinrichtungen wie Magnetplatten- und Diskettenlaufwerken etc..

Weiter können technische Systeme wie Fertigungsanlagen, Hochregallager etc. angeschlossen sein, die über den Rechner im Online-Betrieb, d.h. direkt gesteuert werden.

Die Rechnersoftware besteht aus dem Betriebssystem und aus Dienstprogrammen.

Das Betriebssystem ist das wesentliche Programm, das alle Vorgänge im Rechner steuert.

Dienstprogramme unterstützen die problembezogene Arbeit des Benutzers am Rechner. Es sind dies Programme zur Textverarbeitung (Editor), Übersetzer (Compiler) für verschiedene Sprachen wie FORTRAN, COBOL, ALGOL, PASCAL, MODULA, C, SIMULA, C++ etc. und ein Binder (Linker), mit dem übersetzte Programme lauffähig gemacht werden.

Die Dokumentation besteht aus Handbüchern (manuals), die die Hardware und die Software des Rechners verständlich beschreiben sollten.

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

5.1 Eine Rechnerkonfiguration

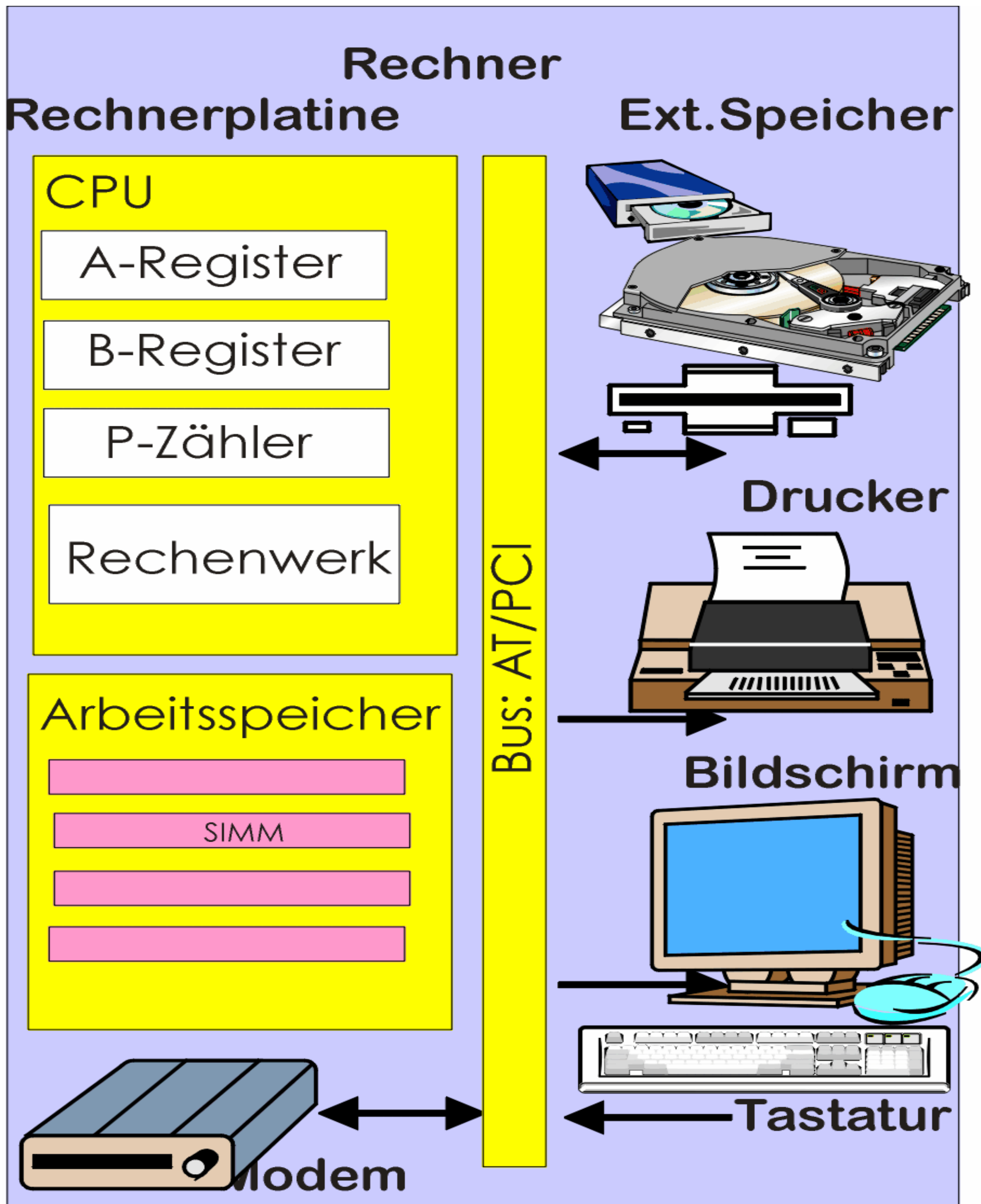


Abbildung 6 Rechnerkonfiguration

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

5.2 Betriebssysteme

Der Hochschule stehen zur Zeit (2000) in einem Netzwerk (Ethernet) mehrere Rechner RS6000, diverse lokale Laborrechner und Personal Computer (PC) in Pools zur Verfügung.

Die Laborrechner und die Rechner der Netze stehen für Forschungszwecke zur Verfügung. Studierende können in der Regel bei Studien- und Diplomarbeiten an diesen Rechnern arbeiten.

Für Ausbildungszwecke stehen hauptsächlich die PCs im Pool zur Verfügung.

Jede StudentIN wird im Laufe seines Studiums an mehreren unterschiedlichen Rechnern arbeiten. Es ist also wichtig, die Prinzipien der Rechnerhandhabung zu verstehen. Den aktuellen Rechner erarbeitet man sich anhand der Manuals.

Rechner	Hersteller	Betriebssystem
RS6000	IBM	AIX
VAX	Digital	VMS
PC Intel/AMD	Diverse	DOS/Windows/Linux
HP 9000	Hewlett Packard	HP-UX

Abbildung 7   Rechner und Betriebssysteme an der GhK (Auszug)

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

5.3 Dokumentation

Zu jedem gekauften Rechner wird in der Regel die Dokumentation, meist in Englisch und sehr umfangreich (10 cm bis 1 m), mitgeliefert. Das Hochschulrechenzentrum HRZ (Mönchebergstr. 11) erstellt für die meisten Rechner Dokumentation in deutscher Sprache und führt Einführungskurse durch, die am Schwarzen Brett des HRZ angekündigt werden.

6 Übungsaufgaben

Im Laufe der Lehrveranstaltung werden betreute Übungssitzungen direkt an Rechnern im HRZ mit dem Betriebssystem AIX (Unix) abgehalten. In diesen Übungen werden Lösungen zu Aufgaben aus der Vorlesung erarbeitet, deren Abgabe zu vorgegebenen Terminen obligatorisch sind.

In der ersten Übung ist ein vorgegebenes Programm HELLO an den Übungsrechnern einzugeben und zum Ablauf zu bringen. Damit soll der Übungsrechner als Instrument mit seiner Betriebssoftware und der vorhandenen Dokumentation erarbeitet werden.

Die Lösung zur Aufgabe des Försters MINMAX wird als C- und FORTRAN-Programm ausgegeben. Hier wird gezeigt, dass eine Aufgabe unabhängig von der Programmiersprache zu gleichen Ergebnissen führt.

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

6.1 C/C++-Quellcode HELLO

```
/* Beispiel: Compilertest  (GST rei/fps/ipl/mb/GhK)
   helloc.c  (Prog.Sprache C/C++ 13.10.99 rei)
*/
#include <stdio.h>

int main(void){
      fprintf(stdout, "=== Hello World aus C/C++ ===\n");
      return 0;
}
```

Übersetzen:
gcc –c helloc.c

Binden:
gcc –o helloc.e helloc.o

Starten:
helloc.e

Ergebnis:
=== Hello World aus C/C++ ===

6.2 FORTRAN-Quellcode HELLO

```
* Beispiel: Compilertest (GST rei/fps.ipl/mb/GhK)
* hellof.f  (Prog.Sprache FORTRAN  13.10.99 rei)

      program hellof
      write(UNIT=*, FMT=100)
100   format(1X, '=== Hello World aus FORTRAN77 ===')
      end
```

Übersetzen:
f77 –c hellof.f

Binden:
f77 –o hellof.e hellof.o

Starten:
hellof.e

Ergebnis:
=== Hello World aus FORTRAN77 ===

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

6.3 MinMax in C/C++

```
// Beispiel: Foerster+Gehilfe  (Gst fps/ipl/GhK)
// minmaxc.cpp (Prog.Sprache C/C++ 19.10.98 rei)
#include <stdio.h>
#include <stdlib.h>
        FILE *finp;

int main(void){
// Vereinbarung von Groessen
      int anzahl;
      float aktWert, sumWert, minWert, maxWert;
      float mittlWert;
// Programmstart auf Bildschirm meldern
      fprintf(stdout,
              " == Programm minmaxc.cpp liest Datei: baeume.inp ==\n");
// Initialisierung der Groessen
      anzahl = 0;
      sumWert = 0.0;
      minWert = 99999.9;
      maxWert = -minWert;
      mittlWert = 0.0;
// Datei oeffnen
      finp = fopen("baeume.inp", "r");
        if (finp == NULL){
           fprintf(stdout, "## Datei: baeume.inp nicht vorhanden ##\n");
           exit(1);
      }
// Lesen und verarbeiten
      do {
         fscanf(finp, " %f\n", &aktWert);
         fprintf(stdout, " %8.2f\n", aktWert);
         sumWert = sumWert + aktWert;
         anzahl = anzahl + 1;
         if (aktWert > maxWert)maxWert = aktWert;
         if (aktWert < minWert)minWert = aktWert;
      } while (!feof(finp));

        fclose(finp);
// Ergebnis ausgeben
      if (anzahl > 0)mittlWert = sumWert / anzahl;
      fprintf(stdout,
           " == Ergebnis:\n     Min.   Mittel     Max.  Anzahl\n");
        fprintf(stdout, " %8.2f %8.2f %8.2f %4i\n",
                        minWert, mittlWert, maxWert, anzahl);
        return 0;
}
```

6.4 MinMax in FORTRAN

```
* Beispiel: Foerster+Gehilfe  (Gst fps/ipl/GhK)
* minmaxf.f (Prog.Sprache FORTRAN 19.10.98 rei)
      program minmaxf
        integer inpKanal
* Vereinbarung von Groessen
      integer anzahl
      real aktwert, sumWert, minWert, maxWert
      real mittlWert
* Programmstart auf Bildschirm melden
      write(*, FMT=100)
100   format(1X, '== Programm minmaxf.f   liest Datei: baeume.inp ==')
* Initialisierung der Groessen
        inpKanal = 10
      anzahl = 0
      sumWert = 0.0
      minWert = 99999.9
      maxWert = -minWert
      mittlWert = 0.0
* Datei oeffnen
      open(inpKanal, FILE='baeume.inp', STATUS= 'OLD', IOSTAT=istatus)
        if (istatus .EQ. 0)goto 10
      write(*, FMT=99)
99    format('## Datei: baeume.inp nicht vorhanden ##')
        stop
* Lesen und verarbeiten
10    read(inpKanal, FMT=*, ERR=20, IOSTAT=iread)aktWert
      if (iread .NE. 0)goto 20
      write(*, FMT=200)aktWert
      sumWert = sumWert + aktWert
      anzahl = anzahl + 1
      if (aktWert .GT. maxWert)maxWert = aktWert
      if (aktWert .LT. minWert)minWert = aktWert
      goto 10
20    continue
      close(inpKanal)
* Ergebnis ausgeben
      write(*, FMT=300)
300   format(' == Ergebnis:', /, '     Min.   Mittel     Max.  Anzahl')
      if (anzahl .GT. 0)mittlWert = sumWert / anzahl
        write(*, FMT=200)minWert, mittlWert, maxWert, anzahl
200   format(1X, F8.2, 1X, F8.2, 1X, F8.2, 1X, I4)
        end
```

6.5 Rechnernetzwerk

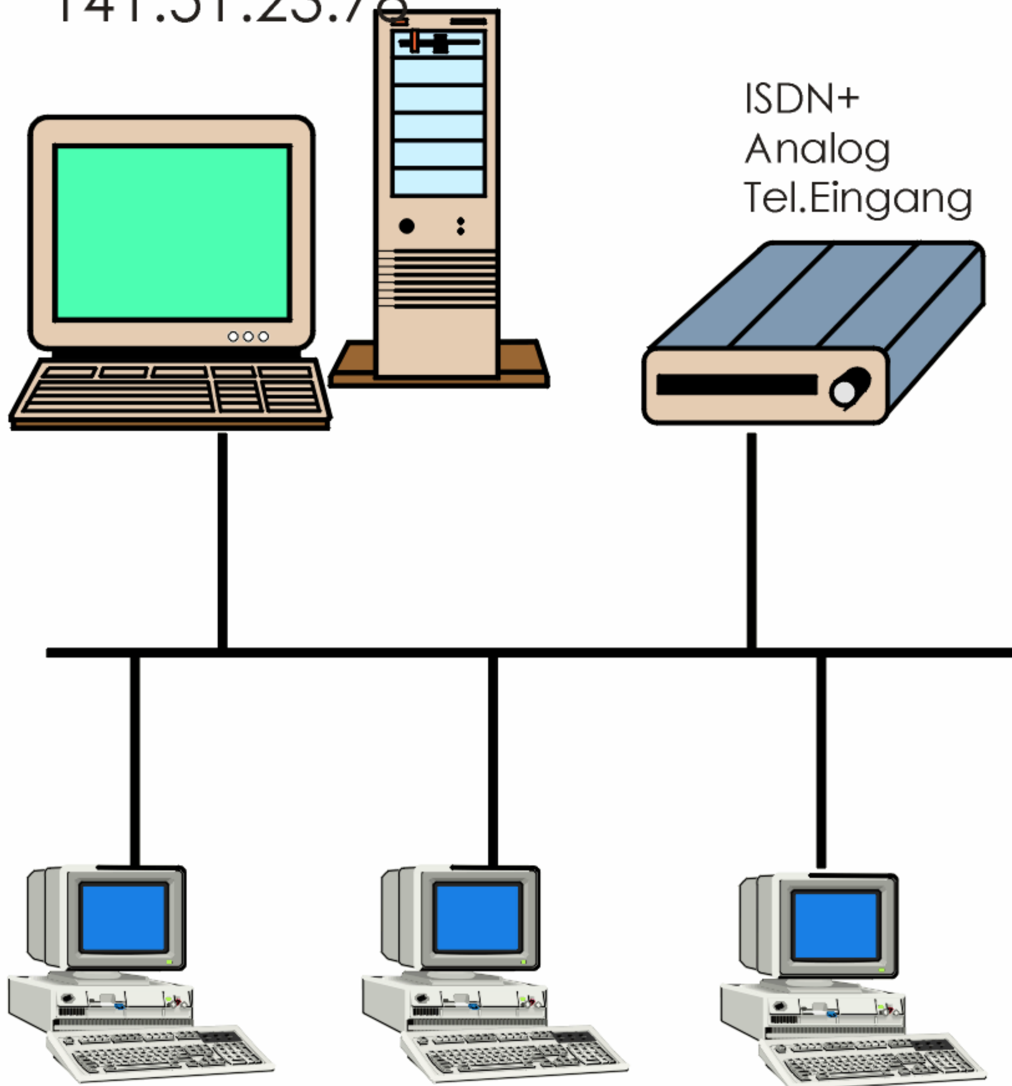


Abbildung 7 Netzwerk des Hochschulrechenzentrums (HRZ)

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

6.6 Dateistruktur unter UNIX

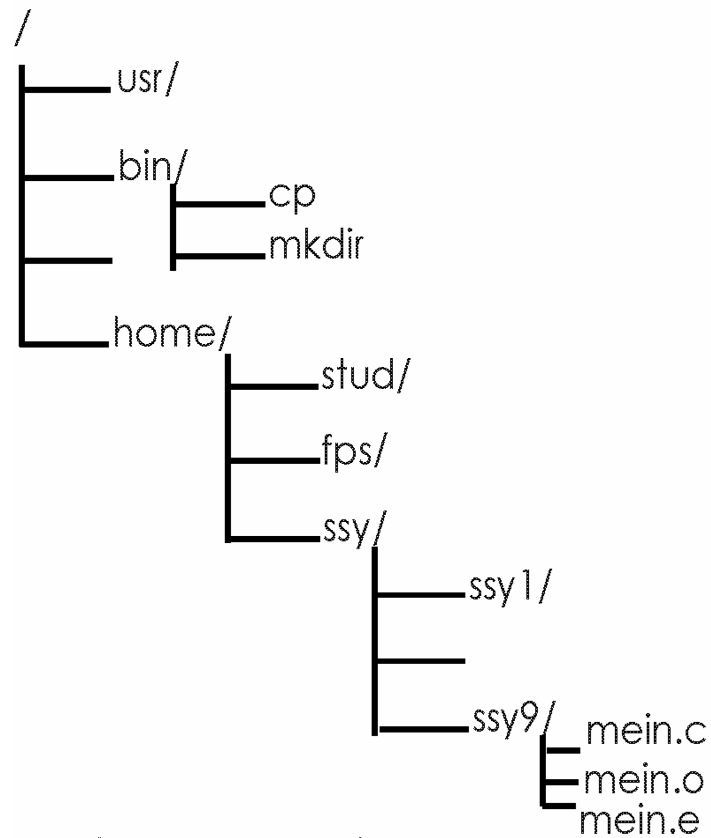


Abbildung 8 Dateistruktur (Unix-Rechner)

© A. Reinhardt, fps/ipl/mb/unikassel, WS 2003/04

7 Anhang

Zahlenfolge: 37.0   3.0   11.0   40.0 9.0 0.0

Name	Zeile	Nummer
Aktuelle Laenge		01
AnzahlLaengen		02
SummeLaengen		03
KleinsteLaenge		04
GroessteLaenge		05

Abbildung 9 Tafel des Försters

(2.2.2) Den ersten Wert abrufen und in das Feld "aktuelle Länge" setzen

(3) Verarbeiten der Längen

(3.1) Solange der Wert in "aktuelle Länge" größer "Null" ist, tue folgendes:

1. "Anzahl der Längen" um "eins" erhöhen
2. "Summe der Längen" um den Wert in "aktuelle Länge" erhöhen
3. Wenn der Wert in "aktuelle Länge" kleiner ist als der Wert in "kleinste Länge", dann übertrage den Wert aus "aktuelle Länge" in "kleinste Länge"
4. Wenn der Wert in "aktuelle Länge" größer ist als der Wert in "größte Länge", dann übertrage den Wert aus Aaktuelle Länge≅ in "größte Länge"
5. Rufe den nächsten Wert ab und übertrage ihn in das Feld "aktuelle Länge"

(4) Ergebnis an den Vertrieb melden

Grundlagen der Softwaretechnik

Teil 2

Modellbildung mit C und C++

Univ.-Prof. Dipl.-Ing. A. Reinhardt
Produktionssysteme
Inst. f. Produktionstechnik und Logistik
Universität Gesamthochschule Kassel
Tel. +49 561 804-2693, Fr. Treskow
www.fps.maschinenbau.uni-kassel.de

WS 2003/04

Inhaltsverzeichnis

1 Einleitung 3
2 Grundlagen 3
3 C ganz einfach 7
  3.1 Formaler Aufbau eines Programmes 7
  3.2 Einfache Datentypen 10
  3.3 Wertzuweisung 11
  3.4 Ein-/ Ausgabeanweisungen 13
  3.5 Lösung der Aufgabe 14
4 Kontrollstrukturen 16
  4.1 Sequenz 16
  4.2 Schleifen 16
    4.2.1 for-Schleife 16
    4.2.2 while-Schleife 18
    4.2.3 do-while-Schleife 19
  4.3 Alternation und Auswahl 20
    4.3.1 Alternation 20
    4.3.2 Auswahl 21
  4.4 Ablaufsprung mit GOTO 23
  4.5 Lösung der Aufgabenstellung 24
  4.6 Übungsaufgaben 25
5 Ablauf- und Datenverbunde 26
  5.1 Ablaufverbund Funktion 26
  5.2 Datenverbunde 33
  5.3 Konstanten und Felder 37
    5.3.1 Konstanten 37
    5.3.2 Felder 38
    5.3.3 Beipiel zu Konstanten und Feldern 42
  5.4 Zeiger und temporäre Datenverbunde 44
  5.5 Lösung der Aufgabenstellung 50
6 Objektorientierte Programmierung 51
  6.1 Klassen 52
  6.2 Sortierer mit Klassen 54
7 Verkettete Objekte 60
  7.1 Ketten 60
  7.2 Listen 70
  7.3 Aufgabenstellung: Lagerverwaltung 79
8 Anhang 86
  8.1 Protokoll einer Tageskasse 86
  8.2 Kartoffelsortierer 87

1 Einleitung

Das vorliegende Skript soll als Leitfaden für die Einarbeitung in die Modellbildung mit C und C++ dienen. Es ist also keine in sich abgeschlossene Beschreibung der Sprachen.

Die Aufgaben in der Lehrveranstaltung lassen sich grundsätzlich in allen bekannten Sprachen lösen.

Die einzelnen Kapitel des Skriptes sind so aufgebaut, dass eine vorangestellte Aufgabenstellung zum Schluss des Kapitels gelöst werden kann.

Für die Einarbeitung in die Programmiersprache C wird das Buch "Programmieren in C",von den Entwicklern Kernighan und Ritchie empfohlen; für C++ das Buch "Die Programmiersprache C++“ von Bjarne Stroustrup, dem Entwickler von C++.

Zur Geschichte der Programmiersprachen finden sie ein Papier auf der Fps-Homepage. http://www.fps.maschinenbau.uni-kassel.de unter Forschung und Publikationen

2 Grundlagen

Programm

Der Begriff Programm kann in zwei unterschiedlichen Ebenen erklärt werden.

Ein Programm ist zunächst die Beschreibung oder Abbildung eines Objektbereiches. Diese Abbildung (Modell) beschreibt

- Objekte und deren Eigenschaften
- ein Verfahren oder einen Ablauf, nach dem die Objekte verknüpft oder zu verknüpfen sind.

Ein Programm ist weiter eine Arbeitsvorschrift in einer formalen Sprache, nach der ein Automat

- die Eigenschaften von Objekten als Daten abbildet
- diese Daten verarbeitet.

Daten

Daten sind demnach im Rechner abgebildete Objekteigenschaften, d.h. gespeicherte Größen oder Eigenschaftswerte.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Bezeichner (Identifier)

Bezeichner sind Namen oder Symbole für Objekteigenschaften bzw. Namen für Speicherzellen, die der Benutzer einer Sprache für die Beschreibung frei wählt.

Bezeichner werden aus einem Wertevorrat von Zeichen

- Buchstaben A...Z, a...z , _ (underline)
- Ziffern 0...9

zusammengesetzt.

Ein Bezeichner beginnt mit einem Buchstaben. Er kann im Prinzip beliebig lang sein. Bezeichner sollten immer aus so vielen Zeichen bestehen, wie für eine lesbare Beschreibung notwendig ist. Große und kleine Buchstaben werden unterschieden.

Vokabular

Programmiersprachen sind formale Sprachen. Ihnen liegt wie jeder Sprache, d.h. auch natürlichen Sprachen, ein Vokabular (Wortschatz) zugrunde, dessen Elemente einzelne Worte sind. In formalen Sprachen werden diese als Grundsymbole oder Schlüsselworte bezeichnet.

Formale Sprachen haben ein abgeschlossenes Vokabular.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Schlüsselworte in C:

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				

Schlüsselworte sind reservierte Worte, sie dürfen weder getrennt noch anderweitig im Programm benutzt werden.

Syntax und Semantik

Die Syntax (Grammatik) beschreibt den Aufbau von Symbolfolgen bzw. Sätzen oder Anweisungen. Die Satzstruktur ist wesentlich für die Erkennung der Semantik (Bedeutung) eines Satzes.

Anweisung

Eine Anweisung ist ein syntaktisch richtiger Satz nach der Konvention einer formalen Sprache.

Sprachreport

Das Vokabular, die Syntax und die Semantik natürlicher Sprachen haben sich als Konvention aus der menschlichen Kommunikation entwickelt. Programmiersprachen sind künstliche Sprachen, deren Sprachkonvention in Sprachbeschreibungen oder -reports niedergelegt sind.

Beispiele sind:

- o Algol - Report
- o Simula - Report
- o Fortran - Ansi - Norm
- o Basis - Pearl - Beschreibung
- o ANSI Programming Language C, X3.159-1989

Die Sprachbeschreibung kann verbal oder formal (Backus-Naur, Syntax-Diagramme) sein. In den in der Einleitung aufgeführten Büchern sind die C-Sprachen vollständig beschrieben.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

3 C ganz einfach

Aufgabenstellung:

Ein Obsthändler verkauft Äpfel, Birnen und Zwetschgen. Der Preis für das Obst soll ermittelt werden.

3.1 Formaler Aufbau eines Programmes

Der einfache Aufbau eines C-Programmes besteht aus

- Programmbeschreibendem Kommentar
- Umgebungsanweisungen (preprocessor)
- Hauptteil (main)

Kommentar wird zwischen die Symbolklammern **/*** und ***/** gesetzt.
Zur **Programmbeschreibung** sollten Programmname, Datum und Autoren angegeben werden:

/* meinprogramm, 24.10.03, Ich, Er, Sie */

Umgebungsanweisungen werden in einer ersten Übersetzungsphase vom Compiler verarbeitet. Sie stellen für ein Programm Funktionsköpfe (**header**) mit der Anweisung

#include <header-datei>

und **globale Vereinbarungen** mit

#define <wert>

zur Verfügung.

Der Hauptteil enthält

- Vereinbarung
- Ablaufteil

In der Vereinbarungsbeschreibung werden die Bezeichner festgelegt, mit denen der (äußere) Objektbereich abgebildet wird bzw. mit denen die zu speichernden (intern) Eigenschaften der Objekte als Daten anzusprechen sind.

Im Ablaufteil wird durch einzelne Anweisungen beschrieben wie die bezeichneten Eigenschaften initialisiert, eingelesen, verknüpft und ausgeschrieben werden.

```
/* meinprog.c, 30.10.03, Mustermensch */

#include <stdio.h>

int main(void) {
   int faktor;
   int AktuellerWert, Ergebnis;

   faktor = 3;

   fprintf(stdout, "Bitte einen Wert eingeben\n");
   fscanf(stdin, " %i", &AktuellerWert);

   Ergebnis = AktuellerWert * faktor;

   fprintf(stdout, "Das Ergebnis ist %4i\n", Ergebnis);
   return 0;
}
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Für die Aufgabenstellung (Obsthändler) müssen Bezeichner vereinbart werden, die die wesentlichen Eigenschaften sinnfällig beschreiben.

Beispiel:

Birnenpreis
Aepfelpreis
Zwetschgenpreis
Obstpreis

(vgl. Datentypen, nächstes Kapitel)

In der Ablaufbeschreibung wird durch Anweisungen festgelegt, wie die bezeichneten Objekteigenschaften bzw. Daten zu behandeln sind.

Beispiel:

Birnenpreis einlesen
Aepfelpreis einlesen
Zwetschgenpreis einlesen
Obstpreis ermitteln
Obstpreis ausgeben

Der Hauptteil eines C-Programmes wird zwischen die Anweisungsklammern

{ ... }

gesetzt.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

3.2 Einfache Datentypen

Für die Abbildung von Objekteigenschaften ist neben der Festlegung der Eigenschaftsbezeichner der Typ der Eigenschaftswerte zu vereinbaren.

Werte von Eigenschaften können beispielsweise sein

- ganze Zahlen
- gebrochene Zahlen
- Zeichen

Die entsprechende Typvereinbarung erfolgt mit den C-Schlüsselworten

- **int**
- **float**
- **char**

Die Form der Vereinbarung (einfacher Datentypen) ist folgende:

<typ> < Bezeichner >;

Bezeichner vom gleichen Typ können als Liste getrennt durch Komma vereinbart werden:

<typ> < BezeichnerA, BezeichnerB, BezeichnerC>;

Beispiele:

(1)
```
int aepfel;
int birnen;
int Zwetschgen;
int Obstmenge;
```

(2)
```
float Aepfelpreis;
float Birnenpreis;
float Zwetschgenpreis;
float obstpreis;
```

Ein Bezeichner darf nicht mehrfach vereinbart werden (logisch klar!). Die Vereinbarungsbeschreibung steht in C vor dem Verarbeitungsteil. Die Typbezeichner geben vor wie im Rechner, genauer im Speicher, die bezeichneten Werte als Bit-Muster abgelegt werden.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

3.3 Wertzuweisung

Im Programmablauf werden Objektabbilder durch Anweisungen in Verbindung gebracht. Die einfachste und grundlegende Anweisung ist die der Wertzuweisung. In dieser Anweisung wird einer bezeichneten Eigenschaft ein Wert zugewiesen.

Form:

< Bezeichner > = < Wert >;

Die Wertzuweisung besteht aus

- Eigenschaftsbezeichner auf der linken Seite
- Zuweisungsoperator
- Zuweisungswert auf der rechten Seite
- Semikolon

Der Zuweisungswert kann der Wert einer Konstanten, eines Bezeichners (Variable) oder der Wert eines Ausdruckes sein.

Ausdrücke sind Verknüpfungen von Werten, die durch Konstante, Bezeichner oder Ausdrücke gegeben sein können.

Beispiele:

```
preis = 22;
obst  = aepfel + birnen;
```

Die Verknüpfung von Werten in Ausdrücken erfolgt über Operatoren.

Liste von arithmetischen und logischen Operatoren, geordnet nach ihrer Auswertungsreihenfolge:

```
()          Klammer für Ausdrücke
!           Nicht, Verneinung
*           Multiplikation
/           Division
+           Addition
-           Subtraktion
<           kleiner
<=          kleiner gleich
>           größer
>=          größer gleich
==          gleich
!=          ungleich
&&          und
||          oder
```

Stehen mehrere Operatoren in einem Ausdruck, so werden die Verknüpfungen in der aufgeführten Reihenfolge ausgewertet, d.h. ein Klammerausdruck () wird zuerst ausgewertet. Für die praktische Anwendung sollten in Zweifelsfällen Teilausdrücke in runden Klammern gebildet werden.

Beispiel:

(1) Anteil = aepfel + birnen / obst;

(2) Anteil = (aepfel + birnen) / obst;

Im Fall (1) wird das Ergebnis unsinnig. Im Fall (2) wird im Ergebnis der relative Anteil von Äpfel und Birnen an Obst ausgedrückt.

Sind zwei Operanden eines arithmetischen Ausdrucks (+ - *) vom Typ **int**, so ist das Ergebnis der Operation ebenfalls vom Typ **int**. Ist ein Operand vom Typ float, so ist das Ergebnis vom Typ **float.**

3.4 Ein-/ Ausgabeanweisungen

Ein-/Ausgabeanweisungen beschreiben den Datentransport zwischen programminternen Bezeichnern und externen Geräten.

Form:

< Transportanweisung> (
<dateivariable>, <formatstring>, <liste von Werten>
);

Die Eingabeanweisung **fscanf** liest Werte von einem Gerät oder einer Datei und legt sie unter den mit Adressoperator **&** (WICHTIG!) versehenen Bezeichnern ab.

```
int Artikel, Menge;
fscanf(stdin, " %i %i", &Artikel, &Menge);
```

Die Ausgabeanweisung **fprintf** schreibt programmintern bezeichnete Werte nach außen auf Geräte oder Dateien aus.

```
fprintf(stdout,  "Werte fuer Artikel = %3i  fuer Menge = %3i\n", Artikel, Menge);
```

Für den Zugriff auf Dateien werden Dateivariable vom Typ **FILE** als Zeiger vereinbart und mit **fopen** auf die Dateien gelenkt. Die Datei wird mit **fclose** geschlossen (vgl. Bücher).

```
FILE *fout;
...
fout = fopen(“daten.out“, “w“);
fprintf(fout, “%8.2f\n“, wert);
fclose(fout);
```

Die Dateivariable **stdout** ist bereits in der Programmumgebung vereinbart und auf den Bildschirm gelenkt. Die Dateivariable **stdin** erwartet Eingaben von der Tastatur.

Der Formatstring besteht aus einem Text in Anführungszeichen mit typbezogenen Formathinweisen an den Stellen, wo die Werte in den String einzusetzen sind

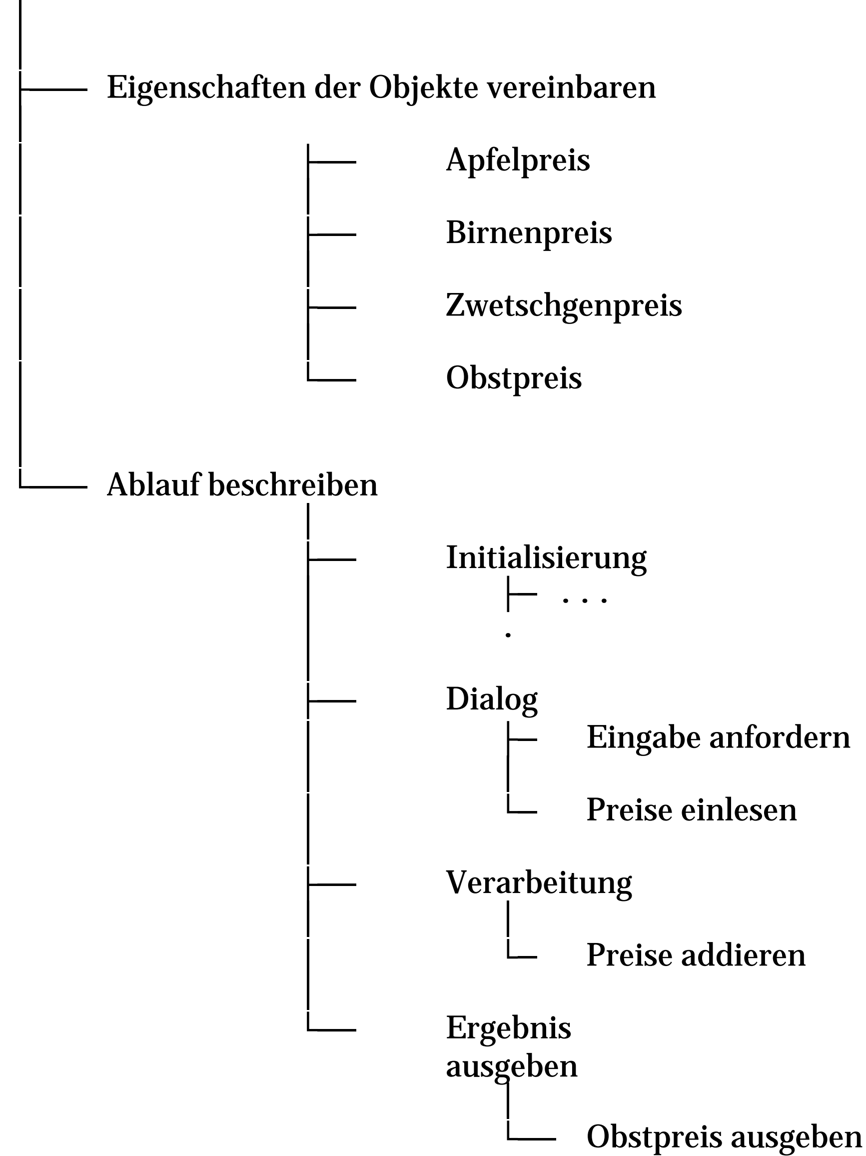
%i für ganzzahlige Werte, **%f** für gebrochene Werte und **%c** für Zeichen

und Steuerzeichen

\n für neue Zeile, **\f** für Seitenvorschub und **\t** für Tabulatorzeichen

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

3.5 Lösung der Aufgabe



© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Programm Obstkasse

```
/* Programm Obstkasse, 27.04.93 rei */

#include <stdio.h>

int main (void)
{
     float Aepfelpreis, Birnenpreis,
       Zwetschgenpreis, Obstpreis;

     Aepfelpreis = 0.0;
     Birnenpreis = 0.0;
     Zwetschgenpreis = 0.0;
     Obstpreis   = 0.0;

     fprintf (stdout,
     "Aepfel-, Birnen- und Zwetschgenpreise eingeben:\n");

     fscanf  (stdin, " %f %f %f",
          &Aepfelpreis, &Birnenpreis, &Zwetschgenpreis);

     Obstpreis = Aepfelpreis +
                Birnenpreis + Zwetschgenpreis;

     fprintf (stdout,
       "\nDas Obst kostet %5.2f DM \n\n", Obstpreis);
     return 0;
} /* main */
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

4 Kontrollstrukturen

Beispiel: Aus einer Reihe von Messwerten sind Minimum, Maximum und Mittelwert zu bestimmen. Das Ende der Reihe sei durch einen Messwert kleiner gleich Null gegeben.

4.1 Sequenz

Die Ablaufbeschreibung bestand in der bisherigen Ausführung aus einer Folge von Anweisungen, die in der niedergeschriebenen Reihenfolge, d.h. sequentiell, ausgeführt wurde. Eine derartige Sequenz von Anweisungen kann durch Anweisungsklammern

{ ... }

zu einem Anweisungsverbund (compound statement) zusammengefasst werden.

Ein Anweisungsverbund wird von außen als einfache Anweisung behandelt.

4.2 Schleifen

Für die mehrfache Wiederholung einer Anweisung oder eines Anweisungsverbundes sind drei Schleifenkonstruktionen vorgesehen.

4.2.1 for-Schleife

Die **for**-Konstruktion wird für den Fall eingesetzt, daß die Zahl der Wiederholungen vorher bekannt ist.

for (<Zaehlerinit>; <Zaehlerkontrolle>; <Zaehlermodifiz.>)
<Schleifenanweisung>

Der erste Ausdruck <Zählerinit> in der **for**-Anweisung wird einmal ausgewertet. Er dient deshalb zur Initialisierung des Schleifenzählers. Der Typ des Ausdrucks kann beliebig sein. Der zweite Ausdruck <Zählerkontrolle> wird vor jedem Eintritt in die Schleife bewertet. Ist der Wert 0 oder FALSCH, so wird die Schleife nicht mehr ausgeführt. Der dritte Ausdruck <Zählermodifiz.> wird nach jedem Schleifendurchlauf ausgewertet. Er modifiziert den Zähler.

```
/*  for.c  02.05.93 rei */

#include <stdio.h>

int main (void){
  int Anfwert, Endwert, Aktwert;
  float Summe = 0.0;

  fprintf(stdout, "\nAnfangs- und Endwert eingeben: ");
  fscanf(stdin, " %i %i", &Anfwert, &Endwert);

  for (Aktwert = Anfwert;
           Aktwert <= Endwert; Aktwert++)
    Summe = Summe + Aktwert;

  fprintf(stdout,
      "\nDie Summe %2i bis %3i ergibt %6.2f \n\n",
       Anfwert, Endwert, Summe);
  return 0;
}  /* main */
```

In diesem Beispiel wird eine einzelne Anweisung über die **for**-Konstruktion kontrolliert.

Im **for**-Beispiel wurde die Operation **Inkrement** <Variable>++ eingeführt. Der Wert der Variablen wird um 1 erhöht. In C existiert auch das Gegenstück **Dekrement** <Variable>**--**.

4.2.2 while-Schleife

Mit der **while**-Konstruktion wird eine Anweisung oder ein Anweisungsverbund so lange ausgeführt wie ein bool'scher Kontrollausdruck wahr ist.

while (< bool. ausdruck>)
< anweisung >

Der Kontrollausdruck wird vor Eintritt in die Schleife ausgewertet. Ein bool'scher Ausdruck hat den Wert TRUE (!= 0) oder FALSE (== 0), abhängig von den Werten der beteiligten Operanden. Der Wert des Ausdruckes kann sich nach Eintritt in die Schleife nur ändern, wenn die Werte der Operanden im Schleifenkörper verändert werden.

```
/* while.c 02.05.93 rei */

#include <stdio.h>

int main (void){
  float Summe, AktWert;

  Summe = 0.0;

  fprintf(stdout,
    "\nWerte > 0.0 eingeben fuer while-Beispiel: \n");
  fscanf(stdin, " %f", &AktWert);

  while (AktWert > 0.0){
    Summe = Summe + AktWert;
    fscanf(stdin, " %f", &AktWert);
  }  /* while */

  fprintf(stdout,
     "\nDie Summe betraegt %6.2f\n\n", Summe);
  return 0;
}  /* main */
```

Über die **while**-Konstruktion wird hier im Beispiel eine Anweisung kontrolliert und zwar eine Verbundanweisung, die mehrere Anweisungen durch {...} zusammenfasst. Im Schleifenkörper tritt ein Operand (AktWert) aus dem bool'schen Ausdruck des Schleifenkopfes auf.

4.2.3 do-while-Schleife

In der **do-while**-Konstruktion, auch repeat-Schleife genannt, wird der Schleifenkontrollausdruck am Ende der Schleife ausgewertet. Die Schleife wird wiederholt, wenn der Kontrollausdruck >0 bzw. WAHR ist.

> **do**
> < Anweisung >
> **while** < bool. Ausdruck >;

Die Schleife wird also mindestens einmal durchlaufen.

```
/* repeat.c 02.05.93 rei */

#include <stdio.h>

int main (void){
  float AktWert;
  int Zaehler;

  fprintf(stdout,
   "\nWerte > 0.0 eingeben fuer Repeat-Beispiel: \n");

  Zaehler = 0;
  do{
    Zaehler++;
    fscanf(stdin, " %f", &AktWert);
  } while (AktWert > 0.0);

  fprintf(stdout,
     "\nDas waren %2i Eingaben\n\n", Zaehler);
  return 0;
}  /* main */
```

Der Schleifenkontrollausdruck wird, wie man sieht, im Schleifenkörper verändert. Was passiert, wenn dies nicht der Fall ist ?

4.3 Alternation und Auswahl

Für die Beschreibung alternativer Aktionen oder einer Auswahl an Aktionen sind zwei Sprachkonstruktionen vorgesehen.

4.3.1 Alternation

Wenn eine Bedingung erfüllt ist, dann wird die Folgeanweisung ausgeführt, sonst wird ein **else**-Zweig ausgeführt.

if (< bool. Ausdruck >)
  < Anweisung >
else < Anweisung >

Soll keine alternative Aktion beschrieben werden, dann kann der **else**-Zweig in der Konstruktion wegfallen.

if (< bool. Ausdruck >)
  < Anweisung >

Zur Wiederholung soll gesagt sein, dass eine Anweisung auch eine Verbundanweisung sein kann.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

```
/* if.c 02.05.93 rei */

#include <stdio.h>

int main (void){
  float AktWert;

  fprintf(stdout,
    "\nWerte fuer Alternation (if) eingeben: \n");

  do{
    fscanf(stdin, " %f", &AktWert);

    if ( AktWert > 0)
      fprintf (stdout,
            "Wert ist groesser 0.0\n");
    else
      fprintf (stdout,
       "Wert ist nicht groesser 0.0\n");
  }while (AktWert != 0.0);
  return 0;
  }  /* main */
```

4.3.2 Auswahl

Die Konstruktion der Auswahl (**switch**) besteht aus einem Selektorausdruck vom Typ '**int**' und einer Blockkonstruktion { } mit einer Liste von markierten Anweisungen, die jeweils mit einer oder mehreren Konstanten aus dem Wertebereich des Selektors markiert sind. Zeichen werden in '**int**'-Typen umgewandelt.

Der Wert des Selektorausdruckes dient als Einsprung an einer Marke. Es werden ab dort alle Anweisungen bis zu einer break-Anweisung ausgeführt.

switch (< Selektorausdruck >) {
  case <marke 1> : <anweisung>; **break**;
    ....
  case <marke n> : <anweisung>;
  case <marke m> : <anweisung>; **break**;
  default : <anweisung>; **break;**
}

Eine **default**-Markierung dient für alle Selektorwerte, die keinen Einsprung finden, sie kann als Fehlerausgang etc. dienen.

```
/* case.c 02.05.93 rei */

#include <stdio.h>

int main (void)
  {
  char Operation;

  fprintf(stdout,
     "\nEingabe '+ - *' oder 'x=Ende' \n");
  do{
     fscanf(stdin, " %c", &Operation);

    switch (Operation){
      case '+': fprintf(stdout,
             " %c bedeutet Addition\n", Operation);
          break;
      case '-': fprintf(stdout,
               " %c bedeutet Subtraktion\n", Operation);
          break;
      case '*': fprintf(stdout,
              " %c bedeutet Multiplikation\n", Operation);
          break;
      default : fprintf(stdout,
                 " %c hat keine Bedeutung\n", Operation);
                break;
    }  /* switch */
  } while (Operation != 'x');
  fprintf (stdout, "\nProgrammende\n\n");
  return 0;
}  /* main */
```

Bemerkung: In der Zeile **fscanf,** dort im Formatstring **" %c"** ist zu beachten, dass das **blank** vor dem **%** eine besondere Bedeutung hat. Das **blank** überliest alle unsichtbaren Zeichen (white chars) in Eingabepuffer **stdin** bis ein echtes Zeichen gefunden wird. (In der Übung ohne **blank** ausprobieren und eine Erklärung suchen)

4.4 Ablaufsprung mit GOTO

Betrachtet man ein Programm als Abbildung eines realen Objektbereiches, so ist eine Sprunganweisung in einem Programm unlogisch. Die Sprunganweisung enthält keine Abbildungsqualität. Sie wird deshalb hier nicht behandelt. Für unentwegte Programmierkünstler wurde auch in C die berüchtigte **GOTO** - Anweisung aufgenommen.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

4.5 Lösung der Aufgabenstellung

```
/* minmax.c 02.05.93 rei */

#include <stdio.h>

int main (void){
  float AktWert,
        MinWert, MaxWert, Summe;
  int   Anzahl;

  MinWert =  99999.99,
  MaxWert = -99999.99,
  Summe = 0.0;
  Anzahl = 0;

  fprintf(stdout, "\nMesswerte > 0.0 eingeben: \n");
  fscanf(stdin, " %f", &AktWert);

  while (AktWert > 0.0){
    Anzahl++;
    Summe = Summe + AktWert;
    if (AktWert > MaxWert)
      MaxWert = AktWert;
    if (AktWert < MinWert)
      MinWert = AktWert;
    fscanf(stdin, " %f", &AktWert);
  }  /* while */

  fprintf(stdout,
   "Anzahl der Werte = %6i\n", Anzahl);
  if (Anzahl > 0) {
    fprintf(stdout,
      "Mittelwert       = %6.2f\n",(Summe/Anzahl) );
    fprintf(stdout,
      "Maximaler Wert   = %6.2f\n", MaxWert);
    fprintf(stdout,
      "Minimaler Wert   = %6.2f\n", MinWert);
  }
  else fprintf(stdout, "===Keine Eingaben===\n");
  return 0;
}  /* main */
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

4.6 Übungsaufgaben

Entwickeln Sie aus dem Beispiel OBSTKASSE ein Programm, das wie bisher Einzelabrechnungen vornimmt und für die Tagesabrechnung folgende Ergebnisse liefert:

- Anzahl der Einkäufe
- Umsatz der einzelnen Obstsorten und Gesamtumsatz
- Anteile der einzelnen Umsätze in % am Gesamtumsatz
- Mittlere Einkaufssumme
- Wann wurde der minimale und wann der maximale Einkaufspreis bezahlt?

Die Gestaltung der Ein-/Ausgabeform des Programmes wird in der Übung vorgegeben.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

5 Ablauf- und Datenverbunde

Aufgabenstellung:

Klassifizieren von Messwerten

Die Bearbeitungszeiten eines Werkstückspektrums sollen untersucht werden. Die Häufigkeit der Werte sind in zehn gleichen Klassen darzustellen. Werte, die die äußere Grenze überschreiten, sind ebenfalls darzustellen.
Es ist der minimale, der maximale und der mittlere Wert der Zeiten zu ermitteln.

```
     Klassen

   1      2      3      4      5      6      7      8      9      10     11
----+------+------+------+------+------+------+------+-----+------------------------------
    20     40     60     80    100    120    140    160    180   200   >200

     Klassengrenze
```

5.1 Ablaufverbund Funktion

Der Entwurf eines Programmes als Beschreibung eines Objektbereiches wird zunächst immer als Grobstruktur entwickelt. In dieser Ebene werden Programmeinheiten als Ablaufverbunde festgelegt, die dann schrittweise verfeinert werden.

In einem baumorientierten Entwurf wird in Stamm, Äste, Zweige und Blätter gegliedert. Dabei können Daten- und Ablaufstrukturen getrennt betrachtet werden. Das Programm wird als Stamm gesehen. Äste oder Zweige können mit Namen belegt und dadurch über Bezeichner angesprochen werden.

Hier sollen zunächst Ablaufäste behandelt werden. Datenäste werden unter Punkt Datenverbund und Felder behandelt.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Ein Ablaufast kann als Zusammenfassung von Einzelaktionen verstanden werden. C, wie auch andere Programmiersprachen, bietet dazu die Konstruktion **Funktion.**

Eine Funktion besteht in Analogie zu einem **main**-Teil aus

<Resultattyp> <Funktionsname> (evt.Parameterdefinition)
{
<Vereinbarungen>

<Ablaufanweisungen>

<Resultatzuweisung>
}

Der Resultattyp gibt an, welchen Datentyp (int, float, char ...) die Funktion liefert. Der Typ void zeigt an, dass die Funktion keinen Wert zurückliefert.

Der Funktionsname bezeichnet den Ablaufverbund und dient als Name für den Aufruf.

Die Parameterdefinition dient als Schnittstelle für die Datenübernahme aus der Umgebung. Sie besteht aus der Vereinbarung eines oder mehrerer Bezeichner. Mehrere Vereinbarungen werden durch Komma getrennt.

Die Parameterliste ist für die Funktion Teil der Variablenvereinbarung.

Die Bezeichner in der Parameterliste werden formale Parameter genannt. Beim Aufruf der Funktion werden aktuelle Parameter übergeben. Die Werte der aktuellen Parameter werden dann in die formalen übernommen.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Beispiele für Funktionsköpfe (Prototypen):

```
void druckergebnis (int anzahl, float summe);

float verarbeite (float wert, float summe);

float minimum (float werta, float wertb);
```

Beispiele für Funktionsaufrufe:

```
druckergebnis (messungen, SummeDerWerte);

ergebnis = verarbeite (messwert, summe);

MinWert = minimum (AktWert, MinWert);
```

Der Funktionskörper ist wie ein main-körper aufgebaut, d.h. er besteht aus

- Vereinbarungsbeschreibung
- Ablaufbeschreibung
- return-Anweisung

Die vereinbarten Bezeichner sind allerdings nur in der Funktion bekannt. Es sind also lokale Bezeichner. Sie sind außerhalb der Funktion nicht bekannt und daher auch nicht ansprechbar.

In der Funktion können jedoch alle Bezeichner angesprochen werden, die außerhalb der Funktion vereinbart wurden, also alle globalen Bezeichner aus dem Blickwinkel der Funktion.

Die **return**-Anweisung entfällt, wenn der Resultattyp **void** ist.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Beispiele für Funktionen:

```
void druckergebnis(int anzahl, float summe)  {
  float MittelWert;

  if (anzahl > 0)
    MittelWert = summe/anzahl;
  else
    MittelWert = 0.0;
  fprintf(stdout,
    "Mittelwert %5.1f aus %3i Messungen\n",
     MittelWert, anzahl);
} /* druckergebnis */

float verarbeite( float wert, float summe) {
  return (summe + wert);
}

float minimum( float werta, float wertb) {
  if (werta < wertb)
    return werta;
  else
    return wertb;
}
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Im Folgenden soll ein vertiefendes Programmbeispiel dargestellt werden.

```
/* verwirr1.c 12.05.93 rei */

#include <stdio.h>

int b, c, x;

int tuwasmit (int a, int x)
{
  x = x + a;
  return  (x * x);
}  /*      tuwasmit */

int main (void)
{
  b = c = x = 0;

  fprintf(stdout, "\nWerte fuer b und c eingeben: ");
  fscanf(stdin, " %i %i", &b, &c);

  b = tuwasmit (b, c);

  fprintf(stdout,
     "\nb = %2i, c = %2i, x = %2i\n", b, c, x);
  return 0;
}  /* main */
```

Fragen:

- Welche Werte liefert das vorliegende Programm für b=2 und c=5 ?
- Welche Werte liefert das Programm, wenn die Vereinbarung des Bezeichners x aus dem Prozedurkopf und dem Aufruf gestrichen wird ?

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Die Vereinbarung einer Funktion erfolgt vor dem **main**-Teil bzw. vor der aufrufenden Funktion. In C allerdings kann auch nur der Prototyp, d.h. der Kopf einer Funktion vereinbart werden. Die volle Funktion muss dann allerdings noch an anderer Stelle, z.B. nach dem **main**-Teil als Quelltext oder beim Binden als Objekt geliefert werden.

Die **header**-Dateien, wie sie bisher über **#include** in den Quelltext eingefügt wurden, sind Prototypen. Ihre volle Funktion wird beim Binden aus Bibliotheken geladen.

Die Anweisungen **fprintf** und **fscanf** sind nichts anderes als der Aufruf vordefinierter Funktionen in der **header**-Datei **stdio.h.**

Ein Programm mit Funktionen

Wie im folgenden Beispiel gezeigt wird, kann eine Grobstruktur entworfen werden, in der bestimmte Funktionen als Prototypen für die Lösung einer Aufgabenstellung festgelegt werden. Die Verfeinerung der Äste der Grobstruktur, also die Ausführung der Funktionen, kann in Untergruppen oder durch einzelne Mitarbeiter einer Gruppe erfolgen.

Die Vorgehensweise des baumorientierten Entwerfens ist insbesondere für die Bearbeitung komplexer Projekte geeignet.

In der Projektgruppe sind gemeinsam abzusprechen

- die Grobstruktur
- die Schnittstellen
- der Zeitplan
- die Inbetriebnahme

Die Grobstruktur beschreibt in einfacher und verständlicher Art die Lösung der Aufgabenstellung und ist damit gleichzeitig ein lesbares Dokument für eventuelle Anwender.

Die baumorientierte Strukturierungsweise ist unabhängig von einer Programmiersprache. Sie entspricht einer systemtechnischen Denkweise, die bei der Entwicklung von Sprachen berücksichtigt wird.

```
 /* funktion.c 10.05.93 rei */
#include <stdio.h>
#include <math.h>
int Summiere1(int WertA, int WertB);
int Summiere2(int WertA, int WertB);
float Wuerfel(void);
void DruckWerte(int Stichprobe);

int main(void) {
  int Anfang, Ende;
  int Stichprobe;
  int Summe;
  fprintf(stdout, "Gib Anfangs- und Endwert ein\n");
  fscanf(stdin, " %i %i", &Anfang, &Ende);

  Summe = Summiere1(Anfang, Ende);
  fprintf(stdout, "Ergebnis Summiere1 = %i\n", Summe);

  Summe = Summiere2(Anfang, Ende);
  fprintf(stdout, "Ergebnis Summiere2 = %i\n", Summe);

  fprintf(stdout, "Gib Stichprobe fuer Wuerfel ein\n");
  fscanf(stdin, " %i", &Stichprobe);
  DruckWerte(Stichprobe);

  fprintf(stdout, "Ende des Programms funktion.c\n");
  return 0;
} /* main */

int Summiere1(int WertA, int WertB){
  int Zaehler;
  int Summe = 0.0;

  for (Zaehler = WertA; Zaehler <= WertB; Zaehler++)
    Summe = Summe + Zaehler;
  return Summe;
};
int Summiere2(int WertA, int WertB){
  int Anzahl;
  Anzahl = WertB - WertA + 1;
  return ((WertA + WertB) * Anzahl) / 2;
};
float Wuerfel(void){
  return ( -83.5 * log(1.0 - rand()/(32767 + 1.0)));
};
void DruckWerte(int Stichprobe){
  int Zaehler;
  for (Zaehler = 1; Zaehler <= Stichprobe; Zaehler++)
    fprintf(stdout, "WuerfelNr = %4i Wert = %6.1f\n",
            Zaehler, Wuerfel());
};
```

5.2 Datenverbunde

Bisher wurden zur Beschreibung technischer Systeme Eigenschaftsbezeichner vereinbart als Träger einfacher Datentypen (float, integer, char ...), wie sie in allen bekannten Programmiersprachen in ähnlicher Form bekannt sind.

Die Ablaufbeschreibung wurde durch die Einführung benannter Anweisungsverbunde (Funktionen) strukturiert. Diese Ablaufstruktur gilt ebenfalls für gängige Programmiersprachen (z.B. FORTRAN, COBOL, PASCAL, MODULA,...).

Allgemeine Beschreibung

Neuere Programmiersprachen bieten für eine strukturierende Systembeschreibung höhere Datentypen an, die ein Objekt als benannte Einheit von Grundeigenschaften modellieren.

Maschinen eines Maschinenparks können beispielsweise durch Eigenschaften ihrer Identität und ihres Belegungszustandes wie folgt beschrieben werden:

- Maschinennummer
- Hersteller
- Baujahr
- Anschaffungsjahr
- Planzeiten
- Planoperationen

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Beispiele für die Vereinbarung von Datenverbunden in der Form von Strukturen mit den Schlüsselworten **typedef** und **struct** und Zugriff auf die Komponenten des Verbundes:

- Vereinbarung des Typs

```
typedef
struct {
        int   nummer;
        char  hersteller;
        int   baujahr;
        int   anschaffungsjahr;
        float planzeiten;
        int   planoperationen;
} typMaschine;
```

- Vereinbarung eines Bezeichners vom Typ

```
typMaschine station;
```

- Zugriff auf Komponenten mit Punktnotation

```
station.nummer = 37;

station.hersteller = 'A';

station.baujahr = 1991;

station.anschaffungsjahr = 1993;

station.planzeiten += operationszeit;

station.planoperation++;
```

- Zugriff auf Komponenten eines Datenverbundes
  über die Adresse (z.B. in einer Funktion) mit Pfeilnotation (->)

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Beispiel Messwerterfassung mit Datenverbundtyp

Zur Demonstration der Verbunddatentypen werden im Folgenden Beispielprogramm minmaxi.c Messwerte statistisch verarbeitet.
Zur Erarbeitung der charakteristischen Werte wird ein Datenverbund **typRegister** mit den Komponenten **MinWert, MaxWert, Summe** und **Anzahl** als vorhanden angenommen, sowie die Funktionen **iRegister, sRegister und dRegister**, die mit diesem Typen arbeiten, wenn sie dessen Adresse über die Parameterliste erhalten.

```
/* minmaxi.c 09.05.93 rei */

#include <stdio.h>
#include "regis.h"

int main (void){
  float AktWert;
  typRegister Messung;

  iRegister(&Messung);

  fprintf(stdout, "\nMesswerte > 0.0 eingeben: \n");
  fscanf(stdin, " %f", &AktWert);

  while (AktWert > 0.0){
    sRegister(AktWert, &Messung);
    fscanf(stdin, " %f", &AktWert);
  }  /* while */

  dRegister(&Messung);
  return 0;
}  /* main */
```

Der Datenverbund typRegister und die Prototypen der Funktionen werden über

```
#include "regis.h"
```

in den Quelltext geladen. Die ausformulierten Funktionen, die in der Datei regis.c enthalten sind, werden als Programmodul compiliert und zum Hauptprogramm dazugelinkt.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

```
/* regis.h 12.05.93 rei */

typedef struct {
  float MinWert, MaxWert, Summe;
  int Anzahl;
} typRegister;

void iRegister(typRegister *reg);
void sRegister(float Wert, typRegister *reg);
void dRegister(typRegister *reg);
```

```
/* regis.c 12.05.93 rei */
#include <stdio.h>
#include "regis.h"

void iRegister(typRegister *reg){
  reg->Anzahl  =       0;
  reg->Summe   =     0.0;
  reg->MinWert =  9999.0;
  reg->MaxWert = -9999.0;
} /* iRegister */

void sRegister(float Wert, typRegister *reg){
  reg->Anzahl++;
  reg->Summe = reg->Summe + Wert;
  if (Wert > reg->MaxWert)
          reg->MaxWert = Wert;
  if (Wert < reg->MinWert)
          reg->MinWert = Wert;
} /* sRegister */

void dRegister(typRegister *reg){
  fprintf(stdout,
   "Anzahl der Werte = %6i\n", reg->Anzahl);
  if (reg->Anzahl > 0) {
    fprintf(stdout,
     "Mittelwert       = %6.2f\n",
                       (reg->Summe/reg->Anzahl));
    fprintf(stdout,
     "Maximaler Wert   = %6.2f\n", reg->MaxWert);
    fprintf(stdout,
     "Minimaler Wert   = %6.2f\n", reg->MinWert);
  }
  else fprintf(stdout, "===Keine Eingaben===\n");
} /* dRegister */
```

Übung: OBSTKASSE MIT TAGESUMSATZ mit Verbundtypen lösen

5.3 Konstanten und Felder

5.3.1 Konstanten

Konstanten sind Elemente eines Wertebereiches (ganze Zahlen, gebrochene Zahlen, Zeichen etc.). In C und anderen Sprachen können Bezeichner für Elemente eines Wertebereiches vereinbart werden. Dadurch besteht die Möglichkeit, Konstantenbezeichner zur verständlichen Beschreibung von festen Objekteigenschaften heranzuziehen.

Konstantenbezeichner werden mit unveränderlichem Wert vereinbart, d.h. im Programmablauf kann der Wert des Konstantenbezeichners nicht verändert sondern nur benutzt werden.

Im Gegensatz dazu waren die bisher besprochenen Bezeichner variabel.

Konstantenbezeichner enthalten eine Typvereinbarung mit Wertinitialisierung, und sie beginnen mit dem Schlüsselwort **const**

```
const int MAXKISTEN = 20;
const char ZUGRIFF = 'r';
const float intervall = 20.0;
const float pi = 3.14;
```

Konstantenbezeichner können im Variablenteil benutzt werden. Sie werden beispielsweise zur problemspezifischen Festlegung eines Programmes eingesetzt.

In C ist es bisher üblicher, Konstante mit dem Schlüsselwort **define** als globale Größe zu vereinbaren (vgl. Kap.3.1).

```
#define MAXWERT 37
```

Der Vorteil mit **const** liegt darin, dass der Typ der Konstante für den Compiler beim Übersetzen kontrollierbar ist.

5.3.2 Felder

Felder sind eine feste Anzahl von Elementen des gleichen Typs, wobei die Elemente des Feldes über einen Feldindex angesprochen werden können.

Der Feldindex kann aus operativen Verknüpfungen errechnet und zur Adressierung eines Feldelementes benutzt werden.

Vereinbarung:

< Typ > <Feldbezeichner > [<Laenge>];

Sind beispielsweise in einem Prozess mehrere Temperaturfühler abzulesen, so kann ein Temperaturfeld über einen Feldbezeichner vereinbart werden.

Beispiele:

(1)
```
float temperatur[10];
```

(2)
```
const int MAXSTELLEN = 20;

float temperatur[MAXSTELLEN];
int Stelle;
...
for (Stelle = 0; Stelle < MAXSTELLEN; Stelle++)
  fscanf(stdin, " %f", &temperatur[Stelle]);
...
```

Werden die Feldlängen über Konstantenbezeichner vereinbart, so muss bei einer Programmerweiterung nur die Konstantenvereinbarung entsprechend geändert werden (z.B. MAXSTELLEN).

!!!! WICHTIG !!!! Fehlerquelle mit Verlust an Lebenszeit !!!

Die Feldlänge 10 bedeutet in C, dass 10 Elemente im Feld vorhanden sind, jedoch hat das erste Feld den Index 0 und das letzte Feld den Index 9.

!!!! WICHTIG !!!! Fehlerquelle mit Verlust an Lebenszeit !!!

Zeichenketten (String) werden als Feld von Zeichen vereinbart. Sie können mit dem Format **"%s"** eingelesen und ausgeschrieben werden. Für die Verarbeitung von Strings gibt es in C eine Vielzahl von Verarbeitungsfunktionen. Die Prototypen für Stringoperatoren stehen in der **Header**-Datei **string.h**, die mit **#include** in die Programmumgebung eingebunden wird.

Für die Operationen mit Strings ist das Zeichen **'\0'** als Endekriterium für Strings festgelegt. Bei der Vereinbarung einer Zeichenkette muss also genügend Platz reserviert werden.

Ein Beispiel mit Tücken:

```
/* str00.c 09.05.93  rei */

#include <stdio.h>

#define StrLaenge 20

void ZeichenAus(char z[]);
void ZeichenEin(char z[]);

int main(void) {
  char Zeichenkette[StrLaenge];
  int Index = 0;

  fprintf(stdout,
     "Gib Zeichenkette bis %i Zeichen ein\n", StrLaenge);
  ZeichenEin(Zeichenkette);
  while (Zeichenkette[Index] != '\0') Index++;
  ZeichenAus(Zeichenkette);
  fprintf(stdout, "Index = %3i\n", Index);
} /* main */
void ZeichenAus(char z[]) {
  fprintf(stdout, ">%s<\n", z);
}
void ZeichenEin(char z[]) {
  fscanf(stdin, "%s", z);
}
```

Ein Beispiel für Grafik mit Zeichen:

```
/* chargraf.c 15.10.98  rei */

#include <stdio.h>

#define MAXSTRING 31

int main(void){
  /* Zeichenkette mit fester Laenge ohne Init. */
  char Z_Kette[MAXSTRING];

  /* Zeichenkette mit Laenge durch Init. */
  char KopfZeile[] = "> char-graphik 11.05.93 rei <";

  char Merker;
  int z, stelle;

  /* Initialisierung char-Feldes mit EndeZeichen */
  for (stelle = 0; stelle < MAXSTRING - 1; stelle++)
    if (stelle % 2 == 0) Z_Kette[stelle] = '.';
    else Z_Kette[stelle] = ' ';

  Z_Kette[stelle] = '\0';

  fprintf(stdout, "\n%s\n", KopfZeile);
  fprintf(stdout, "%s\n", Z_Kette);

  for (z = 0; z < 15; z++){
    stelle = z * 2;
    Merker = Z_Kette[stelle];
    Z_Kette[stelle] = '*';
    fprintf(stdout, "%s\n", Z_Kette);
    Z_Kette[stelle] = Merker;
  }
  fprintf(stdout, "%s\n", Z_Kette);
  return 0;
} /* main */
```

Bemerkung: Im **chargraf.c**, dort in der ersten **for**-Schleife gibt es den Ausdruck **stelle % 2**. Dieser Ausdruck enthält eine Moduloperation (Restwertoperation). Was ist das, wie wirkt die Operation?

Ausgabe zum Beispiel Zeichenkette:

```
> char-graphik 15.10.98 rei <
. . . . . . . . . . . . . . .
* . . . . . . . . . . . . . .
. * . . . . . . . . . . . . .
. . * . . . . . . . . . . . .
. . . * . . . . . . . . . . .
. . . . * . . . . . . . . . .
. . . . . * . . . . . . . . .
. . . . . . * . . . . . . . .
. . . . . . . * . . . . . . .
. . . . . . . . * . . . . . .
. . . . . . . . . * . . . . .
. . . . . . . . . . * . . . .
. . . . . . . . . . . * . . .
. . . . . . . . . . . . * . .
. . . . . . . . . . . . . * .
. . . . . . . . . . . . . . *
. . . . . . . . . . . . . . .
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

5.3.3 Beipiel zu Konstanten und Feldern

In dem Beispiel wird ein Feld von Kisten zum Einsortieren von Messwerten benutzt. Die Anzahl der Kisten wird über eine Konstante vorgegeben. Die Messwerte werden über eine Funktion Wuerfel mit unbekannten Werten erzeugt.

Wenn die Anzahl der Klassen verändert werden soll, dann muss nur die Konstante MAXKISTEN verändert werden. Das Programm muß allerdings neu übersetzt werden!
(Wir werden später eine bessere Lösung erarbeiten.)

```
/* felderi.c  12.05.93  rei */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "kiste.h"

#define MAXKISTEN 20

float Wuerfel(void){
  float wert;
  wert = -83.0 * log(1.0 - rand()/(RAND_MAX + 1.0));
  return wert;
}
int main(void){
  int KistenNr;
  TypKiste Sortierer[MAXKISTEN];
  float Durchmesser;
  int AktZiehung;
  int Stichprobe;

  InitKiste(20.0, Sortierer, MAXKISTEN);

  fprintf(stdout, "Stichprobe eingeben\n");
  fscanf(stdin, " %i", &Stichprobe);
  for  (AktZiehung  =  1;  AktZiehung  <=  Stichprobe;
    AktZiehung++){
   Durchmesser = Wuerfel();
   SchreibKiste(Durchmesser, Sortierer, MAXKISTEN);
  }
  DruckKiste(Sortierer, MAXKISTEN);
  return 0;
} /* main */
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

```
/* kiste.h 12.05.93  rei */

typedef struct Kiste {
  float Grenze;
  int Inhalt;
} TypKiste;

void InitKiste(int Interv, TypKiste Sort[], int Anzahl);
void SchreibKiste(float Wert, TypKiste Sort[]);
void DruckKiste(TypKiste Sort[], int Anzahl);
```

```
/* kiste.c 12.05.93  rei */

#include <stdio.h>
#include "kiste.h"

void InitKiste(int Interv, TypKiste Sort[], int Anzahl){
  int KistenNr;

  for (KistenNr = 0; KistenNr < Anzahl; KistenNr++){
    Sort[KistenNr].Inhalt = 0;
    Sort[KistenNr].Grenze = (KistenNr + 1) * Interv;
  }
  Sort[Anzahl - 1].Grenze = 99999.9;
}
void SchreibKiste(float Wert, TypKiste Sort[]){
   int KistenNr = 0;
   while (Wert > Sort[KistenNr].Grenze) KistenNr++;
   Sort[KistenNr].Inhalt++;
};
void DruckKiste(TypKiste Sort[], int Anzahl){
  int KistenNr;
  fprintf(stdout,"Nr.  Grenze  Inhalt\n");
  for (KistenNr = 0; KistenNr < Anzahl; KistenNr++){
    fprintf(stdout, "%3i %7.1f %5i\n",
       KistenNr, Sort[KistenNr].Grenze,
       Sort[KistenNr].Inhalt);
  }
};
```

5.4 Zeiger und temporäre Datenverbunde

Bisher wurden Datenverbunde als **struct**-Typen deklariert und in Vereinbarungen von Bezeichnern benutzt. Bei der Vereinbarung eines Bezeichners vom bestimmten Typ reserviert der Compiler den nötigen Speicherplatz. Dieser bleibt über die Programmlaufzeit erhalten und kann nicht anderweitig verwendet werden.

Eine andere Möglichkeit in C und anderen Sprachen besteht darin, zur Laufzeit aufgabenbezogen Speicher zu belegen und wieder freizugeben. Die Datenobjekte existieren nur temporär, d.h. über die Zeit, in der sie gebraucht werden.

Zur Verwaltung temporärer Datenverbunde bzw. Speicherbereiche werden Adressen bzw. Zeiger auf Speicherbereiche vereinbart.

Am Beispiel eines Kartoffelsortierers wird die Anzahl der Kisten zur Laufzeit bestimmt und der benötigte Speicher erst dann reserviert.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Folgende Funktionen sind für das Verfahren in C vorhanden (vgl. Bücher):

malloc (<Speicherbedarf>)

Reserviert Speicher der angegebenen Größe und liefert die Adresse des Speicherbereiches oder den Wert **NULL** wenn keiner frei ist.

free (<Adresse>)

Gibt den über die Adresse bezeichneten und über **malloc** reservierten Speicherbereich wieder frei.

Vereinbarung eines Datenverbundes

```
typedef struct {
        float Grenze;
        int Inhalt;
} TypKiste;
```

Vereinbarung von Zeigern auf einen Datenverbund mit Sternnotation (*):

```
TypKiste *Erste, *Letzte;
TypKiste *zAktuell;
```

Erzeugen und Zugriff mit Pfeilnotation (->):

```
zAktuell = (TypKiste *)malloc(sizeof(TypKiste));
zAktuell->Grenze = 37.0;
Erste = zAktuell;
Erste->Inhalt = 41;
```

Löschen und Speicherfreigabe:

```
free (zAktuell);
zAktuell = NULL;
Erste = NULL;
```

Funktionen und Parameter:

- Vereinbarung

```
TypKiste *ErzKiste(float Wert){
        TypKiste *zK;
        zK = (TypKiste *)malloc(sizeof(TypKiste));
        zK->Grenze = Wert;
        zK->Inhalt = 0;
        return zK;
}
```

- Aufruf

```
Erste = ErzKiste(20.0);
```

Zeiger in einem Datenverbund:

- Typvereinbarung

```
typedef struct Kiste {
        float Grenze;
        int Inhalt;
        struct Kiste *Nachfolger;
} TypKiste;
```

Im Typ gibt es als Komponente **Nachfolger** einen Zeiger auf ein weiteres Objekt. Da im Verbund der Typname noch nicht bekannt ist, wird der Zwischentyp '**struct Kiste**' benutzt.

- Anwendung

```
TypKiste *Erste, *zKiste;

zKiste = ErzKiste(20.0);
Erste = zKiste;
zKiste->Nachfolger = ErzKiste(40.0);
zKiste = zKiste->Nachfolger;
```

```
/* zsorter.c  31.05.93 rei */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "histo.h"

float Wuerfel(void){
  float wert;
  wert = -83.0 * log(1.0 - rand()/(RAND_MAX + 1.0));
  return wert;
}
int main(void){
  TypHKlasse *zSorter;
  float Intervall; int Anzahl;

  float Durchmesser;
  int AktZiehung, Stichprobe;

  fprintf(stdout,
     "Stichprobe, Intervall, KlassenAnzahl ?\n");
  fscanf(stdin,
     " %i %f %i", &Stichprobe, &Intervall, &Anzahl);
  zSorter = iHistogramm(Intervall, Anzahl);

  for (AktZiehung = 1;
     AktZiehung <= Stichprobe; AktZiehung++){
   Durchmesser = Wuerfel();
   sHistogramm(Durchmesser, zSorter);
  }
  dHistogramm(zSorter);
  return 0;
} /* main */
```

```
/* histo.h 31.05.93 rei */

typedef struct HKlasse {
  float Grenze;
  int Inhalt;
  struct HKlasse *nKlasse;
} TypHKlasse;

TypHKlasse *iHistogramm(float Interv, int Anzahl);
void sHistogramm(float Wert, TypHKlasse *zKlasse);
void dHistogramm(TypHKlasse *zKlasse);
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

```
/* histo.c 31.05.93 rei */

#include <stdio.h>
#include <stdlib.h>
#include "histo.h"

TypHKlasse *ErzHKlasse(float GrenzWert) {
   TypHKlasse *zKlasse;

   zKlasse = (TypHKlasse *)malloc(sizeof(TypHKlasse));
   zKlasse->Grenze = GrenzWert;
   zKlasse->Inhalt = 0;
   zKlasse->nKlasse = NULL;
   return zKlasse;
}
TypHKlasse *iHistogramm(float Interv, int Anzahl){
  TypHKlasse *zAnfang, *zKlasse;
  int KlassenNr;
  zAnfang = ErzHKlasse(Interv);
  zKlasse = zAnfang;
  for (KlassenNr = 2; KlassenNr <= Anzahl; KlassenNr++){
    zKlasse->nKlasse = ErzHKlasse(KlassenNr * Interv);
    zKlasse = zKlasse->nKlasse;
  }
  zKlasse->Grenze = 99999.9;
  return zAnfang;
}
void sHistogramm(float Wert, TypHKlasse *zKlasse){
   while (Wert > zKlasse->Grenze)
         zKlasse = zKlasse->nKlasse;
   zKlasse->Inhalt++;
};
void dHistogramm(TypHKlasse *zKlasse){
  int KlassenNr = 0;
  fprintf(stdout,"Nr.  Grenze  Inhalt\n");
  while (zKlasse != NULL) {
    KlassenNr++;
    fprintf(stdout, "%3i %7.1f %5i\n", KlassenNr,
      zKlasse->Grenze, zKlasse->Inhalt);
    zKlasse = zKlasse->nKlasse;
  }
};
```

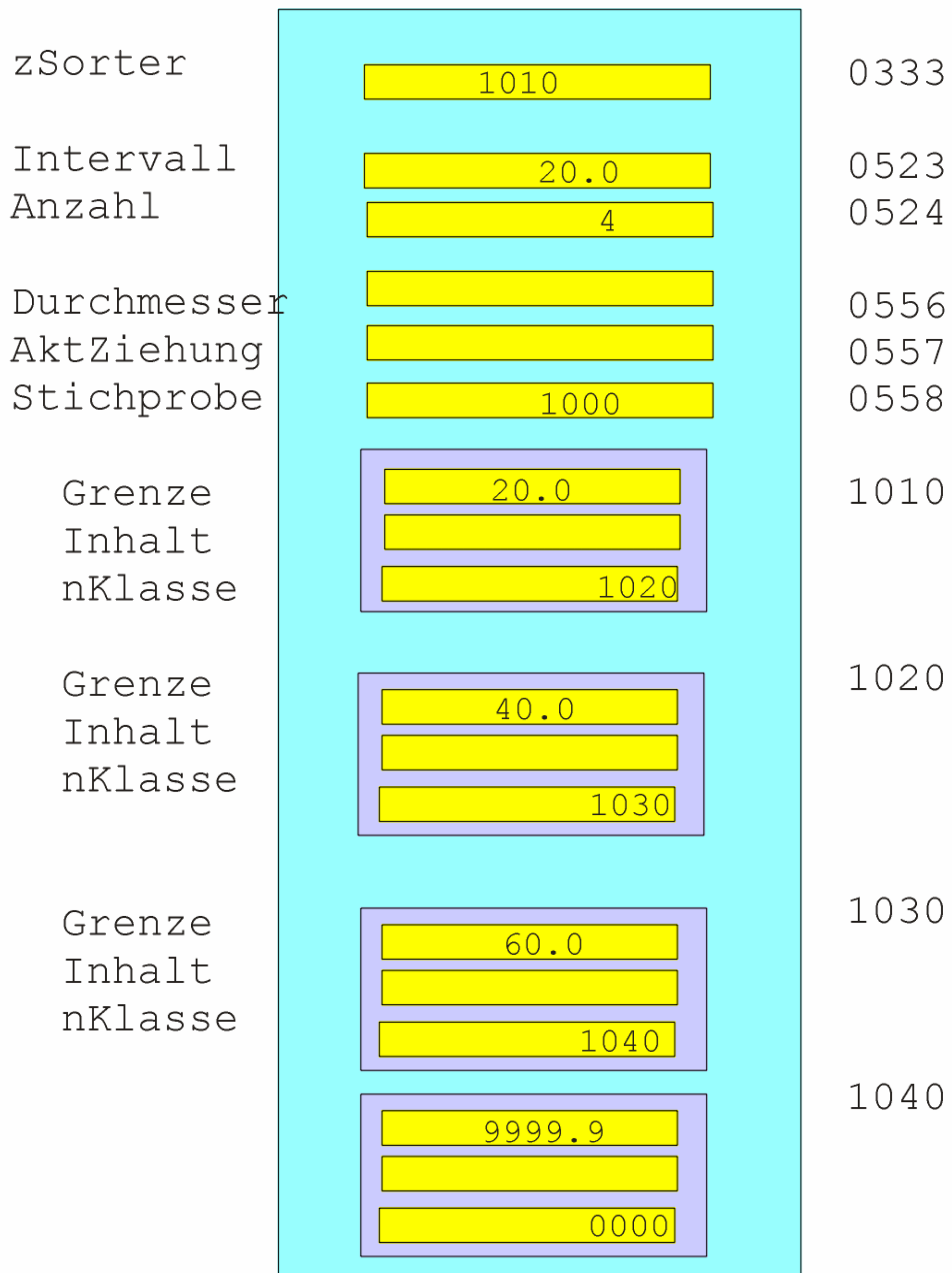


Abb. 1 Speicherbild zum Beispiel ZSORTER.C

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

5.5 Lösung der Aufgabenstellung

Die Aufgabenstellung des Kapitels soll anhand der ausgeführten Beispiele gelöst werden. Das Programm soll die Grobstruktur erkennen lassen (Daten- und Ablaufverbunde: schon ganz schön schwer ?).

Die Messwerte sollen aus der Funktion Wuerfel gezogen werden. Es sind 3 Versionen zu erarbeiten. Nehmen Sie als Basis Beispiel **felderi.c, zsorter.c** (vgl.Kap.5) und **s-class.cpp** (vgl.Kap.6) und erzeugen Sie für eine beliebige Stichprobe folgende Ergebnisse nach Form und Inhalt:

```
Stichprobe, Intervall, Anzahl eingeben
10000 20.0 15
Anzahl der Werte =  10000
Mittelwert       =  81.45
Maximaler Wert   = 862.97
Minimaler Wert   =   0.01

MaxInhalt = 2172

Nr. GrenzeInhalt  0       20      40       60       80     100%
 1    20.0 2172   |       .       .        .        .      *
 2    40.0 1688   |       .       .        .    *   .      |
 3    60.0 1359   |       .       .        .*       .      |
 4    80.0  997   |       .       . *      .        .      |
 5   100.0  844   |       .     * .        .        .      |
 6   120.0  637   |       .   *   .        .        .      |
 7   140.0  497   |       .*      .        .        .      |
 8   160.0  382   |    *  .       .        .        .      |
 9   180.0  327   |    *  .       .        .        .      |
10   200.0  252   |   *   .       .        .        .      |
11   220.0  179   |  *    .       .        .        .      |
12   240.0  137   |  *    .       .        .        .      |
13   260.0  133   |  *    .       .        .        .      |
14   280.0   92   | *     .       .        .        .      |
15 99999.9  304   |    *  .       .        .        .      |
```

6 Objektorientierte Programmierung

Gegenstände

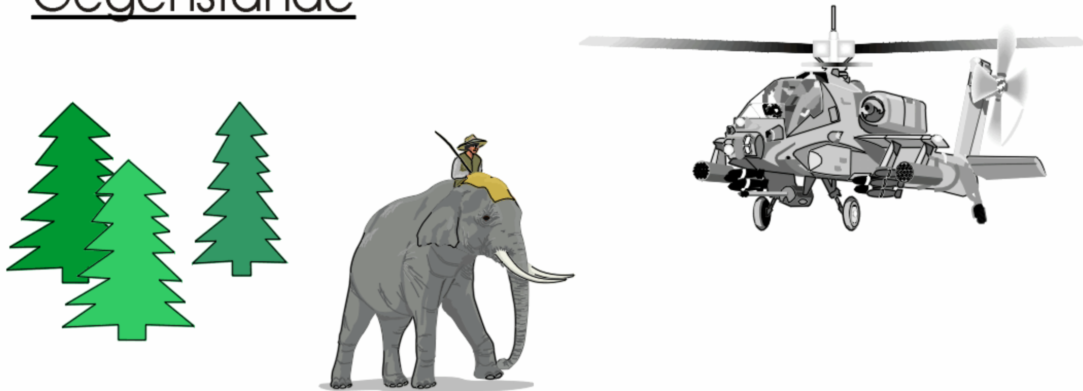


Abbildung mit Prog.Sprachen im Rechner

Einfache Datentypen + Funktionen

int

float

char

Operation1

Operation2

Datenverbunde + Funktionen

struct

int

float

char

Operation1

Operation2

Klassen: Datentypen+Methoden

Vererbung

class

struct

int

flaot

char

Operation1

Operation2

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

6.1 Klassen

Die Entwicklung neuer Programmiersprachen wird getrieben durch das Bemühen, komplexe Zusammenhänge als Ausschnitte einer betrachteten Realität, softwaretechnisch einfach und transparent zu beschreiben. Dies ist gleichbedeutend mit der kostengünstigen Entwicklung und Pflege von Software.

Die objektorientierte Programmierung unterstellt, dass eine Sprache, hier eine Programmiersprache, eine bestimmte Sichtweise auf die Welt (World View of Language) beinhaltet. Diese Sichtweise wurde Ende der 60er Jahre in der Sprache SIMULA in Norwegen implementiert und hauptsächlich in Hochschulen eingesetzt. Die Vorteile dieses Konzeptes, insbesondere in Bezug auf Kosten, wurden allerdings damals im industriellen Bereich nicht erkannt. Der Ansatz ging verloren.
Die Schmerzen der hohen Softwarekosten haben heute zur Wiederbelebung dieses Ansatzes in den Sprachen wie SMALLTALK, C++ und anderen geführt.

Das Wesentliche des Konzeptes lässt sich durch drei Merkmale beschreiben:

- Ein Gegenstand der realen Welt wird modelliert bzw. programmiert als Einheit aus Attributen und Operationen auf diese. Eine Klasse (class) ist eine Einheit aus Daten (Attribute) und Ablaufverbunden (Methoden bzw. Funktionen), die auf diese wirken.

- Die Attribute einer Klasse können gegen Zugriff von außen geschützt (hidden) werden.

- Die Eigenschaften einer Klasse lassen sich auf andere Objekte vererben.

Ein Mensch kann beispielsweise aufgabenbezogen gesehen werden als Objekt. Er hat dann Eigenschaften wie einen Standort im Raum; er kann hören und sehen, und er hat die Fähigkeit, seinen Standort zu ändern, d.h. sich zu bewegen, wenn er will.

Ein Handwerker ist ein Mensch. Er hat Werkzeuge und die Fähigkeit, mit diesen ein Bild an die Wand zu nageln, d.h. die Fähigkeit, Aufgaben mit Werkzeugen zu lösen. Da er ein Mensch ist, kann er sich an verschiedene Orte bewegen und dort Aufgaben lösen.

Am Beispiel Sortierer aus dem vorangegangenen Kapitel soll eine Lösung mit Klassen aufgezeigt werden.

Vereinbarung einer Klasse

```
class Mensch {
public:
  Mensch(){ // Konstruktor
    ...................
  }
  ~Mensch(){ // Destruktor
    ....................
  }
  < Methoden um die eigenen Attribute
    zu ändern und nach außen zu wirken
  >
protected:
  < Attribute>

}
```

Vereinbarung einer abgeleiteten Klasse

```
class Tischler : public Mensch {
public:
  < zusätzliche Methoden >
protected:
  < zusätzliche Attribute >
}
```

Erzeugen eines Klassenobjektes

```
Tischler *Neuer;

Neuer = new Tischler();
```

Der Generator **new** erzeugt ein Klassenobjekt (Instanz) im Speicher und liefert die Speicheradresse. Der **Konstruktor** initialisiert die Datenstruktur und Attribute der Klasse.

Zugriff auf Attribute über die Methoden

```
Neuer-><Methode>;
```

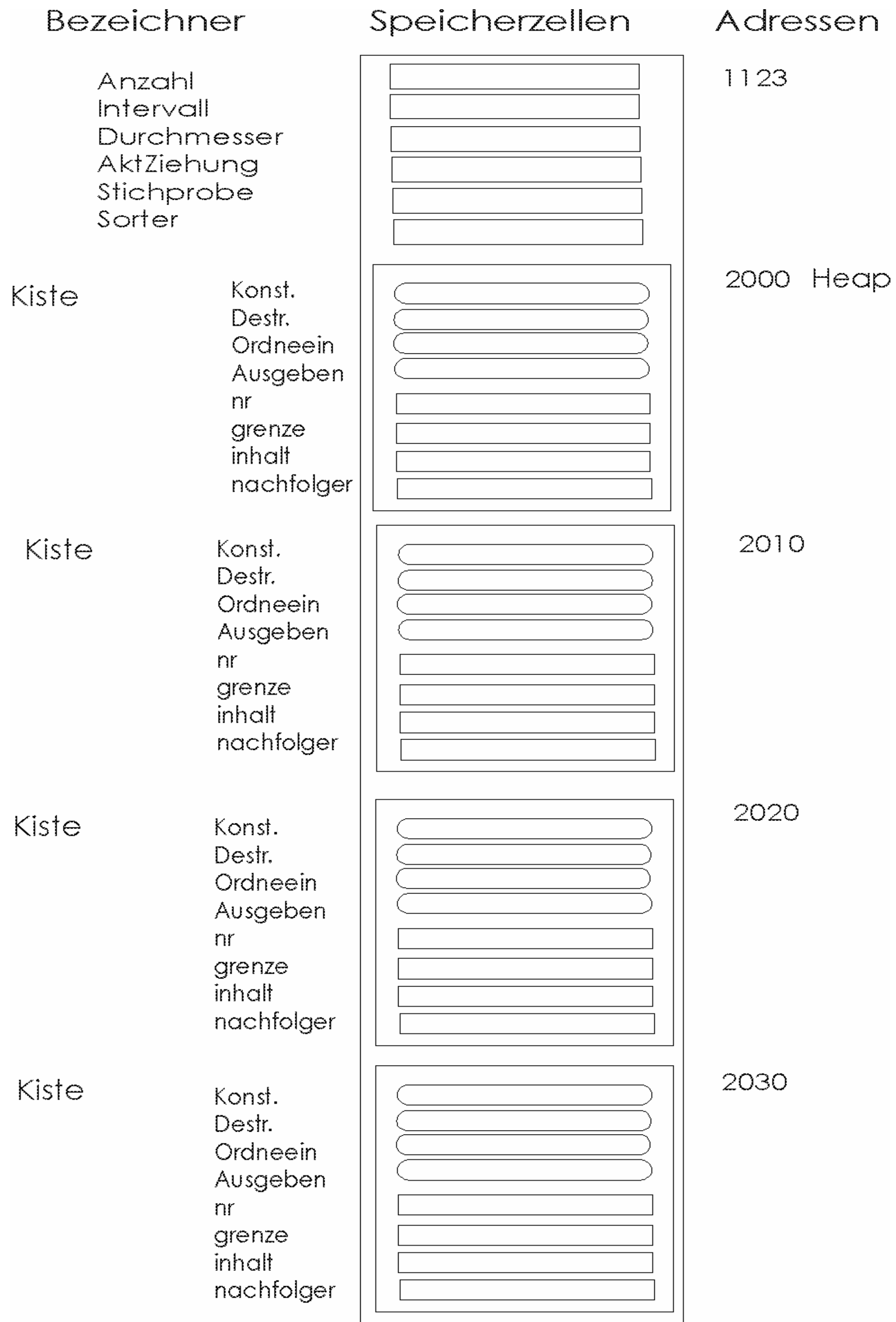
Löschen einer Klasseninstanz

```
delete Neuer;
```

Mit dem Operator **delete** wird eine Klasseninstanz aus dem Speicher entfernt. Der **Destruktor** der Klasse löscht dabei die Datenstruktur und gibt den Speicher wieder frei.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

6.2 Sortierer mit Klassen



© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Lösung: Sortierer mit Klassen

```
/* s-class0.cpp  13.12.99 rei */

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

#include "h-class0.h"

float Wuerfel(void){
  float wert;
  wert = -83.0 * log(1.0 - rand()/(32767+ 1.0));
  return wert;
}
int main(void){
  Kiste *Sorter;
  float Intervall; int Anzahl;

  float Durchmesser;
  int AktZiehung, Stichprobe;

  fprintf(stdout,
     "Stichprobe, Intervall, KlassenAnzahl ?\n");
  fscanf(stdin,
     "%i %f %i", &Stichprobe, &Intervall, &Anzahl);

  fprintf(stdout, "Sorter wird initialisiert\n");
  Sorter = new Kiste(0, Intervall, Anzahl);

  for (AktZiehung = 1;
     AktZiehung <= Stichprobe; AktZiehung++){
   Durchmesser = Wuerfel();
   Sorter->Ordneein(Durchmesser);
  }
  fprintf(stdout, " Nr. Grenze Inhalt\n");
  Sorter->Ausgeben();

  fprintf(stdout, "Speicher freigeben \n");
  delete Sorter;
  return 0;
} /* main */
```

Modul: Sortierer mit Klassen

```
/* h-class0.h 13.12.99 rei */

class Kiste{
public:
   Kiste(int n, float gr, int max);
   ~Kiste();
   void Ordneein(float wert);
   void Ausgeben();
protected:
   int nr;
   float grenze;
   int inhalt;
   Kiste *nachfolger;
};
```

```
/* h-class0.cpp 13.12.99 rei */

#include <stdio.h>
#include "h-class0.h"

Kiste::Kiste(int n, float gr, int max){
   nr = n;
   grenze = gr + nr * gr;
   inhalt = 0;
   nachfolger = NULL;
   fprintf(stdout,
      "Kiste %2i ist im Speicher initialisiert\n", nr);
   if (nr < max)
        nachfolger = new Kiste(nr + 1, gr, max);
   else
        grenze = 9999.9;
}
Kiste::~Kiste(){
   fprintf(stdout,
      "Kiste %2i gibt den Speicher frei\n", nr);
   delete nachfolger;
};
void Kiste::Ordneein(float wert){
   if (wert > grenze)
      nachfolger->Ordneein(wert);
   else inhalt++;
}
void Kiste::Ausgeben(){
      fprintf(stdout, " %3i %6.1f %4i\n",
                 nr, grenze, inhalt);
      if (nachfolger != NULL)nachfolger->Ausgeben();
}
```

Ablaufprotokoll: Sortierer mit Klassen

```
Stichprobe, Intervall, KlassenAnzahl ?
1000 20.0 15
Sorter wird initialisiert
Kiste  0 ist im Speicher initialisiert
Kiste  1 ist im Speicher initialisiert
Kiste  2 ist im Speicher initialisiert
Kiste  3 ist im Speicher initialisiert
......
Kiste  9 ist im Speicher initialisiert
Kiste 10 ist im Speicher initialisiert
Kiste 11 ist im Speicher initialisiert
Kiste 12 ist im Speicher initialisiert
Kiste 13 ist im Speicher initialisiert
Kiste 14 ist im Speicher initialisiert
Kiste 15 ist im Speicher initialisiert
 Nr. Grenze Inhalt
   0   20.0  208
   1   40.0  185
   2   60.0  132
   3   80.0  101
   4  100.0   84
   5  120.0   58
   6  140.0   47
   7  160.0   26
   8  180.0   32
   9  200.0   29
  10  220.0   21
  11  240.0   11
  12  260.0   19
  13  280.0   10
  14  300.0    7
  15 9999.9   30
Speicher freigeben
Kiste  0 gibt den Speicher frei
Kiste  1 gibt den Speicher frei
Kiste  2 gibt den Speicher frei
Kiste  3 gibt den Speicher frei
......
Kiste  9 gibt den Speicher frei
Kiste 10 gibt den Speicher frei
Kiste 11 gibt den Speicher frei
Kiste 12 gibt den Speicher frei
Kiste 13 gibt den Speicher frei
Kiste 14 gibt den Speicher frei
Kiste 15 gibt den Speicher frei
```

Modul: Register mit Klassen

```
/* regis-pp.h 14.12.99 rei */

class Register {
public:
   Register();
   ~Register();
   void sRegister(float wert);
   void dRegister();
protected:
   float sum, min, max;
   int anzahl;
};
```

```
/* regis-pp.cpp 14.12.99 rei */

#include <stdio.h>
#include "regis-pp.h"

Register::Register(){
   anzahl = 0;
   sum = 0.0;
   min = 99999.9;
   max = -min;
//   fprintf(stdout, "Register initialisiert\n");
};
Register::~Register(){
//   fprintf(stdout, "Register geloescht\n");
};
void Register::sRegister(float wert){
   anzahl++;
   sum += wert;
   if (wert > max) max = wert;
   if (wert < min) min = wert;
};
void Register::dRegister(){
   float mittlWert = 0.0;
   if (anzahl > 0) mittlWert = sum/anzahl;
   fprintf(stdout, "Anzahl     = %7i\n", anzahl);
   fprintf(stdout, "Minimum    = %7.1f\n", min);
   fprintf(stdout, "Mittelwert = %7.1f\n", mittlWert);
   fprintf(stdout, "Maximum    = %7.1f\n", max);
};
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Ergebnisprotokoll: Sortierer mit Register+Graph

```
Stichprobe, Intervall, KlassenAnzahl ?
1000 20.0 20
Messung wird initialisiert
Anzahl     =    1000
Minimum    =     0.1
Mittelwert =    83.4
Maximum    =   506.9
Maxinhalt 208
 Nr. Grenze Inhalt
   0   20.0  208 |#################################################|
   1   40.0  185 |############################################-----|
   2   60.0  132 |###############################------------------|
   3   80.0  101 |########################-------------------------|
   4  100.0   84 |####################-----------------------------|
   5  120.0   58 |#############------------------------------------|
   6  140.0   47 |###########--------------------------------------|
   7  160.0   26 |######-------------------------------------------|
   8  180.0   32 |#######------------------------------------------|
   9  200.0   29 |######-------------------------------------------|
  10  220.0   21 |#####--------------------------------------------|
  11  240.0   11 |##-----------------------------------------------|
  12  260.0   19 |####---------------------------------------------|
  13  280.0   10 |##-----------------------------------------------|
  14  300.0    7 |#------------------------------------------------|
  15  320.0    6 |#------------------------------------------------|
  16  340.0    5 |#------------------------------------------------|
  17  360.0    6 |#------------------------------------------------|
  18  380.0    3 |-------------------------------------------------|
  19  400.0    3 |-------------------------------------------------|
  20 9999.9    7 |#------------------------------------------------|
Speicher freigeben
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

7 Verkettete Objekte

Es gibt eine Vielzahl von Aufgaben, in denen Gegenstände oder Wesen in einer Ordnung zueinander stehen. Beispiele dafür sind allgemein Warteschlangen an Kassen, beim Arzt usw., Artikel in einem Lager, Werkstücke vor einer Maschine, Karteikarten in einem Kasten oder auch ein flexibler Terminkalender. Solche Zusammenhänge werden im Speicher als Struktur dargestellt, in der das Ganze (Lager, Karteikasten..) und die Teile (Artikel, Karteikarte..) abgebildet werden.
Zur Lösung solcher Aufgaben werden schrittweise Verfahren erarbeitet, die zur Lösung einer Aufgabenstellung LAGERVERWALTUNG führen oder die Basis für ein Adressenverwaltungssystem darstellen.

7.1 Ketten

An einem Beispiel soll gezeigt werden, wie eine offene Kette von Objekten mit Vorgänger- und Nachfolger-Beziehungen aufgebaut und verwendet wird.

Die Kette von n Objekten, beispielsweise Menschen, mit Namen und Jahrgang wird aufgebaut. Namen und Alter vom ersten bis zum letzten Menschen in der Kette werden ausgegeben.

Folgende Aufgaben sind zu erarbeiten:

- Ausgabe von Namen und Alter vom letzten bis zum ersten Menschen
- Wie kann sich ein neuer Mensch in die Kette fügen, an den Anfang, an das Ende oder in der Mitte?
- Wie kann ein Mensch sich aus der Mitte der Kette lösen, ohne die Kette zu zerstören?
- Wie kann der erste oder letzte Mensch aus der Kette gelöst werden?

Bezeichner
Speicherzellen
Adressen
Jahrgang
Name
Erster
Letzter
Kumpel
Anzahl
1123
Mensch A
Konstruktor
....
Vorgaenger
Nachfolger
ObjJahrgang
ObjName
ObjNachfolger
ObjVorgaenger
2000 Heap
Mensch B
Konstruktor
....
Vorgaenger
Nachfolger
ObjJahrgang
ObjName
ObjNachfolger
ObjVorgaenger
2010
Mensch C
Konstruktor
....
Vorgaenger
Nachfolger
ObjJahrgang
ObjName
ObjNachfolger
ObjVorgaenger
2020
Mensch X
Y
Z
Konstruktor
....
Vorgaenger
Nachfolger
ObjJahrgang
ObjName
ObjNachfolger
ObjVorgaenger
3011

```
// kette.cpp 03.01.00 rei

#include <stdio.h>
#include "mensch1.h"

int main(void){
   int Jahrgang = 1960;
   char Name = 'A';

   Mensch* Erster;
   Mensch* Letzter;
   Mensch* Kumpel;
   int Anzahl  = 0;

   fprintf(stdout,
        "Gib Anzahl der Menschen in der Kette an: ");
   fscanf(stdin, " %i", &Anzahl);

   fprintf(stdout, "Kette1: Initialisieren\n");
   for (int Zaehler = 1; Zaehler <= Anzahl; Zaehler++){
     Kumpel = new Mensch(Name++, Jahrgang++);
     if (Zaehler == 1)
        Erster = Letzter = Kumpel;
     else {// Setze Objekt an das Ende
        Kumpel->gehHinter(Letzter);
        Letzter = Kumpel;
     } //if
  } //for

  // DruckeVorwaerts
  fprintf(stdout,"Kette1: DruckeVorwaerts\n");
  Kumpel = Erster;
  while (Kumpel != NULL) {
     Kumpel->schreibIdentitaet(2000);
     Kumpel = Kumpel->Nachfolger();
  }

  // DruckeRueckWaerts  ???
  // Setze neues Objekt an den Anfang ???
  // Setze neues Objekt hinter 2. Objekt ???
  // Setze neues Objekt vor letztes Objekt ???

  // Kette loeschen
  fprintf(stdout, "Kette1: Loeschen\n");
  while (Erster != NULL) {
     Kumpel = Erster;
     Erster = Kumpel->Nachfolger();
     delete Kumpel;
  }
  return 0;
} //main
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

```
// mensch1.h  03.01.00 rei
class Mensch {
public:
   Mensch(char name, int jahr);
   ~Mensch();
   int Alter(int AktJahr);
   char Name();
   Mensch* Vorgaenger();
   Mensch* Nachfolger();
   void gehVor(Mensch* wen);
   void gehHinter(Mensch* wen);
   void schreibIdentitaet(int AktJahr);
protected:
   void setVorgaenger(Mensch* wer);
   void setNachfolger(Mensch* wer);
   int ObjJahrgang;
   char ObjName;
   Mensch* ObjNachfolger;
   Mensch* ObjVorgaenger;
};
// mensch1.cpp  03.01.00 rei
#include <stdio.h>
#include "mensch1.h"
   Mensch::Mensch(char name, int jahr){
      ObjJahrgang = jahr;
      ObjName = name;
      ObjNachfolger = NULL;
      ObjVorgaenger = NULL;
      fprintf(stdout,
      "Der Mensch %c, %4i ist im Speicher\n", ObjName, ObjJahrgang);
   };
   Mensch::~Mensch(){
     fprintf(stdout,
      "Der Mensch %c, %4i loescht sich\n", ObjName, ObjJahrgang);
   };
   Mensch* Mensch::Vorgaenger(){ return ObjVorgaenger;
   };
   Mensch* Mensch::Nachfolger(){ return ObjNachfolger;
   };
   void Mensch::setVorgaenger(Mensch* wer){
      ObjVorgaenger = wer;
   };
   void Mensch::setNachfolger(Mensch* wer){
      ObjNachfolger = wer;
   };
   void Mensch::gehVor(Mensch* wen){
      wen->setVorgaenger(this);   // ???
   };
   void Mensch::gehHinter(Mensch* wen){
      wen->setNachfolger(this);    // ???
   };
   void Mensch::schreibIdentitaet(int AktJahr){
      fprintf(stdout,
        "Name %c  Alter %4i\n", ObjName, AktJahr - ObjJahrgang);
   };
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Ablaufprotokoll Kette1:

```
Gib Anzahl der Menschen in der Kette an: 3
Kette1: Initialisieren
Der Mensch A, 1960 ist im Speicher
Der Mensch B, 1961 ist im Speicher
Der Mensch C, 1962 ist im Speicher
Kette1: DruckeVorwaerts
Name A  Alter   40
Name B  Alter   39
Name C  Alter   38
Kette1: Loeschen
Der Mensch A, 1960 loescht sich
Der Mensch B, 1961 loescht sich
Der Mensch C, 1962 loescht sich
```

Ablaufprotokoll Kette2:

```
Gib Anzahl der Menschen in der Kette an: 3
Kette2: Initialisieren
Kette2: DruckeVorwaerts
Name A  Alter   40
Name B  Alter   39
Name C  Alter   38

Kette2: DruckeRueckwaerts
Name C  Alter   38
Name B  Alter   39
Name A  Alter   40

Setze Objekt X an den Anfang

Kette2: DruckeVorwaerts
Name X  Alter    0
Name A  Alter   40
Name B  Alter   39
Name C  Alter   38

Setze Objekt Y hinter 2.Objekt
Name A  Alter   40

Kette2: DruckeVorwaerts
Name X  Alter    0
Name A  Alter   40
Name Y  Alter    1
Name B  Alter   39
Name C  Alter   38

Setze Objekt Z vor das letzte Objekt

Kette2: DruckeVorwaerts
Name X  Alter    0
Name A  Alter   40
Name Y  Alter    1
Name B  Alter   39
Name Z  Alter   12
Name C  Alter   38
Kette2: Loeschen
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

```
// kette2.cpp 03.01.00 rei
#include <stdio.h>
#include "mensch2.h"

void DruckeVorwaerts(Mensch *mensch);
void DruckeRueckWaerts(Mensch *mensch);
void LoescheKette(Mensch *Anfang);

int main(void){
   int Jahrgang = 1960;
   char Name = 'A';

   Mensch *Erster, *Letzter;
   Mensch *Kumpel;
   int Anzahl  = 0;

   fprintf(stdout, "Gib Anzahl der Menschen in der Kette an: ");
   fscanf(stdin, " %i", &Anzahl);

   fprintf(stdout, "Kette2: Initialisieren\n");
   for (int Zaehler = 1; Zaehler <= Anzahl; Zaehler++){
     Kumpel = new Mensch(Name++, Jahrgang++);
     if (Zaehler == 1) Erster = Letzter = Kumpel;
     else {// Setze Objekt an das Ende
        Kumpel->gehHinter(Letzter);
        Letzter = Kumpel;
     } //if
  } //for
  DruckeVorwaerts(Erster);
  DruckeRueckWaerts(Letzter);

  fprintf(stdout, "\nSetze Objekt X an den Anfang\n");
  Kumpel = new Mensch('X', 2000);
  Kumpel->gehVor(Erster);
  Erster = Kumpel;

  DruckeVorwaerts(Erster);
  //DruckeRueckWaerts(Letzter);

  fprintf(stdout, "\nSetze Objekt Y hinter 2.Objekt\n");
  Mensch *zweiter = Erster->Nachfolger();
  zweiter->schreibIdentitaet(2000);
  Kumpel = new Mensch('Y', 1999);
  Kumpel->gehHinter(zweiter);

  DruckeVorwaerts(Erster);
  //DruckeRueckWaerts(Letzter);

  fprintf(stdout, "\nSetze Objekt Z vor das letzte Objekt\n");
  Kumpel = new Mensch('Z', 1988);
  Kumpel->gehVor(Letzter);

  DruckeVorwaerts(Erster);

  LoescheKette(Erster);
  return 0;
} //main
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

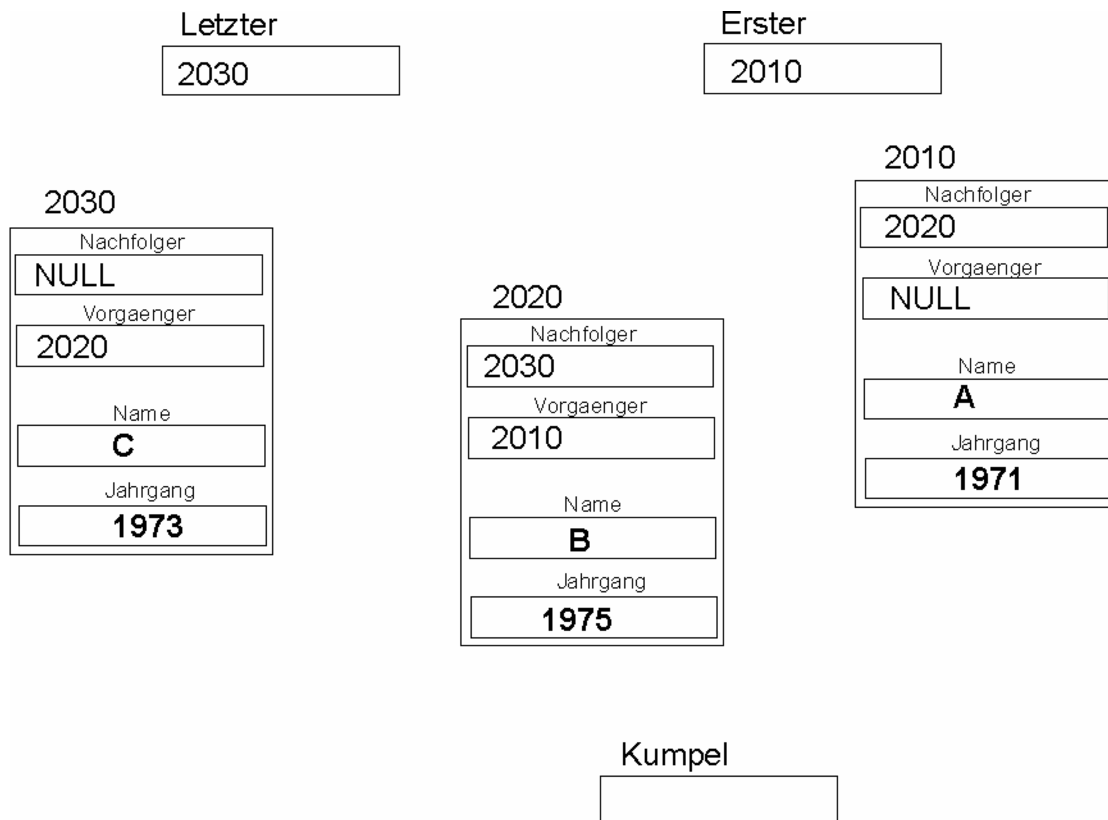
```
// Kette2; Quellcode fuer Kettenfunktionen
void DruckeVorwaerts(Mensch *mensch){
   fprintf(stdout, "\nKette2: DruckeVorwaerts\n");
   Mensch *Kumpel = mensch;
   while (Kumpel != NULL) {
      Kumpel->schreibIdentitaet(2000);
      Kumpel = Kumpel->Nachfolger();
   }
}

void DruckeRueckWaerts(Mensch *mensch){
   fprintf(stdout, "\nKette2: DruckeRueckwaerts\n");
   Mensch *Kumpel = mensch;
   while (Kumpel != NULL) {
      Kumpel->schreibIdentitaet(2000);
      Kumpel = Kumpel->Vorgaenger();
   }
}
void LoescheKette(Mensch *Anfang){
   fprintf(stdout, "Kette2: Loeschen\n");
   Mensch *Kumpel;
   while (Anfang != NULL) {
      Kumpel = Anfang;
      Anfang = Kumpel->Nachfolger();
      delete Kumpel;
   }
}
```

```
// mensch2.h  03.01.00 rei

class Mensch {
public:
   Mensch(char name, int jahr);
   ~Mensch();
   int Alter(int AktJahr);
   char Name();
   Mensch* Vorgaenger();
   Mensch* Nachfolger();
   void gehVor(Mensch* wen);
   void gehHinter(Mensch* wen);
   void schreibIdentitaet(int AktJahr);

protected:
   void setVorgaenger(Mensch* wer);
   void setNachfolger(Mensch* wer);
   int ObjJahrgang;
   char ObjName;
   Mensch* ObjNachfolger;
   Mensch* ObjVorgaenger;
};
```

```
// mensch2.cpp  03.01.00 rei

#include <stdio.h>

#include "mensch2.h"

   Mensch::Mensch(char name, int jahr){
      ObjJahrgang = jahr;
      ObjName = name;
      ObjNachfolger = NULL;
      ObjVorgaenger = NULL;
   };
   Mensch::~Mensch(){
     //fprintf(stdout,
     // "Der Mensch %c, %4i loescht sich\n", ObjName, ObjJahrgang);
   };
   Mensch* Mensch::Vorgaenger(){
      return ObjVorgaenger;
   };
   Mensch* Mensch::Nachfolger(){
      return ObjNachfolger;
   };
   void Mensch::setVorgaenger(Mensch* wer){
      ObjVorgaenger = wer;
   };
   void Mensch::setNachfolger(Mensch* wer){
      ObjNachfolger = wer;
   };
   void Mensch::gehVor(Mensch* wen){
      ObjNachfolger = wen;
      ObjVorgaenger = wen->Vorgaenger();
      wen->setVorgaenger(this);
      if (ObjVorgaenger != NULL)
         ObjVorgaenger->setNachfolger(this);
   };
   void Mensch::gehHinter(Mensch* wen){
      ObjVorgaenger = wen;
      ObjNachfolger = wen->Nachfolger();
      wen->setNachfolger(this);
      if (ObjNachfolger != NULL)
         ObjNachfolger->setVorgaenger(this);
   };
   void Mensch::schreibIdentitaet(int AktJahr){
      fprintf(stdout,
        "Name %c  Alter %4i\n", ObjName, AktJahr - ObjJahrgang);
   };
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

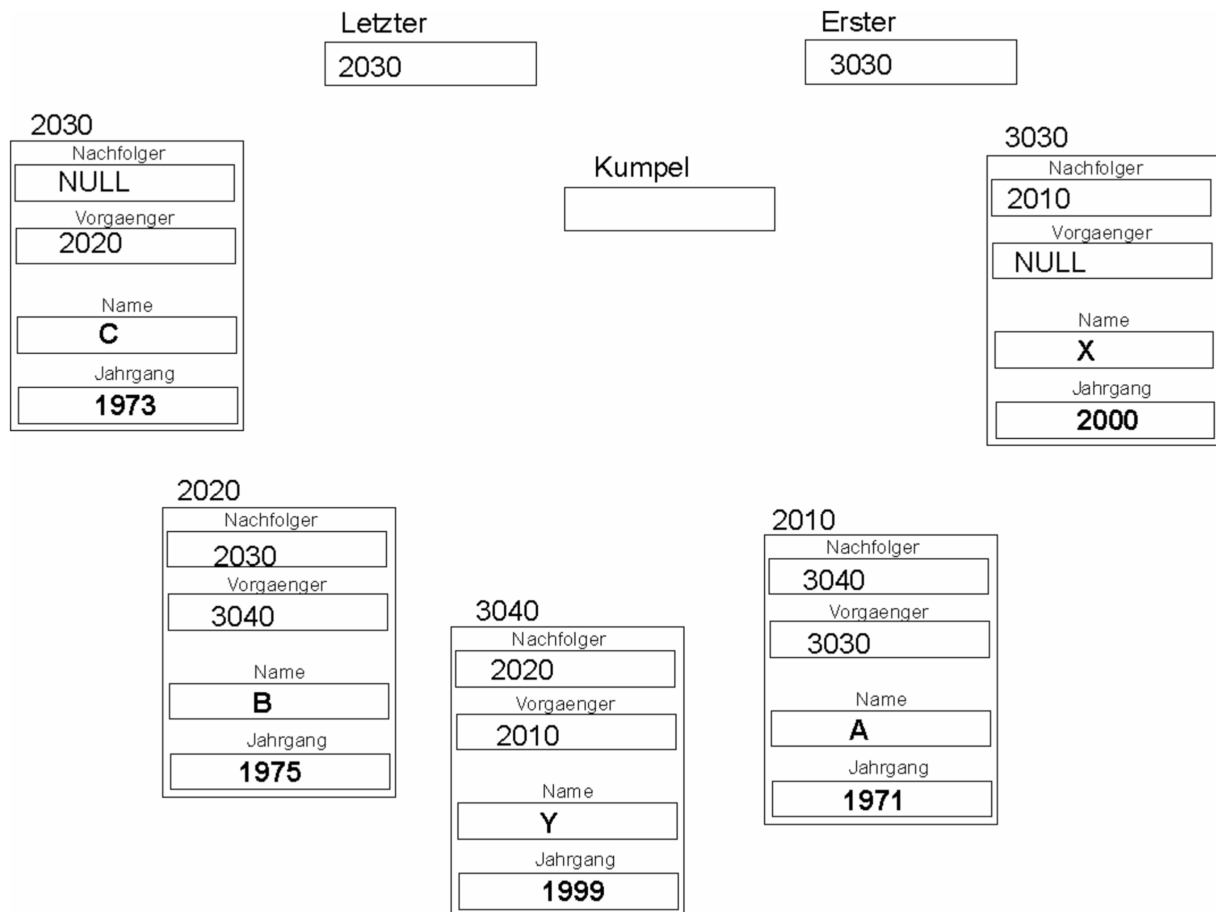
Beipiele für Ausgliedern:




```
Mensch *Kumpel;

......

Kumpel = Erster;        Kumpel = Letzter;     Kumpel = Erster->Nachf..;

// Ausgliedern ?         // Ausglieder ?       // Ausgliedern ?
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Beipiele für Suchen und Ausgliedern:




```
// Suche Kumpel mit Alter 1: ?
```

```
// Kumpel ausgliedern: ?
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

7.2 Listen

Bei der Bearbeitung von Elementen mit Vorgänger- und Nachfolgerbeziehung in einer Kette zeigt sich, dass die Operationen auf die Struktur Kette abhängig sind vom Zustand der Kette und dem Platz der Elemente in der Kette.

Unsere Operation muss unterscheiden, ob die Kette leer oder gefüllt ist, ob wir am Anfang, am Ende oder in der Kette Veränderungen vornehmen.

Für die Abbildung von Puffern mit Werkstücken soll die Struktur **Liste** mit verkettungsfähigen Elementen **Knoten** entwickelt werden.

Mit Sprachen wie FORTRAN, PASCAL und C sind diese Strukturen relativ schwer und klobig auszuführen. Es entsteht ein Mix aus datentechnischen und aufgabenbezogenen Strukturattributen. Die Entwicklung objektorientierter Sprachen wie C++ führt zu besseren Lösungen.

Für die Abbildung von Puffern oder Lager mit Werkstücken werden Datenverbunde entworfen, die Verkettungsattribute (Nachfolger, Vorgänger, Nummer ...) aus den **Knoten** und Werkstückattribute (Bearbeitungszeit ...) aus der abgeleiteten Klasse **Werkstueck** enthalten.

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Lagerverwaltung: 2 Werkstuecke im Lager, ein drittes wird eingefügt

```
// Neu->Angliedern(Alt)  ??
```

```
// ?? Ausgliedern ??
```

Ablaufprotokoll LAGERV1:

```
Knoten erzeugt
Liste erzeugt:

Lagerverwaltung:

      e - Eintragen
      a - Austragen
      d - Drucken
      x - Ende

Symbol fuer Operation : e
Bearbeitungseit fuer Werkstueck eingeben: 37.0
Knoten erzeugt
Werkstueck erzeugt:  Nr =   1, Bearbeitungszeit =  37.0

Lager enhaelt 1 objekte
 Nr =   1, Bearbeitungszeit =  37.0

Lagerverwaltung:

      e - Eintragen
      a - Austragen
      d - Drucken
      x - Ende

Symbol fuer Operation : e

Lagerverwaltung:

      e - Eintragen
      a - Austragen
      p - Drucken
      x - Ende

Symbol fuer Operation : e
Bearbeitungseit fuer Werkstueck eingeben: 61.0
Knoten erzeugt
Werkstueck erzeugt:  Nr =   2, Bearbeitungszeit =  61.0

Lager enhaelt 2 Objekte
 Nr =   1, Bearbeitungszeit =  37.0
 Nr =   2, Bearbeitungszeit =  61.0
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

```
Lagerverwaltung:

      e - Eintragen
      a - Austragen
      d - Drucken
      x - Ende

Symbol fuer Operation : e        Bild Seite 71 vor angliedern

Bearbeitungseit fuer Werkstueck eingeben: 53.1
Knoten erzeugt
Werkstueck erzeugt:  Nr =   3, Bearbeitungszeit =  53.1

Lager enhaelt 3 Objekte
 Nr =   1, Bearbeitungszeit =  37.0
 Nr =   2, Bearbeitungszeit =  61.0
 Nr =   3, Bearbeitungszeit =  53.1
                                         Bild Seite 71 nach angliedern
Lagerverwaltung:

      e - Eintragen
      a - Austragen
      d - Drucken
      x - Ende

Symbol fuer Operation : a

Werkstueck geloescht:  Nr =   1, Bearbeitungszeit =  37.0
~Knoten geloescht: Knoten::Ausgeben: Nummer = 1, Typ = 1

Lagerverwaltung:

      e - Eintragen
      a - Austragen
      d - Drucken
      i - InfoTypen

      x - Ende

Symbol fuer Operation : x

Programmende
Ende der Lagerverwaltung
~Knoten geloescht: Knoten::Ausgeben: Nummer = 2, Typ = 1
~Knoten geloescht: Knoten::Ausgeben: Nummer = 3, Typ = 1
Liste geloescht:
~Knoten geloescht: Knoten::Ausgeben: Nummer = 0, Typ = 0
```

Lagerverwaltung (V1): Hauptprogramm (main) + Dialog

```
/* lagerv1.cpp 04.01.00 rei */
#include <stdio.h>
#include "liste1.h"
#include "lagerf1.h"

int main(void){
  char Operation;
  Liste *Lager = new Liste();
  do {
    fprintf(stdout, "\nLagerverwaltung:\n");
    fprintf(stdout, "\n\t e - Eintragen \n\t a - Austragen");
    fprintf(stdout, "\n\t p - Drucken ");
  // <Uebung: weitere Funktionen eintragen>
    fprintf(stdout, "\n\t x - Ende\n");
    fprintf(stdout, "\nSymbol fuer Operation : ");
    fscanf(stdin, " %c", &Operation);
    switch (Operation){
      case 'e' : Eintragen(Lager);   break;
      case 'a' : Austragen(Lager);   break;
    // <Uebung: weitere Funktionen eintragen>
      case 'd' : Lager->ausgeben();   break;
      case 'x' : fprintf(stdout, "\nProgrammende\n");   break;
      default: break;
    }
  }while (Operation != 'x');
  fprintf(stdout, "Ende der Lagerverwaltung\n");
  delete Lager;
  return 0;
}//main
```

Lagerfunktionen: Werkstueck + E/A-Funktionen (Header)

```
/* lagerf1.h 04.01.00 rei */

class Werkstueck : public Knoten{
public:
  Werkstueck(int artNr, float zeit); //Konstruktor
  ~Werkstueck(); //Destruktor
  float Bearbeitungszeit();
  void ausgeben();
protected:
  float ObjBearbeitungszeit;
};  //Werkstueck

void Eintragen(Liste *inLager);
void Austragen(Liste *ausLager);
// <Uebung: weitere Funktionen eintragen>
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Lagerfunktionen:
Werkstueck + E/A-Funktionen(Quellcode)

```
// lagerf1.cpp  04.01.00 rei */
#include <stdio.h>
#include "liste1.h"
#include "lagerf1.h"

  int ArtikelNr = 1;

// Vereinbarung dynamischer Objekte

  Werkstueck::Werkstueck(int artNr, float zeit){ //Konstruktor
    ObjNummer = artNr;
    ObjTyp = 1;
    ObjBearbeitungszeit = zeit;
    fprintf(stdout, "Werkstueck erzeugt: ");
    ausgeben();
  }
  Werkstueck::~Werkstueck(){ //Destruktor
    fprintf(stdout, "Werkstueck geloescht: ");
    ausgeben();
  }
  float Werkstueck::Bearbeitungszeit(){
    return ObjBearbeitungszeit;
  }
  void Werkstueck::ausgeben(){
    fprintf(stdout, " Nr = %3i, Bearbeitungszeit = %5.1f\n",
                Nummer(), Bearbeitungszeit());
  };
  void Eintragen(Liste *inLager){
    Knoten *Neu;
    float zeit;
    fprintf(stdout,
       "Bearbeitungseit fuer Werkstueck eingeben:");
    fscanf(stdin, "%f", &zeit);
    Neu = new Werkstueck(ArtikelNr++, zeit);
    inLager->gliedereEnde(Neu);
    inLager->ausgeben();
  };
  void Austragen(Liste *einLager){
     Knoten *Teil;

    Teil = einLager->Erstes();
    Teil->ausgliedern();
    delete (Werkstueck *)Teil;
  };
```

Listensystem: Knoten + Liste (Header)

```
/* Liste1.h (04.01.00 rei) */

#ifndef _CLISTE_H
#define _CLISTE_H  1

class Knoten {
public:
  Knoten();
  ~Knoten();
  Knoten* Vorgaenger();
  Knoten* Nachfolger();
  void angliedern(Knoten* anwen);
  void ausgliedern();
  int istUnverkettet();
  virtual void ausgeben();
  int Nummer();
  int Typ();
protected:
  int ObjNummer;
  int ObjTyp;
  Knoten* ObjVorgaenger;
  Knoten* ObjNachfolger;
}; // Knoten

class Liste : public Knoten {
public:
  Liste();
  ~Liste();
  void gliedereAnfang(Knoten *wen);
  void gliedereEnde(Knoten *wen);
  int  istUngleichKopf(Knoten *wer);
  Knoten *Erstes();
  Knoten *Letztes();
  int istLeer();
  int Anzahl();
  void ausgeben();
}; // Class Liste

#endif /* _CLISTE_H */
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Listensystem: Knoten + Liste (Header)

```
/* Liste1.cpp (04.01.00 rei) */
#include <stdio.h>
#include "liste1.h"

  Knoten::Knoten() {
      ObjVorgaenger= this; ObjNachfolger= this;
      ObjTyp = -1;
      fprintf(stdout, "Knoten erzeugt\n");
  }
  Knoten::~Knoten() {
      if (!istUnverkettet()) ausgliedern();
      fprintf(stdout, "~Knoten geloescht: ");
      ausgeben();
  }
  Knoten *Knoten::Vorgaenger() {
       return ObjVorgaenger;
  }
  Knoten *Knoten::Nachfolger() {
       return ObjNachfolger;
  }
  void Knoten::angliedern(Knoten *anwen) {
       ObjNachfolger= anwen->ObjNachfolger;
       ObjNachfolger->ObjVorgaenger= this;
       ObjVorgaenger= anwen;
       anwen->ObjNachfolger= this;
  }
  void Knoten::ausgliedern() {
       ObjVorgaenger->ObjNachfolger = ObjNachfolger;
       ObjNachfolger->ObjVorgaenger = ObjVorgaenger;
       ObjNachfolger= this;
       ObjVorgaenger= this;
  }
  /*virtual*/ void Knoten::ausgeben() {
      fprintf(stdout,
      "Knoten::Ausgeben: Nummer = %i, Typ = %i\n",
       ObjNummer, ObjTyp);
  }
  int Knoten::istUnverkettet() {
     return ObjNachfolger == this;
  }
  int Knoten::Nummer() {
     return ObjNummer;
  }
  int Knoten::Typ() {
     return ObjTyp;
  }
// Knoten
```

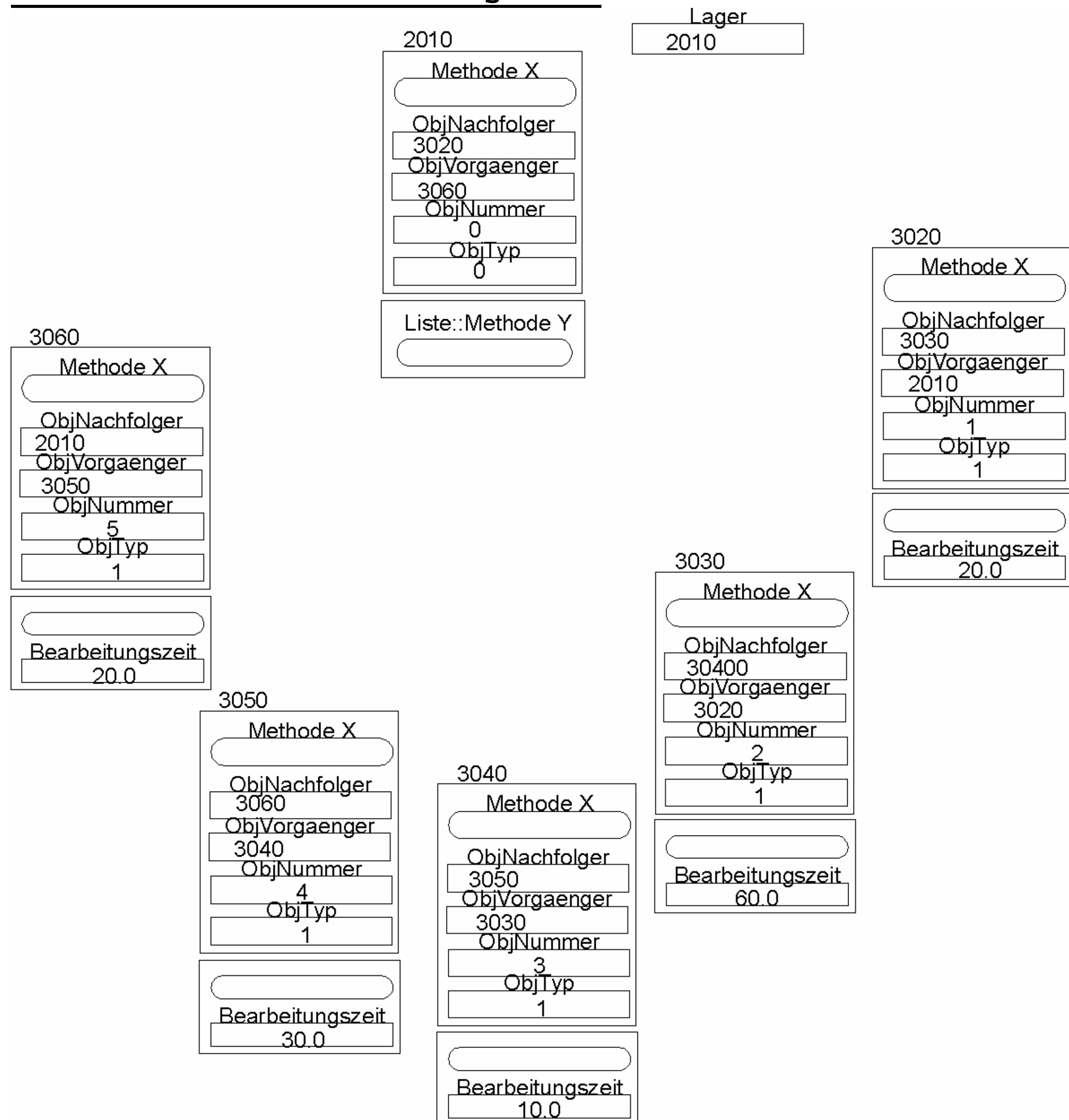
```
Liste::Liste(){
   ObjNummer = 0;             ObjTyp = 0;
   ObjVorgaenger = this;      ObjNachfolger = this;
   fprintf(stdout, "Liste erzeugt:\n");
}
Liste::~Liste() {
   Knoten *k;
   while (!istLeer()) {
    k = Erstes(); k->ausgliedern(); delete k;
   }
   fprintf(stdout, "Liste geloescht:\n");
}
void Liste::gliedereAnfang(Knoten *wen) {
   wen->angliedern(this);
}
void Liste::gliedereEnde(Knoten *wen) {
   wen->angliedern(Letztes());
}
int  Liste::istUngleichKopf(Knoten *wer) {
   return (wer != this);
}
Knoten *Liste::Erstes() {
   return ObjNachfolger;
}
Knoten *Liste::Letztes() {
   return ObjVorgaenger;
}
int Liste::istLeer() {
   return (ObjVorgaenger == this);
}
int Liste::Anzahl() {
  int zaehler = 0;
  if (!istLeer()) {
    Knoten *k = Erstes();
    while (istUngleichKopf(k)){zaehler++;k= k->Nachfolger();
    };
  }
  return zaehler;
} // anzahl
void Liste::ausgeben() {
  if (istLeer()) fprintf(stdout, "\nLager ist leer\n");
  else { fprintf(stdout,
              "\nLager enhaelt %i Objekte\n", Anzahl());
     Knoten *k = Erstes();
     while (istUngleichKopf(k)){
       k->ausgeben();  k = k->Nachfolger();
     } //while
  } // else
} // ausgeben
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

7.3 Aufgabenstellung: Lagerverwaltung

Ausgehend von dem einfachen Lagerverwaltungssystem in Kap. 7.2 sollen die Lagerfunktionen erweitert werden. Die zu erstellenden Lagerfunktionen sind im Header-File LAGERF.H dargestellt. Das Ablaufprotokoll auf der nächsten Seite ist die Grundlage für die Lösung.

Datenstruktur nach 5 Eingaben:



Neu

Ablaufprotokoll der Lagerverwaltung:

```
Lagerverwaltung (24.01.2000):
      e - neuesEintragen
      a - erstesAustragen
      d - druckeLager
      1 - druckeMinZeit
      2 - druckeMaxZeit
      3 - druckeMitZeit
      4 - druckeWerkstueckMitNummer
      5 - druckeWerkstueckMitZeit
      6 - loescheMinZeit
      7 - loescheMaxZeit
      8 - ordneLagerSteigend
      9 - ordneLagerFallend
      x - Ende

Symbol fuer Operation eingeben: e
Bearbeitungseit fuer Werkstueck eingeben:   20.00

.................
Symbol fuer Operation eingeben: e
Bearbeitungseit fuer Werkstueck eingeben:   60.00

.................
Symbol fuer Operation eingeben: e
Bearbeitungseit fuer Werkstueck eingeben:   10.00
.................
Symbol fuer Operation eingeben: e
Bearbeitungseit fuer Werkstueck eingeben:   30.00

Lager enhaelt 4 Objekte
 Nr =   1, Bearbeitungszeit =  20.0
 Nr =   2, Bearbeitungszeit =  60.0
 Nr =   3, Bearbeitungszeit =  10.0
 Nr =   4, Bearbeitungszeit =  30.0
....................
Symbol fuer Operation eingeben: e
Bearbeitungseit fuer Werkstueck eingeben:   20.00

Lager enhaelt 5 Objekte
 Nr =   1, Bearbeitungszeit =  20.0
 Nr =   2, Bearbeitungszeit =  60.0
 Nr =   3, Bearbeitungszeit =  10.0
 Nr =   4, Bearbeitungszeit =  30.0
 Nr =   5, Bearbeitungszeit =  20.0
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

```
Lagerverwaltung (24.01.2000):
         ...
         1 - druckeMinZeit
         ...
      x - Ende

Symbol fuer Operation eingeben: 1
 Nr =   3, Bearbeitungszeit =  10.0

Lagerverwaltung (24.01.2000):
         ...
         2 - druckeMaxZeit
         ...
      x - Ende

Symbol fuer Operation eingeben: 2
 Nr =   2, Bearbeitungszeit =  60.0


Lagerverwaltung (24.01.2000):
         ...
         3 - druckeMitZeit
         ...
      x - Ende

Symbol fuer Operation eingeben: 3
MitZeit im Lager betraegt: 28.00

Lagerverwaltung (24.01.2000):
         ...
         4 - druckeWerkstueckMitNummer
         ...
      x - Ende

Symbol fuer Operation eingeben: 4
Geben Sie die WerkstueckNummer ein:  2
 Nr =   2, Bearbeitungszeit =  60.0

Lagerverwaltung (24.01.2000):
         ...
         5 - druckeWerkstueckMitZeit
         ...
      x - Ende

Symbol fuer Operation eingeben: 5
Geben Sie eine Bearbeitungszeit ein:   20.00
 Nr =   1, Bearbeitungszeit =  20.0
 Nr =   5, Bearbeitungszeit =  20.0
```

```
Lagerverwaltung (24.01.2000):
         ...
         6 - loescheMinZeit
         ...
      x - Ende

Symbol fuer Operation eingeben: 6

Lagerverwaltung (24.01.2000):
         ...
         d - druckeLager
         ...
      x - Ende

Symbol fuer Operation eingeben: d

Lager enhaelt 4 Objekte
 Nr =   1, Bearbeitungszeit =  20.0
 Nr =   2, Bearbeitungszeit =  60.0
 Nr =   4, Bearbeitungszeit =  30.0
 Nr =   5, Bearbeitungszeit =  20.0

Lagerverwaltung (24.01.2000):
         ...
         7 - loescheMaxZeit
         ...
      x - Ende

Symbol fuer Operation eingeben: 7

Lagerverwaltung (24.01.2000):
         ...
         d - druckeLager
         ...
      x - Ende

Symbol fuer Operation eingeben: d

Lager enhaelt 3 Objekte
 Nr =   1, Bearbeitungszeit =  20.0
 Nr =   4, Bearbeitungszeit =  30.0
 Nr =   5, Bearbeitungszeit =  20.0
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

```
Lagerverwaltung (24.01.2000):
         ...
         8 - ordneLagerSteigend
         ...
      x - Ende

Symbol fuer Operation eingeben: 8

Lagerverwaltung (24.01.2000):
         ...
         d - druckeLager
         ...
      x - Ende

Symbol fuer Operation eingeben: d

Lager enhaelt 3 Objekte
 Nr =   1, Bearbeitungszeit =  20.0
 Nr =   5, Bearbeitungszeit =  20.0
 Nr =   4, Bearbeitungszeit =  30.0

Lagerverwaltung (24.01.2000):
         ...
      9 - ordneLagerFallend
      x - Ende

Symbol fuer Operation eingeben: 9

Lagerverwaltung (24.01.2000):
         ...
         d - druckeLager
         ...
      x - Ende

Symbol fuer Operation eingeben: d

Lager enhaelt 3 Objekte
 Nr =   4, Bearbeitungszeit =  30.0
 Nr =   1, Bearbeitungszeit =  20.0
 Nr =   5, Bearbeitungszeit =  20.0

Lagerverwaltung (24.01.2000):
         ...
      x - Ende

Symbol fuer Operation eingeben: x

Programmende
Ende der Lagerverwaltung
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04

Lagerverwaltung: main mit Dialog

```
/* lagerv2.cpp 24.01.00 Vorlage */

extern "C" {
#include <stdio.h>
}
#include "liste2.h"
#include "lagerf2.h"

int main(void){
  char Operation;
  Liste *Lager = new Liste();
  do {
    fprintf(stdout, "\nLagerverwaltung (24.01.2000):");
    fprintf(stdout, "\n\t e - neuesEintragen");
    fprintf(stdout, "\n\t a - erstesAustragen");
    fprintf(stdout, "\n\t d - druckeLager");
    fprintf(stdout, "\n\t 1 - druckeMinZeit");
    fprintf(stdout, "\n\t 2 - druckeMaxZeit");
    fprintf(stdout, "\n\t 3 - druckeMitZeit");
    fprintf(stdout, "\n\t 4 - druckeWerkstueckMitNummer");
    fprintf(stdout, "\n\t 5 - druckeWerkstueckMitZeit");
    fprintf(stdout, "\n\t 6 - loescheMinZeit");
    fprintf(stdout, "\n\t 7 - loescheMaxZeit");
    fprintf(stdout, "\n\t 8 - ordneLagerSteigend");
    fprintf(stdout, "\n\t 9 - ordneLagerFallend");
    fprintf(stdout, "\n\t x - Ende\n");
    fprintf(stdout, "\nSymbol fuer Operation eingeben: ");
    fscanf(stdin, " %c", &Operation);
    fprintf(stdout, "%c\n", Operation);
    switch (Operation){
      case 'e' : Eintragen(Lager);   break;
      case 'a' : if (Lager->istLeer ())
                    fprintf (stdout, "Lager ist leer!\n");
                 else Austragen(Lager);
                 break;
      case 'd' : Lager->ausgeben();   break;
      case '1' : if (Lager->istLeer ())
                    fprintf (stdout, "Lager ist leer!\n");
                 else sucheMinZeit (Lager)->ausgeben ();
                 break;
      ..................................................
      ..................................................

      case '9' : Lager = ordneFallend  (Lager);  break;
      case 'x' : fprintf(stdout, "\nProgrammende\n");   break;
      default: break;
    }
  }while (Operation != 'x');
  fprintf(stdout, "Ende der Lagerverwaltung\n");
  delete Lager;
  return 0;
}//main
```

Header-File für die Lagerverwaltung:

```
/* lagerf2.h 07.02.02 rei */

class Werkstueck : public Knoten {
public:
  Werkstueck (int artNr, float zeit); //Konstruktor
  ~Werkstueck(); //Destruktor
  float Bearbeitungszeit();
  void ausgeben();
protected:
  float ObjBearbeitungszeit;
};  //Werkstueck

void Eintragen(Liste *inLager);
void Austragen(Liste *ausLager);
// <Uebung: weitere Funktionen eintragen>
Knoten *zeigMinWerkstueck(Liste *inLager);
Knoten *zeigMaxWerkstueck(Liste *inLager);
float gibMitBearbeitungszeit(Liste *inLager);
Knoten *zeigWerkstueckMitNummer(Liste *wsLager, int Nr);
Knoten *zeigWerkstueckMitZeit(Liste *wsLager, float Zeit);
Liste *ordneSteigend(Liste *inLager);
Liste *ordneFallend(Liste *inLager);
```

Beispiele für implementierte Methoden

```
Knoten *zeigMinWerkstueck(Liste *inLager) {
   Knoten *such = inLager->Erstes();
   Knoten *min = such;
   while (inLager->istUngleichKopf(such)) {
     if (((Werkstueck *)such)->Bearbeitungszeit()
         < ((Werkstueck *)min)->Bearbeitungszeit())
        min = such;
     such = such->Nachfolger();
   }
   return min;
}
Knoten *zeigMaxWerkstueck(Liste *inLager) {
   Knoten *such = inLager->Erstes();
   Knoten *max = such;
   while (inLager->istUngleichKopf(such)) {
     if (((Werkstueck *)such)->Bearbeitungszeit()
         > ((Werkstueck *)max)->Bearbeitungszeit())
        max = such;
     such = such->Nachfolger();
   }
   return max;
}
```

8 Anhang

8.1 Protokoll einer Tageskasse

```
GST WS 03/04, Aufgabe: tagesk.c (08.10.03 rei)

Aepfel-,Birnen- und Zwetschgenpreise eingeben:
1.0 2.0 3.0

Das Obst kostet  6.00 EURO

Weitere Geschaefte [j/n]:
j

Aepfel-,Birnen- und Zwetschgenpreise eingeben:
9.0 5.0 3.0

Das Obst kostet 17.00 EURO

Weitere Geschaefte [j/n]:
j

Aepfel-,Birnen- und Zwetschgenpreise eingeben:
2.0 1.0 3.0

Das Obst kostet  6.00 EURO

Weitere Geschaefte [j/n]:
n
Anzahl der Werte =      3
Mittelwert       =   9.67
Maximaler Wert   =  17.00
Minimaler Wert   =   6.00

=== Das waren      3  Einkaeufe ===
SummeObstpreis         = EURO   29.00
SummeAepfelpreis       = EURO   12.00
SummeBirnenpreis       = EURO    8.00
SummeZwetschgenpreis   = EURO    9.00

AnteilAepfelpreis      =    %   41.38
AnteilBirnenpreis      =    %   27.59
AnteilZwetschgenpreis  =    %   31.03

Mittlere Einkaufsumme  = EURO    9.67, in DM   18.85

1.EinKauf MinObstpreis = EURO    6.00
2.EinKauf MaxObstpreis = EURO   17.00
```

8.2 Kartoffelsortierer

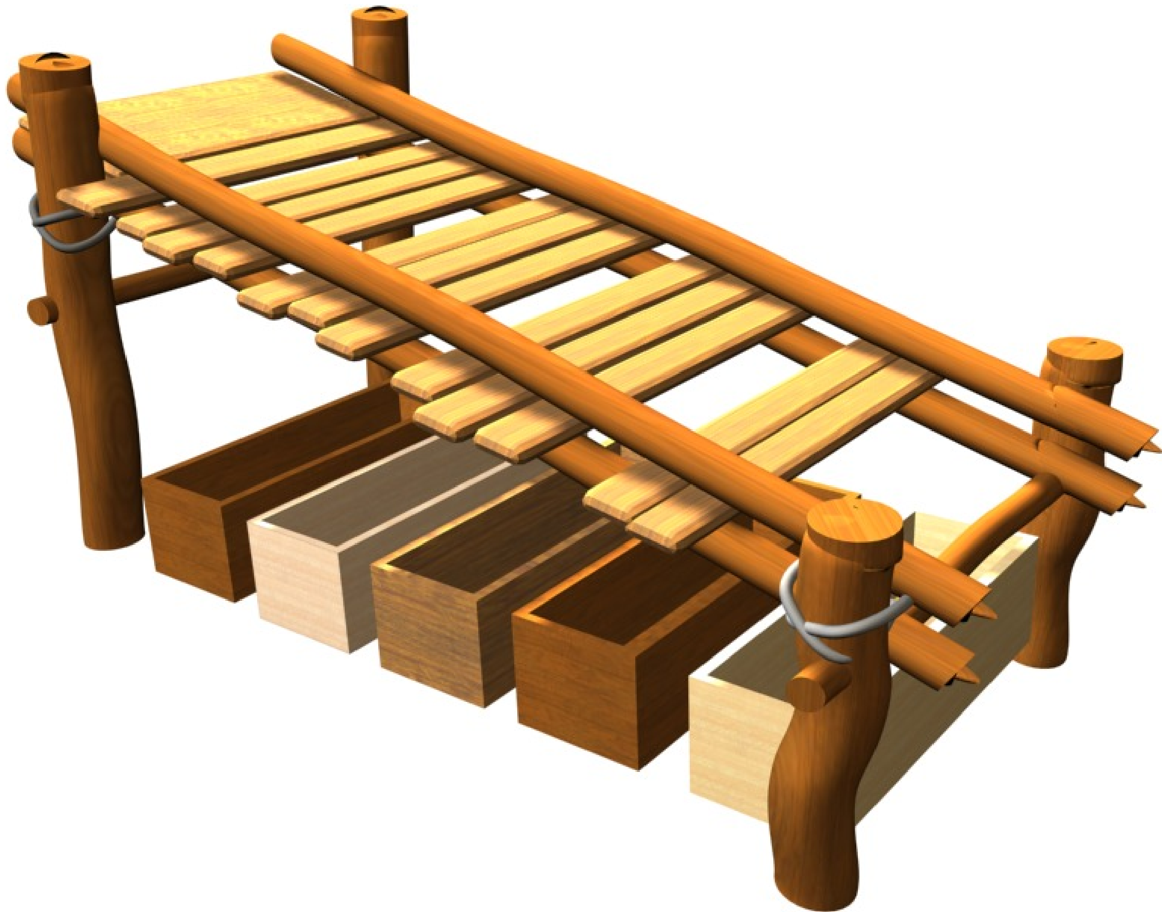


Abbildung : Feuerstein's Kartoffelsortierer (vgl. Kap. 5)

Eigenschaften:

```
Kartoffel    -> Durchmesser

Sortierer    -> Schlitzbreite, Inhalt, Nr von n Kisten
```

Verfahren:

```
setze Nr auf 0;
solange  (Durchmesser  > Schlitzbreite der Kiste[Nr])
                                  inkrementiere Nr;
inkrementiere den Inhalt der Kiste[Nr];
```

© A. Reinhardt, fps/ipl/mb/Uni Kassel, WS 2003/04