

Inhaltsverzeichnis

Kapitel 1 – Einführung	3
Spezifikation.....	3
Algorithmus.....	3
Kapitel 2 – Graphische Darstellungsmittel	4
Struktogramme.....	4
Datenflussdiagramme.....	4
Programmablaufpläne.....	4
Kapitel 3 – Die Entwicklungsumgebung	4
Projekte.....	4
Kapitel 4 – Erste Schritte	6
Kapitel 5 – Variablen, Konstanten	7
Kapitel 6 – Numerische Datentypen	8
Ganze Zahlen.....	8
Zahlensysteme.....	8
Operatoren.....	9
Punktzahlen.....	9
Zeichen.....	11
Typumwandlung.....	11
Kapitel 7 – Eingabe – Ausgabe	12
Ausgabefunktionen.....	12
Eingabefunktionen.....	13
Kapitel 8 – Ausdrücke	13
Kapitel 9 – Selektionen	16
Die if-Anweisung.....	16
Die switch-Anweisung.....	17
Kapitel 10 – Iterationen	17
Die for-Anweisung.....	18
Die while-Anweisung.....	18
Die do - while - Anweisung.....	19
Die break-Anweisung.....	19
Die continue-Anweisung.....	19
Schachteln von Schleifen.....	19
Kapitel 11 – Funktionen	19
Definition einer Funktion.....	19
Deklaration einer Funktion.....	20
Unterschied zwischen Deklaration und Definition.....	21
Kapitel 12 – Speicherklassen	21
Lokale Variablen.....	22
Globale Variablen.....	22
Kapitel 13 – Felder	23
Eindimensionale Felder.....	23
Mehrdimensionale Felder.....	24
Initialisieren von Feldern.....	24
Konstante Felder.....	24
Felder als Parameter.....	24
Einfache Sortierverfahren.....	25
Einfache Suchverfahren.....	25
Kapitel 14 – Zeiger	26
Zeiger als Parameter:.....	27
Die „Dualität“ von Zeigern und Feldern.....	27
Zeigerarithmetik.....	27
Konstante Zeiger.....	28
Zeiger auf Funktionen.....	28
Kapitel 15 – Zeichenketten	29
Elementare Funktionen für Zeichenketten:.....	30

Kapitel 16 – Abgeleitete Datentypen.....	31
Strukturen.....	31
Bitfelder.....	33
Aufzählungen.....	33
Variante Strukturen.....	34
Rekursive Strukturen.....	34
Deklaration von abgeleiteten Datentypen.....	35
Typdefinition mit typedef.....	35
Kapitel 17 – Dateien.....	35
Datenströme.....	35
Öffnen und Schließen von Datenströmen.....	36
Ein- und Ausgabe.....	37

Kapitel 1 – Einführung

Spezifikation

genaue Definition der Software, jedoch keine detaillierten Realisierungsvorschläge; vielmehr wird die Leistung des zu entwickelnden Programmes genau festgelegt
auch diverse Festlegungen, wie Art der Fehlerbehandlung, Testmechanismen, Möglichkeiten der Wartung oder Benutzerführung

- Vorbedingung: Voraussetzung
- Nachbedingung: Leistungen

Methoden:

- Top-Down-Methode (Gesamtspezifikation -> Teilprobleme)
- Bottom-Up-Methode (Teilprobleme -> Gesamtspezifikation)
- Middle-Out-Methode

Algorithmus

„Rechenvorschrift“ - von persischem Mathematiker
Der Algorithmus ist als Oberbegriff zu „Programm“ zu betrachten, bei dem im Gegensatz zu einem Programm die strengen syntaktischen Regeln der Programmiersprache nicht eingehalten zu werden brauchen.

Grundlegende Eigenschaften:

- Das Verfahren ist durch einen endlichen Text beschreibbar.
- Es läuft in einzelnen, wohldefinierten Rechenschritten ab, die in einer eindeutigen Reihenfolge durchzuführen sind.
- Die Rechnung besitzt gewisse Parameter (Eingabegrößen) und wird für jede Eingabe nach endlich vielen Rechenschritten abbrechen und ein eindeutig bestimmtes Ergebnis liefern.

Bedingungen:

- Finitheit der Beschreibung
- Effektivität
- Terminiertheit
- Determiniertheit

Verifikation von Algorithmen:

Durch Testen kann man nur die Anwesenheit von Fehlern nachweisen, nicht aber deren Abwesenheit.

Kapitel 2 – Graphische Darstellungsmittel

Struktogramme

- Darstellung von sequenziellen Abläufen
- Eigene Symbole für Grundelemente wie Blöcke, Selektionen, Schleifen oder Funktionsaufrufe.
- Werden von oben nach unten gelesen

Datenflussdiagramme

- Hilfsmittel zur Modellierung von Datenflüssen
- geben eine Übersicht über Funktionen oder Prozesse

Elemente:

- Funktion (Prozess)
- Übergang
- Schnittstelle (Datenquellen und Datensenken)
- Speicher

Produzent: Funktion, die Daten erzeugt

Konsument: Funktion, die Daten erhält

Programmablaufpläne

Beschreiben den Ablauf eines Programmes mit Hilfe von definierten Symbolen

Nachteile: keine Symbole für elementare Befehle wie z.B. Mehrfachauswahl

Kapitel 3 – Die Entwicklungsumgebung

Umfasst alle notwendigen Werkzeuge und Methoden um Software zu planen, entwickeln und verwalten zu können.

Mindestausstattung:

- Projektverwaltung
- Programmierumgebung (Editor)
- Übersetzer
- Testumgebung

Projekte

Vorgänge wie Planung, Verwaltungsaufwand, Dokumentation, Versionskontrolle, Codierungs- und Testaufwand um ein spezifiziertes Software-Produkt herzustellen.

Ein Projekt besteht (vereinfacht) aus:

- Programmdateien

- externen Modulen
- Bibliotheken

Arten von Dateien:

- Quelltext-Dateien („Modul“)
- Header-Dateien (Deklarationen)

Makefiles:

Übersetzungsabfolge wird beschrieben; Übersetzungszeigen werden minimiert, da nur noch die veränderten Module neu übersetzt werden

Compiler:

- zur Übersetzung des Programmtextes („Quellcode“) wird ein Compiler verwendet.
- Das gesamte Programm wird in Maschinensprache übersetzt.
- Anschließend liegt das Programm als ausführbares Maschinenprogramm vor.

Interpreter:

- benötigt Quelltextdateien bei der Ausführung
- Programmtext wird oft als Bytecode vorkompiliert, wodurch nur noch Codes statt Text interpretiert werden müssen -> schnellere Ausführungszeiten

Übersetzungsvorgang:

- Präprozessor (simpler Textersetzer)
- Syntaxüberprüfung
- Compiler (C-Befehle -> Assemblersprache)
- eventuell Optimierung (nach Laufzeit bzw. Codegröße)
- Assembler (Assemblersprache -> Maschinensprache)

Linker:

fügt alle Objektdateien und Bibliotheken (enthalten Ansammlungen von Funktionen) zu einem ausführbaren Programm zusammen.

Compiler-Optionen:

- Optimierung (-O)
- Debugger (-g)
- Bibliotheken linken (-l)
 - Um Bibliotheken verwenden zu können, muss die Header-Datei derselbigen in den Quelltext eingebunden werden (Bibliotheken und deren Header-Dateien haben meist nicht den selben Namen)

Der Präprozessor:

- #include-Anweisung
 - dient dazu, Header-Dateien in den Programmtext einzufügen
- #define-Anweisung
 - Definition von Präprozessorkonstanten
- Makros

- Bsp.: #define ERRLOG(text) fprintf(stderr, text)
- keine Typüberprüfungen der Parameter
- keine Beachtung von Prioritäten
- keine Syntaxüberprüfung
- Vordefinierte Präprozessorkonstanten
 - `__FILE__`
 - `__LINE__`
- #if-Anweisung
 - Debug-Anweisungen
 - Demo-Versionen
- #ifdef-Anweisung / #ifndef-Anweisung
 - Programmtexte werden ein- und ausgeblendet, je nachdem ob die angegebene Präprozessorkonstante bereits definiert wurde oder nicht.

Kapitel 4 – Erste Schritte

- C wurde im Jahr 1972 von Dennis M. Ritchie entwickelt.
- C ist eine syntaktisch „kleine“, jedoch semantisch „große“ Programmiersprache.
- C ist eine Hochsprache, ermöglicht aber maschinennahe Programmierung.

Syntax:

beschreibt die exakte Form, in der Befehle anzugeben sind.

Semantik:

Aufbauend auf der Syntax umfasst die Semantik die Bedeutung der Zusammenhänge von Wörtern in einem Text.

Die **Semantik** (Bedeutungslehre) ist das Teilgebiet der [Sprachwissenschaft](#) (Linguistik), das sich mit **Sinn** und **Bedeutung** von [Sprache](#) beziehungsweise sprachlichen [Zeichen](#) befasst. Die Semantik kümmert sich um die Frage, wie Sinn und Bedeutung von komplexen Begriffen aus denen von einfachen Begriffen abgeleitet werden können und stützt sich dabei in der Regel auf die [Syntax](#). Hierbei bezeichnet nach [Gottlob Frege](#)

- **Sinn** (engl. `sense') den Inhalt, der sich aus den Relationen der Zeichen, Wörter, Sätze usw. untereinander im System der Sprache ergibt,
- **Bedeutung** (engl. `reference') den Inhalt, der sich aus der Relation zwischen Zeichen und Welt ergibt.

Warum C?

- C ist eine mächtige und flexible Programmiersprache
- C ist weit verbreitet
- C ist eine portable Sprache
- C ist eine Sprache mit wenig „Worten“, sogenannten Schlüsselwörtern
- C unterstützt modulares Programmieren

Programmtext-Dateien:

- Programmtext („.c“)
- Header-Dateien („.h“)
 - enthalten Informationen, die in mehreren C-Dateien gebraucht werden
 - Definition von Präprozessorkonstanten
 - Typ- und Strukturdefinitionen
 - Funktionsdeklarationen
 - Header-Dateien dürfen jedoch keine ausführbaren Programmtexte (Definitionen von Funktionen) enthalten!

Kommentare:

werden vom Präprozessor vor dem eigentlichen Übersetzen durch den Compiler entfernt.

Kapitel 5 – Variablen, Konstanten

Eigenschaften von Variablen und Konstanten:

- Name
- Ort
- Datentyp
- Wert
- Gültigkeitszeitraum
- Sichtbarkeitsbereich

C bietet keine Vergabe von Einschränkungen an Zugriffsrechten auf Variablen an. Moderne Programmiersprachen (wie C++ oder Java) haben aus diesem Grund das Programmierparadigma der Datenkapselung eingeführt, das die Zugriffsrechte regelt.

Definition von Variablen:

Um eine Variable verwenden zu können, muss sie definiert oder deklariert werden. Dabei wird ihr Datentyp, ihr Name, sowie ihre Speicherklasse festgelegt. Der Unterschied der Definition gegenüber der Deklaration ist, dass Speicher für die Variable erzeugt wird.

Namen von Variablen:

Namen von Variablen dürfen nur mit einem Buchstaben oder dem Unterstrich '_' beginnen und können ab dem zweiten Zeichen auch Ziffern enthalten. Variablenamen sind case sensitive.

Initialisierung von Variablen:

Unter dem Initialisieren einer Variablen versteht man das Setzen eines Wertes gleich bei der Definition, also beim Anlegen der Variablen. Nicht initialisierte Variablen haben „irgendeinen“ Wert.

Konstanten:

Konstanten müssen initialisiert werden.

Kapitel 6 – Numerische Datentypen

Ein Datentyp ist eine Definitionsmenge von Daten inklusive aller Operatoren und Funktionen, die auf dieser Menge definiert sind.

Qualifizierer:

werden dem Datentyp vorangestellt und können seinen Wertebereich verändern.

Speicherverbrauch einzelner Datentypen:

<i>Datentyp</i>	<i>Bits</i>
short int	16
long int	32
float	32
double	64
long double	≥ 64
char	8
Zeiger	32 (32-Bit Rechnerarchitektur) 64 (64-Bit Rechnerarchitektur)

Literale:

Ein Literal, häufig auch Konstante genannt, ist eine Zeichenfolge mit der Werte formuliert werden.

Ganze Zahlen

- Qualifizierer:
 - signed (wird angenommen wenn nichts angegeben)
 - unsigned
 - short
 - long
- Literale:
 - 1234
 - -4

Zahlensysteme

In C ist der Datentyp *int* meist als Binärcode im Zweierkomplement implementiert. Dabei wird eine Zahl in einem modifizierten 2er-Zahlensystem – dem binären Zahlensystem – dargestellt.

Grundlegende Eigenschaften von Zahlensystemen:

- Das n -Zahlensystem hat n Symbole zur Darstellung von Zahlen
- Mit m Stellen lassen sich n^m Zahlen darstellen, von 0 bis $n^m - 1$
- Reichen die vorhandenen Stellen zur Darstellung der Zahl nicht aus, so kann die Zahl nicht dargestellt werden. Es sind weitere Stellen notwendig.

Zweierkomplement:

- Darstellung negativer Zahlen:
 - Ist x die darzustellende negative Zahl, so wird sie als Zweierkomplement von $-x - 1$ codiert, damit es nicht 2 Formen von '0' gibt.
 - Alle Bits werden invertiert!
 - Bei negativen Zahlen ist das MSB (most significant bit) gesetzt.
 - Es existiert immer eine negative Zahl mehr als es positive gibt.

Zahlensysteme in C:

In C können Zahlen im

- Dezimal-,
 - Oktal- und
 - Hexadezimalsystem
- dargestellt werden.

Umwandlungen:

- Umwandlung von Binär-, Oktal- und Hexadezimalzahlen
- Umwandlung einer Binärzahl ins Dezimalsystem
- Umwandlung einer Dezimalzahl ins Binärsystem
 - Man dividiere die Zahl d durch 2 und schreibe den Rest der Division an. Dieser Vorgang wird solange wiederholt, bis d Null ist, wobei die Restbeträge von rechts nach links (!) angeschrieben werden.

Operatoren

- $+$, $-$, $*$, $/$, $\%$
- Das Ergebnis einer Ganzzahldivision ist eine ganze Zahl!
- Der Modulo-Operator ' $\%$ ' berechnet den Rest der Ganzzahldivision (funktioniert auch für negative Zahlen)
 - $5 \% 2$ liefert 1
 - $-5 \% 2$ liefert -1
 - $5 \% -2$ liefert 1
 - $-5 \% -2$ liefert -1

Punktzahlen

Zur Darstellung einer Punktzahl werden entweder Festpunkt- oder Gleitpunkt-Zahlensysteme verwendet.

mb^e

m ... Mantisse

b ... Basis

e ... Exponent (Festpunkt-Zahlensystem -> e fix vorgegeben)

Punktzahlen in C sind im Gleitpunkt-Zahlensystem mit der Basis 2 implementiert. Bei der Codierung von Gleitpunkt-Zahlen werden, da die Basis 2 implizit verwendet ist, die Mantisse, der Exponent und das Vorzeichen gespeichert.

2 normierte Grundformate:

- Einfach langes Format:
 - v ... 1 Bit
 - e ... 8 Bit
 - m ... 23 Bit
- Doppelt langes Format
 - v ... 1 Bit
 - e ... 11 Bit
 - m ... 52 Bit

Konventionen des IEEE Gleitpunkt-Zahlensystems:

- Die Mantisse m ist normalisiert bzw. für sehr kleine Zahlen denormalisiert. Die original Mantisse m wird solange „nach links bzw. rechts geschoben“ bis die Zahl '1' hinter dem Punkt steht.
- Das erste Bit der Mantisse wird weggelassen, da die '1' hinter dem Punkt implizit ist. Man spart dadurch ein Bit.
- Weiters sind spezielle Bit-Kombinationen für die Darstellung von $+\infty$, $-\infty$, 0 oder NaN definiert.

Literale:

- double
 - 46.5656L
 - 3.1415
 - 0.147 oder .147 oder 147e-3 oder 147E-3
- float
 - 46.5656F
 - 3.1415F
 - 0.147F oder .147F oder 147e-3F oder 147E-3F

Operatoren:

- +, -, *, /
- Es existiert kein Modulo-Operator '%'
- Der Divisionsoperator '/' liefert eine reelle Zahl.

Mathematische Funktionen:

- Die Genauigkeit der Funktionswerte ist, da es sich um numerische Verfahren handelt, fehlerbehaftet.
- Um die Funktionen verwenden zu können, muss die Header-Datei *math.h*

inkludiert und die Mathematik-Bibliothek *m* zum Programm hinzugelinkt werden (Linker-Option *-lm*)

Zeichen

Zeichen existieren in C nur indirekt. Soll ein Zeichen gespeichert werden, wird sein ASCII-Code abgespeichert – eine Zahl, die dem Zeichen entspricht. Ein `char` ist 8 Bit lang und kann die Zahlen im Wertebereich `-128 ... 127` annehmen.

Literale:

- Zeichen können mit den einfachen Hochkommata angegeben werden
- Escape-Sequenzen können ebenfalls angegeben werden.

Der ASCII-Code:

- Der ASCII-Code legt die Codierung im Wertebereich `0 ... 127` (7 Bits-Code) fest.
- Die unteren 32 Codes sind Steuercodes.
- Die oberen 96 Codes dienen der Darstellung von Buchstaben, Ziffern und diversen Zeichen.
- Die im Deutschen verwendeten Umlaute sowie einige Buchstaben mancher anderer Sprachen sind im ASCII-Code nicht genormt, werden aber im Wertebereich `128 ... 255` dargestellt (erweiterter ASCII-Code).

Funktionen für Zeichen:

Die C-Standard-Bibliothek bietet einige Funktionen für Zeichen. Um diese Funktionen verwenden zu können, muss die Header-Datei *ctype.h* inkludiert werden.

Wahrheitswerte:

Wahrheitswerte sind in C nicht typorientiert implementiert. (Es wird meist *int* verwendet)

void:

Der Typ *void* bedeutet in C soviel wie „nichts“ oder „unbekannt“.

Typumwandlung

Implizite Typumwandlung:

- Findet statt, wenn ein Wert eines eingebauten Datentyps in einen anderen eingebauten Datentyp umgewandelt werden soll. Diese Typumwandlung wird zur Zeit der Übersetzung festgestellt.
- Es kann dabei auch ein Datenverlust auftreten.

Explizite Typumwandlung:

- Eine explizite Typumwandlung findet dann statt, wenn explizit ein Zieldatentyp angegeben wird.
- Dabei ist ebenfalls zwischen werterhaltender und verlustbehafteter

Umwandlung zu unterscheiden.

- Bei einer Umwandlung einer Gleitpunkt-Zahl in eine ganze Zahl kommt es zu einem Datenverlust – die Nachkommastellen werden abgetrennt. Wird eine sehr hohe Zahl ($> 2^{31} - 1$) konvertiert, kann die Zahl in einem *long* nicht dargestellt werden.

sizeof:

Mit Hilfe der Funktion *sizeof()* kann festgestellt werden, wieviele Bytes ein Datentyp oder auch eine Variable oder Konstante verbraucht. Die Anzahl wird zur Übersetzungszeit ermittelt.

sizeof ist streng genommen ein Operator, der zur Übersetzungszeit durch den Compiler ausgewertet wird, hat jedoch in seiner Verwendung die Eigenschaften einer Funktion.

Kapitel 7 – Eingabe – Ausgabe

Ein- und Ausgabefunktionen der Standard-Bibliothek (*stdio.h* muss geladen werden).

Ausgabefunktionen

printf:

- Escape-Sequenzen
 - `\n` ... Neue Zeile
 - `\t` ... Tabulator
 - `\\` ... backslash
 - `\"` ... Anführungszeichen
 - `\b` ... Klingelton
- Platzhalter für ganze Zahlen
 - `%ld` ... long
 - `%lo` ... Oktalsystem
 - `%lx` ... Hexadezimalsystem
 - `%d` ... int
 - `%hd` ... short int
 - `%u` ... unsigned int
- Platzhalter für Gleitpunkt-Zahlen
 - `%e` ... rundet das Argument ... `[-]d.ddde±dd`
 - `%f` ... rundet das Argument ... `[-]ddd.ddd`
 - `%g` ... schaltet zwischen `%e` und `%f` hin und her (je nach die wie groß das Argument ist)
- Platzhalter für Zeiger
 - `%p`
- Platzhalter für Zeichen
 - `%c`
- Platzhalter für Zeichenketten
 - `%s`

Weitere Formatierungsmöglichkeiten

Will man die Ausgabebreite verändern, so muss das Zeichen '*' verwendet werden.

Eingabefunktionen

scanf:

Die Funktion gibt die Anzahl der gelesenen Werte zurück oder die Konstante EOF, wenn ein Fehler aufgetreten ist, oder die Eingabe beendet wurde.

- Platzhalter für ganze Zahlen
 - %li ... gestattet Eingabe einer Zahl in verschiedenen Zahlensystemen (beginnt die Zahl mit 0, so wird das Oktalsystem, beginnt sie mit 0x, das Hexadezimalsystem, ansonsten das Dezimalsystem verwendet.
- Platzhalter für Gleitpunkt-Zahlen
 - %f ... float
 - %lf ... double
- Platzhalter für Zeichen
 - %c
- Platzhalter für Zeichenketten
 - %s (Der Text wird jedoch nur bis zum ersten Whitespace (Leerzeichen, Tabulator oder neue Zeile) gelesen.

Probleme mit *scanf*:

Das Lesen einer Variablen mit *scanf* passiert in 3 Schritten:

1. Es wird solange Text (!) von der Tastatur gelesen, bis der Benutzer die Eingabetaste betätigt.
2. Die Eingabe wird in den gewünschten Datentyp konvertiert. Falls das Konvertieren fehlschlägt, bleibt der Eingabepuffer unverändert (!), *scanf* bricht ab und die Variable bleibt ebenfalls unverändert.
3. Der Wert wird in den Speicher geschrieben. Stimmt der gelesene Datentyp, der im Platzhalter angegeben ist, nicht mit dem Datentyp der Variable, in die gespeichert werden soll, überein, so können schwerwiegende Programmfehler auftreten.

Ein- und Ausgabe von Zeichen:

getchar:

liefert 1 Zeichen als int-Wert zurück oder EOF wenn ein Fehler aufgetreten ist.

putchar:

gibt 1 Zeichen am Bildschirm aus; erwartet ein Zeichen als Argument und gibt EOF im Fehlerfall zurück

Kapitel 8 – Ausdrücke

Befehle werden in C in sogenannten Ausdrücken (expressions) angeschrieben. C

kennt eine Reihe von Operatoren, die in Ausdrücken verwendet werden. Manche von ihnen (z.B. Vergleichsoperatoren oder logische Operatoren) werden fast ausschließlich in Selektionen verwendet, andere (z.B. der Komma-Operator) kommen hauptsächlich in Iterationen zum Einsatz.

Allgemeines:

Die einfachste Anweisung in C ist der Strichpunkt ';' für sich alleine. Er bedeutet: „Tue nichts.“ Genau genommen ist der Strichpunkt jedoch keine Anweisung. Er dient vielmehr als Ende-Markierung, beispielsweise von Ausdrücken.

Ausdrücke bestehen aus Literalen, Konstanten, Variablen und Operatoren. Ausdrücke werden ausgewertet. Dabei geht C in einer genau definierten Reihenfolge vor. Der einfachste Ausdruck ist ein Literal, eine Konstante oder eine Variable. Wird der Ausdruck ausgewertet, ist das Ergebnis des Ausdruckes das Literal selbst, der Wert der Konstanten oder der Wert der Variablen.

Reihenfolge der Auswertung:

Zuerst wird der Operator mit der höchsten Priorität ausgeführt, dann der Operator mit der nächst niedrigen, usw. Haben 2 mögliche Operatoren die selbe Priorität, so entscheidet die Assoziativität.

Ein sogenannter Lvalue (Links-Wert) ist eine besondere Art eines Ausdruckes, der für eine Speicherzelle steht. Der einfachste Lvalue ist eine Variable.

Die runde Klammer () ist kein Operator. Sie dient nur der Zusammenfassung von Ausdrücken.

Assoziativitäten:

Assoziativitäten beschreiben die Bindungen zwischen gleichwertigen Operatoren.

- Alle binären (z.B. + oder * ... haben genau 2 Operanden) Operatoren mit Ausnahme der Zuweisungsoperatoren sind links bindend.
- Alle unären (z.B. ! ... haben genau 1 Operanden) sind rechts bindend.

Bei der Auswertung von Ausdrücken entstehen sogenannte temporäre Werte. Sie benötigen natürlich auch Speicherplatz, welcher im Zuge der Auswertung automatisch erzeugt und freigegeben wird.

Die Reihenfolge der Auswertung von Operanden ist im ANSI-Standard nicht festgelegt (z.B. a() + b() oder (1 * 2) + (3 * 4) + (5 * 6)).

Operatoren:

Alle Operatoren haben etwas gemein: Sie haben immer einen Rückgabewert.

- Arithmetische Operatoren
- Die Zuweisung

- Auf der linken Seite der Zuweisung steht prinzipiell eine Variable, der sogenannte Lvalue, dem das Ergebnis des Ausdruckes auf der rechten Seite zugewiesen wird. Das Ergebnis einer Zuweisung ist ein Wert, mit dem weitergerechnet werden kann.
- Inkrement und Dekrement
 - wert = wert + 1;
 - wert = wert - 1;
 - wert++;
 - wert--;
 - Diese Ausdrücke werden heutzutage völlig gleich übersetzt.
 - Der Operator ++ bedeutet genau genommen „Das nächste“. Wird er auf eine ganze Zahl angewendet, so wird die Variable auf den nächsten Wert gesetzt. Ist die Variable vom Typ double, so wird sie um 1.0 erhöht, ist sie ein Zeiger, so kommt die Zeigerarithmetik zum Einsatz.
 - Steht das ++ bzw. -- vor dem Operanden, so nennt man die Anweisung Präinkrement bzw. Prädekrement.
 - Steht das ++ bzw. -- nach dem Operanden, so nennt man die Anweisung Postinkrement bzw. Postdekrement
 - Sowohl das Präinkrement als auch das Prädekrement können nur auf Variablen und nicht auf Ausdrücke angewendet werden.
- Abkürzungen
 - x1 *= 10;
 - x2 += 10;
 - Derartige Ausdrücke werden in C gerne verwendet, da sie die Lesbarkeit erhöhen, und Tippfehler von ähnlich geschriebenen Variablennamen minimiert werden.
- Logische Operatoren
 - 0 bedeutet „falsch“
 - Jede Zahl ungleich 0 bedeutet „wahr“
 - Erhält man „wahr“ als Ergebnis einer Auswertung von Vergleichsoperatoren und logischen Operatoren, so erhält man immer die 1.
- Vergleichsoperatoren
 - <, <=, >=, >, ==, !=
 - Die exakte Abfrage auf Gleichheit bei double ist aufgrund von Fehlern numerischer Verfahren nicht möglich, vielmehr muss bestimmt werden, ob das Resultat hinreichend genau ist ... ((wert - eps) < d) && (d < (wert + eps))

Der Operator &&

Mit dem Operator && können Ausdrücke logisch Und verknüpft werden. Der Operator liefert dann „wahr“ (1), wenn beide Operanden logisch „wahr“ (1) ergeben.

Der Operator && hat eine wesentliche Eigenschaft: Der zweite Ausdruck wird nur dann ausgewertet, wenn er für das Gesamtergebnis notwendig ist. Ergibt der erste Ausdruck bereits 0, so ist der zweite Ausdruck unerheblich und wird nicht ausgeführt und ausgewertet.

Der Operator ||

Mit dem Operator || werden Ausdrücke logisch Oder verknüpft. Der Operator liefert dann „wahr“ (1) wenn einer der beiden Operanden logisch „wahr“ (1) ergibt.

Genau wie der Operator && hat der Operator || die Eigenschaft, dass der zweite Ausdruck nur dann ausgewertet wird, wenn er für die Berechnung des Gesamtergebnisses notwendig ist.

Der Operator !

Zum Negieren von Wahrheitswerten wird er Operator ! verwendet. Der Wert „wahr“ wird in „falsch“ gewandelt, der Wert „falsch“ in „wahr“.

Für das Vereinfachen komplexer Beziehungen eignen sich beispielsweise Karnaugh-Veitch-Diagramme oder das Verfahren nach Quine und McCluskey.

Der Operator ,

Der Komma-Operator gleicht dem Semikolon. Beide Zeichen dienen zum Trennen von Ausdrücken. Der einzige Unterschied: Das Komma ist ein Operator, das Semikolon nicht. Der Komma-Operator ist ein binärer Operator – die Teilasdrücke werden daher von links nach rechts bearbeitet.

Rückgabewert: Wert des letzten Ausdrucks

Der Operator ? :

Bedingung ? If-Ausdruck : else-Ausdruck

Abstände

In C werden sämtliche Füllzeichen, also Abstände, Tabulatorsprünge oder Leerzeichen ignoriert.

In C sind Abstände notwendig, um Operatoren zu trennen, wenn Mehrdeutigkeiten entstehen könnten. Abstände trennen auch Namen bzw. Schlüsselwörter voneinander.

Kapitel 9 – Selektionen

Selektionen werden in der Programmierung eingesetzt, um verschiedene Programmteile in Abhängigkeit einer Bedingung auszuführen.

Die if-Anweisung

Die if-Anweisung ermöglicht das alternative Ausführen zweier Programmteile abhängig davon, ob eine Bedingung „wahr“ oder „falsch“ ergibt.

Syntax:

```
if (Bedingung)
    Block
else
```

Block

Besonderes Konstrukt: der sogenannte „if-Rechen“:

```
if (a == b)
    printf(„a und b sind gleich groß.\n“);
else if
    printf(„a ist kleiner als b.\n“);
else
    printf(„a ist größer als b.\n“);
```

Die switch-Anweisung

Die switch-Anweisung ermöglicht das selektive Ausführen von beliebig verschiedenen Programmteilen:

Syntax:

```
switch (Ausdruck)
{ case Marke1: Anweisungen
      break;
  case Marke2: Anweisungen
      break;
  ...
  default:    Anweisungen
      break;
}
```

Die Reihenfolge der Marken ist beliebig – es sei denn, es wird ein break weggelassen.

Der Ausdruck kann ein beliebiger C-Ausdruck sein. Er wird zu Beginn der Anweisung ausgewertet. Stimmt der Wert des Ausdruckes mit dem Namen einer Sprungmarke (label) überein, wird zu dieser Sprungmarke verzweigt. Existiert die erforderliche Sprungmarke nicht, so wird zur default-Sprungmarke verzweigt. Existiert auch diese Sprungmarke nicht, so wird die gesamte switch-Anweisung übersprungen.

Wurde zu einer Sprungmarke verzweigt, so wird die Ausführung unabhängig von weiteren Sprungmarken solange fortgesetzt, bis ein break auftritt oder die switch-Anweisung beendet ist.

Sprungmarken dürfen nur Zeichen oder ganze Zahlen sein.

Kapitel 10 – Iterationen

Eine Schleife wird in C solange durchlaufen, solange eine Bedingung – die Schleifenbedingung – erfüllt ist.

- vorprüfende Schleifen (prüfen vor dem Schleifenrumpf)
- nachprüfende Schleifen (prüfen nach dem Schleifenrumpf)

Die for-Anweisung

- komplexeste Schleife in C
- vorprüfende Schleife
- kommen zum Einsatz, wenn sogenannte „Iteratoren“ verwendet werden

Iteratoren:

In prozeduralen Hochsprachen, wie C, sind Iteratoren zumeist als simple Zähler oder als Zeiger implementiert. Iteratoren werden verwendet, um sich durch eine Menge von Objekten zu bewegen.

```
for (Initialisierung; Bedingung; Inkrement)
    Block
```

- Initialisierungsteil
 - wird vor dem ersten Durchlauf einer Schleife ausgeführt
- Bedingung
 - wird vor jedem Schleifendurchlauf überprüft
 - Wird eine for-Schleife ordnungsgemäß beendet, so hat nach der Schleife der Zähler den Wert, der die Bedingung nicht mehr erfüllt.

Besonderheiten:

- Weglassen von Ausdrücken
 - Ist kein Inkrement als solches notwendig, so sollte eine while-Schleife verwendet werden.
- Mehrerer Zähler
 - Möglichkeit mehrere Initialisierungen bzw. Inkremente durchzuführen

Die while-Anweisung

- mächtigste Schleife in C
- vorprüfende Schleife
- Alle anderen Schleifenarten können letztendlich durch eine while-Schleife ausgedrückt werden, auch wenn aus semantischen Gründen davon abgeraten wird.
- kommen zum Einsatz, wenn sich die an der Schleifenbedingung beteiligten Variablen innerhalb des Schleifenrumpfes ändern.

Syntax:

```
while (Bedingung)
    Block
```

- Bedingung
 - wird immer zu Schleifenbeginn ausgewertet
 - Ändern sich während des Schleifendurchlaufes die Variablen, die in der Schleifenbedingung verwendet wurden, so wird dies erst bei der Auswertung der Bedingung beim nächsten Schleifendurchlauf berücksichtigt.

Die do - while – Anweisung

- nachprüfende Schleife
- Der Block wird mindestens einmal durchlaufen, bevor das erste Mal die Bedingung abgefragt wird.

Syntax:

```
do
  Block
while (Bedingung);
```

Die break-Anweisung

Um eine Schleife abubrechen, kann die break-Anweisung verwendet werden. Sie wird dann eingesetzt, wenn ein Ergebnis innerhalb der Schleife auftritt, wodurch diese beendet werden soll.

Die break-Anweisung funktioniert für alle Schleifen (while, do-while und for) gleich.

Die continue-Anweisung

Die continue-Anweisung wird dazu verwendet, vorzeitig die nächste Schleifeniteration auszulösen.

Die continue-Anweisung innerhalb einer while-Anweisung hat ein anderes Verhalten als innerhalb einer for-Anweisung (es wird kein Inkrementteil ausgeführt – es existiert ja keiner).

Bei einer for-Schleife beginnt die nächste Schleifeniteration mit der Auswertung des Inkrement-Ausdruckes, noch bevor die Bedingung ausgewertet wird.

Schachteln von Schleifen

Schachtelung durch äußere und innere Schleife.

Kapitel 11 – Funktionen

Funktionen werden in C benötigt, um

- die Funktionalität
- das Programm zu strukturieren
- immer wieder vorkommende Programmteile einsetzen zu können
- in sich abgeschlossene definierte Teilaufgaben zu programmieren.

Definition einer Funktion

Unter einer Funktionsdefinition versteht man die Angabe des Funktionsnamens, des Rückgabetyps, der Parameter und des kompletten Funktionsrumpfes – die eigentliche Funktion.

```
Typ Funktionsname (Parameterliste)
{ Funktionsrumpf
} //end Funktionsname
```

Namen von Funktionen:

Funktionsnamen unterliegen den selben Einschränkungen, wie Namen von Variablen und Konstanten.

Parameter:

Funktionen können eine beliebige Anzahl von Parametern haben. Sie werden durch Beistriche getrennt in der Parameterliste innerhalb der runden Klammern nach dem Funktionsnamen angegeben. Dabei wird für jeden Parameter auch sein Typ festgelegt.

Die Reihenfolge der Parameter ist für die Parameterübergabe signifikant.

Rückgabewerte:

Funktionen können einen Rückgabewert haben. Der Typ des Rückgabewertes muss gleich bei der Funktionsdefinition unmittelbar vor dem Funktionsnamen angegeben werden.

Wird der Rückgabotyp weggelassen, so wird automatisch der Typ *int* angenommen. Soll die Funktion keinen Wert zurückliefern, sollte der Rückgabotyp als *void* definiert werden.

Ist der Typ des Rückgabewertes unterschiedlich zu dem des tatsächlich zurückgegebenen Wertes, findet eine Typumwandlung statt.

Der Rückgabewert der Funktion *main*:

Die Funktion *main* hat den Rückgabewert *int*. Mit diesem Rückgabewert signalisiert das Hauptprogramm dem Aufrufer, ob ein Fehler aufgetreten ist.

- Unix-Welt:
 - 0 ... „Programm erfolgreich“
 - ≠0 ... Fehlernummer
- MSDOS-Welt:
 - 0 ... Fehler
 - 1 ... „Programm erfolgreich“

Ablauf eines Funktionsaufrufes:

Bei einem Funktionsaufruf werden Werte an die Funktionsparameter in C immer in Form von Kopien (call by value) übergeben, nicht im Original (call by reference).

Deklaration einer Funktion

Unter einer Deklaration einer Funktion versteht man eine Bekanntmachung

einer Funktion innerhalb einer C-Datei. Da ein Compiler den Quelltext von oben nach unten liest und übersetzt, ist es notwendig Funktionen bekanntzumachen, bevor sie verwendet werden.

Funktionsdeklarationen werden üblicherweise in eigenen Header-Dateien abgelegt.

Eine Deklaration teilt dem Compiler folgendes mit:
Den Funktionsnamen, die Anzahl und die Typen der Argumente, den Typ des Rückgabewertes. Diese Informationen reichen aus, um Funktionsaufrufe auf syntaktische Korrektheit überprüfen zu können.

Die Namen der Parameter können in der Deklaration weggelassen werden.

Wird eine Funktion verwendet, ohne dass sie zuvor definiert oder deklariert wurde, so nimmt der C-Compiler automatisch den Rückgabewert *int* an.

Unterschied zwischen Deklaration und Definition

<i>Merkmal</i>	<i>Definition</i>	<i>Deklaration</i>
Eigenschaft	Die Definition einer Funktion ist die komplette Beschreibung der Funktion. Die Definition enthält also auch den Funktionsrumpf.	Die Deklaration ist eine Bekanntmachung einer Funktion an den Compiler.
Sichtweise	Die komplette Funktion wird von innen betrachtet und beschreibt die Implementierung von Algorithmen.	Die Funktion wird als Black Box von außen betrachtet.
Parameter	Parameter müssen jeweils mit Datentyp und Name angegeben werden.	Für die Sichtweise von außen sind die Namen der Parameter nicht relevant. Sie können weggelassen werden.

Externe Funktionen:

Alle Funktionen sind automatisch extern definiert, es sei denn, das Schlüsselwort *static* ist im Funktionskopf angegeben.

Statische Funktionen:

Statische Funktionen können nicht exportiert werden (sind nur innerhalb eines Moduls sichtbar).

Kapitel 12 – Speicherklassen

Speicherklassen legen den Gültigkeitszeitraum und den Sichtbereich von Variablen fest.

Lokale Variablen

Die Lokalität beschränkt sich dabei auf die Funktionen, in denen sie definiert wurden. Der Vorteil von lokalen Variablen ist, dass sie innerhalb dieser Funktion gekapselt sind (Ausnahme: call by reference).

Variablendefinitionen müssen zu Beginn eines Blockes stehen.

Variablen der Speicherklasse *auto*:

Alle lokalen Variablen, denen kein spezielles Schlüsselwort, wie *register* oder *static* vorangestellt ist, sind automatische Variablen. Sie heißen automatische Variablen, da sie zu Blockanfang automatisch für den Block angelegt und zu Blockende automatisch gelöscht werden.

Variablen der Speicherklasse *register*:

Die Variablen werden nicht im Speicher angelegt, ihnen wird vielmehr ein Register des Prozessors zugeordnet.

Es sind nur die Datentypen *char*, *long*, *double* (auch *float*) und Zeiger erlaubt (keine selbstdefinierten Datentypen).

Sind jedoch keine Register mehr frei, werden Variablen, die mit *register* definiert wurden, wie automatische Variablen behandelt. Die Optimierer vieler Compiler setzen häufig gebrauchte Variablen aber automatisch als Registervariablen um.

Da Register keine Adresse haben, kann der Adressoperator nicht auf Registervariablen angewendet werden.

Variablen der Speicherklasse *static*:

Lokale Variablen der Speicherklasse *static* unterscheiden sich von automatischen Variablen durch ihren Gültigkeitszeitraum:

- Sie existieren, solange das Programm läuft.
- Der Sichtbarkeitsbereich ist trotzdem noch lokal.

Die Initialisierung findet jedoch nur zu Programmstart statt. Wird kein Initialwert angegeben, wird die Variable automatisch mit 0 initialisiert.

Globale Variablen

Globale Variablen werden, wenn sie nicht explizit initialisiert werden, automatisch mit 0 initialisiert. Ihre Initialwerte müssen entweder Literale sein oder zur Übersetzungszeit feststellbar sein.

Globale Variablen haben den Nachteil, dass sie auch verschiedensten Funktionen heraus modifiziert werden können.

Variablen der Speicherklasse *extern*:

Globale Variablen, die nicht mit dem Schlüsselwort `static` definiert sind, sind sogenannte externe Variablen. Das Schlüsselwort `extern` darf allerdings bei einer Definition nicht angegeben werden.

Sie können auch exportiert werden. Das bedeutet, dass ihr Name und ihr Datentyp in anderen Dateien, die zum selben Projekt bzw. Programm gehören, bekanntgemacht wird.

Nur externe globale Variablen können deklariert werden. Variablen anderer Speicherklassen können nur definiert werden.

Variablen der Speicherklasse *static*:

Soll eine globale Variable nicht exportiert werden, ist sie mit dem Schlüsselwort `static` zu versehen. Die Speicherklasse der globalen statischen Variable ist verwandt mit der einer lokalen statischen Variable. Beide existieren solange das Programm läuft.

Kapitel 13 – Felder

Felder (arrays) stellen einen Verbundtyp dar. Sie werden verwendet, um mehrere Daten des selben Typs zu speichern.

Eindimensionale Felder

```
Feldtyp feldname[Feldlaenge];
```

- *feldtyp* gibt den Datentyp für alle Feldelemente an. Es können alle Datentypen in C (mit Ausnahme von *void*) und selbstdefinierte Datentypen verwendet werden.
- *feldlaenge* legt die Anzahl der Elemente des Feldes fest. Die Feldlänge kann nach der Definition nicht mehr geändert werden. Seit dem C-Standard 1999 kann die Feldlänge ein allgemeiner Ausdruck sein.
- *feldname* ist der Name des Feldes.

Alle Objekte (Elemente) eines Feldes werden hintereinander angelegt. Der Name des Feldes steht für die Adresse, an der das Feld im Speicher liegt.

Die Nummerierung des Feldindizes beginnt immer bei 0.

Wird außerhalb der Feldgrenzen gelesen oder geschrieben, können 2 Dinge passieren:

- Das Programm stürzt ab, da vom Betriebssystem ein illegaler Speicherzugriff auf einen Speicherbereich, der dem Programm nicht zugeordnet ist, erkannt wird.
- Es passiert nichts sofort Erkennbares, weil der Speicherbereich, auf den zugegriffen wurde, zum Programm gehört: Es werden andere Daten oder

sogar ein Teil des Programmes direkt überschrieben.

Mehrdimensionale Felder

C unterstützt auch mehrdimensionale Felder. Jedes Feldelement hat dadurch zwei oder mehr Indizes.

Die Feldlängen können, wie bei eindimensionalen Feldern allgemeine Ausdrücke sein, wobei der erste die Zeilen und der zweite die Spalten angibt.

Mehrdimensionale Felder sind in C in Wirklichkeit eindimensionale Felder, bei denen jedes Feldelement wieder ein Feld ist. Sie werden zeilenweise abgespeichert.

Initialisieren von Feldern

Beim Initialisieren von Feldern muss für jedes Feldelement der Reihe nach ein Wert angegeben werden:

```
long zahlen[5] = {11, 22, 33, 44, 55};
```

Ein Überspringen von Elementen ist nicht möglich. Werden zu wenig Werte angegeben, so werden nur die ersten Elemente aufgefüllt, die restlichen werden auf 0 gesetzt. Werden bei der Definition eines Feldes jedoch keine Initialwerte angegeben, so werden nur globale und statische Felder automatisch mit 0 initialisiert.

Ein ganzes Feld mit 0 initialisieren:

```
long zahlen[5] = {0};
```

Die Feldlänge kann bei der Initialisierung auch weggelassen werden:

```
long primzahlen = {2, 3, 5, 7, 11, 13, 17, 19};
```

Konstante Felder

Sind die Elemente eines Feldes Konstanten, so sollte das Schlüsselwort `const` bei der Definition des Feldes angeschrieben werden:

```
const long primzahlen = {2, 3, 5, 7, 11, 13, 17, 19};
```

Felder als Parameter

Felder werden immer im Original (call by reference) übergeben. Der Grund dafür ist, dass das Erzeugen von Kopien von Feldern zu einem zu hohen Speicherverbrauch und Geschwindigkeitsverlust führen würde.

Genau genommen steht der Name des Felde für die Adresse, an der das Feld im Speicher stehen. Bei genauerem Hinsehen wird also sehr wohl eine Kopie übergeben: Die Adresse des Feldes! Greift die Funktion allerdings auf die Daten

zu, die an dieser Adresse stehen, wird auf die Originalelemente des Feldes zugegriffen.

Um einer Funktion die Länge eines Feldes mitzuteilen, muss ein weiterer Parameter für die Feldlänge übergeben werden. Durch das leere eckige Klammernpaar bei der Definition wird ausgedrückt, dass für den Parameter `feld` ein eindimensionales Feld beliebiger Länge erwartet wird (falls die Feldlänge trotzdem angegeben wird, wird sie durch den Compiler ignoriert).

Die Größe der ersten Dimension eines mehrdimensionalen Feldes muss durch einen weiteren Parameter an die Funktion übergeben werden. Alle anderen Dimensionen müssen in der Funktionsdefinition korrekt angegeben werden.

Um genau zu sein, handelt es sich innerhalb der Funktionen nicht mehr um Felder, sondern um konstante Zeiger.

Einfache Sortierverfahren

Vorsortierte Daten erleichtern die Suche nach einem bestimmten Wert bzw. Datensatz erheblich. Welcher Algorithmus für ein bestimmtes Problem gut geeignet ist, hängt von den Eigenschaften der zu sortierenden Daten ab.

Minimum-Suche:

„Minimum-Suche“ oder auch „Selection Sort“ ist einer der einfachsten Sortieralgorithmen.

Verfahren:

Suche in einer Folge von Zahlen das kleinste Element und tausche es gegen das Element an Position 0. Suche nun das zweitkleinste Element und tausche es gegen das Element an Position 1. Fahre so fort, bis das gesamte Feld sortiert ist.

„Minimum-Suche“ ist ein langsames, einfaches Verfahren. Für kleine Zahlenmengen ist es allerdings durchaus ausreichend.

Bubble Sort

Ein anderes Sortierverfahren ist „Bubble Sort“. Dabei werden in einem Schritt immer zwei benachbarte Elemente miteinander verglichen. War das Element mit dem kleineren Index größer als das andere, wird getauscht. Ist kein Tausch mehr erforderlich, so ist das Feld sortiert.

Einfache Suchverfahren

Sequenzielles Suchen

Sequenzielles Suchen ist das einfachste Suchverfahren. Eine Datenmenge wird beim ersten Element beginnend bis zum letzten durchsucht. Dieses Verfahren eignet sich nur für nicht sortierte, kleine Datenmengen.

Der Aufwand dieses Verfahrens ist proportional der Feldlänge. Ist das Feld sehr groß, ist der Suchaufwand enorm.

Binäres Suchen

Bei großen Datenmengen kann die Suchdauer durch das Schema „Teile und Eroberer“ (divide and conquer) erheblich verringert werden:

- Teile die Menge der Daten in 2 Hälften.
- Bestimme den Teil der den gewünschten Wert enthalten kann und
- setze das Schema in diesem Teil fort
- bis das Datum gefunden ist
- oder festgestellt ist, dass das gesuchte Datum nicht in der Datenmenge enthalten ist

Der Aufwand dieses Verfahrens ist proportional dem Logarithmus der Feldlänge N – also $\log_2(N)$. Hat das Feld die Länge $N = 2^x - 1$, so ist der Eintrag mit maximal x Vergleichen gefunden.

Kapitel 14 – Zeiger

Ein Zeiger (pointer) ist eine Variable, die eine Adresse enthält. Da eine Variable ein Objekt darstellt, ist streng genommen auch ein Zeiger ein Objekt (ein Objekt ist in C eine Variable oder ein beliebiger Lvalue). Dennoch spricht man im Zusammenhang mit Zeigern nicht von Objekten.

Zeiger sind oft die einzige Möglichkeit der Realisierung mancher Programmierkonzepte (aufgrund fehlender Konzepte der Datenorganisation).

Der unäre Adressoperator & kann nur auf Lvalues angewendet werden, also nicht auf Ausdrücke, Konstante oder *register*-Variable.

Will man das Objekt erreichen, auf das ein Zeiger zeigt, so muss der unäre Inhaltsoperator * dem Zeiger vorangestellt werden.

Dereferenzieren:

Wird einem Zeiger der Operator * vorangestellt, so erhält man das Objekt, auf das er zeigt. Man sagt: Der Zeiger wird „dereferenziert“.

Zeiger haben wie reguläre Variablen, einen Datentyp:

```
long *ptr;
```

Der Datentyp von *ptr* ist *long **.

Zeiger sollten immer initialisiert werden. Manchmal ist es aber auch wichtig, Zeiger als ungültig zu markieren, um auszusagen, dass sie auf kein Objekt zeigen (Adresse 0).

Zeiger als Parameter:

Soll eine Funktion mehr als einen Wert an den Aufrufer zurückgeben, müssen Zeiger als Parameter verwendet werden (call by reference).

Die „Dualität“ von Zeigern und Feldern

Software-technisch gesehen sind Zeiger und Felder zwei völlig verschiedene Konstrukte. In C wurden jedoch gewollt semantische Parallelen eingeführt, die oft leider verwirrend sind.

- Feld ... Verbundtyp
- Zeiger ... Variable, die eine Adresse speichert

Hauptverantwortlich dafür, dass man überhaupt von einer „Dualität“ von Zeigern und Feldern sprechen kann, ist, dass in C der Name eines Feldes als konstanter Zeiger verwendet werden kann – der Feldname steht für die Adresse, an der sich das Feld befindet. Er kann somit einem Zeiger zugewiesen werden.

Auf Grund der „Zeiger-Feld-Dualität“ kann der Index-Operator [] auch auf Zeiger, der Dereferenzierungsoperator * auch auf Felder angewendet werden.

Die sprachliche „Dualität“ von Zeigern und Feldern tritt auch dann zu Tage, wenn Felder als Parameter an eine Funktion übergeben werden (konstante Zeiger!).

Zeigerarithmetik

Wird ein Zeiger um n erhöht, so wird seine Adresse auf das n-te im Speicher unmittelbar folgende Objekt vom selben Typ gesetzt – ähnlich der Indizierung in einem Feld. Die Objektgröße wird dabei also berücksichtigt.

Durch das Erhöhen eines Zeigers kann also über ein Feld iteriert werden:

	<i>Zeigerschreibweise</i>	<i>Feldschreibweise</i>
Adresse:	ptr + n	&feld[n]
Objekt:	*(ptr + n)	feld[n]

Weitere sinnvolle Operatoren im Zusammenhang mit Zeigern sind:

- Subtraktion (ptr2 – ptr1)
- Vergleichsoperatoren <, <=, >=, >, == und !=

Komplexere Fälle: Felder von Zeigern, Zeiger auf Zeiger

Zeiger können sehr gut eingesetzt werden, wenn viele Felder unterschiedlicher Länge verwaltet werden sollen. Eine häufige Anwendung sind Felder von Zeigern auf Zeichenketten.

```
long a1[6];
long a2[3];
long a3[5];

long *p[3] = {&a1[0], &a2[0], &a3[0]};
```

Will man auf das 3. Element mit dem Index 2 in `a1` zugreifen, so kann einer der folgenden Ausdrücke geschrieben werden, da `a1` und `p[0]` das selbe Feld bezeichnen:

```
a1[2]
p[0][2]
```

`p[0]` ist ein Zeiger, der mit dem Index `[2]` dereferenziert wird (`*(p[0] + 2)`). Diese Schreibweise ist aufgrund der „Zeiger-Feld-Dualität“ möglich.

Die Verwaltung der Längenangaben kann mit einem separaten Feld geschehen. Bei Zeichenketten ist dies allerdings nicht erforderlich, da sie mit einem Null-Byte abgeschlossen sind.

Konstante Zeiger

Um das Schreiben auf einen Speicherbereich, auf den ein Zeiger weist, zu verhindern, muss der Zeiger als konstant definiert werden.

```
const char *ptr;
```

`ptr` zeigt auf ein Objekt vom Typ `char`, das konstant ist.

```
char * const ptr = &feld[0];
```

`ptr` ist ein konstanter Zeiger auf ein Objekt vom Typ `char` und wird mit der Adresse des ersten Elementes des Feldes `feld` initialisiert.

```
const char * const ptr = &feld[0];
```

`ptr` ist ein konstanter Zeiger auf ein Objekt vom Typ `char`, das ebenfalls konstant ist. Der Zeiger wird mit der Adresse des ersten Elementes des Feldes `feld` initialisiert.

Zeiger auf Funktionen

In C können auch Zeiger auf Funktionen vereinbart werden. Sie sind bei der Verwendung von externen Bibliotheken, wie beispielsweise graphischen Benutzerschnittstellen (graphical user interfaces, GUIs), kaum wegzudenken.

```
&HalloWelt; //Adresse der Funktion HalloWelt (empfohlen)
HalloWelt; //Adresse der Funktion HalloWelt (nicht empfohlen)
```

```
HalloWelt(); //Aufruf der Funktion HalloWelt
```

Definition eines Zeigers auf eine Funktion:

```
Typ (* Funktionsname) (Parameterliste);
```

Die einzigen Operatoren, die auf Zeiger auf Funktionen angewendet werden können, sind der Dereferenzierungsoperator `*` und der Adressoperator `&`. Die Verwendung des Dereferenzierungsoperators `*` ist allerdings nutzlos und wahrscheinlich nur der Vollständigkeit halber implementiert.

Einsatzgebiete für Zeiger auf Funktionen sind beispielsweise call back functions (Funktionen, die vom Betriebssystem beim Eintreten eines Ereignisses aufgerufen werden) oder Menüs.

Die Verwendung von Zeigern auf Funktionen beim Aufruf unterscheidet sich nicht von gewöhnlichen Funktionsaufrufen.

Typdefinitionen mit *typedef*

Mit *typedef* können Typnamen für Zeiger auf Funktionen generiert werden:

```
typedef long Funktion_t(long, long);
Funktion_t *fptr;
```

Der Vorteil der Vereinbarung *typedef* liegt in der erhöhten Übersichtlichkeit und geringeren Fehleranfälligkeit dieser Schreibweise.

Kapitel 15 – Zeichenketten

Zeichenketten sind in C nicht als unabhängiger Datentyp implementiert. Zeichenketten sind „nullterminierte“ Folgen von Zeichen, die in Feldern von Zeichen abgespeichert werden. Nullterminiert bedeutet, dass eine Zeichenkette mit einer `'\0'` abgeschlossen wird – dem sogenannten Null-Byte (ASCII-Code 0). Das Null-Byte wird bei einer Zeichenkette, die in Anführungszeichen angegeben ist, automatisch angehängt. Der Text „Hallo“ ist beispielsweise 5 Zeichen lang, er benötigt aber 6 Zeichen Speicherplatz.

- Vorteil der Nullterminiertheit: Es können beliebig lange Texte abgespeichert werden.
- Nachteil der Nullterminiertheit: Die Textlänge steht im Vorhinein nicht fest (*strlen* muss verwendet werden).

Literale:

Literale von Zeichenketten sind Zeichenfolgen, die in doppelte Anführungszeichen gesetzt sind:

```
„Hallo Welt!“;
```

Stehen zwischen den Textteilen nur Füllzeichen (whits spaces), wie Leerzeichen, Tabulatoren oder der Zeilenvorschub, so gelten die Textteile als eine Zeichenkette.

Datentyp

Einfache Zuweisungen sind nur bei der Initialisierung möglich, ansonsten muss *strcpy* verwendet werden.

```
strcpy(text, „Hallo“);
```

Der Text „Hallo“ ist genau gesagt ein Feld – und zugleich ein konstanter Zeiger -> Zeiger auf *char* werden gerne für Texte verwendet.

Initialisierung

```
char text[6] = „Hallo“;
```

Die Feldlängen sollten nur bei konstanten Feldern weggelassen werden.

```
const char text[] = „Hallo“;
```

Elementare Funktionen für Zeichenketten:

Funktionen zur Manipulation von Zeichenketten aus *string.h*:

- `strcat(s, t)`
- `strncat(s, t, n)`
- `strcmp(s, t)`
- `strncmp(s, t, n)`
- `strcpy(s, t)`
- `strncpy(s, t, n)`
- `strlen(s)`
- `strchr(s, c)`
- `strrchr(s, c)`
- `strstr(s, t)`

Beim Verwenden der Funktionen `strcpy` und `strcat` ist Vorsicht geboten. Sind die zu kopierenden Texte länger als das Feld selbst, so wird über die Feldgrenzen hinausgeschrieben!

Ausgabefunktionen für Zeichenketten aus *stdio.h*:

- `printf(f, ...)`
- `snprintf(s, n, f, ...)` //schreibt max. n Zeichen in das Feld s

Ausgabefunktionen für Zeichenketten aus *stdlib.h*:

- `atol(s)` //wandelt die Zeichenkette s in ein long um und liefert dieses zurück

Felder von Zeigern auf Zeichenketten

Argumente der Funktion *main*

Der Parameter *argv* (argument value) ist ein Feld von Zeigern auf Zeichenketten. Die erste Zeichenkette ist immer der Programmname selbst. Der erste Parameter *argc* (argument counter) gibt die Länge des Feldes *argv* an.

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

Kapitel 16 – Abgeleitete Datentypen

Sogenannte Verbunddatentypen ermöglichen durch das Kombinieren mehrerer Elemente unterschiedlichen Datentyps die Definition komplexerer Datentypen. Aufzählungen werden zur Gruppierung von Konstanten mit Namen verwendet.

Datenstrukturen haben erheblichen Einfluss auf die darauf angewendeten Algorithmen und Funktionen eines Programmes.

- Datenstrukturen müssen in Hinblick auf die zu verwendenden Funktionen und Algorithmen entworfen werden.
- Die Elemente `struct` und `union` sollen ihrem Einsatzzweck entsprechend verwendet werden.
- Datenstrukturen sollten möglichst einfach und verständlich aufgebaut sein.
- Zu tiefe Schachtelungsebenen sollen vermieden werden.
- Datenstrukturen mit einer hohen Anzahl an Attributen (Komponenten) sollen vermieden werden.
- Zu allen Datenstrukturen sollen die benötigten Sets an Funktionen, sogenannte Zugriffs- oder Schnittstellenfunktionen (interface sets), definiert werden, um sie zu bedienen.
- Die Informationen sollen stets redundanzfrei und konsistent bleiben.

Strukturen

Strukturen können beliebig komplex sein und aus unterschiedlichen Datentypen bestehen.

Einfache Strukturen

```
struct Strukturname
{ Typ Attributname;
  // ...
};
```

Mit der Typdefinition wird jedoch kein Speicherplatz reserviert, es wird lediglich die Struktur eines neuen Datentyps beschrieben.

Variablen, die in einer Struktur angegeben sind, nennt man Komponenten oder Attribute.

Variablendefinition:

```
struct Strukturname Variablenliste;
```

Anonyme Strukturen:

Bei kombinierten Typ- und Variablendefinitionen könnte nach dem C-Standard sogar der Typname entfallen, wenn der Datentyp nur an einer Stelle im Programm verwendet wird. Die Struktur wird dann anonym definiert:

```
struct
{ Typ Attributname;
  // ...
} Variablenliste;
```

- Struktur-Variablen können auch initialisiert werden. Die Initialwerte werden dabei in geschwungene Klammern gefasst und der Reihe nach angegeben.
- Um auf die Attribute so definierter Struktur-Variablen zuzugreifen, wird der Punkt-Operator '.' - der Selektionsoperator - verwendet.
- Strukturen können auch verschachtelt werden.
- Strukturen können als Ganzes an Funktionen übergeben werden und auch als Funktionswert zurückgegeben werden. Es empfiehlt sich jedoch, bei zeitkritischen Funktionen bereits ab einer Strukturgröße von 4 bis 8 Byte nicht die Kopien von Strukturen, sondern ihre Adressen zu übergeben (call by reference)

Zeiger auf Strukturen

```
struct Punkt_s p1 = {4, 3};
struct Punkt_s *pp = &p1;
```

Um ein Attribut ansprechen zu können, gibt es 2 Notationen:

- (*pp).x
- pp->x

Felder von Strukturen

Strukturen werden gerne eingesetzt, um in Programmen zusammengehörige Daten zusammenzufassen.

Wieviele Bytes bzw. Bits genau für eine Struktur verwendet werden, ist compiler- und systemabhängig. Die Abweichungen von der theoretischen Größe entsteht durch Füllbytes.

Felder von Strukturen können aber auch initialisiert werden. Dabei gilt, wie für alle Felder, dass bei Teilinitialisierungen – also bei nicht vollständigen Initialisierungen – nicht initialisierte Einträge auf 0 gesetzt werden.

Attribute in Strukturen werden wie Variablen verwendet.

Enthält eine Struktur Zeichenketten oder Felder allgemein, ist zu überlegen, ob

statt Felder nicht besser Zeiger verwendet werden sollen. Felder haben den Vorteil, dass sie einfacher zu verwalten sind – sie werden automatisch mit der geforderten Feldlänge angelegt. Felder sind aber starr. Für viele Namen wird die Feldlänge des Attributs *name* überdimensioniert sein. Für lange Namen ist sie zu kurz.

Zeiger auf Strukturen sind genau so zu behandeln wie Zeiger auf einfache Datentypen.

Bitfelder

In Bitfeldern werden mehrere ganzzahlige Objekte in einem Maschinenwort zusammengefasst, was beispielsweise auch zur Nachbildung von Hardwareregistern vorteilhaft ist:

```
struct Bitfelder_s
{ unsigned int a : 1; //Bitfeld der Länge 1
  unsigned int b : 2; //Bitfeld der Länge 2
  unsigned int c : 3; //Bitfeld der Länge 3
  unsigned int d : 5; //Bitfeld der Länge 5
};
```

- Bitfelder dürfen nur als *int* vereinbart werden, wobei das Schlüsselwort *signed* oder *unsigned* aus Gründen der Portabilität erforderlich ist.
- Wird der Variablenname weggelassen (das Bitfeld ist anonym), so wird ein Zwischenraum geschaffen.
- Wird für anonyme Bitfelder die Länge mit 0 angegeben, so wird das nächste Bitfeld in ein neues Maschinenwort gesetzt.

Bitfelder werden meist eingesetzt, um Speicherplatz zu sparen. Bitfelder sind maschinenabhängig und haben keine Adresse, weshalb auch keine Zeiger auf Bitfelder definiert werden können. Ebenso sind Felder von Bitfeldern nicht möglich.

Aufzählungen

Aufzählungen werden in C verwendet, um mehrere Konstanten zu definieren und zu einem Typ zu kombinieren. Alle Konstanten müssen ganzzahlig sein. Diese Zahlen werden der Reihe nach aufsteigend mit 0 beginnend vergeben.

```
enum Enumname_e
{ NAME1,
  NAME2,
  NAME3
};
```

Den einzelnen Konstanten können auch ganzzahlige Werte explizit zugewiesen werden. Welcher Datentyp verwendet wird (*short int* oder *long int*) ist compiler- bzw. maschinenabhängig.

Werden Zahlenwerte weggelassen, so werden Zahlen wieder aufsteigend vergeben.

Aufzählungstypen werden gerne eingesetzt, um Mengen gleichartiger Namen, zu erzeugen, die voneinander unterschieden werden sollen. Der Aufzählungstyp soll garantieren, dass jeder Name für einen eindeutigen Zahlenwert steht.

Ein weiterer Vorteil von Aufzählungstypen ist, dass die Eindeutigkeit auch beim Umsortieren der Namen gewährleistet bleibt, sofern keine expliziten Werte angegeben werden.

Variante Strukturen

Bei regulären Strukturen werden die Attribute hintereinander im Speicher angelegt, bei varianten Strukturen im selben Speicherbereich (ihre Attribute liegen quasi übereinander).

Variante Strukturen werden eingesetzt, wenn verschiedene Attribute zu einer Struktur zusammengefasst werden sollen, die jedoch nicht gleichzeitig auftreten können.

```
union Unionname_u
{ Typ Attributname;
  // ...
};
```

Anonyme variante Strukturen:

```
union
{ Typ Attributname;
  // ...
} Variablenliste;
```

Die Länge einer varianten Struktur entspricht der Länge des größten Attributs. Die Information, welches Attribut gültig ist, muss separat gespeichert werden.

Da es bei tief verschachtelten Strukturen oft zu sehr langen Variablennamen kommen kann, werden hier gerne anonyme variante Strukturen verwendet. Dies erfordert allerdings die Definition der *union* innerhalb der Struktur.

Rekursive Strukturen

Rekursive Strukturen sind Strukturen, die Attribute enthalten, die auf Strukturen des selben Typs vereisen.

```
struct Eintrag_s
{ struct Eintrag_s *naechster;
  long wert;
};
```

Deklaration von abgeleiteten Datentypen

Ein abgeleiteter Datentyp kann auch deklariert werden. Dies ist speziell dann erforderlich, wenn Zeiger auf Objekte des Datentyps verwendet werden sollen, die Interna des Datentyps zugleich jedoch nicht interessieren. Die Definition des Datentyps selbst erfolgt dann in einem anderen Modul.

Ist innerhalb eines Moduls ein Datentyp nur deklariert nicht jedoch definiert, können keine Objekte dieses Datentyps angelegt werden. Weiters können Zeiger auf Objekte dieses Datentyps nicht dereferenziert werden, denn es ist weder die Objektgröße noch der Aufbau des Datentyps bekannt.

Analog zur Deklaration einer Funktion kann zusammenfassend gesagt werden, dass die Deklaration eines Datentyps lediglich die Bekanntmachung an den Compiler ist, dass ein Datentyp dieses Namens existiert. Die Deklaration entspricht somit der „Sicht von außen“ - im Gegensatz zur Definition, die den Datentyp näher beschreibt als „Sicht von innen“.

Typdefinition mit *typedef*

Mit der *typedef*-Vereinbarung können neue Typnamen erzeugt werden. Die neu erzeugten Typnamen sind zu der Bezeichnung, für die sie stehen, äquivalent.

```
typedef char *string;
```

Wird eine Variable mit dem Namen *string* definiert, so wird sie so definiert, wie eine gedachte Variable *string* in der Typdefinition.

Die *typedef*-Vereinbarung kann auch mit der Strukturdefinition kombiniert werden.

Es sei betont, dass mit der *typedef*-Vereinbarung kein neuer Datentyp definiert wird. Es wird vielmehr ein weiterer Name für einen bestehenden Typ geschaffen, der ebenfalls verwendet werden kann.

Kapitel 17 – Dateien

Dateien sind Objekte, die vom Betriebssystem verwaltet werden. Die Standard-Bibliothek stellt eine Reihe an Funktionen zur Verfügung, die das komfortable Arbeiten mit Dateien ermöglichen.

Datenströme

Die Standard-Bibliothek bietet zur Ein- und Ausgabe sogenannte Datenströme – kurz Ströme (streams) – an. Ströme sind Objekte, in die Informationen geschrieben oder aus denen Informationen gelesen werden können. Will man von einer Datei lesen oder in eine Datei schreiben, so benötigt man einen Strom, über den die gesamte Kommunikation aber auch Positionierungen

innerhalb der Datei erfolgen.

- **Gepufferte Ströme** geben die Informationen nicht gleich weiter sondern sammeln sie, bis genug Daten vorhanden sind, um als Ganzes oder in größeren Portionen beispielsweise auf die Festplatte oder den Bildschirm geschrieben zu werden.
- **Nicht gepufferte Ströme** geben die Daten sofort weiter, was bei block-orientierten Medien, wie z.B. der Festplatte, nicht sinnvoll ist.

Standardmäßig geöffnete Ströme:

- stdout
- stdin
- stderr (sollte für Fehlermeldungen benutzt werden – In Unix-Systemen ist diese Konvention sehr sinnvoll, da die Ströme umgelenkt, gefiltert oder entfernt werden können)

Öffnen und Schließen von Datenströmen

Ströme werden durch sogenannte Filehandles – Objekte des Typs *FILE* repräsentiert. Für das Arbeiten mit einer Datei wird jedoch nur ein Zeiger auf das Filehandle verwendet. Die Informationen, die im Filehandle stehen, sind für den Programmierer nicht interessant. Sie werden vom Betriebssystem verwaltet und dürfen nicht geändert werden.

- Um auf eine Datei zugreifen zu können, muss diese zuerst mit einem Filehandle verknüpft werden.
- Ist das Arbeiten mit einem Strom abgeschlossen, muss er geschlossen werden, wobei bei Ausgabe-Strömen alle gepufferten Daten geschrieben werden.
- Standard-Ströme sind bereits geöffnet. Werden sie geschlossen, ist ein Öffnen nicht mehr möglich.

Grundlegende Funktionen für Ströme:

- `fopen(s, m)` //Öffnet die Datei `s` mit dem Modus `m` und gibt einen Strom `F` zurück
- `fclose(F)` //Schließt den Strom `F`
- `fflush(F)` //Leert den Strom `F`

Der Befehl `fopen` liefert einen Zeiger auf ein Filehandle zurück. Konnte der Strom nicht geöffnet werden, wird 0 zurückgegeben.

Der Befehl `fclose` schließt einen Strom und löst die Verknüpfung zur Datei. Eventuell noch gepufferte Daten werden in die Datei geschrieben.

Modi für `fopen`: „r“, „r+“, „w“, „w+“, „a“, „a+“

Mit der Funktion *fflush* werden gepufferte Daten in die Datei geschrieben. Der Befehl *fflush* wird in C implizit bei Ausgabe von '\n', durch Eingabefunktionen, durch *fclose* oder mit Programmende aufgerufen.

Die Befehle *fclose* und *fflush* liefern 0 zurück, wenn die Befehle ordnungsgemäß ausgeführt wurden, einen anderen Wert im Fehlerfall.

Ein- und Ausgabe

Textbasierte Ein- und Ausgabefunktionen für Ströme:

- `fprintf(F, f, ...)` //wie printf - Ausgabe in Strom F
 - `fscanf(F, f, ...)` //wie scanf - lesen von Strom F
 - `fgets(s, n, F)` //liest eine Zeile aus dem Strom F und schreibt sie nach s, aber nicht mehr als max. n Zeichen
 - `fgetc(F)` //liest 1 Zeichen aus dem Strom F
 - `fputc(c, F)` //schreibt das Zeichen c in den Strom F
- Der Rückgabewert von *fprintf* ist die Anzahl der Zeichen, die geschrieben werden konnten.
 - Die Funktion *fscanf* retourniert die Anzahl der an Variablen zugewiesenen Werte.
 - Im Fehlerfall oder wenn das Dateiende erreicht ist, retournieren *fscanf*, *fgetc* und *fputc* die Konstante *EOF* (end of file).
 - Die Funktion *fgets* gibt im Fehlerfall 0 zurück.
 - Die Funktionen *fgetc* und *fputc* liefern bzw. erwarten ein Zeichen als *int*. Das bedeutet, dass ein Zeichen vom Typ *char* in den Typ *int* umgewandelt wird. Das ist notwendig, um die Konstante *EOF* (-1) vom Zeichen mit dem ASCII-code 255 unterscheiden zu können.

Sollen Strukturen oder binäre Daten gelesen oder geschrieben werden, können die Funktionen *fread* und *fwrite* verwendet werden.

Funktionen für blockweises Lesen und Schreiben:

- `fread(b, g, n, F)` //liest n Elemente der Größe g Byte aus F und speichert sie in b
 - `fwrite(b, g, n, F)` //schreibt n Elemente der Größe g Byte von b nach F
- Beide Funktionen liefern die Anzahl der erfolgreich gelesenen bzw. geschriebenen Elemente als Ergebnis zurück – im Fehlerfall 0 oder eine ganze Zahl kleiner als n.
 - Dateien, die mit *fwrite* geschrieben wurden, können nur noch mit dem Befehl *fread* gelesen werden.

Dateien müssen aber nicht nur sequenziell gelesen werden Die Funktionen *fseek* und *ftell* ermöglichen das Positionieren innerhalb einer Datei. Das nächste Schreiben oder Lesen findet dann an der gewünschten Position

innerhalb des Files statt.

- `fseek(F, n, p)` //setzt die Position in F auf n Bytes relativ zu p
- `ftell(F)` //liefert die aktuelle Position in Bytes relativ zum Dateianfang von F
- Die Schreib-Lese Position wird mit `fseek` gesetzt. Dabei wird von einer durch p bestimmten Position ausgegangen und n Bytes addiert. Der Wert von n kann auch negativ sein.
- Präprozessorkonstanten für p:
 - `SEEK_SET` ... Positionsangabe relativ zum Dateianfang
 - `SEEK_CUR` ... Positionsangabe relativ zur aktuellen Cursorposition
 - `SEEK_END` ... Positionsangabe relativ zum Dateiende
- Die Funktionen `fseek` und `ftell` retournieren -1, wenn ein Fehler aufgetreten ist.