

Überblick

- ◆ Struktur eines C-Programms
- ◆ Datentypen und Variablen
- ◆ Anweisungen
- ◆ Funktionen
- ◆ C-Präprozessor
- ◆ Programmstruktur und Module
- ◆ Zeiger(-Variablen)
- ◆ sizeof-Operator
- ◆ Explizite Typumwandlung — Cast-Operator
- ◆ Speicherverwaltung
- ◆ Felder
- ◆ Strukturen
- ◆ Ein- / Ausgabe
- ◆ Fehlerbehandlung

Systemprogrammierung Einführung in die Programmiersprache C

■ Literatur zur C-Programmierung:

- ◆ Darnell, Margolis. *C: A Software Engineering Approach*. Springer 1991
- ◆ Kernighan, Ritchie. *The C Programming Language*. Prentice-Hall 1988
- ◆ Dausmann, Bröckl, Schoop, et al. *C als erste Programmiersprache: Vom Einsteiger zum Fortgeschrittenen*. (Als E-Book aus dem Uninetz verfügbar; PDF-Version unter /proj/i4sp1/pub). Vieweg+Teubner, 2010.

Struktur eines C-Programms

globale Variablendefinitionen

Funktionen

```
int main(int argc, char *argv[]) {
    Variablendefinitionen
    Anweisungen
}
```

■ Beispiel

```
int main(int argc, char *argv[]) {
    printf("Hello World!\n");
    return(0);
}
```

■ Übersetzen mit dem C-Compiler:

```
cc -o hello hello.c
```

■ Ausführen durch Aufruf von ./hello

Datentypen und Variablen

- **Datentyp := (<Menge von Werten>, <Menge von Operationen>)**
 - Literal Wert im C-Quelltext (z. B. `4711`, `0xff`, `'a'`, `3.14`)
 - Konstante Bezeichner für einen Wert
 - Variable Bezeichner für einen Speicherplatz, der einen Wert aufnehmen kann
 - Funktion Bezeichner für eine Sequenz von Anweisungen, die einen Wert zurückgibt
- ➔ Literale, Konstanten, Variablen, Funktionen haben einen (Daten-)Typ
- **Datentyp legt fest:**
 - Repräsentation der Werte im Rechner
 - Größe des Speicherplatzes für Variablen
 - erlaubte Operationen

3.1 Primitive Datentypen in C

- Ganzzahlen/Zeichen: `char, short, int, long, long long`
 - Wertebereich ist compiler-/prozessorabhängig
es gilt: `char ≤ short ≤ int ≤ long ≤ long long`
 - Zeichen werden als Zahlen im ASCII-Code (8 Bit) dargestellt
 - Zeichenketten (Strings) werden als Felder von `char` dargestellt
- Fließkomazahlen: `float, double, long double`
 - Wertebereich/Genauigkeit ist compiler-/prozessorabhängig
- Leerer Datentyp: `void`
 - Wertebereich: \emptyset
 - Einsatz: Funktionen ohne Rückgabewert
- Boolescher Datentyp: `_Bool` (C99)
 - Bedingungsausdrücke (z. B. `if(...)`) sind in C aber vom Typ `int!`
- Durch vorangestellte Typ-Modifier kann die Bedeutung verändert werden
 - vorzeichenbehaftet: `signed`, vorzeichenlos: `unsigned`, konstant: `const`

- ## 3.2 Variablen (2)
- Typ und Bezeichner werden durch eine **Variablen-Deklaration** festgelegt (= dem Compiler bekannt gemacht)
 - ◆ reine Deklarationen werden erst in einem späteren Kapitel benötigt
 - ◆ vorerst beschränken wir uns auf Deklarationen in **Variablen-Definitionen**
 - eine **Variablen-Definition** deklariert eine Variable und reserviert den benötigten Speicherbereich
 - ◆ Beispiele

```
int a1;
float a, b, c, dis;
int anzahl_zeilen=5;
const char trennzeichen = ':';
```

3.1 Primitive Datentypen in C

- Ganzzahlen/Zeichen: `char, short, int, long, long long`
 - Wertebereich ist compiler-/prozessorabhängig
es gilt: `char ≤ short ≤ int ≤ long ≤ long long`
 - Zeichen werden als Zahlen im ASCII-Code (8 Bit) dargestellt
 - Zeichenketten (Strings) werden als Felder von `char` dargestellt

3.2 Variablen

- Variablen werden definiert durch:

- ◆ **Namen** (Bezeichner)
- ◆ Typ
- ◆ zugeordneten Speicherbereich für einen Wert des Typs
Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
- ◆ **Lebensdauer**

- Variablenname

- ◆ Buchstabe oder `_`,
evtl. gefolgt von beliebig vielen Buchstaben, Ziffern oder `_`

3.2 Variablen (3)

- Position von Variablendefinitionen im Programm:

- ◆ nach jeder `"{"`
- ◆ außerhalb von Funktionen
- ◆ ab C99 auch an beliebigen Stellen innerhalb von Funktionen und im Kopf von `for`-Schleifen

- Wert kann bei der Definition initialisiert werden

- Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar

- Lebensdauer ergibt sich aus Programmstruktur

4.4 Kontrollstrukturen — Schleifensteuerung

■ break

◆ bricht die umgebende Schleife bzw. **switch**-Anweisung ab

```
int c;

do {
    if ( (c = getchar()) == EOF ) break;
    putchar(c);
} while ( c != '\n' );
```

■ continue

◆ bricht den aktuellen **Schleifendurchlauf** ab

◆ setzt das Programm mit der Ausführung des Schleifenkopfes fort

5.2 Beispiel Sinusberechnung

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

double sinus (double x)
{
    double summe;
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```

```
int main()
{
    double wert;

    printf("Berechnung des Sinus von ");
    scanf("%lf", &wert);
    printf("sin(%lf) = %lf\n",
           wert, sinus(wert));
    return(0);
}
```

■ beliebige Verwendung von **sinus** in Ausdrücken:

```
y = exp(tau*t) * sinus(f*t);
```

Funktionen

■ Funktion =

Programmstück (Block), das mit einem **Namen** versehen ist, dem zum Ablauf **Parameter** übergeben werden können und das bei Rückkehr einen **Rückgabewert** zurückliefern kann.

■ Funktionen sind die elementaren Bausteine für Programme

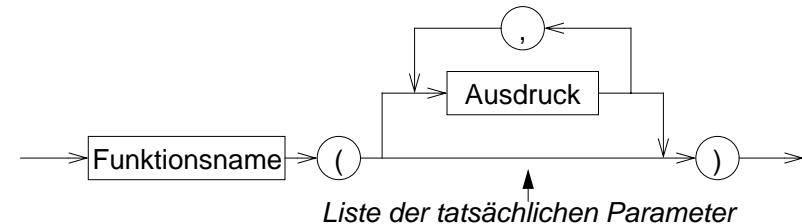
- ➔ verringern die **Komplexität** durch Zerteilen umfangreicher, schwer überblickbarer Aufgaben in kleine Komponenten
- ➔ erlauben die **Wiederverwendung** von Programmkomponenten
- ➔ verbergen **Implementierungsdetails** vor anderen Programmteilen (**Black-Box-Prinzip**)

5.1 Funktionsdefinition

■ Schnittstelle = Ergebnistyp, Name, (formale) Parameter

■ + Implementierung

Funktionsaufruf



Liste der tatsächlichen Parameter

- Die Ausdrücke in der Parameterliste werden ausgewertet, **bevor** in die Funktion gesprungen wird
➔ **tatsächliche Parameter** (*actual parameters*)
- Anzahl und Typen der Ausdrücke in der Liste der tatsächlichen Parameter müssen mit denen der **formalen** Parameter in der Funktionsdefinition übereinstimmen
- Die Auswertungsreihenfolge der Parameterausdrücke ist **nicht** festgelegt

5.4 Regeln

- Funktionen werden global definiert
- **main()** ist eine normale Funktion, die aber automatisch als erste beim Programmstart aufgerufen wird
- rekursive Funktionsaufrufe sind zulässig
 - ▶ eine Funktion darf sich selbst aufrufen

Beispiel Fakultätsberechnung:

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

- Funktionen müssen **deklariert** sein, bevor sie aufgerufen werden
 - = Rückgabetyp und Parametertypen müssen bekannt sein
 - ◆ durch eine Funktionsdefinition ist die Funktion automatisch auch deklariert
- wurde eine verwendete Funktion vor ihrer Verwendung nicht deklariert, wird automatisch angenommen
 - ▶ Funktionswert vom Typ **int**
 - ▶ 1. Parameter vom Typ **int**
 - ▶ **schlechter Programmierstil → fehleranfällig**
 - ▶ **ab C99 nicht mehr zulässig**

- ## 5.5 Funktionsdeklaration
- soll eine Funktion vor ihrer Definition verwendet werden, kann sie durch eine **Deklaration** bekannt gemacht werden (Prototyp)
 - ◆ Syntax:

Typ Name (Liste formaler Parameter);

▶ Parameternamen können weggelassen werden, die Parametertypen müssen aber angegeben werden!

◆ Beispiel:

double sinus(double);

5.6 Funktionsdeklarationen — Beispiel

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

double sinus(double);
/* oder: double sinus(double x); */

int main()
{
    double wert;

    printf("Berechnung des Sinus von ");
    scanf("%lf", &wert);
    printf("sin(%lf) = %lf\n",
           wert, sinus(wert));
    return(0);
}

double sinus (double x)
{
    double summe,
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```

5.7 Parameterübergabe an Funktionen

- allgemein in Programmiersprachen vor allem zwei Varianten:
 - call by value (wird in C verwendet)
 - call by reference (wird in C **nicht** verwendet)
- call-by-value: Es wird eine Kopie des tatsächlichen Parameters an die Funktion übergeben
 - die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
 - die Funktion kann den Wert des formalen Parameters (also die Kopie!) ändern, ohne dass dies Auswirkungen auf den Wert des tatsächlichen Parameters beim Aufrufer hat
 - die Funktion kann über einen Parameter dem Aufrufer keine Ergebnisse mitteilen

- Makros ermöglichen einfache textuelle Ersetzungen (parametrierbare Makros werden später behandelt)
- ein Makro wird durch die **#define**-Anweisung definiert
- Syntax:

```
#define Makroname Ersatztext
```

- eine Makrodefinition bewirkt, dass der Präprozessor im nachfolgenden Text der C-Quelle alle Vorkommen von **Makroname** durch **Ersatztext** ersetzt

- Beispiel:

```
#define EOF -1
```

C-Präprozessor

- bevor eine C-Quelle dem C-Compiler übergeben wird, wird sie durch einen Makro-Präprozessor bearbeitet
- Anweisungen an den Präprozessor werden durch ein **#**-Zeichen am Anfang der Zeile gekennzeichnet
- die Syntax von Präprozessoranweisungen ist unabhängig vom Rest der Sprache
- Präprozessoranweisungen werden nicht durch **;** abgeschlossen!
- wichtigste Funktionen:
 - #define** Definition von Makros
 - #include** Einfügen von anderen Dateien

6.2 Einfügen von Dateien

- **#include** fügt den Inhalt einer anderen Datei in eine C-Quelldatei ein
- Syntax:


```
#include < Dateiname >
oder
#include "Dateiname "
```
- mit **#include** werden Header-Dateien mit Daten, die für mehrere Quelldateien benötigt werden, einkopiert
 - Deklaration von Funktionen, Strukturen, externen Variablen
 - Definition von Makros
- wird **Dateiname** durch **< >** geklammert, wird eine **Standard-Header-Datei** einkopiert
- wird **Dateiname** durch **" "** geklammert, wird eine Header-Datei des Benutzers einkopiert (vereinfacht dargestellt!)

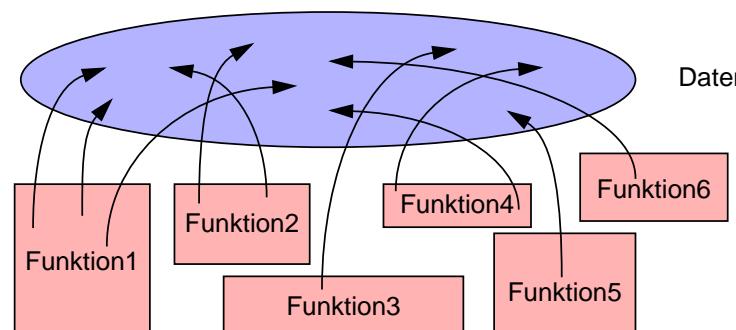
Programmstruktur & Module

7.1 Softwaredesign

- Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
 - Verschiedene Design-Methoden
 - ◆ Top-down Entwurf / Prozedurale Programmierung
 - traditionelle Methode
 - bis Mitte der 80er Jahre fast ausschließlich verwendet
 - an Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert
 - ◆ Objekt-orientierter Entwurf
 - moderne, sehr aktuelle Methode
 - Ziel: Bewältigung sehr komplexer Probleme
 - auf Programmiersprachen wie C++, Smalltalk oder Java ausgerichtet

7.2 Top-down Entwurf (2)

- Problem:
Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten
 - Gefahr:
Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten



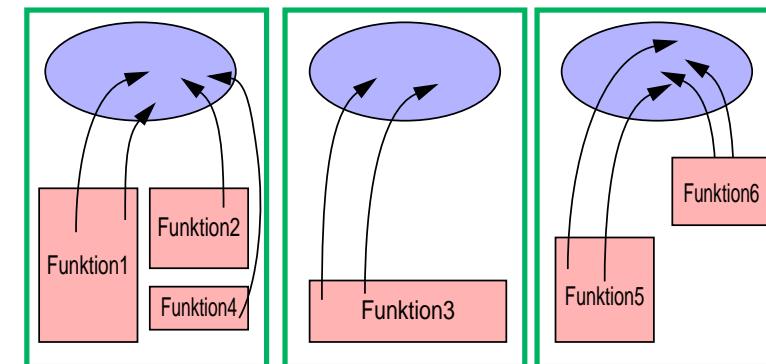
7.2 Top-down Entwurf

- Zentrale Fragestellung
 - ◆ was ist zu tun?
 - ◆ in welche Teilaufgaben lässt sich die Aufgabe untergliedern?
 - Beispiel: Rechnung für Kunden ausgeben
 - Rechnungspositionen zusammenstellen
 - Lieferungsposten einlesen
 - Preis für Produkt ermitteln
 - Mehrwertsteuer ermitteln
 - Rechnungspositionen addieren
 - Positionen formatiert ausdrucken

[7.2](#) Top-down Entwurf (3) — Modul-Bildung

- Lösung:
Gliederung von Datenbeständen zusammen mit Funktionen, die darauf operieren

Modu



7.3 Module in C

- Teile eines C-Programms können auf mehrere .c-Dateien (C-Quelldateien) verteilt werden
- Logisch zusammengehörende Daten und die darauf operierenden Funktionen sollten jeweils zusammengefasst werden
 - ▶ **Modul**
- Jede C-Quelldatei kann separat übersetzt werden (Option **-c**)
 - ▶ Zwischenergebnis der Übersetzung wird in einer .o-Datei abgelegt
- Das Kommando **cc** kann mehrere .c-Dateien übersetzen und das Ergebnis — zusammen mit .o-Dateien — binden:

```
% cc -c prog.c          (erzeugt Datei prog.o)
% cc -c f1.c            (erzeugt Datei f1.o)
% cc -c f2.c f3.c       (erzeugt f2.o und f3.o)
```

- ## 7.4 Gültigkeit von Namen
- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
 - Mehrere Stufen
 1. Global im gesamten Programm
(über Modul- und Funktionsgrenzen hinweg)
 2. Global in einem Modul
(auch über Funktionsgrenzen hinweg)
 3. Lokal innerhalb einer Funktion
 4. Lokal innerhalb eines Blocks
 - Überdeckung bei Namensgleichheit
 - ▶ eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
 - ▶ eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

7.3 Module in C (2)

- !!! .c-Quelldateien auf keinen Fall mit Hilfe der #include-Anweisung in andere Quelldateien einkopieren
- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie **deklariert** werden
 - ▶ Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefasst
 - ◆ Header-Dateien werden mit der #include-Anweisung des Präprozessors in C-Quelldateien einkopiert
 - ◆ der Name einer Header-Datei endet immer auf .h

- ## 7.4 Gültigkeit von Namen
- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
 - Mehrere Stufen
 1. Global im gesamten Programm
(über Modul- und Funktionsgrenzen hinweg)
 2. Global in einem Modul
(auch über Funktionsgrenzen hinweg)
 3. Lokal innerhalb einer Funktion
 4. Lokal innerhalb eines Blocks
 - Überdeckung bei Namensgleichheit
 - ▶ eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
 - ▶ eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

7.3 Module in C (2)

- !!! .c-Quelldateien auf keinen Fall mit Hilfe der #include-Anweisung in andere Quelldateien einkopieren
- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie **deklariert** werden
 - ▶ Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefasst
 - ◆ Header-Dateien werden mit der #include-Anweisung des Präprozessors in C-Quelldateien einkopiert
 - ◆ der Name einer Header-Datei endet immer auf .h

- ## 7.4 Gültigkeit von Namen
- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
 - Mehrere Stufen
 1. Global im gesamten Programm
(über Modul- und Funktionsgrenzen hinweg)
 2. Global in einem Modul
(auch über Funktionsgrenzen hinweg)
 3. Lokal innerhalb einer Funktion
 4. Lokal innerhalb eines Blocks
 - Überdeckung bei Namensgleichheit
 - ▶ eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
 - ▶ eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

7.3 Module in C (2)

- !!! .c-Quelldateien auf keinen Fall mit Hilfe der #include-Anweisung in andere Quelldateien einkopieren
- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie **deklariert** werden
 - ▶ Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefasst
 - ◆ Header-Dateien werden mit der #include-Anweisung des Präprozessors in C-Quelldateien einkopiert
 - ◆ der Name einer Header-Datei endet immer auf .h

- ## 7.4 Gültigkeit von Namen
- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
 - Mehrere Stufen
 1. Global im gesamten Programm
(über Modul- und Funktionsgrenzen hinweg)
 2. Global in einem Modul
(auch über Funktionsgrenzen hinweg)
 3. Lokal innerhalb einer Funktion
 4. Lokal innerhalb eines Blocks
 - Überdeckung bei Namensgleichheit
 - ▶ eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
 - ▶ eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

7.3 Module in C (2)

- !!! .c-Quelldateien auf keinen Fall mit Hilfe der #include-Anweisung in andere Quelldateien einkopieren
- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie **deklariert** werden
 - ▶ Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefasst
 - ◆ Header-Dateien werden mit der #include-Anweisung des Präprozessors in C-Quelldateien einkopiert
 - ◆ der Name einer Header-Datei endet immer auf .h

- ## 7.4 Gültigkeit von Namen
- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
 - Mehrere Stufen
 1. Global im gesamten Programm
(über Modul- und Funktionsgrenzen hinweg)
 2. Global in einem Modul
(auch über Funktionsgrenzen hinweg)
 3. Lokal innerhalb einer Funktion
 4. Lokal innerhalb eines Blocks
 - Überdeckung bei Namensgleichheit
 - ▶ eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
 - ▶ eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

7.5 Globale Variablen (2)

■ Probleme mit globalen Variablen

- ◆ Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren
- ◆ Funktionen können Variablen ändern, ohne dass der Aufrufer dies erwartet (Seiteneffekte)
- ◆ Programme sind schwer zu pflegen, weil bei Änderungen der Variablen erst alle Programmteile, die sie nutzen gesucht werden müssen

► globale Variablen möglichst vermeiden

7.6 Einschränkung der Gültigkeit auf ein Modul

- Zugriff auf eine globale Variable oder Funktion kann auf das Modul (= die Datei) beschränkt werden, in der sie definiert wurde
 - Schlüsselwort **static** vor die Definition setzen
 - Beispiel: **static int a;**
 - **extern**-Deklarationen in anderen Modulen sind nicht möglich
- die **static**-Variablen bilden zusammen den Zustand eines Moduls, die Funktionen des Moduls operieren auf diesem Zustand
- Hilfsfunktionen innerhalb eines Moduls, die nur von den Modulfunktionen benötigt werden, sollten immer static definiert werden
 - sie werden dadurch nicht Bestandteil der Modulschnittstelle (= des "Vertrags" mit den Modulbenutzern)
- !!! das Schlüsselwort **static** gibt es auch bei lokalen Variablen (mit anderer Bedeutung! - dort jeweils *kursiv* geschrieben)

7.5 Globale Funktionen

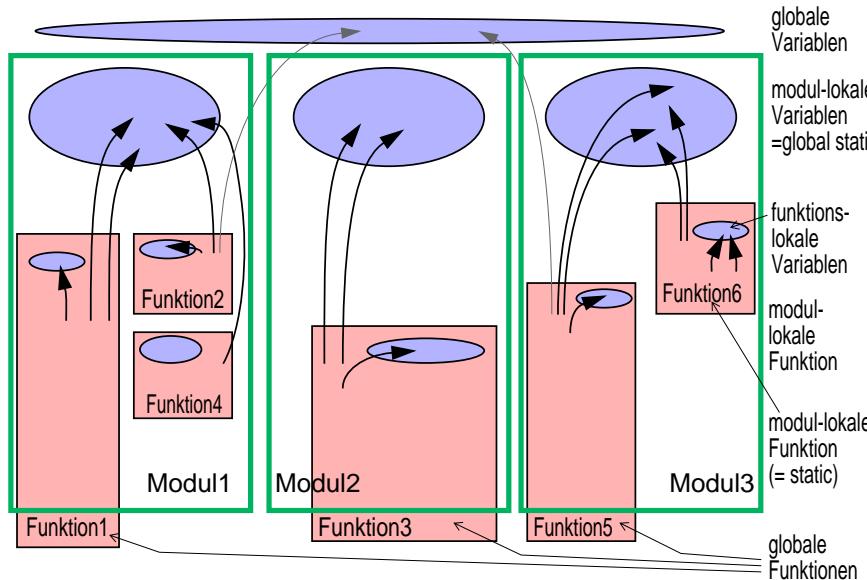
- Funktionen sind generell global
(es sei denn, die Erreichbarkeit wird explizit auf das Modul begrenzt)
- Funktionen aus anderen Modulen müssen ebenfalls vor dem ersten Aufruf **deklariert** werden
(= Typ, Name und Parametertypen bekanntmachen)
- Das Schlüsselwort **extern** ist bei einer Funktionsdeklaration nicht notwendig
- Beispiele:


```
double sinus(double);
float power(float, int);
```
- Globale Funktionen (und soweit vorhanden die globalen Daten) bilden die äußere Schnittstelle eines Moduls
 - "vertragliche" Zusicherung an den Benutzer des Moduls

7.7 Lokale Variablen

- Variablen, die innerhalb einer Funktion oder eines Blocks definiert werden, sind lokale Variablen
- bei Namensgleichheit zu globalen Variablen oder lokalen Variablen eines umgebenden Blocks gilt die jeweils letzte Definition
- lokale Variablen sind außerhalb des Blocks, in dem sie definiert wurden, nicht zugreifbar und haben dort keinen Einfluss auf die Zugreifbarkeit von Variablen

7.8 Gültigkeitsbereiche — Übersicht



7.9 Lebensdauer von Variablen (2)

auto-Variablen

- Alle lokalen Variablen sind automatic-Variablen
 - der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
 - der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr sicher verfügbar!
- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
 - die Initialisierung wird bei jedem Eintritt in den Block wiederholt
 - !!! wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert (= irgendwas)**

7.9 Lebensdauer von Variablen (3)

static-Variablen

- Der Speicher für alle globalen Variablen ist generell von Programmstart bis Programmende reserviert
- Lokale Variablen erhalten bei Definition mit dem Schlüsselwort **static** eine **Lebensdauer über die gesamte Programmausführung hinweg**
 - der Inhalt bleibt bei Verlassen des Blocks erhalten und ist bei einem erneuten Eintreten in den Block noch verfügbar
 - !!! Das Schlüsselwort **static** hat bei globalen Variablen eine völlig andere Bedeutung (Einschränkung des Zugriffs auf das Modul)**
- Static-Variablen können durch beliebige konstante Ausdrücke initialisiert werden
 - die Initialisierung wird nur einmal beim Programmstart vorgenommen (auch bei lokalen Variablen!)
 - erfolgt keine explizite Initialisierung, wird automatisch mit 0 vorbelegt

Zeiger(-Variablen)

8.1 Einordnung

■ Konstante:

Bezeichnung für einen Wert

'a' ≡ 0110 0001

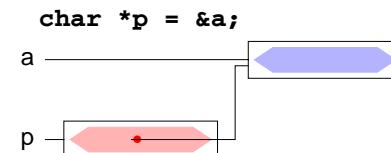
■ Variable:

Bezeichnung für ein Datenobjekt



■ Zeiger-Variable (Pointer):

Bezeichnung einer Referenz auf ein Datenobjekt



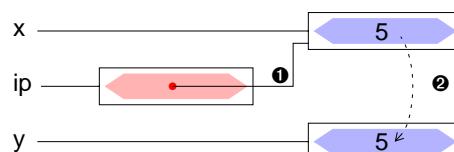
8.3 Definition von Zeigervariablen

■ Syntax:

Typ *Name ;

▲ Beispiele

```
int x = 5;
int *ip;
int y;
ip = &x; ①
y = *ip; ②
```



Zeiger(-Variablen)

8.1 Einordnung

■ Konstante:

Bezeichnung für einen Wert

- Eine Zeigervariable (**pointer**) enthält als Wert die Adresse einer anderen Variablen
 - der Zeiger verweist auf die Variable
- Über diese Adresse kann man **indirekt** auf die Variable zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
 - Funktionen können (indirekt) ihre Aufrufparameter verändern (**call-by-reference**)
 - dynamische Speicherverwaltung
 - effizientere Programme
- Aber auch Nachteile!
 - Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variable zugreifen?)
 - häufigste Fehlerquelle bei C-Programmen

8.3 Definition von Zeigervariablen

■ Syntax:

Typ *Name ;

▲ Beispiele

```
int x = 5;
int *ip;
int y;
ip = &x; ①
y = *ip; ②
```

Zeiger(-Variablen)

8.2 Überblick

■ Konstante:

Bezeichnung für einen Wert

■ Variable:

Bezeichnung für ein Datenobjekt

■ Zeiger-Variable (Pointer):

Bezeichnung einer Referenz auf ein Datenobjekt

■ Adressoperator &

&x

der unäre Adress-Operator liefert eine Referenz auf den Inhalt der Variablen (des Objekts) **x**

■ Verweisoperator *

***x**

der unäre Verweisoperator ***** ermöglicht den Zugriff auf den Inhalt der Variablen (des Objekts), auf die der Zeiger **x** verweist

★ Unterschied des Symbols *

in einer Variablendefinition und in einem Ausdruck

► `int *ip;` * in einer Variablendefinition:
ip ist eine Variable vom Typ (**int ***),
eine Variable die auf ein Objekt vom Typ (**int**) verweist

► `y = *ip;` * als Operator in einem Ausdruck:
ip ist eine Variable, die auf ein Objekt vom Typ (**int**) verweist,
der Ausdruck `*ip` ermittelt den Inhalt dieses Objekts, also den int-Wert
→ das Ergebnis des Ausdrucks `*ip` ist ein Wert vom Typ (**int**)

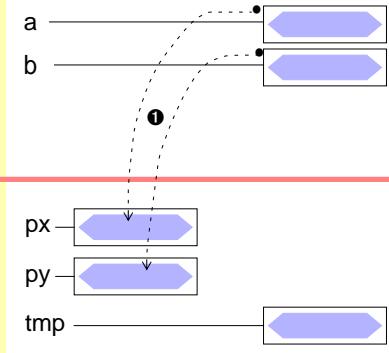
8.5 Zeiger als Funktionsargumente

- Parameter werden in C *by-value* übergeben
- die aufgerufene Funktion kann den tatsächlichen Parameter beim Aufrufer nicht verändern
- auch Zeiger werden *by-value* übergeben, d. h. die Funktion erhält lediglich eine Kopie des Adressverweises
- über diesen Verweis kann die Funktion jedoch mit Hilfe des *-Operators auf die zugehörige Variable zugreifen und sie verändern
→ *call-by-reference*

8.5 Zeiger als Funktionsargumente (2)

- Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b); ①
    ...
}
```



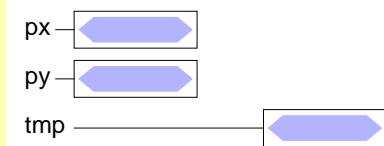
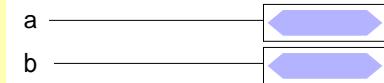
8.5 Zeiger als Funktionsargumente (2)

- Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}
```



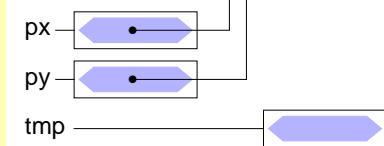
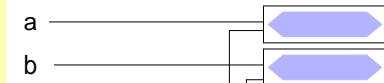
8.5 Zeiger als Funktionsargumente (2)

- Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}
```



8.5 Zeiger als Funktionsargumente (2)

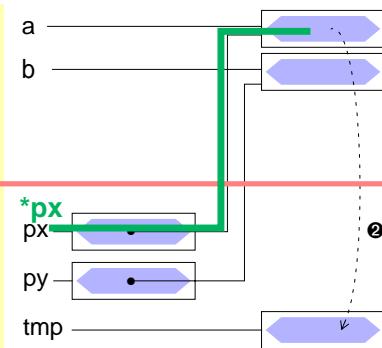
Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ②
    *px = *py;
    *py = tmp;

}
```



8.5 Zeiger als Funktionsargumente (2)

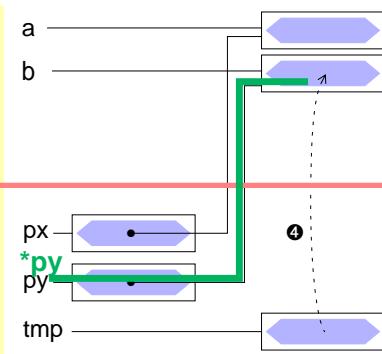
Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ④
    *px = *py;
    *py = tmp; ④

}
```



8.5 Zeiger als Funktionsargumente (2)

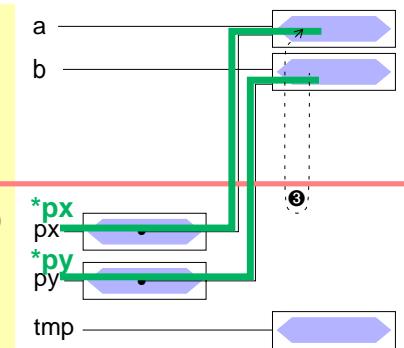
Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ③
    *px = *py;
    *py = tmp;

}
```



8.5 Zeiger als Funktionsargumente (2)

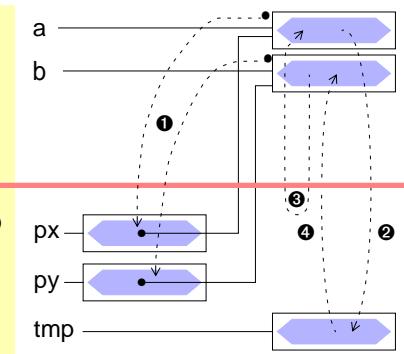
Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b); ①
    ...
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ②
    *px = *py; ③
    *py = tmp; ④

}
```



8.6 Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
 - Adresse einer Struktur mit &-Operator zu bestimmen
- Beispiele

```
struct person stud1;
struct person *pstud;
pstud = &stud1;           /* ⇒ pstud → stud1 */
```

- Besondere Bedeutung zum Aufbau verketteter Strukturen

8.7 Zusammenfassung

- Variable


```
int a;
```
- Struktur


```
struct s{int a; char c;};
struct s s1 = {2, 'a'};
```
- Zeiger


```
int *p = &a;
```
- Zeiger auf Struktur


```
struct s *sp = &s1;
```

- Zugriff auf Strukturkomponenten über einen Zeiger
 - *-Operator liefert die Struktur
 - .-Operator zum Zugriff auf Komponente
 - Operatorenvorrang beachten

⇒ `(*pstud).alter = 21;`

nicht so gut leserlich!

- Syntaktische Verschönerung

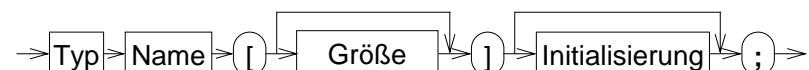
⇒ `->-Operator`

`pstud->alter = 21;`

Felder

9.1 Eindimensionale Felder

- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefasst werden
- bei der Definition wird die Größe des Felds angegeben
 - Größe muss eine Konstante sein
 - ab C99 bei lokalen Feldern auch zur Laufzeit berechnete Werte zulässig
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes

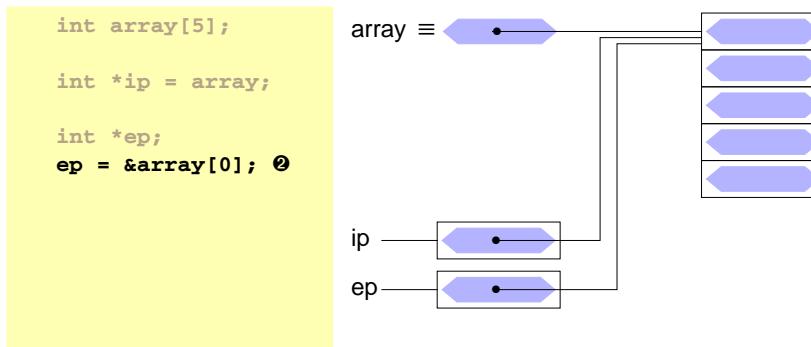


- Beispiele:

```
int x[5];
double f[20];
```

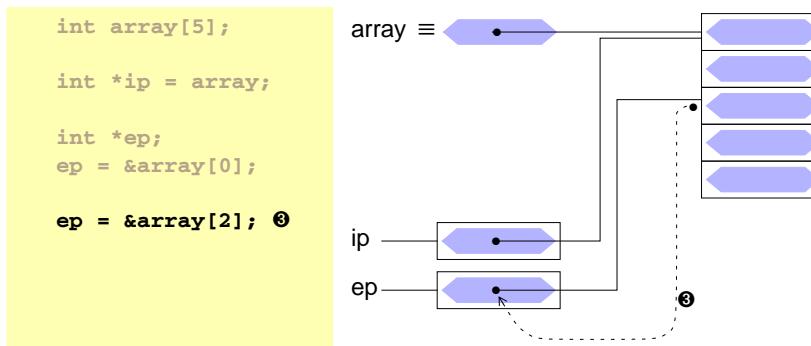

Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
`array ≡ &array[0]`
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



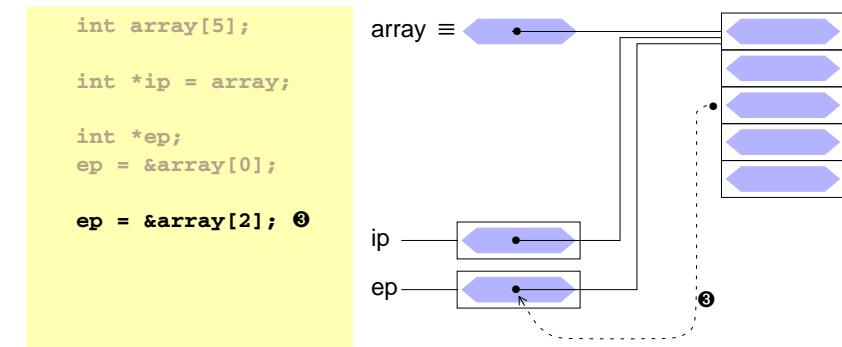
Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
`array ≡ &array[0]`
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



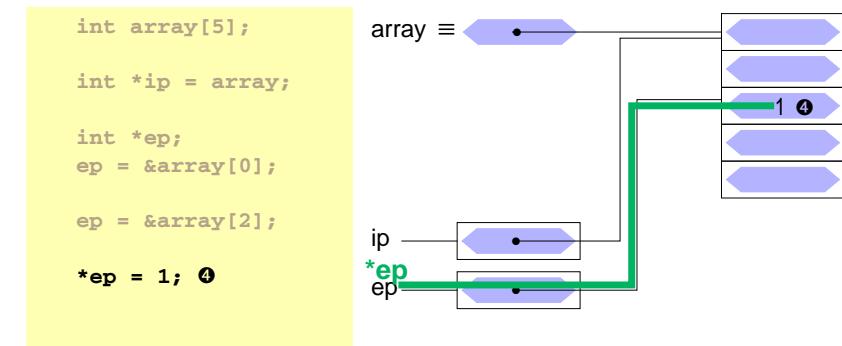
Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
`array ≡ &array[0]`
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



Zeiger und Felder

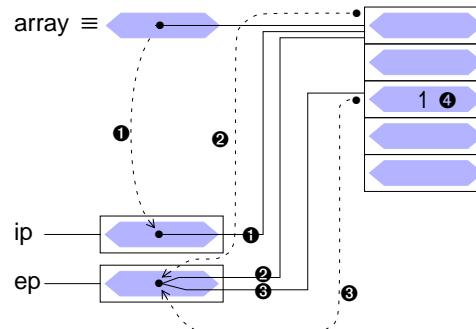
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
`array ≡ &array[0]`
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
`array ≡ &array[0]`
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];
int *ip = array; ①
int *ep;
ep = &array[0]; ②
ep = &array[2]; ③
*ep = 1; ④
```



10.2 Zeigerarithmetik und Felder

- Ein Feldname ist eine Konstante für die Adresse des Feldanfangs
 - Feldname ist ein ganz normaler Zeiger
 - Operatoren für Zeiger anwendbar (*, [])
 - aber keine Variable → keine Modifikationen erlaubt
 - keine Zuweisung, kein ++, --, +=, ...
- In Kombination mit Zeigerarithmetik lässt sich in C jede Feldoperation auf eine äquivalente Zeigeroperation abbilden
 - für `int, array[N], *ip = array;` mit $0 \leq i < N$ gilt:

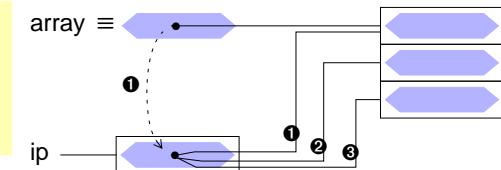

```
array ≡ &array[0] ≡ ip           ≡ &ip[0]
*array ≡ array[0] ≡ *ip           ≡ ip[0]
*(array + i) ≡ array[i] ≡ *(ip + 1) ≡ ip[i]
array++ ≠ ip++
```

Fehler: array ist konstant!
- Umgekehrt können Zeigeroperationen auch durch Feldoperationen dargestellt werden (nur der Feldname darf nicht verändert werden)

- ++ -Operator: Inkrement = nächstes Objekt

```
int array[3];
int *ip = array; ①

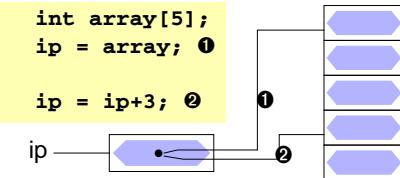
ip++; ②
ip++; ③
```



- -Operator: Dekrement = vorheriges Objekt

- +, -
 - Addition und Subtraktion von Zeigern und ganzzahligen Werten.

Dabei wird immer die Größe des Objekttyps berücksichtigt!



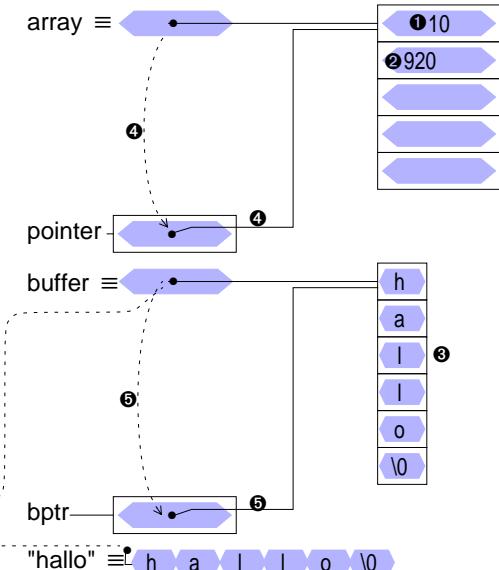
!!! Achtung: Assoziativität der Operatoren beachten

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;

① array[0] = 10;
② array[1] = 920;
③ strcpy(buffer, "hallo");
④ pointer = array;
⑤ bptr = buffer;
```

Fomale Parameter
der Funktion strcpy

ziel → **hallo** ≡ h a l l o \0
quelle →



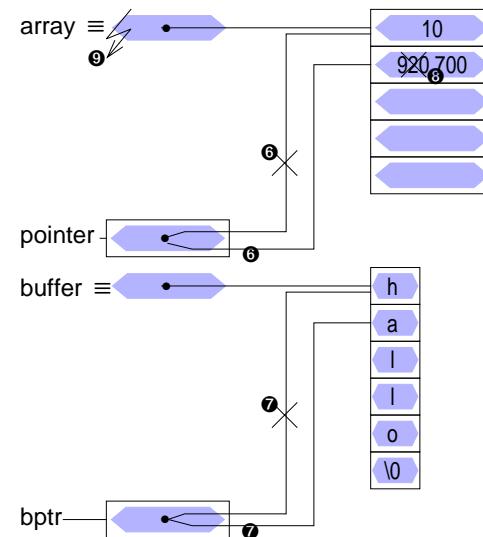
10.2 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;

① array[0] = 10;
② array[1] = 920;
③ strcpy(buffer,"hallo");
④ pointer = array;
⑤ bptr = buffer;

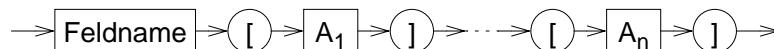
⑥ pointer++;
⑦ bptr++;
⑧ *pointer = 700;

⑨ array++;
```



10.4 Zugriffe auf Feldelemente bei mehrdim. Feldern

- Indizierung:



wobei: $0 \leq A_i < \text{Größe der Dimension } i \text{ des Feldes}$
 $n = \text{Anzahl der Dimensionen des Feldes}$

- Beispiel:

```
int feld[5][8];
feld[2][3] = 10;
```

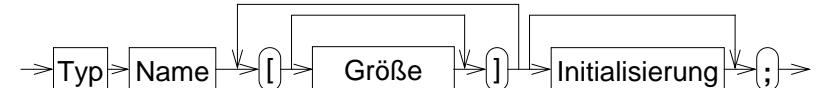
◆ ist äquivalent zu:

```
int feld[5][8];
int *f1;
f1 = (int*)feld;
f1[2*8 + 3] = 10;
oder
*(f1 + (2*8 + 3)) = 10;
```

10.3 Mehrdimensionale Felder

- neben eindimensionalen Felder kann man auch mehrdimensionale Felder vereinbaren

- Definition eines mehrdimensionalen Feldes



- Beispiel:

```
int matrix[4][4];
```

- Realisierung:

- in der internen Speicherung werden die Feldelemente zeilenweise hintereinander im Speicher abgelegt
- Felddefinition: `int f[2][2];`
 Ablage der Elemente: `f[0][0], f[0][1], f[1][0], f[1][1]`
`f` ist ein Zeiger auf `f[0][0]`

- ein mehrdimensionales Feld kann - wie ein eindimensionales Feld - durch eine Liste von konstanten Werten, die durch Komma getrennt sind, initialisiert werden

- wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Größe des Feldes

- Beispiel:

```
int feld[3][4] = {
    { 1, 3, 5, 7 }, /* feld[0][0-3] */
    { 2, 4, 6 } /* feld[1][0-2] */
};

feld[1][3] und feld[2][0-3] werden in dem Beispiel mit 0 initialisiert!
```

Dynamische Speicherverwaltung

- Felder können (mit einer Ausnahme im C99-Standard) nur mit statischer Größe definiert werden
- Wird die Größe eines Feldes erst zur Laufzeit des Programms bekannt, kann der benötigte Speicherbereich dynamisch vom Betriebssystem angefordert werden: Funktion **malloc**
 - Ergebnis: Zeiger auf den Anfang des Speicherbereichs
 - Zeiger kann danach wie ein Feld verwendet werden ([]-Operator)

void *malloc(size_t size)

```
int *feld;
int groesse;
...
feld = (int *) malloc(groesse * sizeof(int));
if (feld == NULL) {
    perror("malloc feld");
    exit(1);
}
for (i=0; i<groesse; i++) { feld[i] = 8; }
...
```

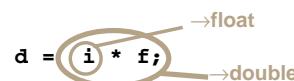
cast-Operator sizeof-Operator

Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```



- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)

◆ Syntax:

(Typ) Variable

Beispiele:

```
(int) a      (int *) a
(float) b     (char *) a
```

◆ Beispiel:

```
feld = (int *) malloc(groesse * sizeof(int));
...
```

malloc liefert Ergebnis vom Typ (void *)
cast-Operator macht daraus den Typ (int *)

Dynamische Speicherverwaltung (2)

- Dynamisch angeforderte Speicherbereiche können mit der **free**-Funktion wieder freigegeben werden

void free(void *ptr)

```
double *dfeld;
int groesse;
...
dfeld = (double *) malloc(groesse * sizeof(double));
...
free(dfeld);
```

- die Schnittstellen der Funktionen sind in in der include-Datei stdlib.h definiert

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

Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```



- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)

◆ Syntax:

(Typ) Variable

Beispiele:

```
(int) a      (int *) a
(float) b     (char *) a
```

◆ Beispiel:

```
feld = (int *) malloc(groesse * sizeof(int));
...
```

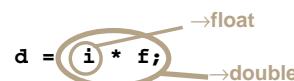
malloc liefert Ergebnis vom Typ (void *)
cast-Operator macht daraus den Typ (int *)

Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```



- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)

◆ Syntax:

(Typ) Variable

Beispiele:

```
(int) a      (int *) a
(float) b     (char *) a
```

◆ Beispiel:

```
feld = (int *) malloc(groesse * sizeof(int));
...
```

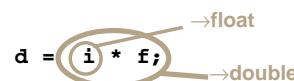
malloc liefert Ergebnis vom Typ (void *)
cast-Operator macht daraus den Typ (int *)

Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```



- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)

◆ Syntax:

(Typ) Variable

Beispiele:

```
(int) a      (int *) a
(float) b     (char *) a
```

◆ Beispiel:

```
feld = (int *) malloc(groesse * sizeof(int));
...
```

malloc liefert Ergebnis vom Typ (void *)
cast-Operator macht daraus den Typ (int *)

Eindimensionale Felder als Funktionsparameter

- ganze Felder können in C **nicht by-value** übergeben werden
- wird einer Funktion ein Feldname als Parameter übergeben, wird damit der Zeiger auf das erste Element "by value" übergeben
 - die Funktion kann über den formalen Parameter (=Kopie des Zeigers) in gleicher Weise wie der Aufrufer auf die Feldelemente zugreifen (und diese verändern!)
- bei der Deklaration des formalen Parameters wird die Feldgröße weggelassen
 - die Feldgröße ist automatisch durch den tatsächlichen Parameter gegeben
 - die Funktion kennt die Feldgröße damit nicht
 - ggf. ist die Feldgröße über einen weiteren **int**-Parameter der Funktion explizit mitzuteilen
 - die Länge von Zeichenketten in **char**-Feldern kann normalerweise durch Suche nach dem \0-Zeichen bestimmt werden

Eindimensionale Felder als Funktionsparameter (3)

■ Beispiel 1: Bestimmung der Länge einer Zeichenkette (*String*)

```
int strlen(const char string[])
{
    int i=0;
    while (string[i] != '\0') ++i;
    return(i);
}
```

Eindimensionale Felder als Funktionsparameter (2)

- wird ein Feldparameter als **const** deklariert, können die Feldelemente innerhalb der Funktion nicht verändert werden
- Funktionsaufruf und Deklaration der formalen Parameter am Beispiel eines **int**-Feldes:


```
int a, b;
int feld[20];
func(a, feld, b);
...
int func(int p1, int p2[], int p3);
oder:
int func(int p1, int *p2, int p3);
```
- die Parameter-Deklarationen **int p2[]** und **int *p2** sind vollkommen äquivalent!
 - im Unterschied zu einer Variablendefinition


```
int f[] = {1, 2, 3}; // initialisiertes Feld mit 3 Elementen
int f1[];           // ohne Initialisierung oder Dimension nicht erlaubt!
int *p;             // Zeiger auf einen int
```

Eindimensionale Felder als Funktionsparameter (4)

■ Beispiel 2: Konkateneiere Strings

```
void strcat(char to[], const char from[])
{
    int i=0, j=0;
    while (to[i] != '\0') i++;
    while ( (to[i++] = from[j++]) != '\0' )
        ;
}
```

◆ Funktionsaufruf mit Feld-Parametern

- als tatsächlicher Parameter beim Funktionsaufruf wird einfach der Feldname angegeben

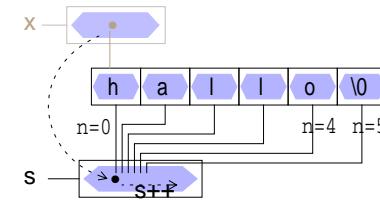
```
char s1[50] = "text1";
char s2[] = "text2";
strcat(s1, s2); /* → s1= "text1text2" */
strcat(s1, "text3"); /* → s1= "text1text2text3" */
```

Zeiger, Felder und Zeichenketten

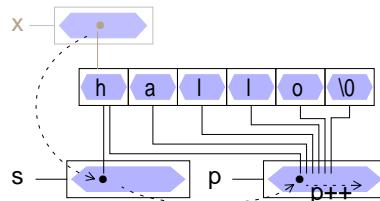
- Zeichenketten sind Felder von Einzelzeichen (**char**), die in der internen Darstellung durch ein '\0'-Zeichen abgeschlossen sind

- Beispiel: Länge eines Strings ermitteln — Aufruf **strlen(x);**

```
/* 1. Version */
int strlen(const char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



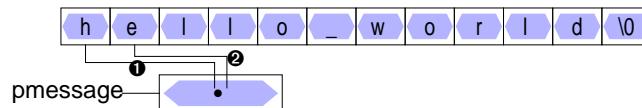
```
/* 2. Version */
int strlen(const char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return(p-s);
}
```



Zeiger, Felder und Zeichenketten (3)

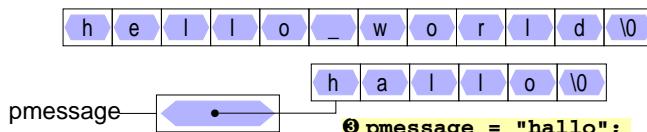
- wird eine Zeichenkette zur Initialisierung eines **char**-Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

```
char *pmassage = "hello world";
```



```
pmassage++; ②
printf("%s", pmassage); /* gibt "ello world" aus */
```

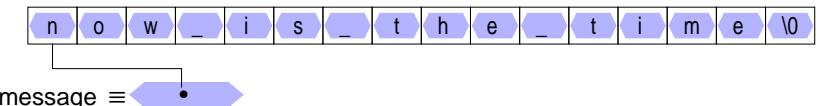
→ wird dieser Zeiger überschrieben, ist die Zeichenkette nicht mehr adressierbar!



Zeiger, Felder und Zeichenketten (2)

- wird eine Zeichenkette zur Initialisierung eines char-Feldes verwendet, ist der Feldname ein konstanter Zeiger auf den Anfang der Zeichenkette

```
char amessage[] = "now is the time";
```

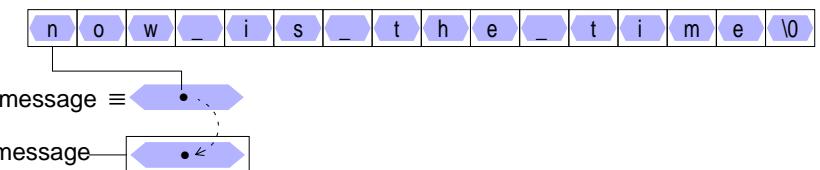


Zeiger, Felder und Zeichenketten (4)

- die Zuweisung eines **char**-Zeigers oder einer Zeichenkette an einen **char**-Zeiger bewirkt kein Kopieren von Zeichenketten!

```
pmassage = amessage;
```

weist dem Zeiger pmassage lediglich die Adresse der Zeichenkette "now is the time" zu



- wird eine Zeichenkette als tatsächlicher Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

Zeiger, Felder und Zeichenketten (5)

- Zeichenketten kopieren

```
/* 1. Version */
void strcpy(char to[], const char from[])
{
    int i=0;
    while ( (to[i] = from[i]) != '\0' )
        i++;
}

/* 2. Version */
void strcpy(char *to, const char *from)
{
    while ( (*to = *from) != '\0' )
        to++, from++;
}

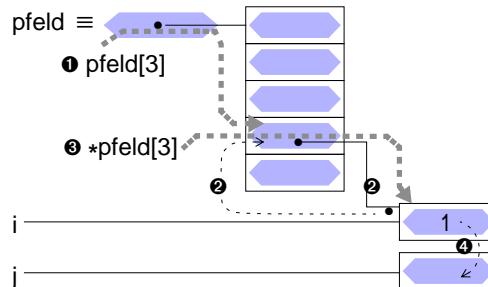
/* 3. Version */
void strcpy(char *to, const char *from)
{
    while ( *to++ = *from++ )
        ;
}
```

Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden

- Deklaration

```
int *pfeld[5];
int i = 1
int j;
```



- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i; ②
```

①

- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

```
j = *pfeld[3]; ④
```

③

④

Zeiger, Felder und Zeichenketten (6)

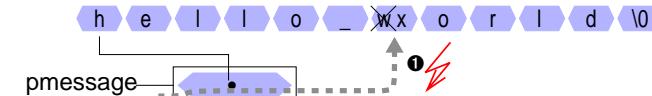
- in ANSI-C können Zeichenketten in nicht-modifizierbaren Speicherbereichen angelegt werden (je nach Compiler)

→ Schreiben in Zeichenketten
(Zuweisungen über dereferenzierte Zeiger)
kann zu Programmabstürzen führen!

- Beispiel:

```
strcpy("zu ueberschreiben", "reinschreiben");
```

```
char *pmassage = "hello world";
```



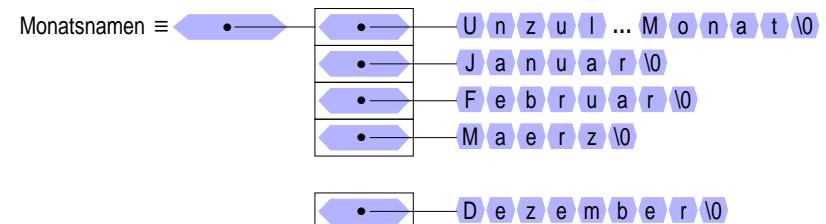
aber! `char amessage[] = "hello world"; ok!`

Felder von Zeigern (2)

- Beispiel: Definition und Initialisierung eines Zeigerfeldes:

```
char *month_name(int n)
{
    static char *Monatsnamen[] = {
        "Unzulaessiger Monat",
        "Januar",
        ...
        "Dezember"
    };

    return ( (n<0 || n>12) ? Monatsnamen[0] : Monatsnamen[n] );
}
```



Argumente aus der Kommandozeile

- beim Aufruf eines Kommandos können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion **main()** durch zwei Aufrufparameter ermöglicht:

```
int main (int argc, char *argv[])
{
    ...
}
```

oder

```
int main (int argc, char **argv)
{
    ...
}
```

- der Parameter **argc** enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter **argv** ist ein Feld von Zeiger auf die einzelnen Argumente (Zeichenketten)
- der Kommandoname wird als erstes Argument übergeben (**argv[0]**)

17.2 Zugriff — Beispiel: Ausgeben aller Argumente (1)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int main (int argc, char *argv[])
{
    int i;
    for ( i=1; i<argc; i++) {
        printf("%s%c", argv[i],
               (i < argc-1) ? ' ' : '\n' );
    }
    ...
}
```

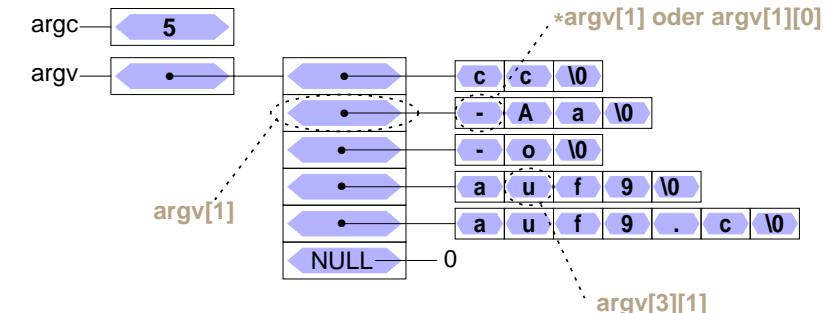
1. Version

17.1 Datenaufbau

Kommando: **cc -Aa -o auf9 auf9.c**

Datei cc.c:

```
...
main(int argc, char *argv[])
{
    ...
}
```



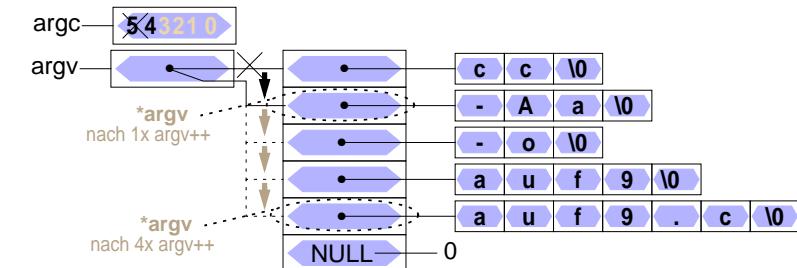
17.2 Zugriff — Beispiel: Ausgeben aller Argumente (2)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int main (int argc, char **argv)
{
    while (--argc > 0) {
        argv++;
        printf("%s%c", *argv, (argc>1) ? ' ' : '\n' );
    }
    ...
}
```

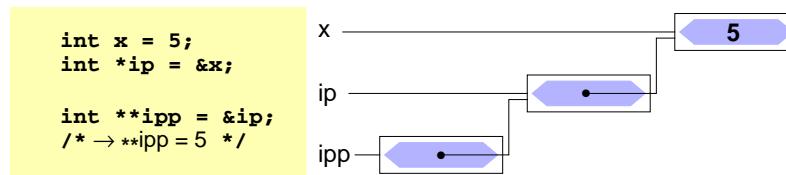
2. Version

linksseitiger Operator:
erst dekrementieren,
dann while-Bedingung prüfen
→ Schleife läuft für argc=4,3,2,1



Zeiger auf Zeiger

- ein Zeiger kann auf eine Variable verweisen, die ihrerseits ein Zeiger ist



- wird vor allem bei der Parameterübergabe an Funktionen benötigt, wenn ein Zeiger "call by reference" übergeben werden muss (z. B. swap-Funktion für Zeiger)

Strukturen

- Beispiele

```
struct student {
    char nachname[25];
    char vorname[25];
    char gebdatum[11];
    int matrnr;
    short gruppe;
    char best;
};
```

```
struct komplex {
    double re;
    double im;
};
```

- Initialisierung
- Strukturen als Funktionsparameter
- Felder von Strukturen
- Zeiger auf Strukturen

19.1 Initialisieren von Strukturen

- Strukturen können — wie Variablen und Felder — bei der Definition initialisiert werden
 - die Zuordnung zu den Komponenten erfolgt entweder aufgrund der Reihenfolge oder aufgrund des angegebenen Namens (in C++ nur aufgrund der Reihenfolge möglich!)

Beispiele

```
struct student stud1 = {
    "Meier", "Hans", "24.01.1970", 1533180, 5, 'n'
};

struct komplex c1 = {1.2, 0.8}, c2 = {.re=0.5, .im=0.33};
```

!!! Vorsicht

bei Zugriffen auf eine Struktur werden die Komponenten immer durch die Komponentennamen identifiziert,
bei der Initialisierung nach Reihenfolge aber nur durch die Position
 ► potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration

19.2 Strukturen als Funktionsparameter

- Strukturen können wie normale Variablen an Funktionen übergeben werden
 - ◆ Übergabesemantik: **call by value**
 - Funktion erhält eine Kopie der Struktur
 - auch wenn die Struktur ein Feld enthält, wird dieses komplett kopiert!
 - !!! Unterschied zur direkten Übergabe eines Feldes

- Strukturen können auch Ergebnis einer Funktion sein
 - Möglichkeit mehrere Werte im Rückgabeparameter zu transportieren

Beispiel

```
struct komplex komp_add(struct komplex x, struct komplex y) {
    struct komplex ergebnis;
    ergebnis.re = x.re + y.re;
    ergebnis.im = x.im + y.im;
    return(ergebnis);
}
```

19.3 Felder von Strukturen

- Von Strukturen können — wie von normalen Datentypen — Felder gebildet werden

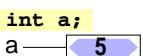
- Beispiel

```
struct student gruppe8[35];
int i;
for (i=0; i<35; i++) {
    printf("Nachname %d. Stud.: ", i);
    scanf("%s", gruppe8[i].nachname);
    ...
    gruppe8[i].gruppe = 8;

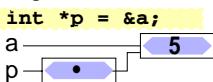
    if (gruppe8[i].matrnr < 1500000) {
        gruppe8[i].best = 'y';
    } else {
        gruppe8[i].best = 'n';
    }
}
```

19.5 Zusammenfassung

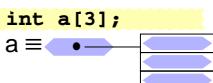
- Variable



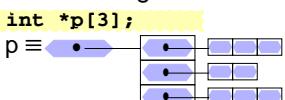
- Zeiger



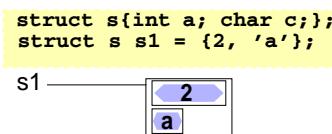
- Feld



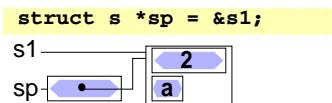
- Feld von Zeigern



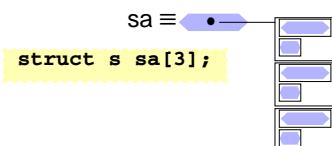
- Struktur



- Zeiger auf Struktur



- Feld von Strukturen



19.4 Zeiger auf Felder von Strukturen

- Ergebnis der Addition/Subtraktion abhängig von Zeigertyp!

- Beispiel

```
struct student gruppe8[35];
struct student *gp1, *gp2;

gp1 = gruppe8; /* gp1 zeigt auf erstes Element des Arrays */
printf("Nachname des ersten Studenten: %s", gp1->nachname);

gp2 = gp1 + 1; /* gp2 zeigt auf zweite Element des Arrays */
printf("Nachname des zweiten Studenten: %s", gp2->nachname);

printf("Byte-Differenz: %d", (char*)gp2 - (char*)gp1);
```

Zeiger auf Funktionen

- Datentyp: Zeiger auf Funktion

◆ Variablendef.: <Rückgabetypr> (*<Variablenname>) (<Parameter>);

```
int (*fptr)(int, char*);
```

```
int test1(int a, char *s) { printf("1: %d %s\n", a, s); }
int test2(int a, char *s) { printf("2: %d %s\n", a, s); }
```

```
fptr = test1;
```

```
fptr(42,"hallo");
```

```
fptr = test2;
```

```
fptr(42,"hallo");
```

Ein-/Ausgabe

- E-/A-Funktionalität nicht Teil der Programmiersprache
- Realisierung durch "normale" Funktionen
 - Bestandteil der Standard-Funktionsbibliothek
 - einfache Programmierschnittstelle
 - effizient
 - portabel
 - betriebssystemnah
- Funktionsumfang
 - Öffnen/Schließen von Dateien
 - Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
 - Formatierte Ein-/Ausgabe

- Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:
 - ◆ **stdin** Standardeingabe
 - normalerweise mit der Tastatur verbunden
 - Dateiende (**EOF**) wird durch Eingabe von **CTRL-D** am Zeilenanfang signalisiert
 - bei Programmaufruf in der Shell auf Datei umlenkbar


```
prog <eingabedatei
```

 (bei Erreichen des Dateiendes wird **EOF** signalisiert)
 - ◆ **stdout** Standardausgabe
 - normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden
 - bei Programmaufruf in der Shell auf Datei umlenkbar


```
prog >ausgabedatei
```
 - ◆ **stderr** Ausgabekanal für Fehlermeldungen
 - normalerweise ebenfalls mit Bildschirm verbunden

21.1 Standard Ein-/Ausgabe (2)

- Pipes
 - ◆ die Standardausgabe eines Programms kann mit der Standardeingabe eines anderen Programms verbunden werden
 - Aufruf


```
prog1 | prog2
```
- ! Die Umlenkung von Standard-E/A-Kanäle ist für die aufgerufenen Programme völlig unsichtbar
- automatische Pufferung
 - ◆ Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem **NEWLINE**-Zeichen ('\n') an das Programm übergeben!

21.2 Öffnen und Schließen von Dateien

- Neben den Standard-E/A-Kanälen kann ein Programm selbst weitere E/A-Kanäle öffnen
 - Zugriff auf Dateien
- Öffnen eines E/A-Kanals
 - Funktion **fopen**:

```
#include <stdio.h>
FILE *fopen(char *name, char *mode);
```

name	Pfadname der zu öffnenden Datei
mode	Art, wie die Datei geöffnet werden soll
"r"	zum Lesen
"w"	zum Schreiben
"a"	append: Öffnen zum Schreiben am Dateiende
"rw"	zum Lesen und Schreiben

 - Ergebnis von **fopen**: Zeiger auf einen Datentyp **FILE**, der einen Dateikanal beschreibt im Fehlerfall wird ein **NULL**-Zeiger geliefert

21.2 Öffnen und Schließen von Dateien (2)

■ Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE *eingabe;

    if (argc[1] == NULL) {
        fprintf(stderr, "keine Eingabedatei angegeben\n");
        exit(1); /* Programm abbrechen */
    }

    if ((eingabe = fopen(argv[1], "r")) == NULL) {
        /* eingabe konnte nicht geöffnet werden */
        perror(argv[1]); /* Fehlermeldung ausgeben */
        exit(1); /* Programm abbrechen */
    }

    ... /* Programm kann jetzt von eingabe lesen */
}
```

■ Schließen eines E/A-Kanals

```
int fclose(FILE *fp)
```

► schließt E/A-Kanal **fp**

21.3 Zeichenweise Lesen und Schreiben (2)

■ Beispiel: copy-Programm, Aufruf: **copy Quelldatei Zielfile**

```
#include <stdio.h>                                Teil 1: Aufrufargumente
int main(int argc, char *argv[]) {                  auswerten
    FILE *quelle, *ziel;
    int c;                                         /* gerade kopiertes Zeichen */

    if (argc < 3) { /* Fehlermeldung, Abbruch */

        if ((quelle = fopen(argv[1], "r")) == NULL) {
            perror(argv[1]); /* Fehlermeldung ausgeben */
            exit(EXIT_FAILURE); /* Programm abbrechen */
        }

        if ((ziel = fopen(argv[2], "w")) == NULL) {
            /* Fehlermeldung, Abbruch */
        }

        while ( (c = getc(quelle)) != EOF ) {
            putc(c, ziel);
        }

        fclose(quelle);
        fclose(ziel);
    }
}
```

21.3 Zeichenweise Lesen und Schreiben

■ Lesen eines einzelnen Zeichens

◆ von der Standardeingabe

```
int getchar()
```

► lesen das nächste Zeichen

► geben das gelesene Zeichen als **int**-Wert zurück

► geben bei Eingabe von **CTRL-D** bzw. am Ende der Datei **EOF** als Ergebnis zurück

■ Schreiben eines einzelnen Zeichens

◆ auf die Standardausgabe

```
int putchar(int c)
```

► schreiben das im Parameter **c** übergeben Zeichen

► geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

◆ von einem Dateikanal

```
int getc(FILE *fp)
```

21.3 Zeilenweise Lesen und Schreiben

■ Lesen einer Zeile von der Standardeingabe

```
char *fgets(char *s, int n, FILE *fp)
```

► liest Zeichen von Dateikanal **fp** in das Feld **s** bis entweder **n-1** Zeichen gelesen wurden oder '**\n**' oder **EOF** gelesen wurde

► **s** wird mit '**\0**' abgeschlossen ('**\n**' wird nicht entfernt)

► gibt bei **EOF** oder Fehler **NULL** zurück, sonst **s**

► für **fp** kann **stdin** eingesetzt werden, um von der Standardeingabe zu lesen

■ Schreiben einer Zeile

```
int fputs(char *s, FILE *fp)
```

► schreibt die Zeichen im Feld **s** auf Dateikanal **fp**

► für **fp** kann auch **stdout** oder **stderr** eingesetzt werden

► als Ergebnis wird die Anzahl der geschriebenen Zeichen geliefert

21.4 Formatierte Ausgabe

■ Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int printf(char *format, /* Parameter */ ... );
int fprintf(FILE *fp, char *format, /* Parameter */ ... );
int sprintf(char *s, char *format, /* Parameter */ ... );
int snprintf(char *s, int n, char *format, /* Parameter */ ... );
```

■ Die statt ... angegebenen Parameter werden entsprechend der Angaben im **format**-String ausgegeben

- bei **printf** auf der Standardausgabe
- bei **fprintf** auf dem Dateikanal **fp**
(für **fp** kann auch **stdout** oder **stderr** eingesetzt werden)
- **sprintf** schreibt die Ausgabe in das **char**-Feld **s**
(achtet dabei aber nicht auf das Feldende -> Pufferüberlauf möglich!)
- **snprintf** arbeitet analog, schreibt aber maximal nur **n** Zeichen
(**n** sollte natürlich nicht größer als die Feldgröße sein)

21.5 Formatierte Eingabe

■ Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int scanf(char *format, /* Parameter */ ... );
int fscanf(FILE *fp, char *format, /* Parameter */ ... );
int sscanf(char *s, const char *format, /* Parameter */ ... );
```

- Die Funktionen lesen Zeichen von **stdin** (**scanf**), **fp** (**fscanf**) bzw. aus dem **char**-Feld **s**.
- **format** gibt an, welche Daten hiervon extrahiert und in welchen Datentyp konvertiert werden sollen
- Die folgenden Parameter sind Zeiger auf Variablen der passenden Datentypen (bzw. **char**-Felder bei Format **%s**), in die die Resultate eingetragen werden
- relativ komplexe Funktionalität, hier nur Kurzüberblick
für Details siehe Manual-Seiten

21.4 Formatierte Ausgabe (2)

■ Zeichen im **format**-String können verschiedene Bedeutung haben

- normale Zeichen: werden einfach auf die Ausgabe kopiert
- Escape-Zeichen: z. B. **\n** oder **\t**, werden durch die entsprechenden Zeichen (hier Zeilenvorschub bzw. Tabulator) bei der Ausgabe ersetzt
- Format-Anweisungen: beginnen mit %-Zeichen und beschreiben, wie der dazugehörige Parameter in der Liste nach dem **format**-String aufbereitet werden soll

■ Format-Anweisungen

- | | |
|---------------|---|
| %d, %i | int Parameter als Dezimalzahl ausgeben |
| %f | float Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben |
| %e | float Parameter wird als Fließkommazahl in 10er-Potenz-Schreibweise (z. B. 2.714567e+02) ausgegeben |
| %c | char-Parameter wird als einzelnes Zeichen ausgegeben |
| %s | char-Feld wird ausgegeben, bis ' \0 ' erreicht ist |

21.5 Formatierte Eingabe

21.5 Formatierte Eingabe (2)

■ White space (Space, Tabulator oder Newline **\n**) bildet jeweils die Grenze zwischen Daten, die interpretiert werden

- white space wird in beliebiger Menge einfach überlesen
- Ausnahme: bei Format-Anweisung **%c** wird auch white space eingelesen

■ Alle anderen Daten in der Eingabe müssen zum **format**-String passen oder die Interpretation der Eingabe wird abgebrochen

- wenn im **format**-String normale Zeichen angegeben sind, müssen diese exakt so in der Eingabe auftauchen
- wenn im **format**-String eine Format-Anweisung (...) angegeben ist, muss in der Eingabe etwas hierauf passendes auftauchen
 - diese Daten werden dann in den entsprechenden Typ konvertiert und über den zugehörigen Zeiger-Parameter der Variablen zugewiesen

■ Die **scanf**-Funktionen liefern als Ergebnis die Zahl der erfolgreich an die Parameter zugewiesenen Werte

21.5 Formatierte Eingabe (3)

%d	int
%hd	short
%ld	long int
%lld	long long int
%f	float
%lf	double
%Lf	long double
analog auch %e oder %g	

%c	char
%s	String, wird automatisch mit '\0' abgeschl.

- nach % kann eine Zahl folgen, die die maximale Feldbreite angibt
- %3d = 3 Ziffern lesen
- %5c = 5 char lesen (Parameter muss dann Zeiger auf char-Feld sein)
 - %5c überträgt exakt 5 char (hängt aber kein '\0' an!)
 - %5s liest max. 5 char (bis white space) und hängt '\0' an

Beispiele:

```
int a, b, c, d, n;
char s1[20] = "XXXXXX", s2[20];
n = scanf("%d %2d %3d %5c %s %d",
           &a, &b, &c, s1, s2, &d);
```

Eingabe: 12 1234567 sowas hmm
 Ergebnis: n=5, a=12, b=12, c=345
 s1="67 soX", s2="was"

- Fast jeder Systemcall/Bibliotheksauftrag kann fehlschlagen
 - ◆ Fehlerbehandlung unumgänglich!
- Vorgehensweise:
 - ◆ Rückgabewerte von Systemcalls/Bibliotheksaufträgen abfragen
 - ◆ Im Fehlerfall (meist durch Rückgabewert -1 angezeigt): Fehlercode steht in der globalen Variable **errno**
- Fehlermeldung kann mit der Funktion **perror** auf die Fehlerausgabe ausgegeben werden:

```
#include <errno.h>
void perror(const char *s);
```