Programmieren

C

1. Semester

Klaus Kusche

Inhalt (1)

- Einleitung
- Die Sprache C / C++
- Grundbausteine eines Programms
- “Rechnungen” (+ Zuweisungen, einfaches I/O, ...)
- Befehle (Bedingungen & Schleifen)
- Arrays
- Funktionen

Inhalt (2)

- Die Rekursion
- Speicherklassen
- Zeiger
- Zeiger auf Arrays
- Strings
- Zeichensätze

Inhalt (3)

- Wiederholung Zeiger
- Wiederholung Funktionen
- File-I/O
- Fehlerbehandlung
- Strukturen (+ `typedef`)
- Dynamische Daten
- Bit-Operationen

Inhalt (4)

- Der Präprozessor
- .c-Files und .h-Files
- Union's, Enum's usw.
- Die C Library
- Ausblick

Einleitung

- Was ist Programmieren?
- Wie arbeitet ein Computer?
- Was ist eine Programmiersprache?
- Welche Arten von Sprachen gibt es?
- Wie wird ein Programm ausgeführt?
- Wie programmiert man?
- Was braucht man dazu?

Programmieren?

Programmieren =
Programme schreiben =
“Dem Computer erklären,
was er tun soll”

Hochtrabend:
Programme = *“Software”*
Programme schreiben (professionell) =
“*Software-Entwicklung*”

“**Software Engineering**” =
Die Kunst / Wissenschaft / Technik
der *professionellen Programmierung*
(“Informatik” umfasst mehr, SE ist ein Teilgebiet)

Programme

Ohne Programme macht ein Computer gar nichts:

Beim Strom Einschalten beginnt der Prozessor,
von einer bestimmten Stelle im Speicher
Programm-Befehle zu holen und auszuführen
(beim PC steht dort das BIOS, auch ein "Programm").

Bei PC's, Großrechnern, ...:

Programme sind *Dateien* auf der Platte (**`.exe`**, **`.dll`**), werden *in den Speicher geladen* und ausgeführt.

Bei Steuerungen, Routern, ...:

Programme werden fix in der Hardware gespeichert (meist *in Flash-Chips*)

Wie arbeitet ein Computer? (1)

Alles im Computer
(*Programme & Daten*,
auf Platte, im Speicher, im Prozessor, am USB-Stick...)
*sind viele **0 und 1***
(Spannung niedrig / hoch,
Licht / kein Licht,
Magnetfeld S-N oder N-S, ...)

==> Ein Programm (wie es der Prozessor ausführt)
besteht aus vielen 0 und 1

==> Ein Programm berechnet (aus Hardware-Sicht)
aus vorhandenen 0 und 1 neue 0 und 1

Speicherinhalte

1 mal “0 oder 1” … **1 Bit**
(ein bisschen wenig, unpraktisch)

8 Bits nacheinander (8 mal “0 oder 1”) … **1 Byte**
(kleinste “sinnvolle” Datenmenge: **1 Zeichen**)

Vielfache: 1 **KB** (1000 bzw. 1024 Bytes), 1 **MB**, 1 **GB**, …

Darstellung von Zahlen, Texten, Bildern, Tönen, …
nur mit 0 und 1 bzw. als Folge von Bytes:

- Demnächst kurz hier
- *… und in anderen Fächern!*

Wie arbeitet ein Computer? (2)

- Der Prozessor holt einen Befehl aus dem Speicher, führt ihn aus, holt den nächsten, führt ihn aus, ...
- 1 Befehl (= Bitmuster mit 8...64 Bits)
  - ***liest*** *einen Wert* (1, 2, 4 oder 8 Bytes) aus dem Speicher (= RAM) in den Prozessor
  - oder führt *eine* ***Rechenoperation*** (+, *, ...) durch
  - oder ***schreibt*** *ein Ergebnis* zurück in den Speicher
  - oder ***vergleicht*** */ prüft* Werte (gleich?, ‹0 ?, ...)
  - oder *ändert den* ***Programmablauf*** *(ev. abh. von Vergleich)* (*"nimm nicht den nächsten Befehl im Speicher, sondern den Befehl an Speicherstelle xxx"*).

Wieso Programmiersprachen? (1)

- Das Programm für den Prozessor im Speicher (Millionen 0 und 1) ist für den Menschen völlig *unleserlich / "unschreibar" / unverständlich.*
- "Dasselbe" Programm (aus Benutzersicht!) schaut je nach *Prozessortyp*
  (PC, Handy, Steuerung, ...)
  und je nach *Betriebssystem*
  (Windows, Linux, MacOS, Android, ...)
  komplett *anders* aus (ganz andere 0 und 1)

  ==> *Jedesmal neu schreiben???*

Wieso Programmiersprachen? (2)

==> Eine **Programmiersprache** erlaubt eine

- *menschlich lesbare und (halbwegs) verständliche*
- *und (weitgehend) Hardware- und Betriebsystem-unabhängige*

Beschreibung von Programmen als "Text"

Dieser "Text" (vordef. Worte, math. Formeln, Sonderzeichen, ...)

- muss eine bestimmte *Struktur* haben
- und nach gewissen *formalen Regeln* aufgebaut sein

... sonst "versteht" ihn der Computer nicht!

Die Programmiersprache legt diese Regeln fest!

Was steht in einem Programm?

Ein Programm ist wie ein *Kochrezept*:

Es beschreibt

- zuerst die **Daten** (Zahlen, Texte, ...),
  die man verwendet
  (“*Zutaten*”):
  ==> ***“Deklarationen”***
- und dann der Reihe nach
  die einzelnen **Arbeitsschritte**
  (“*Zubereitung*”, Arbeitsablauf):
  ==> ***Befehle, Anweisungen***

Was sind Arbeitsschritte?

- ***Rechnen*** (math. Formeln)
- Rechenergebnisse ***speichern*** / merken
- *"**Aufrufen**"* anderer / vordefinierter Codestücke
  (z.B. für Ein- und Ausgabe)
- *"**Verzweigen**" = Bedingung, Fallunterscheidung:*
  "Mach entweder dieses oder jenes,
  je nachdem, ob ... ist oder nicht"
- *"**Wiederholen**" = Schleife:*
  "Mach ... immer wieder, bis ... gilt"
  "Zähl von ... bis ... und mach jedesmal ..."

Einteilung Programmiersprachen (1)

- *Maschinennahe Sprachen* = "**Assembler**":

  Menschlich lesbare Notation (Abkürzungen)
  für *einzelne Prozessor-Befehle* (Bitmuster)
  Alt, mühsam, schlecht lesbar, *HW-abhängig!*

- *Prozedurale Sprachen:* **C, Fortran, Cobol, Ada, ...**

  Erste "Hochsprachen":
  HW-unabhängig, halbwegs lesbar,
  komfortablere / abstraktere Konzepte
  und Konstrukte als einzelne Prozessor-Befehle

Einteilung Programmiersprachen (2)

- *Objektorientierte Sprachen:* **C++, Java, C#, Objective C, JavaScript, VB, PHP, Smalltalk, ...**

  Erweiterung von prozeduralen Sprachen zur *besseren Strukturierung* von Code & Daten für *große* Programme

- *Funktionale Sprachen* (**Lisp, Haskell**, ...) und *logische Sprachen* (**Prolog**):

  Logik-basierte Programmier-Konzepte ...
  ... haben sich nie so wirklich durchgesetzt

Verarbeitung von Programmen (1)

Aus der Sicht des Computers ist
ein Programm in einer Programmiersprache
eine *ganz normale Textdatei*
(bis auf die Filenamen-Endung z.B. "`.c`" statt "`.txt`"):

"Quelltext", "Source-File"

==> auch nur Daten, "Ansammlung von Buchstaben"

==> Keine Befehle,
die der Prozessor direkt ausfühen kann!

==> Ein *anderes Programm* *"schaut sich diesen Textfile an",*
erst dadurch bekommt er eine Bedeutung als Programm!

Verarbeitung von Programmen (2)

Dieses “andere Programm” zur Verarbeitung
des Quelltextes kann sein …

Variante 1: Interpreter

Der Interpreter liest eine Quelltext-Zeile,
analysiert sie, führt aus, was sie ihm anschafft,
liest die nächste Zeile, analysiert sie, führt sie aus, usw.
(ohne dass es je einen **.exe***-File für das Programm gibt).*

Vorteil: Komfortabel, das Programm läuft “sofort” los.

Nachteil: (Viel) langsamer, braucht mehr Speicher.

Für alle ***Skriptsprachen*** (JavaSkript, VB, PHP, …).

Verarbeitung von Programmen (3)

Variante 2: Compiler ("Übersetzer")

Der Compiler liest den *gesamten* Source-File
und erzeugt daraus (wenn fehlerfrei)
einen **.exe-File** mit binären Prozessor-Befehlen.

Danach wird dieser **.exe** als
eigenes Programm gestartet.

Vorteil: Das Programm läuft *schnell*.

Nachteil: Eigener *Zwischenschritt* notwendig.

Für fast alle ***"klassischen" Programmiersprachen***
(u.a. C, C++).

Verarbeitung von Programmen (4)

Es gibt viele *Zwischenstufen bzw. Kombinationen*
zwischen Compiler und Interpreter:

- ***Zwischencode-Compiler + Zwischencode-Interpreter***
  (Java, C#):

  - Compiler erzeugt Datei mit "Prozessor-Befehlen"
    für einen *imaginären Prozessor*

    ==> *Compiler ist hardware-unabhängig*

  - Interpreter interpretiert diese "Prozessor-Befehle"

    ==> Geht viel *schneller und leichter*
    als Interpretieren des ursprünglichen Quelltexts

Verarbeitung von Programmen (5)

- ***Interpreter mit JIT ("Just-in-Time-Compiler")***
  (fast alle modernen Skriptsprachen-Interpreter):

  - Interpreter *merkt sich* für jede interpretierte Zeile
    die dafür ausgeführten Prozessor-Befehle

  - Wenn *dieselbe Zeile nochmals* ausgeführt werden soll:
    Nicht mehr nochmals interpretieren,
    sondern gleich gemerkten echten Maschinen-Code
    laufen lassen

  ==> *Schleifen und viel benutzte Codestücke*
  werden viel schneller!

Verarbeitung von Programmen (6)

- Wenn der Quelltext in einer *Assembler-Sprache* (menschliche Notation für einzelne Prozessor-Befehle) geschrieben ist, heißt das Übersetzer-Programm auch

"Assembler" und nicht "Compiler"

Compiler im Detail (1)

Bei C / C++ passieren beim Compilieren 3-4 Schritte:

- Der **C Präprozessor**
  (eigener Vorverarbeitungs-Schritt im Compiler)
  modifiziert den Quelltext
  (hängt z.B. vordefinierte Quelltext-Files dazu: **`#include`** ...).

  Das liefert wieder einen (temporären) Quelltext-File.

- Der **Compiler** *erzeugt Binärcode* für den Quelltext.
  (bei vielen Compilern, z.B. **gcc**, nicht direkt,
  sondern durch Erzeugen von Assembler-Quelltext
  und Aufruf des Assemblers)

  Das liefert einen **`.o`-File** (*"Object-File"*).

...

Compiler im Detail (2)

...

- Der **Linker** (würde eigentlich “Verbinder” heißen, früher “Binder”, heute eingedeutscht) *hängt* den **.o**-File mit **Libraries** bzw. **DLL’s** (enthalten compilierten Binärcode für *vordefinierte Funktionen* und *andere Quell-Dateien*) *zusammen*.

  Erst das liefert den fertigen **.exe-File**.

Compiler im Detail (3)

Der Compiler ist ein ***eigenes Befehlszeilen-Programm*** (ohne Fenster), meist "**cc**" oder "**gcc**".

Bei modernen IDE's (z.B. CodeBlocks) wird der Compiler nicht mehr separat auf der Befehlszeile aufgerufen:

- Der Editor startet den Compiler intern,
  wenn man auf "Compile & Run" klickt.
- Liefert der Compiler Fehler,
  markiert sie der Editor gleich im Quelltext.
- Klappt das Compilieren,
  startet der Editor das frisch compilierte **.exe**-File
  gleich in einem DOS-Fenster.

Schritte beim Programmieren (1)

- **Problem analysieren:**
  Aufgabe *klar* und *vollständig* formulieren:
  - Was ist **gegeben** / wird eingegeben?
  - Was ist **gesucht** / soll ausgegeben werden?
  - Randbedingungen? Sonderfälle?

Ein Problem,
das man nicht richtig verstanden hat,
kann man auch nicht richtig lösen!

Schritte beim Programmieren (2)

- **Lösungsidee entwickeln (= "Algorithmus"):**
  - Welche *Daten* brauche ich intern?
  - Welche *Operationen* muss ich darauf *Schritt für Schritt* machen?

Die Lösungsidee sollte noch unabhängig von der Programmiersprache sein!

(im Prinzip mit "Papier & Bleistift" durchführbar)

Schritte beim Programmieren (3)

- Betreffend *"Schritt für Schritt"*:

  Der Computer ist strohdumm

  ==> *Wie würde ich die Lösungsidee meinem fünfjährigen Bruder erklären?*

  ==> *Jeder einzelne Schritt muss "trivial" sein!*

  (einen Wert berechnen & speichern, einen Wert vergleichen, ein "wenn ... dann ...", einen Wert ausgeben, ...)

Schritte beim Programmieren (4)

- **Lösungsidee (Algorithmus) programmieren:**

  Für jeden einzelnen Schritt
  den entsprechenden Programmbefehl tippen.

  Wenn die Lösungsidee sauber entwickelt war,
  sollte das ohne viel Denken gehen!

- **Programm compilieren & testen:**

  Alle typischen Eingabefälle, Randfälle
  und typische falsche Eingaben
  durchprobieren.

Entwurf eines Algorithmus

In C: Von der Grobstruktur zu den Details,
"von außen nach innen".

- Teile die Lösung in *wenige, grobe Teilschritte*
  (oft: "*Einlesen - Berechnen - Ausgeben*").
- *Zerlege jeden einzelnen Teilschritt* für sich
  wieder in ein paar kleinere Teilschritte,
  jeden davon wieder usw.
- Wiederhole das,
  bis jeder einzelne Teilschritt "trivial" ist.

(siehe später: "Struktogramme" in immer kleinere Kästchen zerteilen)

Was zählt beim Programmieren? (1)

- ***Exaktheit*** *der Eingabe:* Jedes Zeichen, jeder "**;**" zählt…
  Der Computer ist ein Vollidiot,
  er kann sich nicht "denken", was wir von ihm wollen!
- ***Korrektheit*** *der Lösungsidee:* Das Programm
  … stürzt nicht ab
  … liefert das *gewünschte* Ergebnis
  … endet geordnet
- ***Robustheit:*** Das Programm reagiert sinnvoll auf
  … falschen Aufruf und falsche Eingaben
  … zu viele / zu große Daten
  … Fehlerbedingungen (Datei nicht da, Netzwerk weg, …)
- ***Bedienerfreundlichkeit***

Was zählt beim Programmieren? (2)

- ***Effizienz:*** Das Programm
  … rechnet so schnell wie möglich
  … braucht nicht unnötig viel Speicher

  Aber:

  Zuerst richtig machen, dann erst schnell!

- ***Schönheit = Programmierstil:***
  Das Programm soll *lesbar* und *verständlich* sein!

  Weil:

  - Hilft bei Fehlersuche, Erweiterung, Wartung, …
  - In 5 Jahren: *“Was habe ich denn da verbrochen???”*

Zum Programmierstil … (1)

- Systematische *Einrückung*,
  systematische Position der **{ }**
- *Zwischenräume* und Leerzeilen
- *Gute, "sprechende" Namen:*

  "Was bedeutet der Wert in dieser Variable?"

  "Was macht diese Funktion (dieses Stück Code)?"
- *Kommentare!*

Zum Programmierstil ... (2)

Ist dem Computer völlig egal
(es gibt keine Fehlermeldung für schlechten Stil),
aber hilft dem Menschen sehr viel

==> Selbstdisziplin ist gefragt:
Im eigenen Interesse
schön programmieren!

Fast so wichtig
wie viel selber programmieren üben:

Viele bestehende gute Programme lesen!

Was kann schiefgehen? (1)

- ***"Compilerfehler"*:**
  Das Programm entspricht nicht den *formalen Regeln* der Programmiersprache

  ==> der Compiler / Interpreter "versteht es nicht", liefert *schon vor dem Ausführen* Fehlermeldungen.

- ***"Linkerfehler"*:**
  Das Programm ist korrektes C, aber z.B. *fehlen* Dinge (oder häufig: Ein *Funktionsname* ist *falsch* getippt)

  ==> Auch Fehlermeldung *schon vor dem Ausführen*, aber vom Linker, nicht vom Compiler

Was kann schiefgehen? (2)

- ***"Laufzeitfehler":***

  Das Programm ist zwar formal richtig geschrieben
  und *lässt sich ausführen*,
  hat aber einen ***Denkfehler*** in der Lösungsidee
  oder einen ***Programmierfehler*** in deren Umsetzung

  ==> Es *"rechnet falsch"* oder macht Blödsinn:

  - ***Falsche*** Ausgabe
  - ***Absturz*** (= gewaltsamer *Abbruch* durch das *Betriebssystem*)
  - Fehlermeldung trotz richtiger Eingabe
  - Programm läuft ***endlos***
  - Abbruch wegen ***Speicher-Überlauf***
  - ...

Was muss man können? (1)

- Eine *Programmiersprache*
  (wir lernen hier C)
- Verständnis von *Aufbau und Funktionsweise eines Computers* (==> Rechnerarchitektur)
  incl. *Datendarstellung* (==> Grundlagen)
- Kenntnis der *vordefinierten Funktionen* der Sprache
  (bestehende Codestücke, die man verwenden kann,
  z.B. Ein- und Ausgabe, Text suchen, Sinus rechnen, ...)
  (die wichtigsten lernen wir, sonst *Online-Doku*)

  (z.B. bei Java: rund 300000 vordef. Funktionen
  ==> nicht lernen, sondern bei Bedarf nachschauen!)

Was muss man können? (2)

- Häufige *Algorithmen und Datenstrukturen*
  (z.B. *"Wie sortiere ich viele Daten am schnellsten"* oder
  "Wie speichere ich viele Daten so,
  dass ich jeden Wert möglichst schnell finde?")
  (==> Studium 2./3. Sem.)
- Umgang mit den *Hilfsmitteln* zum Programmieren
  (Editor, Compiler, Debugger, Online-Hilfe, ...)
  (==> selber anschauen & üben)
- *Projektorganisation*, Methoden und Werkzeuge
  zur Entwicklung *großer* Software-Pakete:
  "Software Engineering"
  (==> Studium ab 4. Sem.)

Hilfsmittel zum Programmieren

- Für den Entwurf der Lösung:
  Struktogramme (zum *"Schüler ärgern"*, ignorieren wir)
  (früher auch: PAP = *"Flussdiagramm"*, heute nicht mehr!)
- Guter *Editor* zum Eintippen (automat. Einrückung, ...)
- *Compiler*
  (Compiler-Fehlermeldungen sind hilfreich,
  verstehen und ernst nehmen!!!)
- *Debugger* zur Fehlersuche
  (machen wir nicht, Interessierte: Selber beibringen...)
- Für Fortgeschrittene: Viele andere SW-Werkzeuge!
  (siehe mein Skript Softwaretechnik)

Die Sprachen C und C++

- Geschichte
- Unterschiede C / C++
- Vor- und Nachteile
- Verwendung
- Lehrstoff

Die Geschichte von C

- C entstand um *1970* in den *Bell Labs*
  (Forschungslabor der US Telefongesellschaft)

  ==> Ist die *älteste* heute verbreitete Sprache

  ==> Ist mit C++ die *am weitesten verbreitete* Sprache
- ... zusammen mit *Unix*
  (C und Unix/Linux sind “siamesische Zwillinge”)
- Autoren: *B. Kernighan, D. Ritchie*
- C ist Urvater / Vorbild fast aller
  modernen Programmiersprachen
  (Java, C#, viele Scriptsprachen, ...)

C Standards

- Ursprüngliches C: "**K&R C**"
  (heute nicht mehr verwendet)
- Erster formaler Standard: ANSI C = ISO C = **C89**
  (heute auch veraltert)
- Lange Zeit "der" C-Standard: **C99**
  (heute absolute Mindestanforderung)
- Heute üblicher Standard: **C11** (C17 ist de facto ident)
  (heute bis auf Microsoft fast überall verwendet)
- Aktueller Standard: **C23**
  (noch selten, *breaking Standard*: *verbietet* einige vor-C99-Konstrukte!)
- In der Open-Source-Welt üblich: **Gnu C11 oder C23**
  (= C11 oder C23 + Sprach-Erweiterungen des Gnu C Compilers **gcc**)

C++

Ist eine nachträgliche *Erweiterung* von C
um **objektorientierte Konzepte**

==> C ist (fast) eine Untermenge von C++,
C++ kann alles, was C kann, und ***viel mehr***

Echte Unterschiede (für uns):

- *Datei- und DOS-Fenster-Ein/Ausgabe* (komplett anders!)
- Dyn. Speicherverwaltung
- Ein Dutzend Details...
  (für Interessierte: ==> Mein Skript C++ 2. Sem, Kap. 1)

Ein- und Ausgabe: C oder C++ ?

- Ich verwende im 1. Sem. *nur C I/O*

  ==> *Im 1. Sem. wird C I/O geprüft!*

- Beides hat Vor- und Nachteile
- *Man sollte beides lesen und verstehen können*

  Im Bereich der Lehre: Mehr C++ I/O (oft auch für C)
  In der Praxis: Eher mehr C I/O !
  (u.a. weil C I/O auch in C++ geht, umgekehrt nicht!)

- Man sollte zumindest eines
  selber gut & sicher schreiben können
  (mit den wichtigsten Erweiterungen)

Vorteile von C / C++

- *Sehr effizient:*

  Läuft schneller, braucht weniger Speicher als Java, C#, JavaScript, ...
- *Direkter Zugriff auf Hardware* möglich

  ==> eignet sich für Steuerungen, Treiber, ...
- *Schult das Verständnis,*
  wie ein Computer und ein Compiler arbeitet

  (relativ “direkte”, nachvollziehbare Abbildung von C auf Maschinenprogramm und Daten im Speicher)

Nachteile von C / C++

- *Unkomfortabel und primitiv:*

  Man muss vieles selbst programmieren,
  was bei Java, ... automatisch geht

- *Nicht "deppensicher":*

  Sehr viele Fehler-Möglichkeiten
  Kaum automatische Fehler-Prüfungen
  ==> Stürzt leicht ohne sinnvolle Meldung ab
  ==> Gefahr unerkannter Fehler groß
  ==> Fehlersuche mühsam
  ==> Anfällig für Sicherheits-Lücken

Verwendung von C / C++ (1)

- *"Systemnahe"* Programmierung:

  Betriebssysteme, Compiler, ...
  Ein Großteil der gesamten Linux- und Open-Source-Software!

- *"Hardware-nahe"* Programmierung:

  Steuerungen, Treiber, ...

- *Effizienz-*, *zeit-* und *platz*kritische Software:

  Echtzeitsysteme, Hochleistungsrechner, ...

- ...

Verwendung von C / C++ (2)

Nicht:

- “Anwendungs-Software” (kaufmännisch, …)
- Webprogrammierung
- Skripts

Größenordungen:

C / C++ ist die derzeit am weitesten verbreitete Sprache!

- Linux, nur Kernel: Knapp 25 Mio. Zeilen C (nicht C++)
- Linux, ganze Distribution: ›› 500 Mio. Zeilen
- Windows: Gut 50 Mio. Zeilen

Unser Lehrstoff

- **1. Sem.:** ***C*** *komplett!*
- **2. Sem.:** ***Objektorientierung, C++*** *(nicht komplett),*
  Algorithmen & Datenstrukturen

Was kommt hier ***nicht****?*

- Grafische Benutzeroberflächen (mit Fenstern), Web-Anwendungen, …
- Datenbanken, Netzwerk-Programmierung

Unsere Programme…

… sind daher **Text-Modus-Programme**:

- Laufen im "**DOS-Fenster**" (= **cmd-Fenster**)
  (oder im Linux-Shell-Fenster)
- Lesen zeichenweise Text von der Tastatur,
  schreiben zeichenweise Text ins DOS-Fenster

Alternativ statt "Tastatureingabe Lesen" im Code:

Angabe der Eingabe-Daten
beim *Programmstart* auf der *Befehlszeile!*
(ist bei Text-Modus-Programmen
üblich und praktisch!!!)

Grundbausteine von Programmen

- Kommentare, White Space
- Fixe Programmbestandteile:
  `#include` und `main`
- Der ;
- Konstanten
- Variablen-Deklarationen,
  Namen & Typen
- Benannte Konstanten

Kommentare

- Werden vom Compiler *ignoriert*
- Sind nur für den Menschen *zur Information*
- *2 Arten:*
  - `//` bis Zeilenende
  - `/*` bis `*/` (auch über mehrere Zeilen)

Achtung: `/* ... */` ist *nicht* geschachtelt möglich!

==> Trick für Fortgeschrittene,
um Programmteile auszukommentieren, die schon `/* ... */` enthalten:

```
#if 0
...
#endif
```

Kommentierung

- **Oben** in jedem Quelltext:

  Programm-Zweck, Aufruf des Programms, Autor, …

- **Mittendrin**:
  - ***Nicht*** im Kommetar wiederholen,
    was ohnehin aus dem Programmcode klar ist (“**Was?**”):

    `tag = tag + 7;` // zähl zu **tag** 7 dazu
  - Sondern *den Sinn / die Idee beschreiben* (“**Warum?**”):

    `tag = tag + 7;` // gehe mit **tag** 1 Woche weiter!
- ***Lieber schön programmieren & nichts kommentieren***
  als grauslichen Code mit Kommentaren retten wollen!

“White Space”

Leerzeilen, Einrückungen (“Tab”), Zwischenräume:

- (Großteils) nach Belieben
- Vom Compiler ignoriert (außer in " " usw.)
- Wichtig für die *Lesbarkeit* des Programms!
- Zeilenlänge ist egal / unbeschränkt,
  aber *max. 80 Zeichen* sind üblich & sinnvoll
  (historische Standard-Breite von DOS-Fenster,
  Textmodus & Drucker)

Am besten (vor allem betr. “Wo Zwischenraum?”):

Guten Code lesen & nachmachen…!

“White Space” - Hinweise

- Gedankliche Blöcke durch *Leerzeilen* trennen
- *Jeder Befehl in eine eigene Zeile*
  (außer zwei Befehle sind eine gedankliche Einheit)
- *Leerzeichen* helfen,
  Zeilen optisch besser zu strukturieren,

  aber an unüblichen Stellen
  tun sie beim Lesen weh!

- *Tab*'s zur Einrückung sind *problematisch*
  (*besser Zwischenräume* verwenden)!
- *Code Formatter* verwenden! (z.B. **`astyle`**)

Systematische Einrückung (1)

Die Einrückung soll genau entsprechen

- … der **Programmstruktur**:

  Der Inhalt eines **`if`** / einer Schleife ist *eingerückt*!

- … der **`{ }`** **Schachtelungstiefe**!

==> **`}`** hat *dieselbe Einrückung*
wie die Zeile mit der zugehörigen **`{`** !

==> Alles *dazwischen* ist *weiter eingerückt:*
2-4 Zwischenräume (oder 1 Tab) pro **`{ }`** *- Ebene!*

==> Alle Befehle, die zur *gleichen* **`{ }`** - Ebene gehören,
sind *gleich weit* eingerückt!

Systematische Einrückung (2)

Zwei weit verbreitete Stilrichtungen, beide ok:

- **`{`** steht *hinten* nach dem **`if`** / dem Schleifenbefehl:

```
if ( … ) {
      …     // eingerückt!
      …
}
```

- **`{`** steht *vorne* in *eigener* Zeile, genau *unter* dem **`if`**:
-

```
if ( … )
{
      …     // eingerückt!
      …
}
```

Fixe Bestandteile

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

int main(int argc, const char *argv[])
{
    ...
    return 0;
}
```

`#include` (1)

`#include` *inkludiert einen "Headerfile".*

Ein Headerfile enthält Deklarationen vordefinierter Funktionen, Typen usw.

Ein **`#include`** *allein pro Zeile, kein* **`;`** *!*

Technisch:

- **`#include`** wird (wie jede #-Zeile) *vom Präprozessor verarbeitet.*
- Der Präprozessor *ersetzt* das **`#include`**
  durch den *Inhalt* des angegebenen Files.
- Der Compiler sieht kein **`#include`** mehr,
  sondern den Quelltext *einschließlich* der inkludierten Header.

`#include` (2)

- Includes *mit* **`.h`**: **`C`** Header, z.B. **`stdio.h`**, **`math.h`**

  ==> brauchen *kein* "**`using namespace std;`**"

  ==> Funktionieren in C *und* C++

- Includes *ohne* **`.h`**: **`C++`** Header, z.B. **`iostream`**, **`cmath`**

  ==> *brauchen* ein "**`using namespace std;`**"
  gleich unter dem **`#include <...>`**

  ==> Funktionieren *nur in C++* (Programmname **`.cpp`**)

`#include` (3)

- Zu (fast) jedem vordefinierten Standard-C-Headerfile gibt es den entsprechenden C++-Headerfile:

  Die C++-Version des C-Headers **`<xxx.h>`** ist **`<cxxx>`**
  (enthält die gleichen Dinge, aber für **`namespace std`**)

- Es gibt auch *"alte" C++-Header* (vor C++98-Standard) mit **`.h`** im Namen, z.B. **`iostream.h`**

  ==> *Nicht mehr verwenden!*

  (Details zu C++ alt / neu und **`using namespace std;`**
  siehe meine alte Schüler-Zusammenfassung)

- Bitte **`stdio.h`** und **`iostream`** Ein- und Ausgabe *nicht in einem Programm mischen!*

#include (4)

Für jede im Programm verwendete Funktion
muss der passende Header inkludiert werden!
(damit der Compiler die Funkion kennt)

Beispiele:

- **printf** usw. (C Ein/Ausgabe) ==> **stdio.h**
- **cout** und **cin** (C++ Ein/Ausgabe) ==> **iostream**
- **atoi**, **rand**, **exit**, **abs**, ... ==> **stdlib.h** / **cstdlib**
- **sqrt**, **M_PI**, ... ==> **math.h** / **cmath**

Der zu einer Funktion gehörige Header-File
steht in der Linux **man**-Page (*Online-Hilfe*)!

`#include` (5)

Unterscheide:

- **`#include <...>`**

  ==> *vordefinierter* System-Headerfile

  ==> wird im *System-Include-Verzeichnis* gesucht
  (Windows: Ordner **`include`** tief im MinGw-Verzeichnis)
  (Linux: **`/usr/include`**)

- **`#include "..."`**

  ==> *selbstgeschriebener* Header

  ==> wird im *eigenen Verzeichnis* gesucht
  (dort wo der **`.c`**-File selbst liegt)

`main` (1)

```
int main( ... )     // void main( ... ) ist >25 Jahre alt und falsch!!!
{
    ...    // hier steht unser Programmcode
}
```

Wir definieren damit eine *Funktion*
(ein Programmstück) namens **`main`** :
Alles zwischen den **`{ }`** ist der *Code* für unser **`main`** .

Die **`main`**-Funktion (die "*Hauptfunktion*")
wird ausgeführt, sobald unser **`.exe`**-File gestartet wird.
Endet **`main`**, endet auch unser Programm.

Details zu Funktionen in eigenem Kapitel. Für Neugierige:
`( ... )` beschreibt, was reinkommt, und ein **`int`** (ganze Zahl) kommt raus.

`main` (2)

- Wenn *kein* Zugriff auf die Befehlszeile nötig ist:
  `int main(void)` (*nur* in C++ gilt auch "**`int main()`**")
- *Mit* Zugriff auf die **Befehlszeile**:
  `int main(int argc, const char *argv[])`
  (oder alt, heißt *dasselbe*: **`int main(int argc, char **argv)`**)
  - **`argc`** … *"Argument count"*, eine ganze *Zahl*:
    Anzahl der Worte auf der Befehlszeile
    incl. Programmname! ==> *immer mindestens* **`1`**
  - **`argv`** … *"Argument values"*, *wortweise* als *Texte*:
    `argv[0]` … **Programmname**
    `argv[1]` bis **`argv[argc-1]`** … **restliche Worte**

    Achtung: Zugriff auf **`argv[argc]`** ==> *Absturz!*

argv

Achtung:

Die **argv**-Werte sind ***immer Texte***,
auch wenn sie eine Zahl enthalten!
Sie werden *nicht* automatisch umgewandelt!

Wenn man sie ***als Zahl*** weiterverarbeiten will:

- **atoi(argv[1])** (**atoi** kommt aus **stdlib.h**)
  liefert **argv[1]** als eine *ganze Zahl* (**int**)
- **atof(argv[1])** (**atof** ist auch aus **stdlib.h**)
  liefert **argv[1]** als eine *Kommazahl* (**double**)

Im Fehlerfall (**argv[1]** enthält *gar keine* Zahl) liefern beide einfach **0**.
Ähnliche vordef. Funktionen mit Fehlererkennung: **strtol**, **strtod**

“**`return …;`**” im **`main`**

- *Beendet* das Programm sofort.
- Meldet “…” als *Fehlercode* an das Betriebssystem:
  - **`0`** oder schöner **`EXIT_SUCCESS`** heißt “*alles ok*”:
    Das Programm ist *erfolgreich* fertig gelaufen
  - **`1`** oder schöner **`EXIT_FAILURE`** heißt “*Fehler*”
    (auch **`2`**, **`3`**, … sind erlaubt, bedeuten auch “Fehler”)
- Der Fehlercode … kann in einem DOS-Batchfile als **`ERRORLEVEL`** (in Linux mit **`$?`**) abgefragt werden
- Schöner als “**`return …;`**” ist “**`exit( … );`**”
  (Unterschied später…)

Der ;

Jede Deklaration und jeder Programmbefehl ***endet*** *mit einem* **;**

Wichtigste Ausnahmen:

- Zeilen mit **#** (*Präprozessor*-Befehle): *Kein* **;** !
- *Nach* einer **}** steht *kein* **;** (*vor* einer **}** *schon*!), wenn *Programmbefehle* in den **{ }** stehen!

  Später: Nach **{ }** für *Initialisierungen und Deklarationen* steht *schon* ein **;** !

- *Nach* der **)** bei **if**, **for** und **while**: *Kein* **;** !

  (denn die folgenden **{ }** *gehören* noch zum **if** / **for** / **while** *dazu!*)

Konstanten

- … sind fix ins Programm geschriebene **Werte**.
- … haben außer einem Wert auch einen **Typ**:
  - *Ganze Zahl* (**`int`**)
  - *Kommazahl* (**`double`**)
  - *Einzelnes Zeichen* (**`char`**)
  - *Text* (**`const char[]`** bzw. **`const char *`**)

Der Typ ergibt sich automatisch
aus der Schreibweise der Konstante!

Konstanten für Zahlen

- **`int`**-Konstanten = *ganze Zahlen*: **`1`**, **`-27`**, …
  Keine Zwischenräume, keine Tausender-Trennung!

  (Für Fortgeschrittene: Auch *hexadezimal* **`0xff00`** oder *oktal* **`0770`**, bei **`gcc`** auch *binär* **`0b01101100`**, Anhang **`L`** für **`long`**-Werte: **`-1L`**)

- **`double`**-Konstanten = *Kommazahlen*:
  - Mit **`.`** (***Punkt, nicht Komma!***): **`0.5`**, **`-1.99`**, …
  - Oder mit **`e`** (für "*mal 10 hoch …*"), oder mit beidem:

    `1e6` (= $1*10^{6}$ = 1000000)
    `-1.25e-12` (= $-1.25*10^{-12}$ = -0,000000000000125)

  Eine Konstante mit **`.`** oder **`e`** ist *immer* vom Typ **`double`**, auch wenn der Wert an sich ganzzahlig ist!

Konstanten für Zeichen

Nicht verwechseln!!!

- **`char`**-Konstanten = *Zeichen-Konstanten*:

  Ein ***einzelnes*** Zeichen in ***einfachen*** Hochkomma:

  `'z'`, **`'0'`**, **`'\n'`**, **`'@'`**, …

- *Text-Konstanten* (heißen "**String**" = *"Zeichenkette"*)
  (deren Typ **`char *`** usw. lernen wir später!):

  Null oder mehr Zeichen in ***doppelten*** Hochkomma:

  `""` (der *"Leerstring"*: Text, der *keine* Zeichen enthält)
  `"Ich bin ein Text!\n"`

Lange Text-Konstanten (1)

Eine Text-Konstante in " " darf *nicht*
über mehrere Zeilen gehen:

- Das schließende " muss in *derselben* Zeile wie das öffnende " sein.
- *Zeilenvorschub* innerhalb " ist *verboten*!

Aber:

Mehrere "..." *nacheinander* werden
zu *einer* Textkonstante zusammengehängt
==> einfach ein "..." pro Zeile machen!

Lange Text-Konstanten (2)

Beispiel:

```
printf("Das ist "
  "ein einziger, "
  "ganz langer Text\n");
```

Für Fortgeschrittene:
In uralten C-Programmen hat man Textkonstanten
mit \+Zeilenvorschub auf die nächste Zeile “verlängert”.
Das gilt heute als unschön!

```
printf("Das ist \
ein einziger, \
ganz langer Text\n");
```

\n usw.

- **\n** usw. codiert *Steuerzeichen*.
- Es liefert *ein* Zeichen, *nicht zwei!*
- Die wichtigsten: (es gibt mehr)

  \n … Neue Zeile
  (In Text-Dateien: Linux & Mac: **\n**, Windows: **\r\n**, alte Mac: **\r**)

  \t … Tab

  \0 … ASCII-Wert 0 (brauchen wir später oft!)
  (auch für beliebige andere Oktalzahlen, z.B. **\033** … **ESC**)

  \\, \', \" … der \, das ' und das " selbst
  (das \\ ist wichtig für \ in *Windows-Filenamen*!)

Datendarstellung (1)

"12345" (mit ") und **12345** (ohne ") sind *ganz verschiedene Dinge*:

"12345": *Text*, Typ **`const char[]`** (bzw. **`const char *`**)
6 Bytes im Speicher (eines pro Zeichen),
zeichenweise ASCII-Werte mit 0 als *Ende-Markierung*:

`00110001 00110010 00110011 00110100 00110101 00000000`
(als 6 einzelne Dezimalzahlen: 49 50 51 52 53 0)

12345: *Ganze Zahl*, Typ **`int`**
4 Bytes im Speicher,
32 bit Zweierkomplement-*Binärzahl*:

`00000000 00000000 00110000 00111001`

Datendarstellung (2)

… und **`12345.0`** (mit **`.`**) ist wieder etwas *ganz anderes*:

`12345.0`: *Komma-Zahl*, Typ **`double`**

8 Bytes im Speicher,

Codierung Vorzeichen + 2er-Exponent + Mantisse

gemäß *IEEE 754 Norm*:

```
01000000 11001000 00011100 10000000
      00000000 00000000 00000000 00000000
```

(Details von ASCII, Binärzahlen, IEEE-Kommazahlen:
==› Grundlagen Informatik, ==› Mein altes Infotech-Skript)

Variablen

Die *änderbaren Werte*,
mit denen unser Programm rechnet

bzw. die *Speicherplätze*,
in denen wir unsere Daten im RAM ablegen,

werden im Programm
durch ***Variablen*** dargestellt.

(so ähnlich wie *x*, *y*, ... in Mathe,
aber meistens mit längeren Namen:
`anzahl`, **`max`**, ...).

Deklarationen (1)

Bevor wir mit einer Variable arbeiten können, müssen wir sie *deklarieren*

= dem Compiler sagen,

- wie sie *heißt*
- und *welche Art von Werten* wir darin speichern wollen
  (damit der Compiler weiß, wie viel *Platz* sie braucht)

Die Deklaration muss
vor der ersten Verwendung stehen!

Deklarationen (2)

Beispiel:

```
int anzahl;
```

"Mach mir bitte eine neue Variable
*mit dem Namen "***`anzahl`***".*

Reserviere dir soviel Speicher dafür,
*wie du für eine ganze Zahl (***`int`***) brauchst.*

*Immer, wenn ich in Zukunft "***`anzahl`***" schreibe,*
möchte ich die Zahl an dieser Speicherstelle
lesen oder speichern."

Deklarationen (3)

Weitere Beispiele:

```
char input;      // Variable für ein Zeichen
```

Mehrere Variablen mit `,` :

```
double a, summe, mittel;
```

Mit *Initialisierung* (d.h. mit einem *Anfangswert*):

```
int zahl = 0, stelle = 1;
```

Wenn *keine* Initialisierung angegeben ist:

*Die Variable hat am Anfang einen zufälligen Wert, d.h. enthält **"Blödsinn"**!!!*

==> Vor allem bei Summen usw.: *Zuerst auf* **`0`** *setzen!*

Deklarationen (4)

- Normalerweise schreibt man die Deklarationen *gleich nach der ersten* **`{`** von **`main`** (*vor* allen Befehlen)
- Seit C99 und in C++ (in C89 nicht!!!) sind Deklarationen auch mittendrin *zwischen* Befehlen möglich.
- Erlaubt sind sie auch innerhalb von *inneren* **`{`** , dann gelten sie nur bis zur dazugehörigen **`}`** .
- Schleifenvariablen darf man direkt **`im for`** deklarieren:

```
for (int i = 1; i < argc; ++i) { … }
```

(erst seit C99, **`i`** gilt dann *nur bis zum Ende der Schleife*)

Zusammenfassung: Variablen …

- Haben einen **Namen**
  (frei wählbar, hoffentlich einen *sinnvollen*!).
- Können sich einen **Wert** merken (speichern).
- Die *Art* des Wertes (Zahl, Buchstabe, Text, …),
  den man in der Variable speichern kann,
  wird durch den **Typ**
  in der *Deklaration* festgelegt.
- Bekommen vom Compiler einen **Speicherplatz**
  im Hauptspeicher (RAM) zugeteilt
  (*je nach Typ* in der richtigen *Größe*).
- Haben eine **Lebensdauer** (“Wie lange bleibt der Speicher reserviert?”) und eine **Sichtbarkeit** (“In welchem Teil des Programmes kann ich darauf zugreifen?”).

Namen

- Aus *Buchstaben* und *Ziffern*
- "**_**" ("Underscore") gilt als Buchstabe,
  sonst *keine Sonderzeichen*!
- *Keine Umlaute*, kein ß !!!
- *Erstes Zeichen* muss *Buchstabe* oder "**_**" sein,
  keine Ziffer: **`x1`** ist ok, **`1x`** ist verboten
- *Keine vordefinierten Wörter* (**`int`**, **`if`**, **`while`**, ...)
- Groß- und Kleinschreibung wird *unterschieden!*

  ==> **`count`**, **`Count`** und **`COUNT`**
  sind *3 verschiedene Namen!*

Üblicher Stil für Namen

- *Zählvariablen* usw.: Meist **`i`**, **`j`**, **`k`**
  (Der Zähl-Ende-Wert heißt oft **`n`**)
- *Häufig benutzte Variablen* mittellang (1 Wort):

  `summe`, **`anzahl`**, **`result`**, …
- *Konstanten, seltene Variablen,* … lang, 2 übliche Stile:

  `langer_var_name` (seit den Anfängen von C, Linux)
  `langerVarName` (*"CamelCase"*, kommt von Java)
- Variablen meist *klein,* jedenfalls *nicht komplett groß*
  (*historisch:* komplett groß nur für **`#define`**-*Konstanten*)
- Kein "Denglisch"! (schlechtes Beispiel: **`drawBalken`** ???)

Typen

- **`int`** … *Ganze Zahl*

  Bei uns 4 Bytes: **-2^{31}** bis **$+2^{31}-1$**
  (d.h. die kleinste negative Zahl hat *kein positives Gegenstück*!)

- **`double`** … *Kommazahl*

  -10^{308} bis 10^{308}, gut 16 Stellen genau, min. 10^{-308}

  Auch: **`float`** (nur $+-10^{+-38}$, nur 6 Stellen genau)

  Braucht 8 (**`double`**) bzw. 4 (**`float`**) Bytes Platz.

- **`char`** … *Ein* Zeichen (Details gleich…)

- **`bool`** … Ein *Wahrheitswert* (kommt später!)

Typ **`auto`**

Wie in **C++**, erst seit **C23** und nur bei
Deklarationen mit Initialisierung:

`auto` *var* **`=`** *wert***`;`**

> "Gib der Variable *var* den Typ,
> den ihr Initial-Wert *wert* hat"

Beispiel: **`auto umfang = 2 * 3.141592;`**

==> **`umfang`** wird als **`double`** deklariert,
weil **`2 * 3.141592`** ein **`double`**-Wert ist

Empfehle ich *nicht*!

`auto` hatte in **K&R C** eine *ganz andere Bedeutung*, z.B. **`auto int i;`**

Mehr **`int`**-Typen

Nur für Fortgeschrittene:

- **`short int`**, **`long int`**, **`long long int`**
  (dabei kann man **`int`** auch *weglassen*)

  ==> Verschiedene *Speichergröße*, verschiedener *Wertebereich*

  Implementierungs-abhängig: **`short`** / **`int`** / **`long`** / **`long long`**
  Windows/Linux 32 bit: 2 / 4 / 4 / 8 Bytes = 16 / 32 / 32 / 64 Bits
  Windows/Linux 64 bit: 2 / 4 / 8 / 8 Bytes = 16 / 32 / 64 / 64 Bits
  Mikro-Kontroller: 1 oder 2 / 2 / 4 / --- Bytes = 8 oder 16 / 16 / 32 / --- Bits

- **`unsigned int`** usw.:

  Vorzeichenlose Werte: Bei n Bit Länge 0 ... 2^n-1 statt -2^{n-1} ... 2^{n-1}-1

- Für Spezialzwecke: **`size_t`** (für Längen), **`time_t`** (Zeiten) usw.
  (**`size_t`** ist meist als Abkürzung für **`unsigned long`** definiert)

- **`char`**, **`bool`** und **`enum`**-Typen sind auch *"mit* **`int`** *verwandt"*

Verwendung von **`double`**

… *nur wenn notwendig:*

- ***Komma***zahlen
- Zahlen, für die **`int` zu klein** ist (ab 10 Stellen)
  (**`double`** ist für *ganze Zahlen mit max. 16 Stellen exakt*!)

weil:

- **`double`** hat *Konvertierungsverluste und Rundungsfehler*

  `0.1` dezimal ist z.B. eine *unendlich-periodische* Binär-Kommazahl
  ==› wird *abgeschnitten*!

- **`double`** ist *langsamer*, braucht *mehr Platz!*

char (1)

- Auch Zeichen sind intern *ganze Zahlen*:

  *Zahlenwert eines Zeichens = sein **ASCII-Wert***

  Beispiele: **'0'** ... 48, **' '** ... 32, **'\n'** ... 10

- Wertebereich von **char** (je nach Compiler):
  -128 bis 127 oder 0 bis 255
  (Für Fortgeschrittene: ev. **signed char** oder **unsigned char** verwenden!)

  ==> Umlaute usw. können *negativ* sein!

  ==> Achtung bei **char**-Werten als *Index* (in **[]**)!

- Braucht 1 Byte Platz
  (Für Fortgeschrittene: Typ **wchar** (*"wide char"*) für *UTF-16 Unicode*: 2 Bytes)

`char` (2)

Man kann mit **`char`**-Werten ganz normal "*rechnen*" und **`char`** und **`int`** beliebig *mischen*.

Beispiele:

- Wenn "**`char c;`**" ein Ziffern*zeichen* **`'0'`** bis **`'9'`** ist und "**`int z;`**" der Ziffern*wert* werden soll:

  ```
  z = c - '0';
  ```

- Wenn "**`int z;`**" eine *Zahl* zwischen **`0`** und **`9`** ist und "**`char c;`**" die Ziffer (als *Zeichen*) werden soll:

  ```
  c = z + '0';
  ```

Benannte Konstanten (1)

- Verbessern die *Leserlichkeit:*

  "Was bedeutet diese Zahl???"
  (Limits, Naturkonstanten, Umrechnungsfaktoren, ...)

  `M_PI` sagt auf einen Blick mehr als **`3.14159265359`** !

- Verbessern die *Änderbarkeit:*

  Verwende **`MAX_MESSWERTE`** statt **`1000`**

  Wenn es später einmal 5000 statt 1000 sein sollen:
  Wert muss *nur an einer Stelle geändert* werden
  statt an vielen (hunderten?) Stellen im Programm

Benannte Konstanten (2)

... werden normalerweise *ganz oben*
(*nach* **`#include`**, aber *vor* **`main`**)
definiert!

3 Möglichkeiten:

- **`#define`**
- **`const`**
- **`enum`** (siehe Ende des Semesters)

#define

#define *NAME wert*

Beispiel: **#define LINELEN 80**

Achtung: *Ein* **#define** pro Zeile, *kein* **=** und *kein* **;** !

Üblich (aber nicht Pflicht):

NAME komplett großschreiben

(siehe vordefinierte Konstanten wie **M_PI**, **M_E** usw.!)

Wirkung:
Der Präprozessor *ersetzt vor* dem eigentlichen Compilieren
überall im Programmtext *NAME* durch *wert* (**LINELEN** durch **80**)

Anmerkung: **#define** ist uralt
==> Statt **#define** sollte *in C++* **const** verwendet werden!

`const` (1)

Normale Variablen-Deklaration mit **`const`** *davor:*

`const` *typ name* **`=`** *wert* **`;`**

`const` heißt

"Der Wert der Variablen ist konstant (fix)":
Er darf nach ihrer Initialisierung
nicht mehr geändert werden

==> *Keine spätere Zuweisung* auf die Variable

==> Kein **`++`** oder **`--`** usw.

Initialisierung (Wertzuweisung) *nur in der Deklaration*
(Eine **`const`**-Variablen-Deklaration *ohne* Initialwert ist *sinnlos*!)

`const` (2)

Beispiel: **`const int LineLen = 80;`**

Funktioniert wie eine normale Variable,
aber **`const`** hat Vorteile:

- Der Leser erkennt sofort:

  Das soll eine Konstante sein, keine Variable!

- Der Compiler bemerkt *versehentliches Ändern*

  ==> *Fehlermeldung*!

- Der Compiler erzeugt mit etwas Glück
  effizienteren Code

`const` (3)

Technischer Unterschied:

- **In C:**

  Eine **`const`**-Variable ist eine *"nur lesbare Variable"*

  ==> Belegt Speicherplatz
  ==> Kann *nicht* verwendet werden,
  wo eine *echte Konstante* (d.h. Zahl) nötig ist

  Behoben in **C23**: **`constexpr`** statt **`const`** erzeugt eine *echte Konstante*, die *wie eine Zahl* verwendet werden kann

- **In C++:**

  Eine **`const`**-Variable ist eine *echte Konstante*
  (wird wie bei **`#define`** direkt *durch den Wert ersetzt*)

“Rechnungen”

- Normale Rechnungen
- Die Zuweisung: Speichern
- Increment (++) und Decrement (--)
- Typumwandlungen
- Funktionsaufrufe
- `printf` usw., C++ Ein- und Ausgabe
- Zufallszahlen

“Rechnungen”

Eine “Rechnung” (Programmierer-Deutsch:
“(Arithmetischer) Ausdruck”, engl. “expression”)
wird ausgerechnet und liefert einen Wert als Ergebnis.

Besteht aus:

- ***Variablen*** (Wert = *aktueller Inhalt* der Variable)
- ***Konstanten***
- ***Operatoren*** = “Rechenzeichen” und ***Klammern***
- ***Funktionsaufrufen*** (wie in Mathe: **`sin(x)`**, **`abs(n)`**, …)
- ***Typumwandlungen*** und **`sizeof`** (siehe bei Arrays!)
- ***Zuweisungen*** (**'='** ist in C auch ein “Rechenzeichen”)

Operatoren

- **`+ - * /`** … wie üblich: **`*`** und **`/`** vor **`+`** und **`-`**

  C kennt *kein* Rechenzeichen für "*hoch*"!
- **`a % b`** … *Rest von ganzzahliger* Division **`a / b`**

  `%` gibt es *nur* für **`int`**, *nicht* für **`double`**
- **`-a`** … Vorzeichen umdrehen
- **`( )`** … Klammern wie üblich: Vorrang
- Weitere folgen…!
- Für Fortgeschrittene: *Bit-Operatoren* (nur für **`int`**, siehe Bit-Kapitel!):

  `& | ^ ~` … bitweises AND, OR, XOR, NICHT

  `<< >>` … bitweises Verschieben nach links / rechts

  (in C++ auch: *Ausgabe und Eingabe*)

Die Tücken des % (Rest)

Für Fortgeschrittene:

Achtung bei negativen Zahlen!

- *Vor C99* war **%** für negative Zahlen *undefiniert*:

  `-10 % 6` durfte **`2`** *oder* **`-4`** liefern!

- *Seit C99* ist das *negative* Ergebnis definiert,
  d.h. **%** entspricht *nicht* dem "**mod**"-Operator in Mathe!

- Leider würde man fast immer das **mod**-Verhalten brauchen
  (d.h. ein garantiert positives Ergebnis!),
  aber das gibt es in C nicht als fertigen Operator...

==> *Am besten kein **%** für negative Zahlen verwenden,*
wenn doch: Aufpassen!

Fehler und Überlauf bei **`int`**

- … **`/ 0`** und … **`% 0`** : Harter *Programmabbruch*
- Alle *Überläufe*:

Keine Meldung, es wird
"im Kreis herum" normal weitergerechnet!

Nach 2147483647 kommt -2147483648 und umgekehrt!
Auch zu große Konstanten werden stillschweigend "passend gemacht"!

Für Techniker:
Nur die *hinteren 32 Bits* werden gespeichert,
das Überlauf-Bit bzw. alles davor wird *ignoriert*.

Für Mathematiker:
Es wird so lange 2^{32} dazu- oder weggezählt,
bis das Ergebnis in -2^{31} … $2^{31}-1$ liegt (d.h. Rechnung mod 2^{32}).

Fehler und Überlauf bei **`double`**

Keine Meldung,
es wird mit einem *Fehlerwert* weitergerechnet:

- "**`Inf`**" (*"Infinity"* = unendlich) bei *Überlauf*

  Zwei separate Werte: *"plus unendlich"* und *"minus unendlich"*

- "**`NaN`**" (*"Not a number"* = undefiniert)
  bei *undefinierten Operationen* (negative Wurzel, ...)

In **`math.h`** gibt es Funktionen **`isinf`**, **`isnan`**, ...,
mit denen man auf diese Fehlerwerte *prüfen* kann.

Für Fortgeschrittene:
Der *Rundungsmodus* (vor allem in der Mathematik / Physik wichtig!)
und die *Fehlerbehandlung* lassen sich *umstellen*!

“Gemischte” Rechnungen (1)

- **Solange *beide* Operanden `int` sind:**
  `int`-Rechnung, **`int`**-Ergebnis
- **Wenn ein Operand `double` ist:**
  Anderer Operand wird *automatisch zuerst* auf **`double`** *umgewandelt*:
  `double`-Rechnung, **`double`**-Ergebnis

Wichtig vor allem bei *Division*:

- **`int / int`**: Division & Ergebnis **`int`**, *abgeschnitten, nicht gerundet*: **`3/4`** ergibt **`0`** !!!
- Wenn *mindestens eine Seite* **`double`** ist:
  Division & Ergebnis **`double`**, *exakt*: **`3./4`** ergibt **`0.75`**

“Gemischte” Rechnungen (2)

Achtung:

Ob **`int`** oder **`double`** gerechnet wird,
wird Schritt für Schritt (*pro Rechenzeichen*) entschieden,
nicht für die ganze Rechnung auf einmal!

Beispiel: **`(i / 4) * M_PI`**

- *Zuerst* **`i / 4`**: Beide Operanden **`int`**

  ==> **`int`**-Rechnung, Ergebnis wird *abgeschnitten*!

- *Dann* **`(...) * M_PI`**: Links **`int`**, rechts **`double`**

  ==> abgeschnittenes **`int`**-Ergebnis von **`(...)`**
  wird in **`double`** umgewandelt, **`double`**-Rechnung

“Gemischte” Rechnungen (3)

Für Fortgeschrittene:

Dasselbe gilt für Rechnungen mit *verschiedenen* **`int`**-*Typen*
oder mit **`float`** und **`double`** gemischt:

- Es wird mit dem Typ des *längeren* Operanden gerechnet,
  der kürzere Operand wird zuerst automatisch umgewandelt.

  Auch das Ergebnis hat den Typ des längeren Operanden.

- Ist ein Operand (oder beide) **`unsigned`**,
  wird **`unsigned`** gerechnet (mit Ergebnis **`unsigned`**).

  Ist der andere Operand negativ,
  wird sein Bitmuster einfach als positive Zahl interpretiert
  (*nicht* das Vorzeichen weggelassen!!!).

(die Details sind sehr kompliziert…)

Die Zuweisung (1)

variable **=** *rechnung* **;**

Zuerst *rechnung ausrechnen*,
dann das Ergebnis in *variable speichern*:
"von rechts nach links"

In Computer-Deutsch: *"**Zuweisung**" (engl. "assignment"):*
"Weise der variable das Ergebnis der rechnung zu!"

Das ist *keine* Gleichung im mathematischen Sinn,
die linke Seite wird *nicht* ausgerechnet!

Beispiele:

```
mittelw = (a + b) / 2;
zahl = zahl * 10 + ziffer;
```

Die Zuweisung (2)

Nicht verwechseln:

- **Ein =** ist eine *Zuweisung*

  Rechte Seite ausrechnen, Ergebnis links **speichern**.

  Zwei == ist ein *Vergleich* (für **`if`** usw.)

  Beide Seiten ausrechnen, Ergebnisse **vergleichen**, liefert *"wahr" oder "falsch"!*

Die Zuweisung (3)

Sonderformen:

- **Mehrfachzuweisung**: `a = b = c = ... = x / 2;`

  `x / 2` wird in *allen* angegebenen Variablen gespeichert (von *rechts nach links*)

- **Abkürzung**: `a += x;`

  Ist *dasselbe* wie `a = a + x;`

  "Zähle **x** *zu* **a** *dazu"*

  Auch für `-=`, `*=`, ...:

  `n *= 10;` // *"Multipliziere* **n** *mit* **10**"

Die Zuweisung (4)

Nur für Fortgeschrittene:

Das *"Ergebnis"* einer Zuweisung
ist der *zugewiesene Wert*,
man kann damit *weiterrechnen*!

==> Eine Zuweisung könnte
mitten in einer Rechnung vorkommen!

Beispiel:

```
q_summe = q_summe + (x2 = x * x);
```

Sehr unschön, nicht nachmachen!!!

Aber: Sehr häufig in *Schleifen-Bedingungen* (siehe dort!)

“Gemischte” Zuweisungen

Grundsätzlich muss der ***Typ***
auf ***beiden Seiten*** einer Zuweisung
(d.h. der Typ des Rechenergebnisses
und der Typ der Variablen)
gleich sein!

`text = zahl;` *und* **`zahl = text;`** *geht nicht!*

Ausnahmen:

Man darf **`int`** auf **`double`** zuweisen und umgekehrt.

Aber:

Bei **`double`** auf **`int`** wird *abgeschnitten, nicht gerundet!*

`++i` und `--i`

- … sind dasselbe wie `i = i + 1;`
  und `i = i – 1;`
- … heißen

  *“**Increment**” = “zähl eins dazu”*

  und

  *“**Decrement**” = “zähl eins weg”*
- … können *überall* verwendet werden,
  nicht nur hinten im **`for`**

`++i` versus `i++` (1)

`i++` und **`i--`** zählen *genauso* eins dazu / weg

- *Kein Unterschied*, wenn sie *allein* stehen
  (z.B. hinten im **`for`**)
- *Verschieden*, wenn sie *innerhalb einer Rechnung* stehen

  (d.h. wenn der Wert von **`++i`** *gleich weiterverwendet* wird):

  - Bei **`++i`** / **`--i`** ist das Ergebnis der **neue** Wert von **`i`**:

    "Zuerst dazuzählen, dann Variable anschauen"
  - Bei **`i++`** / **`i--`** ist das Ergebnis der **alte** Wert von **`i`**
    (*vor* dem Dazuzählen):

    "Zuerst Variable anschauen, dann dazuzählen"

++i versus i++ (2)

```
i = 1;

printf("%d\n", i);
// 1

printf("%d\n", ++i);
// 2 !!!

printf("%d\n", i);
// 2

//=====

i = 1;

printf("%d\n", a[++i]);
// Wert von a[2]
```

```
i = 1;

printf("%d\n", i);
// 1

printf("%d\n", i++);
// 1 !!!

printf("%d\n", i);
// 2

//=====

i = 1;

printf("%d\n", a[i++]);
// Wert von a[1]
```

Falle bei `++i` usw.

Wenn man `++i` in einer Rechnung / Zuweisung verwendet,

darf `i` *kein zweites Mal*
in derselben Rechnung vorkommen!

Weil: Die *Reihenfolge* ist *undefiniert*!!!

Beispiel:

```
i = 0;
a[++i] = i;
```

- Wenn zuerst `++i`, dann rechte Seite: `a[1] = 1;`
- Wenn zuerst rechte Seite, dann `++i`: `a[1] = 0;`

Beides ist zulässig!

Die Typumwandlung

((*neuer typ***)** *variable***)** oder **((***neuer typ***) (***rechnung***))**

- Bei uns vor allem (in Divisionen): **((double) i)**

  "Verwandle den **int i** *in einen* **double***-Wert"*

  Beispiel:

  ((double) i) / j rechnet *exakt* (mit **double**)

  Umgekehrt z.B.:

  ((int) (sqrt(x))) / n rechnet das **/** mit **int**

- Mit **(...)** **nicht** möglich:

  Umwandlungen Zahl ‹==› Text !!!

Die Typumwandlung (2)

Für Fortgeschrittene:

In C++ (*nicht* in C) gibt es *6 verschiedene Typumwandlungen*:

- Die eben beschriebene C-Typumwandlung
- Dasselbe, angeschrieben wie ein Funktionsaufruf:

  neuer_typ`(`*wert*`)` Beispiel: **`double(i)`**
- 4 Spezialkonstrukte (lernen wir hier nicht)

double ==> **int**

Achtung:

Zuweisung **double** auf **int**
und Typumwandlung **((int) (double**-*Rechnung*))

runden nicht, sondern schneiden ab!

Funktionen für *kaufmännische Rundung* (aus **math.h**):

```
i = lround(x);  // double rein, int raus

x = round(x);
   // double rein, ganzzahliger double raus
```

Funktionen (1)

Funktion =
Vordefiniertes, benanntes Programmstück,
das wir in unserem Programm
verwenden (Computer-Deutsch: "aufrufen") können.

Beispiele:

```
c = sqrt(a*a + b*b);
zeichen = getchar();
value = pow(10, expon);
```

Auch *geschachtelt* möglich:

```
anzahl = abs(atoi(argv[1]));
```

Funktionen (2)

Was passiert bei einem Funktionsaufruf?

- Wir stecken *Werte hinein*: Die Werte in den **(…)** (keinen, einen oder mehrere Werte)
- Die *Funktion* wird ausgeführt, *rechnet* etwas
- … und liefert einen *Ergebniswert*
- *Wir rechnen* mit diesem Ergebniswert *weiter* (oder speichern ihn, …), so als ob *statt dem Funktionsaufruf sein Ergebnis* dastünde
- Oft: Die Funktion wird *nicht wegen ihres Ergebniswertes*, sondern wegen ihrer *"Nebenwirkung"* aufgerufen

  Beispiel: **`printf`**, **`exit`**, **`system`**

Funktionen (3)

Die wichtigsten bisher verwendeten Funktionen:

- **`printf`** (Ausgabe): **`stdio.h`**
- **`atoi`**, **`atof`** (Umwandlung Text ==› Zahl): **`stdlib.h`**
- **`system`** (Ausführen eines Windows-Befehls): **`stdlib.h`**
- **`getchar`** (Lesen *eines* Zeichens vom Terminal): **`stdio.h`**
- **`rand`**, **`srand`** (Zufallszahlen): **`stdlib.h`**
- **`abs`**, **`fabs`** (Wert ohne Vorzeichen): **`stdlib.h`** / **`math.h`**
- **`exit`** (Programmende): **`stdlib.h`**
- **`sqrt`**, **`pow`**, **`sin`**, **`log`**, **`lround`** ... (Math. Funkt.): **`math.h`**

printf (1)

printf macht *Ausgabe* mit *Lückentext-Ersetzung*:

- Als erstes kommt der *Lückentext* (in **" "**):

  Jedes **%s**, **%d** usw. im Lückentext
  ist eine *"Lücke"*, wird durch einen *Wert ersetzt*

  Was nicht **%**... ist, wird *unverändert* ausgegeben

- Dann kommen die *einzusetzenden Werte*,
  in der Reihenfolge der **%**

  Die Werte müssen betr. Anzahl und Typ
  zu den **%** *passen!* *(sonst oft: Absturz!)*

```
printf("Hallo %s, Platz %d\n", argv[i], i);
```

`printf` (2)

Die "Lücken":

- **`%d`** (oder gleichbedeutend **`%i`**) … *Dezimalzahl* (**`int`**)
- **`%u`** … "unsigned": vorzeichenlose Dezimalzahl (**`int`**)
- **`%e`**, **`%f`**, **`%g`** … *Komma-Zahl* (**`double`**)
  - **`%e`**: Immer in **`e`**-Darstellung
  - **`%f`**: Immer *mit Komma* aber *ohne* **`e`**
  - **`%g`**: Flexible, kürzestmögliche Darstellung je nach Wert
- **`%s`** … "string": *Text* (**`char[]`** oder **`char*`**)
- **`%c`** … ein *einzelnes Zeichen* (**`char`**)
- **`%%`** … *keine* Lücke, sondern es wird ein **`%`** ausgegeben
- **`%x`**, **`%o`**, **`%b`** … Ganze Zahl (**`int`**) *hexadezimal*, *oktal oder binär*

`printf` (3)

Lücken-Tricks, Formatierung:

- **`%5d`** … mit (mindestens) 5 Zeichen *Breite* ausgeben
- **`%-5s`** … dasselbe, aber *linksbündig* statt rechtsbündig
- **`%05d`** … dasselbe, aber mit führenden **`'0'`** statt **`' '`**
- **`%+d`** … auch **`+`** explizit ausgeben, nicht nur **`-`**
- **`%ld`** *(kleines L)* … Wert ist ein **`long int`** statt **`int`**
- **`%10.4f`** … insges. mit 10 Zeichen Breite ausgeben, davon 4 *Nachkomma*-Stellen
- … und noch ein paar mehr!

Fehlermeldungen (1)

1) Fehlermeldungen werden mit

`fprintf(stderr, "`*lückentext***`", `** *werte***`);`**

ausgegeben statt mit

`printf("`*lückentext***`", `** *werte***`);`**

2) Fehlermeldungen sollten

`argv[0]`

(den *Programmnamen*) enthalten!

Fehlermeldungen (2)

Der **Default-Effekt** ist *derselbe* wie bei **`printf`**:

Die Ausgabe von **`fprintf`** wird
genauso am Bildschirm angezeigt!

Aber:

Die Befehlszeile (oder das übergeordnete Programm)
kann **`stderr`**-Ausgabe
von normaler Ausgabe unterscheiden
und ***getrennt behandeln***
(z.B. bei *Pipes* oder *Umleitung in eine Datei*)

Details bei "*File-I/O*" und "*Fehlerbehandlung*"

Andere C Ein- und Ausgabe

- Es gibt noch *einige weitere Ausgabe-Funktionen*:

  `putchar(c)` gibt z.B. ein *einzelnes Zeichen* **`c`** aus.

- Das Eingabe-Gegenstück zu **`printf`** ist **`scanf`**

  ==> Kompliziert!
  ==> *Fehleranfällig* (!!!) und selten verwendet!

  ==> Eingabe in C in der Praxis:
  Meist *zeilenweise* mit **`fgets`**
  oder *zeichenweise* mit **`fgetc`** oder **`getchar`**

  ==> Formatierte Eingabe ist *in C++ viel praktischer*
  als in C (aber *genauso fehleranfällig!*)

scanf

```
scanf("%d %s %f", &i, txt, &x);
```

- Zuerst beschreibt der "Lückentext",
  welche Art von Daten gelesen werden soll
  (was nicht "Lücke" ist, muss *ident zum Input* sein!)
- Dann kommen die *Variablen*, in die
  die *gelesenen Werte gespeichert* werden sollen

Das & nicht vergessen, außer bei %s !!!

(Erklärung viel später)

- *Fehlerbehandlung* ist in der Praxis Pflicht,
  für uns hier zu aufwändig (siehe Kap. "File-I/O")!

Längenangaben bei **`scanf`**

Grundsätzlich bei **`printf`** und **`scanf`**:

Längenangaben nicht vergessen!!!

`%hd` für **`short int`**, **`%ld`** für **`long int`**,
`%lld` oder **`%Ld`** für **`long long int`**, und weitere

Achtung bei Kommazahlen:

`printf`: Default bei **`%f %e %g`** ist **`double`**

`float` werden *automatisch* als **`double`** übergeben
==> auch **`%f`**

`scanf`: Default bei **`%f %e %g`** ist **`float`** (!!!), *nicht* **`double`**

`double` *muss* mit **`%lf %le %lg`** eingelesen werden!

C++ Ausgabe (1)

- *Oben im Programm (muß C++ sein:* Filename **.cpp**)

  #include <iostream> (statt **<stdio.h>**)
  using namespace std;

- *Ausgabe:*

  cout << *rechnung* **;** oder **cout << "Text";**

  Bei mehreren Werten und Texten:

  cout << "..." << ***rechnung1*** **<< "..." << ... ;**

  Mit Zeilenvorschub:

  cout << ... << endl;

  Der *Typ* der Werte wird *automatisch* erkannt!

C++ Ausgabe (2)

- *Beispiel:*

```
cout << "Hallo, " << argv[i]
<< ", " << i << ". Platz!"
<< endl;
```

Achtung: *Keine* automatischen `' '` zwischen den Werten!

- *Formatierung:*

Feldbreite, führende 0, linksbündig, hex, ...:
Geht auch, ist aber *komplizierter* als bei **`printf`**

==> *Lernen wir nicht, geht mit* **`printf`** *besser!*

(für Interessierte: Suche nach "**`iomanip`**"...)

C++ Eingabe (1)

- **<iostream>** und **using ...** wie bei **cout <<**
- *Eingabe:*

  cin >> *variable* **;**

  Typ der Variable (**int**, Text, ...) wird erkannt, *Umwandlung* erfolgt *automatisch*

- *Beispiel:*

```
int wurf;
cout << "Was hast du gewürfelt? ";
cin >> wurf;
```

C++ Eingabe (2)

- *Leerzeichen und Leerzeilen werden bei* **>>** *übersprungen, es wird immer genau ein Wort gelesen.*

  ==> Bei Text: Man kann so *keine ganze Zeile* lesen!

  ==> Bei **`char`**: Man kann so *kein* **`' '`** oder **`'\n'`** lesen!

- Verhalten bei *Eingabefehlern*
  (z.B. Zahl erwartet, aber Text eingetippt)
  ist *ganz schlimm: Oft Endlosschleife!*

  ==> Darf so in "echten" Programmen
  nicht verwendet werden !!!

  ==> Korrektes, "*deppensicheres*" Einlesen mit **>>**
  ist *sehr kompliziert*, lernen wir hier nicht!

Zufallszahlen

Programme brauchen oft *“zufällige” Werte*:
Für Simulationen, für Kryptographie, zum Programmtest, für zufälliges Verhalten bei Spielen / Lernprogrammen, ...

Unterscheide:

- **“Echte” (physikalische) Zufallszahlen:**

  Nicht standardisiert, kompliziert ==> Lernen wir nicht.

- **Pseudo-Zufallszahlen: Berechnet, nicht zufällig!**

  Viele verschiedene Funktionen, je nach C-Version...

  Am *primitivsten*, immer verfügbar: **`rand()`**
  Liefert *bei jedem Aufruf eine andere ganze Zahl* (**`int`**)
  zwischen 0 und mindestens 32767 (meist 2147483647)

`rand()` und **`srand(...)`** (1)

- **`rand()`** “würfelt” nicht,
  sondern *berechnet* eine *fixe, zyklische Folge* von Zahlen
- **`srand(...)`** setzt den *Startpunkt* in dieser Folge,
  man macht es daher ***einmal** zu Programmstart*:
  - *Kein* **`srand(...)`**
    oder **`srand(...)`** mit *fixem* Wert:

    ==> **`rand()`** liefert bei *jedem Programmlauf*
    exakt dieselbe Folge von Zufallszahlen
  - **`srand(...)`** mit jedesmal *anderem* Wert:

    ==> **`rand()`** liefert bei jedem Programmlauf
    andere Zufallszahlen (beginnt bei anderem Wert)

rand() und **srand(...)** (2)

- Gute Quelle für jedesmal andere Zahl für **srand(...)**:

  time(NULL) … *aktuelle Zeit in Sekunden* seit 1.1.1970
  (kommt aus **time.h**)

  Daher *am Anfang* von **main**: **srand(time(NULL));**

- **rand** wird meist zusammen mit **%** verwendet:

  rand() % 2 … "zufällig" 0 oder 1
  rand() % 6 + 1 … "zufällige" Zahl von 1 bis 6
  rand() % 101 … "zufällige" Zahl von 0 bis 100

  Nach **%** steht immer die ***Anzahl der Möglichkeiten***!

- **rand** und **srand** kommen aus **stdlib.h**

Befehle

- `if` und der "`? :`"-Operator
- Schleifen:

  `while`, `do`, `for`, mehr `for`
  Schleifen-Bedingungen und Hinweise
  `break` und `continue`

- Bedingungen und Wahrheitswerte:

  Vergleiche, `&&` `||` `!`, einfache Werte
  `=` versus `==`, Zuweisungen in Bedingungen

- `switch` und `goto`

`if` (1)

Das **`if`** ist der

“Entweder - oder - Befehl”:

Prüfe eine Bedingung (= Frage)
(meistens: Vergleiche etwas)

und mach abhängig davon
entweder das eine
oder das andere.

`if` (2)

```
if (wenn-frage) {
   dann-befehle;
}
```

```
if (wenn-frage) {
   dann-befehle;
} else {
   sonst-befehle;
}
```

```
if (frage-1) {
   befehle-1;
} else if (frage-2) {
   befehle-2;
} else if (frage-3) {
   befehle-3;
} ...
 ...
} else {
   sonst-befehle;
}
```

`if` (3)

- Wenn die *wenn-frage* "**wahr**" ergibt,
  werden die *dann-befehle einmal ausgeführt*
  (und die *sonst-befehle übersprungen*).
- Wenn die *wenn-frage* "**falsch**" ergibt,
  werden die *dann-befehle übersprungen*
  und die *sonst-befehle einmal ausgeführt*
  (oder *gar nichts*, wenn *kein* **`else`** da ist).
- Bei mehreren zusammenhängenden **`if`** - **`else if`** - …
  wird genau ***eine*** Gruppe von Befehlen **`{`** … **`}`** ausgeführt
  (oder gar keine, wenn alle Fragen falsch ergeben
  und kein **`else`** da ist):
  Die Befehle der *ersten wahren* Bedingung bzw. Frage.

`if` - Beispiele

```
if (x < 0) {
  x = -x;
}
```

```
if (a < b) {
  min = a;
} else {
  min = b;
}
```

```
if (x > 0) {
  printf("Positiv!\n");
} else if (x < 0) {
  printf("Negativ!\n");
} else {
  printf("Null!\n");
}
```

Der “`?  :`”-Operator

frage **`?`** *wahr-rechnung* **`:`** *falsch-rechnung*

Heißt ***“Conditional operator”***

Macht ein **`if`** *innerhalb einer Rechnung*
(d.h. *liefert einen Wert* als Ergebnis!)

1) Die *frage* wird ausgerechnet.
2) Wenn die *frage* “wahr” ergibt,
   wird die *wahr-rechnung* ausgerechnet,
   sonst die *falsch-rechnung*.

   Es wird immer nur ***eine*** der beiden Rechnungen ausgerechnet, nie beide!

Der “**? :**”-Operator: Beispiele

// wenn **a** größer **b** ist, ist das Maximum **a**, sonst **b**

```
max = (a > b) ? a : b;
```

// setze **vorz_x** auf **–1** bei negativem **x**, sonst auf **1**

```
vorz_x = (x < 0) ? –1 : 1;
```

// gib aus, ob **n** gerade oder ungerade ist

```
printf("%d ist %s\n", n,
       (n % 2 == 0) ? "gerade" : "ungerade");
```

// Prozentanteil auch für den Fall, dass alle Werte **0** sind
// Bei **summe == 0** wird “… **/ summe**” *nicht ausgerechnet!*

```
proz = (summe == 0) ? 0 : ((100 * a) / summe);
```

Schleifen

Stärke und Hauptaufgabe von Computern ist,
eine Folge von Befehlen immer wieder zu wiederholen
(z.B. für jede Eingabe, jeden Eintrag einer Datenbank, ...)

Solche ***Wiederhol-Befehle*** heißen ***"Schleifen"*** (***"Loop"***).

Es gibt *zwei Arten:*

- Wiederhole, solange eine *Bedingung gilt*
  (ohne vorher zu wissen, wie oft, und ohne mitzählen)

  ==> **`while`**-Schleife = ***"Solange-Schleife"***
- Wiederhole eine *bestimmte Anzahl* mal, *zähl mit*

  ==> **`for`**-Schleife = ***"Zählschleife"***

`while`

`while (`*solange-bedingung***`) {`**
    *befehle …***`;`**
`}`

Solange die *solange-bedingung* “**wahr**” ergibt,

- *wiederhole* die *befehle* immer wieder
- und *prüfe* die *solange-bedingung* *vor* jeder Wiederholung nochmals.

Ist die *solange-bedingung* “**falsch**”, mach nach der **`}`** weiter.

Die *solange-bedingung* wird ***vor*** den *befehlen* geprüft:
    Ist sie *von Anfang an “falsch”*,
    werden die *befehle* ***gar nicht*** ausgeführt.

`while` - Beispiele

```
summe = 0;
while (n > 0) {
  summe += n % 10;
  n /= 10;
}
```

```
while (b > 0) {
  r = a % b;
  a = b;
  b = r;
}
```

do … **while**

do {
   *befehle***;**
} while (*solange-bedingung***);**

Dasselbe wie **while**, aber
die *solange-bedingung* wird ***nach*** den *befehlen* geprüft:

Die *befehle* werden daher ***mindestens einmal*** ausgeführt,
auch wenn die *solange-bedingung* von Anfang an falsch ist!

Sinnvoll, wenn erst in den *befehlen* berechnet wird,
was in der *solange-bedingung* geprüft wird.

Achtung:
Hier gehört *ausnahmsweise* ein **;** nach den **( )** !!!

do … **while** - Beispiel

```
summe = 0;
do {
    cout << "Was sagt der Würfel? ";
    cin >> wurf;
    summe += wurf;
} while (wurf != 6);
```

(**cout <<** … ist die C++-Ausgabe,
cin >> … ist die C++-Eingabe)

`for` (1)

```
for (anfangszuweisung; solange-bedingung; weiterzählen) {
  befehle;
}
```

- Mach am Anfang *einmal* die *anfangszuweisung*.
- *Solange* die *solange-bedingung* "**wahr**" ergibt,
  wiederhole die *befehle* immer wieder und
  prüfe die *solange-bedingung* ***vor*** jeder Wiederholung.
- Jedesmal ***nach*** den *befehlen*: Führe das *weiterzählen* aus!
  (*vor* der nächsten Prüfung der *solange-bedingung*)
- Wenn die *solange-bedingung* "**falsch**" ergibt,
  mach nach der **}** weiter.

for (2)

- Wenn die *solange-bedingung* schon *von Anfang an "falsch"* ergibt, werden die *befehle* und das *weiterzählen gar nicht* gemacht (aber die *anfangszuweisung* vorher schon!).
- **for (***anfangszuweisung***;** *solange-bedingung***;** *weiterzählen***) {**
  *befehle***;**
  }

  ist ***genau dasselbe*** *wie*

```
anfangszuweisung;                    // ein Mal vor der Schleife!
while (solange-bedingung) {          // jedesmal vor den Befehlen!
  befehle;
  weiterzählen;                      // jedesmal nach den Befehlen!
}
```

Zählen mit **`for`**

- **`for`** wird meistens *zum Zählen verwendet*.
- Die *Zählvariable* heißt normalerweise **`i`**, **`j`** usw..
- Die *anfangszuweisung* setzt **`i`** auf den *Anfangswert*.
- Die *solange-bedingung* muss
  - für *alle Werte im Zählbereich* "***wahr***" sein
  - für *den ersten zu großen (zu kleinen) Wert* "***falsch***" sein.
- Das *weiterzählen* setzt **`i`** auf den *nächsten Wert*.

(das könnte z.B. auch **`i *= 10`** sein:
Die Schleife zählt dann **`1`**, **`10`**, **`100`**, **`1000`**, ...!)

for - Beispiele (1)

- *Beispiel:*

```
for (i = 1; i <= n; ++i) {
  printf("%d ", i);     // 1 2 3 … n
}
```

- *Typische Bedingungen:*

```
i < n    // zählt bis n-1
i <= n   // zählt bis n
i > 0    // zählt herunter bis 1
i >= 1   // zählt auch herunter bis 1
i >= 0   // zählt herunter bis 0
```

for - Beispiele (2)

- *Anderes Weiterzählen* als **++i**:

```
--i          // zählt abwärts
i += 2       // zählt in Zweierschritten
i /= 10      // schneidet jedesmal die letzte Stelle ab
i >>= 1      // schiebt i um ein Bit nach rechts
             // (für Bit-Operationen)
```

- *Beispiel:*

```
for (n = 1; n <= 1024; n *= 2) {
   printf("%d ", n);
}

// 1 2 4 8 16 32 64 128 256 512 1024
```

Nach der Schleife … (1)

… enthält ein ***vor*** *der Schleife deklariertes* **`i`**
den *ersten zu großen* (zu kleinen) Wert

d.h. den ersten Wert,
für den die Bedingung ***falsch*** ist

Beispiel:

```
int i;
for (i = 1; i <= 10; ++i) {
  printf("%d ", i);        // Gibt 1 bis 10 aus
}

// hier enthält i den Wert 11 !
```

Nach der Schleife … (2)

… ist ein ***in** der Schleife deklariertes* **`i`**

unbekannt

(existiert nicht mehr)

Beispiel:

```
for (int i = 1; i <= 10; ++i) {
  printf("%d ", i);       // Gibt 1 bis 10 aus
}

// hier gibt es das Schleifen-i nicht mehr!
```

Fehlende Teile im **`for`**

Jeden Teil in **`for (...; ...; ...)`** kann man *weglassen*

... z.B. wenn der Schleifenzähler schon *vor* dem **`for`** deklariert ist und den richtigen Wert hat

... oder das Weiterzählen im Schleifenrumpf erfolgt

Eine *weggelassene Bedingung* ist *immer "**wahr**"!*

```
for (;;) {      // Endlosschleife!
  ...           // Kann man mit  break  verlassen
                // ... oder mit  return
}
```

`for` mit 2 Zählvariablen

Die *Initialisierungen* und die *Weiterzähl-Befehle*
von *zwei* Zählvariablen werden mit `,` getrennt:

```
for (int i = 0, j = size - 1; i < j; ++i, --j) {
  tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}
```

Achtung:

Die *Deklaration* von zwei Zählvariablen direkt *in der Schleife*
ist *nur bei gleichem Typ* möglich.

Bei *verschiedenem Typ* muss man beide Zählvariablen
vor der Schleife deklarieren!

Der , in Rechnungen

Für Fortgeschrittene:

Mit dem **,** kann man an jeder Stelle,
wo eigentlich nur *eine* Rechnung oder Zuweisung erlaubt ist,
zwei oder mehr Rechnungen (oder Zuweisungen) hinschreiben:

- Es wird *zuerst* das *links* vom **,** und *dann* das *rechts* berechnet
- Das Ergebnis des ganzen **,**-Konstrukts ist das *rechte* Ergebnis
  (das *linke* Ergebnis wird *ignoriert*)

Wird fast nur für **for**-Schleifen mit 2 Zähl-Variablen verwendet.

Anderes Beispiel:

```
while (c = getchar(), c != EOF) { …
```

Hinweis:

Dieses **,** hat *nichts* mit dem **,** in Funktionsaufrufen usw. zu tun!

Geschachtelte **`for`** (1)

Geschachtelte **`for`** müssen

verschiedene Zählvariablen haben!

(denn sonst würde die innere Schleife
ja jedesmal den Zähler der äußeren Schleife verändern!)

In jeder *einzelnen Wiederholung der äußeren Schleife*
(d.h. für jeden Wert der äußeren Zählvariable)
werden *alle Wiederholungen der inneren Schleife* gemacht!

Beispiel:
"In jedem Wort auf der Befehlszeile...
(Äußere Schleife: Zähler für die Worte)
... schau jeden einzelnen Buchstaben an!"
(Innere Schleife: Zähler für die Buchstaben im Wort)

Geschachtelte **`for`** (2)

Verwendung meist mit *zweidimensionalen Arrays*
(= rechteckigen, tabellenartigen Anordnungen von Werten):

`double matrix[size][size]; // matrix` hat **`size`** Zeilen&Spalten

```
// zeile zählt die Zeilen von 0 bis (size - 1) durch
for (int zeile = 0; zeile < size; ++zeile) {
  // für jede einzelne Zeile wieder:
  // spalte zählt die Spalten von 0 bis (size - 1) durch
  for (int spalte = 0; spalte < size; ++spalte) {
    if (zeile == spalte) {
      matrix[zeile][spalte] = 1;  // 1 in der Haupt-Diagonale
    } else {
      matrix[zeile][spalte] = 0;  // ... und 0 überall sonst
    }
  }
}
```

Die Schleifenbedingung… (1)

… ist bei allen Schleifen in C
eine "solange"-Bedingung, ***keine*** *"bis"-Bedingung!*

- *Solange* die Bedingung "**wahr**" ist,
  wiederholt die Schleife die Befehle in den **{ }**.
- Wenn die Bedingung *das erste Mal "falsch"* ist,
  hört die Schleife auf.

Daher:

- Für *alle* Werte, für die die **{ … }** wiederholt werden sollen, muss die Bedingung "**wahr**" ergeben!
- Für den Wert, bei dem die Schleife *aufhören* soll, muss die Bedingung das erste Mal "**falsch**" ergeben!

Die Schleifenbedingung… (2)

- Wenn man in Gedanken eine "**bis**-Bedingung" hat, muss man in die **( )** das *genaue Gegenteil* schreiben!

  "Wiederhole, bis a gleich b ist" (d.h. solange sie verschieden sind):

  ```
  while (a != b) { … }
  ```

  "Zähle herunter, bis i Null ist, Null selbst nicht mehr":

  ```
  for (i = anzahl; i > 0; --i) { … }
  ```

- Eine **for**-Bedingung darf *nicht nur den Endwert* prüfen, sondern muss für *alle Werte bis zum Endwert* wahr sein!

  Falsch: `for (i = 1; i == 100; ++i) { … }`

  Richtig: `for (i = 1; i <= 100; ++i) { … }`

Der böse `;`

Kein `;` ***nach*** der `( )` in `if`, `while` und `for` !

Falsch: `for (i = 0; i < n; ++i); { ...`

Ist *gültiges* C

==> Der Compiler meldet *keinen Fehler!*

Aber: `if`, `while` oder `for` *hören beim* `;` *auf !!! (vor der* `{` *!)*

==> Die Schleife *wiederholt nur das* ***"nichts"***
zwischen `)` und `;` !

==> *Danach* werden die Befehle in `{ }`
genau einmal gemacht (im Beispiel oben für `i` gleich `n`)
(unabhängig von der Schleife bzw. Bedingung!)

Weglassen von **{ }**

- Man kann die **{ }** nach **if**, **else**, **while** und **for** auch *weglassen*, wenn nur ***genau 1 Befehl*** drinsteht:

```
if (x < 0) {
  x = -x;
}

if (x < 0) x = -x;   // Ist genau dasselbe!
```

Empfehle ich aber nur, wenn alles in einer Zeile steht!

- Wenn man ***gar keine Befehle*** in der Schleife hat (bei **for** gibt's das häufig), kann man entweder nur einen **;** oder *leere* **{ }** machen

Ich empfehle leere { }, nicht ; !!! *(sieht man besser)*

break und **continue**

- **break** beendet die Schleife:
  Das Programm macht *hinter* der **}** weiter
- **continue** überspringt
  die restlichen Befehle in den **{ }**:
  Die Schleife *macht mit dem nächsten Durchlauf weiter*:
  - Bei **while** und **do ... while**: Mit der *Bedingung*
  - Bei **for**: Mit dem *Weiterzählen* der Zählvariable
- Beide wirken nur auf die
  unmittelbar umgebende Schleife!

  ==> Man kann mit **break** *nicht mehrere*
  geschachtelte Schleifen auf einmal verlassen!

break und **continue** - Beispiel

```
for (anz = 1; ; ++anz) {  // Endlosschleife, Ende mit break
  cout << "Dein Wurf? ";
  cin >> wurf;
  if ((wurf < 1) || (wurf > 6)) {
    cout << "Du schwindelst!" << endl;
    continue;              // Mach gleich die nächste Eingabe!
  }
  if (wurf == rest) {
    cout << "Du bist mit " << anz << " Mal im Ziel!" << endl;
    break;                 // Fertig, Schleife beenden
  }
  if (wurf > rest) {
    cout << "Leider zu hoch gewürfelt..." << endl;
  } else {
    rest -= wurf;
    cout << "Noch " << rest << " Felder." << endl;
  }
}
```

Bedingungen (1)

- … sind **Fragen**, d.h.

Rechnungen,

*die **“wahr”** oder **“falsch”** ergeben.*

- … sind meistens **Vergleiche**:

== (“gleich”) **!=** (“ungleich”) **<** **<=** **>** **>=**

Beide Seiten werden ausgerechnet
(jede für sich)

und die Ergebnisse verglichen.

Bedingungen (2)

Beispiele:

```
a >= b

// auch char-Werte kann man vergleichen!
c == '\n'

sin(x * M_PI / 180) > sqrt(2) / 2
```

Achtung:

Das *Gegenteil* von **<** ist **>=** (nicht **>** !),
das Gegenteil von **>** ist **<=** .

Bedingungen (3)

Vorsicht bei **`double`** und **`==`** oder **`!=`** :

"Trifft" ev. wegen *Rundungsfehlern* nicht genau
==> *vermeiden!*

Beispiel für "gefährlichen" Code
(**`double x`** wird vielleicht *nie genau gleich* **`ende`**):

```
for (x = anfang; x != ende; x += 0.001) { … }
```

Besser:

- Wenn man *"von einer Seite kommt"*: **`<`** oder **`>`** verwenden
- Sonst *"ungefähr gleich"* (*"auf* **`delta`** *genau"*) verwenden, z.B.:
  `fabs(x – ende) >= delta` oder
  `(x <= ende – delta) || (x >= ende + delta)`

Bedingungen (4)

Jeder Vergleich besteht nur aus *zwei* Werten!

`(min <= x <= max)`

ist ***keine*** sinnvolle Bedingung in C!

(ist formal gültiges C ==> *keine Fehlermeldung*!
aber liefert ein *unsinniges Ergebnis*)

Solche Vergleiche muss man

aus 2 Bedingungen zusammensetzen:

`((x >= min) && (x <= max))`

Boolsche Operatoren (1)

- Logisches "**nicht**": **!** *bedingung*

  Wahr, wenn *bedingung* "falsch" ergibt.
  Falsch, wenn *bedingung* "wahr" ergibt.

- Logisches "**und**": *bedingung1* **&&** *bedingung2*

  Wahr, wenn *beide* Bedingungen "wahr" ergeben.
  Falsch, wenn mindestens eine "falsch" ist.

- Logisches "**oder**": *bedingung1* **||** *bedingung2*

  Wahr, wenn *mindestens eine* Bedingung "wahr" ergibt.
  Falsch, wenn beide "falsch" sind.

- "exklusives oder" gibt es nicht (ev. **!=** verwenden)!

Boolsche Operatoren (2)

Beispiele:

```
while ((x >= min) && (x <= max)) { …
if ((a < 0) || (b < 0)) { …
if ((x == 0) || (pow(x, y) >= 1)) { …
if (!((a == b) && (b == c) && (c == d))) {
    …
while (!isdigit(c)) { …
  // Solange das Zeichen c keine Ziffer ist …
```

Boolsche Operatoren (3)

- **Achtung:**
  Nicht mit bitweisen Operatoren **`& | ~ ^`** *verwechseln!*
- ***Abgekürzte Auswertung:***
  - Bei **`&&`** : Wenn *bedingung1* schon "**falsch**" ist, wird *bedingung2* gar *nicht mehr ausgerechnet.*
  - Bei **`||`** : Wenn *bedingung1* schon "**wahr**" ist, wird *bedingung2* gar *nicht mehr ausgerechnet.*

  Daher:

  - **`(n > 0) && (z % n != 0)`** dividiert nie durch **`0`**
  - **`(x < 0) || (sqrt(x) < delta)`** ist auch sicher

Boolsche Operatoren (4)

"Gegenteil" von Bedingungen mit **&&** *oder* **||**:

- **Entweder:** *"Nicht"* der ganzen Bedingung:

  Ganze Bedingung einklammern und **!** davor

- **Oder:** *"Gesetze von de Morgan"*
  (siehe Digitaltechnik, Mathe, ...)

  1) Zeichen in der Mitte *umdrehen*:

     Aus **&&** wird **||**, aus **||** wird **&&**

  2) Linke und rechte *Teilbedingung* beide in ihr *Gegenteil verwandeln*

Boolsche Operatoren (5)

Beispiele zum ***"Gegenteil"*** zusammengesetzter Bedingungen

(wie mache ich aus einer "bis"-Bedingung eine "solange"-Bedingung?)

- Wie schreibt man

  "Nicht (alter und neuer Wurf gleich 6)" ?

  - Entweder: **`!((alt == 6) && (neu == 6))`**
  - Oder: **`(alt != 6) || (neu != 6)`**
- Wie schreibt man *"Nicht (x ‹ 0 oder y ‹ 0)"* ?
  - Entweder: **`!((x < 0) || (y < 0))`**
  - Oder: **`(x >= 0) && (y >= 0)`**

    Das *Gegenteil* von **<** ist **>=**, nicht **>** !!!

Werte als Bedingung

In C können *beliebige Rechnungen*
(nicht nur Vergleiche und boolsche Operatoren!)
als Bedingung (statt einer Frage) verwendet werden:

Ergebnis der Rechnung ist **`0`** ==> ***"falsch"***

Jedes *andere Ergebnis* ==> ***"wahr"***

Das heißt: Bei einer Bedingung *ohne Vergleich*
muss man sich " ... **`!= 0`** " *dazudenken*!
(bzw. bei *Zeichen* " ... **`!= '\0'`** " und bei *Pointern*: " ... **`!= NULL`** ")

Beispiel:

```
while (1) { ... }  // immer wahr ==> Endlosschleife!
                   // ev. mit break; verlassen
```

= versus == (1)

= *und* == *nicht verwechseln!*

= … *Zuweisung* (speichern):

- Rechnung auf rechter Seite ausrechnen
- Ergebniswert in Variable auf linker Seite *speichern*
- Falls innerhalb einer Rechnung: Mit gespeichertem Wert weiterrechnen

== … *Vergleich*:

- Rechnungen auf linker und rechter Seite ausrechnen
- Beide Ergebnisse *vergleichen*
- Heraus kommt *"wahr"* oder *"falsch"* (**`1`** oder **`0`**)

= versus == (2)

Was passiert bei = statt == im **`if`** ?

```
if (i = n) { ... }    // statt if (i == n) { ... }
```

Ist gültiges C ==> Der *Compiler meldet keinen Fehler!*

- Der Wert von **`n`** wird in **`i`** *gespeichert*
  (wenn **`n`** vorher **`10`** war, ist **`i`** nachher auch **`10`**).
- Das *Ergebnis der Zuweisung* (d.h. der Wert von **`n`**)
  wird dann *implizit mit* **`0`** *verglichen*:
  - Wenn **`n`** gleich **`0`** ist ==> gilt als *"falsch"*
    ==> **`{ ... }`** wird *übersprungen*
  - Wenn **`n`** irgendeinen *anderen* Wert hat ==> gilt als *"wahr"*
    ==> **`{ ... }`** wird *ausgeführt*

Zuweisungen in Bedingungen (1)

Werden vor allem ***in Schleifen-Bedingungen*** verwendet,
dort auch *sinnvoll und üblich*:

`while ((c = getchar()) != EOF) {` …

"Lies ein Zeichen vom Input,
speichere es in **`c`***,*
und prüfe dann, ob es das Dateiende war"

Sonst bräuchte man *zwei Mal* **`c = getchar()`** :
- Einmal *vor* dem **`while`** (für den ersten Durchlauf)
- Einmal als *letzten* Befehl *in* der Schleife (für die restlichen)

Die ***Klammern*** um die Zuweisung sind hier ***notwendig***,
weil die Zuweisung im *Vorrang unter den Vergleichen* steht
(ohne Klammern würde **`(c = (getchar() != EOF))`** gerechnet!)

Zuweisungen in Bedingungen (2)

Wenn eine *Bedingung **nur** aus einer Zuweisung* besteht:

Trotzdem ( ) herum machen,
auch wenn die ( ) nicht notwendig sind!

Ist ***üblich*** als *optisch gut sichtbarer Hinweis*:

"Achtung, hier steht absichtlich eine Zuweisung,
das soll kein Vergleich sein!"

```
for (i = 0; (dest[i] = src[i]); ++i) { }
```

Zuerst wird die *Zuweisung* gemacht,
dann wird der soeben *zugewiesene Wert*
implizit mit **0** *verglichen*

(wenn **0**: Wird *auch noch zugewiesen*, dann Schleifenende)

`bool`

Umgekehrt ist das **Ergebnis** **von Bedingungen `0` oder `1`.**

Es hat **Typ `int`** und *kann in Variablen gespeichert* werden:
Die Variable enthält dann **`0`** für ***“falsch”*** oder **`1`** für ***“wahr”***.

Früher hat man dafür **`int`**-Variablen verwendet,
jetzt (seit C99) gibt es für ***Wahrheitsvariablen*** einen

eigenen Typ `bool`

Für **`bool`** gibt es die beiden *vordefinierten Konstanten*
`true` (***“wahr”***, **`1`**) und **`false`** (***“falsch”***, **`0`**).

In **C** vor **C23** braucht man für **`bool`** den Header **`<stdbool.h>`**,
in **C++** funktioniert **`bool`** *ohne* Header, seit **C23** auch in C.

bool -- Beispiel

```
bool ok;        // ok ... "Alle Werte ok?"    früher: int ok;
...
ok = true;      // früher: ok = 1;

// wiederhole, solange noch Werte da sind und die Werte ok sind
for (int i = 1; (i < argc) && ok; ++i) {
  x = atof(argv[i]);
  ...
  ok = (x >= 0) && (x <= 100);  // Speichere in ok, ob x in 0...100 ist.
  ...
}
...
if (!ok) {      // Prüfung von bool-Werten ohne == !!!
  cout << "Schleifenende wegen falschem Wert!" << endl;
} else {
  cout << "Alle Eingaben erfolgreich gelesen!" << endl;
}
```

Fallunterscheidung: **`switch`** (1)

```
switch (wert) {
   case const1:
      befehle1;
      break;
   case const2:
      befehle2;
      break;
   …
   default:
      sonst-befehle;
      break;
}
```

- Rechne *wert* aus
- Suche das **`case`** mit demselben Wert
- Mach die *befehle* vom passenden **`case`** bis zum *nächsten* **`break`** (oder bis zur **`}`**)
- Wenn kein **`case`** passt: Mach die *befehle* vom **`default`** bis zum nächsten **`break`**

Fallunterscheidung: **`switch`** (2)

- Auf das **`break`** *nicht vergessen*,
  das **Programm macht sonst**
  mit dem *nächsten* `case` weiter!!!

  ==› dadurch kann man mehrere **`case`** *zusammenfassen*!
- Nach dem **`case`** dürfen nur *konstante Werte* stehen,
  keine Variablen oder Rechnungen mit Variablen!
- Die Variable (oder Rechnung) nach dem **`switch`**
  muss ein **`int`** (oder ein **`char`** oder ein **`enum`**) sein:
  `double` ist verboten, *Texte* (generell: Pointer) auch!
- Man sollte *immer* ein **`default:`** angeben,
  denn ohne **`default:`** macht **`switch`** einfach *gar nichts*,
  wenn kein **`case`** zutrifft: Es kommt *keine Fehlermeldung!*

`switch` - Beispiele

```
switch (c) {
  case '(':
  case '[':
  case '{':
    ++offeneKlammern;
    break;
  case ')':
  case ']':
  case '}':
    --offeneKlammern;
    break;
  case '\n':
    ++zeile;
    break;
  default: // nichts zu tun!
    break;
}
```

```
switch (wert % 6) {
  case 0:
    cout << "2 und 3 teilen\n";
    break;
  case 3:
    cout << "nur 3 teilt\n";
    break;
  case 2:
  case 4:
    cout << "nur 2 teilt\n";
    break;
  default:
    cout << "weder 2 noch 3 "
            "teilen\n";
    break;
}
```

Sprungbefehl: **`goto`**

- Man kann vor einen Befehl oder vor eine **`}`**
  ein *Label* (deutsch: *"Sprungmarke"*) schreiben
  (beliebiger *neuer Name* mit "**`:`**" dahinter)
- Beim Befehl "**`goto`** *label***`;`**" wird das Programm
  an der Stelle "*label*" fortgesetzt:

  "*Springe* zum Befehl "*label*" und *mach dort weiter*!"
- *Einschränkungen:*
  - Geht nur *innerhalb* einer Funktion.
  - Man sollte *nicht* von außen in **`{ }`** *hinein*springen

    (praktisch *nie sinnvoll*,
    in bestimmten Fällen technisch *gar nicht möglich*!)

goto: Beispiel (*nicht* Nachmachen!)

```
int main()
{
  ...
  for (...) {
    ...
    if (x == 0) {
      goto err_ende; // Spring zur gemeinsamen Fehlerbehandlung!
    }
  }
  ...
  return 0;   // normalerweise hört das Programm hier mit "ok" auf!
err_ende:     // ... und hierher springt es nur im Fehlerfall
  ...         //  gemeinsame Fehlerbehandlung für alle Fehlerfälle
  cout << "Something bad happened!" << endl;
  return 1;
}
```

Verwendung von **`goto`**

*... am besten **gar nicht!***

Ergibt oft undurchschaubare, fehleranfällige Programme!

==> An Schulen und Uni's normalerweise *verboten*!

==> In vielen Firmen *verboten*!

Ich halte es in bestimmten, seltenen Fällen für vertretbar, z.B.:

- *"Notausgang"* im Fehlerfall
  (Zusammenfassen des Aufräum-Codes z.B. am Funktions-Ende)
- Verlassen *mehrerer Schleifen* auf einmal (geht mit **`break`** nicht!)

Weil: Die **`goto`**-freie Alternative ist in diesen Fällen
oft auch nicht besser verständlich,
und ein *guter Label-Name* hilft beim Lesen des Programms...

Arrays

- Konzept
- Deklaration und Verwendung
- Initialisierung
- Mehrdimensionale Arrays
- `sizeof( ... )`
- Die mem-Funktionen für Arrays

Was ist ein Array? (deutsch: “Feld”)

- “***Array***”:

  Eine Variable, die mehrere Werte speichern kann
  (statt nur einem).

- Die Werte (“*Elemente*”) müssen *alle vom selben Typ* sein (lauter **`int`**, lauter **`double`**, lauter **`char`**, …).

- Die *Anzahl der Werte* muss *in der Deklaration angegeben* werden

  (der Compiler reserviert im Speicher einen fixen Block mit Anzahl mal Platz für einen Wert).

- Die Elemente werden der Reihe nach *durchnummeriert* (die “Hausnummer” heißt *“Index”* oder *“Subscript”*).

Deklaration von Arrays

Mit **`[ ]`** in der Deklaration: Enthält die

Größe des Arrays,
d.h. die Anzahl der Elemente.

```
int a[10];
```

"Lege ein neues Array **`a`** *mit Platz für 10* **`int`***-Werte im Speicher an!"*

Zur Größe von Arrays (1)

Tipp für die Praxis:

Falls sich die *Anzahl der Elemente* jemals *ändern* könnte:

Eine eigene *Konstante* dafür machen,
keine fixe Zahl in die Deklaration schreiben!

==> *Leichter zu finden, zu lesen & zu ändern!*

Ganz oben im Programm:

```
#define SIZE 10
```

Dann statt "`int a[10];`":

```
int a[SIZE];
```

... und auch in den **`for`**-Schleifen **`SIZE`** statt **`10`** verwenden!

Zur Größe von Arrays (2)

Die Größe darf auch
eine *Variable oder Rechnung* sein,
aber sie muss an dieser Stelle
schon einen *Wert* haben.

Beispiel: **`int count[argc];`**

Vorsicht bei **`argc-1`** : Könnte **`0`** werden, Größe **`0`** ist *verboten*!

Achtung:

- Gilt ***nur in C*** (und da erst seit C99 und GnuC)!
  In C++ (und MSVC) muss die Größe immer eine *Konstante* sein!
  (dort für variable Größe **`malloc`** oder **`alloca`** verwenden!)
- Auch bei *globalen und statischen Arrays* (siehe später)
  muss die Größe immer eine *Konstante* sein!

Zur Größe von Arrays (3)

Was bedeuten *leere* **`[]`** ?

*"Größe wird **woanders** festgelegt"*

An 4 Stellen:

- *Funktionsparameter* (z.B. **`argv[]`**)

  ==› Größe wird *im Aufrufer* festgelegt

- Deklaration mit *Initialwerten* (kommt gleich)

  ==› Größe wird durch *Anzahl* der Initialwerte festgelegt

- **`extern`**-Deklaration (kommt später)
  ==› Dazugehörige nicht-**`extern`**-Deklaration legt die Größe fest

- *Hinterstes* Member in einer Struktur
  ==› Struktur muss mit **`malloc`** angelegt werden, das legt die Größe fest

Zugriff auf Array-Elemente

Mit **`[ ]`** *beim Lesen und Speichern*: Enthält die

Nummer (Index) des Elementes,
das gelesen / geschrieben werden soll.

```
sum = sum + a[i];
```

"addiere den Wert von **`a`**'s Element Nummer **`i`** zu **`sum`**"

```
a[0] = 1;
```

"speichere **`1`** im Element Nummer **`0`** von **`a`**"

```
++(a[i]);
```

"zähle zum Element Nummer **`i`** von **`a`** eins dazu"

Anmerkungen zu den [ ]

- Die **[ ]** haben *zwei verschiedene Bedeutungen:*
  - In der **Deklaration**:

    Anzahl der Elemente
  - Beim **Zugriff**:

    Nummer des gewünschten Elementes

- **Achtung:** *Bei der Deklaration und beim Zugriff:*

  In den **[ ]** muss ein **`int`** stehen:
  `double` sind **verboten!**

  (**`char`** sind erlaubt, aber *gefährlich*: **`signed`** oder **`unsigned`**?)

Die Tücken des Index (1)

Die Nummerierung der Elemente
beginnt in C immer bei ***0*** *(nicht bei* ***1*** *!):*

Die Elemente eines Arrays
mit **SIZE** *Elementen*
haben daher die Nummern

0 bis SIZE–1 (*nicht* SIZE !!!).

Daher bei Deklaration **`double a[SIZE];`**

==> *Erstes* Element **`a[0]`**, *letztes* Element **`a[SIZE–1]`** .

==> Es gibt ***kein*** Element **`a[SIZE]`** !

Die Tücken des Index (2)

Wenn das Programm rechnet,
"wissen" die Arrays nicht mehr, *wie groß sie sind!*
(Die Größe wird *nirgends mitgespeichert*!)

==> Zugriffe mit **Index < 0** oder **Index >= Anzahl Elemente**
werden nicht verhindert, es kommt keine Fehlermeldung!

==> Beim Lesen: "Irgendwelche" Werte werden gelesen...
==> Beim Schreiben: *Andere Daten werden **überschrieben**!*

==> Das Programm rechnet mit *falschen Daten* weiter
oder *stürzt* (sofort oder irgendwann später) *ab*.

Einer der Hauptgründe, warum C nicht "deppensicher" ist,
und der Hauptgrund für Sicherheitslücken in C-Programmen!

Was man mit Arrays *nicht* kann...

Mit einem *einzigen* Befehl ...

- einen Wert in allen Elementen eines Arrays speichern.
- alle Elemente eines Array in ein anderes Array kopieren.
- zwei Arrays miteinander vergleichen.
- mit allen Elementen eines Arrays rechnen.
- alle Elemente eines Arrays einlesen oder ausgeben.

==> *Das muss man einzeln* **Element für Element** *machen!*

==> Normalerweise:

for-Schleife

Array: Beispiel (1)

```
double a[SIZE], b[SIZE], c[SIZE];
```

...

```
for (int i = 0; i < SIZE; ++i) {
  a[i] = 1;   // alle Elemente von a auf 1 setzen
```

oder:

```
  b[i] = a[i];   // Inhalt von a nach b kopieren
```

oder:

```
  c[i] = a[i] + b[i]; // elementweise rechnen
}
```

Array: Beispiel (2)

```
int i;   // brauchen wir nach dem for noch!

for (i = 0;
     (i < SIZE) && (a[i] == b[i]);
     ++i) { }   // Leere Schleife!!!

if (i < SIZE) {
  cout << "a und b sind an Stelle "
       << i << " verschieden" << endl;
} else {
  cout << "a und b sind gleich" << endl;
}
```

Initialisierung von Arrays

Elementweise mit " **= { *wert0*, *wert1*, ... }** ".

Größenangabe kann man bei Initialisierung *weglassen*:

Größe des Arrays = Anzahl der Werte in { }

Beispiele:

```
int monatsTage[] = { 31, 28, 31, 30, 31, 30,
                     31, 31, 30, 31, 30, 31 };
char satzZeichen[] =
   { '.', ',', ';', ':', '!', '?' };
```

Nur für die Initialisierung beim Deklarieren möglich,
nicht für eine Zuweisung auf ein bestehendes Array!

Initialisierungs-Trick... (1)

- Wenn Initialisierung *und* Größe angegeben sind...
- ... und *weniger Init-Werte da sind als das Array groß ist*:

==> Die *restlichen Elemente* hinten werden *auf* **`0`** *gesetzt*!

Beispiel:

```
int count[100] = { 1 };
```

setzt das *erste* Element auf **`1`**
und *alle anderen* Elemente von **`count`** auf **`0`**,
egal wie viele es sind!

Aber das ist nur mit *konstanter* Array-Größe erlaubt,
nicht mit **`int count[argc] = { ... };`** !

Initialisierungs-Trick... (2)

Seit **C23**, bei einigen Compilern schon *früher*:

***Leere* `{ }` initialisieren das *gesamte Array* mit `0`**

Beispiel:

```
int count[100] = { };
```

setzt *alle* Elemente von **`count`** auf **`0`**

Funktioniert seit **C23** auch mit *variabel* großen Arrays

Wenn's nicht klappt: Probiere

```
int count[100] = { 0 };
```

==> *Erstes* Element *händisch* **`0`**, *Rest automatisch* **`0`**

Mehrdimensionale Arrays (1)

- *Beispiele*:
  Schachbrett, Matrix in Mathe, Pixel am Schirm, ...
- Deklaration und Zugriff in C mit **[***zeile***][***spalte***]**,
  nicht mit **[***zeile***, *** *spalte***]**:

```
int brett[8][8];
brett[i][j] = DAME;
```

  Merkregel: "*Z*eile *z*uerst, *Sp*alte *sp*äter"
  (Im Speicher: Zuerst 1. Zeile, dann 2. Zeile, ...)
- *Achtung:*
  Nicht alles, was mit **[ ][ ]** angesprochen wird,
  ist ein zweidimensionales Feld
  (Gegenbeispiel: **argv**: Ist *kein* 2-dim. Array, lernen wir später...).

Mehrdimensionale Arrays (2)

- Zugriff normalerweise mit
  zwei ineinander geschachtelten **`for`**-*Schleifen*:

```
for (zeile = 0; zeile < 8; ++zeile) {
  for (spalte = 0; spalte < 8; ++spalte) {
    brett[zeile][spalte] = 0;
  }
}
```

- Initialisierung mit *geschachtelten* **`{ }`** (pro Zeile):

```
double vorz[2][2] = {{1, -1}, {-1, 1}};
```

(nicht zwingend notwendig, aber viel besser lesbar!)

Denn technisch: Zweidimensionales Array = *"Array von Arrays"*

`sizeof( … )`

… liefert die *Größe* von Werten im Speicher ***in Bytes***.

In `( … )` darf ein *Typ* oder ein *Wert* stehen.

Beispiele:

```
cout << sizeof(double);    // 8
cout << sizeof(1.0);       // auch 8 (1.0 ist double)
double a[10];
cout << sizeof(a);    // 80 (10 Elemente mit je 8 Bytes)
cout << sizeof(a[0]);      // 8 (Größe eines Elementes)
```

Daher: `(sizeof(a) / sizeof(a[0]))`

liefert die *Anzahl der Elemente* im Array `a`
(vor allem bei Arrays mit leeren `[]` und Initialisierung!)

Beschränkungen von **`sizeof( ... )`**

Für Fortgeschrittene:

`sizeof` ist *keine* Funktion,
sondern ein eigenes Sprachkonstrukt,
das *zur Compilezeit* fix ausgerechnet wird.

(weil die Größe ja zur Laufzeit nirgends gespeichert ist!)

==> *Der Compiler muss die Größe kennen!*

==> *Klappt nur für lokale / statische / globale Arrays!*

Nicht für:

- ***Array-Parameter*** (liefert stattdessen die *Größe eines Pointers!*)
- mit **`malloc`**, **`alloca`** usw. angelegte Arrays
- **`extern`**-Arrays

Die **mem**-Funktionen

... arbeiten mit einer *gegebenen Anzahl von* ***Bytes***
(*nicht* Anzahl von Elementen!)
==> werden üblicherweise mit **sizeof** aufgerufen!

Die wichtigsten (nur für Fortgeschrittene): (aus **<string.h>**)

memset(*array***,** *wert***,** *anzahl***)**
füllt die ersten *anzahl* Bytes von *array* mit *wert*.

memcpy(*zielArray***,** *quellArray***,** *anzahl***)**
kopiert *anzahl* Bytes von *quellArray* nach *zielArray*.

memmove(*zielArray***,** *quellArray***,** *anzahl***)**
dasselbe, aber Überlappungen sind erlaubt.

memcmp(*array1***,** *array2***,** *anzahl***)**
vergleicht die ersten *anzahl* Bytes von *array1* und *array2*,
Ergebnis ist ein **int**: **<** , **==** , **>** **0** wenn *array1* **<** , **==** , **>** *array2*

Funktionen

- Grundsätzliches
- Parameter, Argumente & Returnwert
- Ablauf beim Aufruf
- Der “Prototyp” einer Funktion
- Kleinigkeiten
- Verwendung, Programmiermethodik

Was ist eine Funktion? (1)

Funktion =
benanntes Programmstück
(Befehle & dazugehörige Variablen),

das *von anderen Programmteilen verwendet*
werden kann.

So ähnlich wie in Mathe:

Funktion =
ein Programmstück (in Mathe: Eine Formel),

in das *Werte eingesetzt werden können*,

und das dann
für die eingesetzten Werte ausgerechnet wird.

Was ist eine Funktion? (2)

Alter Name: *"Unterprogramm"*

Bisher kennen wir...

- ... eine *selbstgeschriebene* Funktion:

  `main`

- ... einige *schon fertig mitgelieferte* Funktionen:

  `printf`, **`system`**, **`atoi`**, **`sqrt`**, **`strlen`**, ...

In Zukunft wollen wir mehr

Funktionen selbst schreiben!

Beispiel (1)

In Mathe:

- Definition einer Funktion: f(x) := x^2 - 2*x + 4
- Einsetzen: f(2) = 2^2 - 2*2 + 4 = 4

In C:

- Definition derselben Funktion:

```
double f(double x)
{
  return x * x – 2 * x + 4;
}
```

- Verwendung z.B.: **`printf("%g\n", f(2.0));`**

  ==> *Gibt 4 aus!*

Beispiel (2)

```
double hoch(double a, int n)
{
  double result = 1.0;

  for (int i = 0; i < n; ++i) {
    result *= a;
  }

  return result;
}
```

Aufruf , d.h. *Verwendung* der Funktion (z.B. im **main**):

```
printf("%g^%d = %g\n", x, i, hoch(x, i));
```

Sprachliches …

- Eine Funktion *“wird aufgerufen”* (englisch: *“function call”*)
  = von einem anderen Programmstück aus gestartet.

  “Aufrufer” = der Code, der die Funktion aufgerufen hat.
- Eine Funktion kann beim Start Werte vom Aufrufer *“übergeben”* bekommen.

  Diese Werte heißen *“Parameter”* (in der Funktion selbst) bzw. *“Argumente”* (im Funktionsaufruf).
- Eine Funktion *“kehrt zurück”* (englisch: *“function return”*), wenn sie fertig gerechnet hat.

  ==› *Das Programm macht im Aufrufer nach dem Aufruf weiter.*
- Eine Funktion kann am Ende ein *Ergebnis* liefern,
  mit dem der Aufrufer weiterrechnet: Den *“Returnwert”*

Die Variablen einer Funktion… (1)

- … haben *nichts zu tun* mit den Variablen
  in **`main`** oder in einer anderen Funktion,

  auch nicht bei Namensgleichheit!!!

Jede Funktion hat ihre **eigenen** *Variablen!*

- … können *nur innerhalb der Funktion* verwendet werden
- … und dürfen daher auch *gleich heißen*
  wie Variablen in anderen Funktionen oder **`main`**

Die Variablen einer Funktion... (2)

... bekommen ihren eigenen *Speicher*,

und zwar *erst dann*, wenn die Funktion *aufgerufen* wird,

... und *verschwinden wieder*, wenn sie zurückkehrt.

==> ... haben *bei jedem Aufruf am Anfang zufällige Werte!*

==> ... *"vergessen"* ihren Wert *zwischen zwei Aufrufen!*

Falls eine *Initialisierung* angegeben ist:

==> ... werden *bei jedem Aufruf neu initialisiert*!

Die Parameter einer Funktion (1)

Die Variablen oben in den **( )** heißen

"Parameter"

Das sind ganz normale,
eigenständige Variablen der Funktion:

- Sie gehören *nur der Funktion* und
  sind *nur innerhalb* der Funktion sichtbar.
- Sie haben *nichts* mit anderen Variablen im Aufrufer,
  in **`main`** oder in anderen Funktionen zu tun
  (auch nicht bei Namensgleichheit!).

Die Parameter einer Funktion (2)

Einziger Unterschied:

Die Parameter werden bei jedem Aufruf mit den im Aufruf angegeben Argumentwerten initialisiert.

(so als ob am Anfang der Funktion eine Deklaration z.B. "**`int param =`** *aufrufswert*`;`" stünde!)

==> Die Parameter sind die

"Platzhalter" *im Code,* für die beim Aufruf *bestimmte Werte aus dem Aufrufer eingesetzt* werden!

Die Parameter einer Funktion (3)

==> Die Parameter dürfen in der Funktion *geändert* werden (mit Zuweisung usw., wie normale Variablen).

Aber:

Die Änderung wirkt nicht
auf die entsprechenden Werte im Aufrufer
(die bleiben unverändert)!

Denn:

Die Parameter einer Funktion enthalten nur eine

Kopie

der Werte im Aufrufer!

==> *es wird nur die lokale Kopie geändert!*

Die Parameter einer Funktion (4)

Man nennt das

"Call by Value"

... weil nur der ***Wert*** *vom Aufrufer kopiert* wird
und nicht die Variable aus dem Aufrufer übergeben wird.

Gegenteil:

"Call by Reference"

... übergibt einen ***Verweis*** *auf die Original-Variable im Aufrufer.*

==> Änderungen in der Funktion
ändern direkt die Variable im Aufrufer!

... lernen wir später!

Die Parameter einer Funktion (5)

Da Arrays bekanntlich nicht als Ganzes
zugewiesen bzw. kopiert werden können:

Arrays können in C/C++

- *weder* als "Call by Value"-Parameter *übergeben*
- *noch* als Returnwert *zurückgegeben* werden!

(*"Call by Reference"* mit Arrays geht, lernen wir später...)

Die Parameter einer Funktion (6)

Deklaration der Parameter in **( )**: Wie Deklarationen, aber...

- *Nur ein* Parameter pro Typangabe!

  Man kann *nicht* mehrere Parameter gleichen Typs zusammenfassen:

  Falsch: **`int ggT(int a, b)`**
  Richtig: **`int ggT(int a, int b)`**

- Getrennt durch '**,**' statt '**;**'.

- Die *Reihenfolge* ist maßgeblich:

  Die Argument-Werte im Aufruf werden *der Reihe nach* auf die Parameter zugewiesen:

  1. Arg. -> 1. Param., 2. Arg. -> 2. Param. usw.

Der “Nichts-Typ” **`void`** (1)

Wenn *in den Klammern* im Funktionskopf **`void`** steht,
werden der Funktion *keine Parameter übergeben*.

Beispiel:

```
int main(void) { ... }
```
(wenn man die Befehlszeile nicht braucht)

```
int rand(void) { ... }
```
(**`rand`** braucht keine Werte vom Aufrufer)

Achtung:

Das **`void`** steht *nur im Funktionskopf, nicht beim Aufruf:*

Im *Aufruf* stehen immer nur *leere* **`()`** *ohne* **`void`** : **`rand()`**

Der “Nichts-Typ” **`void`** (2)

Achtung bei Funktionen *ohne* Parameter:

- In **C** *vor C23* *muss* **`func(void)`** im Funktionskopf stehen:

  `func()` bedeutete früher (K&R C) *etwas anderes*
  (Funktion mit beliebig vielen / unbekannten Parametern)

  und ist seit C99 als *veraltert* deklariert!

  (hier gibt es mit **C23** *inkompatible* Änderungen!)

- In **C++** darf man auch einfach **`func()`** schreiben:

  `func(void)` und **`func()`** bedeuten *genau dasselbe*.

Die Argumente im Aufruf

Die Werte in den **( )** beim *Aufruf* heißen *"Argumente"*
(meistens zumindest, Sprachgebrauch uneinheitlich!)

Sie

- … können *Konstanten*, *Variablen*
  oder *beliebige Rechnungen* sein.
- … müssen vom Typ her *zu den Parameter-Typen passen*
  (wie sie im Funktionskopf deklariert sind).
- … werden *bei jedem Aufruf neu ausgerechnet*,
  bevor die aufgerufene Funktion gestartet wird.

Beispiel:

```
quadrat = hoch((x + y), 2);
```

Der Returntyp

... steht in der "Kopfzeile" der Funktion *vor* dem Funktionsnamen.

... legt fest,

welchen Typ das **Ergebnis** *der Funktion hat*

(= Typ des Wertes, den die Funktion ausrechnet)

... kann alles *außer einem Array* sein.

... muss zu dem Wert passen, der beim **`return`** steht!

Bei unserem Beispiel "**`double hoch( ... )`**":

"Die Funktion **hoch** *liefert*
einen **`double`***-Wert als Ergebnis."*

Der Returnwert

= Das *Ergebnis* des Funktionsaufrufes
"das, was ausgerechnet wurde"

(wie bei einer mathematischen Funktion:
sin(x) liefert als Ergebnis den Sinus von *x*)

Der Aufrufer *rechnet mit dem Returnwert weiter*
(so, als ob in der Rechnung *statt dem Funktionsaufruf der Returnwert* als Konstante gestanden wäre):

```
x = 16;
if (hoch(x, 2) > 2) …
```

ist *nach dem Ausrechnen* von **hoch** dasselbe wie

```
if (256 > 2) …
```

Returntyp **`void`**

Wenn der *Returntyp* einer Funktion
(vor dem Funktionsnamen) "**`void`**" ist,

liefert die Funktion *kein Ergebnis* (*keinen Returnwert*)
an den Aufrufer zurück.

Das kommt vor bei Funktionen,

- die *nur für ihren "Nebeneffekt" aufgerufen* werden

  Beispiel: **`void srand(unsigned int startWert)`**

  Setzt den Startwert von **`rand()`**,
  aber liefert *kein* Ergebnis!

- oder die *gar nicht zurückkehren*

  Beispiel: **`void exit(int status)`**

`return` (1)

1.) **`return`** kehrt aus der gerade laufenden Funktion
sofort in den Aufrufer zurück

==> Das Programm macht im Aufrufer
nach dem Aufruf weiter.

2.) **`return`** gibt an, *welchen Wert*
die Funktion als *Ergebnis* zurückliefert

==> Der Aufrufer kann
mit diesem Wert *weiterrechnen*.

Eine Funktion kann *mehrere* **`return`** enthalten (in **`if`**'s...).

Bitte im **`return`** *keine Klammern um den Wert!*
(nicht falsch, aber unnötig und unüblich!)

`return` (2)

- Funktionen mit Returntyp **`void`** haben *kein* Ergebnis:
  - Bei **`return`** darf *kein Wert* stehen.
  - Die Abarbeitung der Befehle der Funktion darf auch einfach *"unten bei der* **`}`** *rausfallen"* (ohne **`return`**).
- Funktionen mit echtem Returntyp (nicht **`void`**) müssen immer ein *Ergebnis* liefern:
  - Bei **`return`** *muss immer ein Wert* angegeben sein.
  - Der *Typ des Wertes* muss zum *Returntyp* passen.
  - Die Funktion muss *immer* mittels **`return`** enden (*"unten bei der* **`}`** *rausfallen"* ist ***verboten!***).

`exit( ... )` und **`return`**

- **`return`** *kehrt aus der gerade laufenden Funktion zurück:*
  - Bei **`return`** im **`main`**:

    Programm-Ende (= Rückkehr ins Betriebssystem)
    angegebener Wert = Exit-Code
  - Bei **`return`** in *allen anderen Funktionen*:

    Programm macht in aufrufender Funktion weiter!
    angegebener Wert = Returnwert
- **`exit( ... )`** *beendet immer das ganze Programm*
  (egal ob es in **`main`** oder einer anderen Funktion steht)

  ... mit dem in Klammer angegebenen Exit-Code.

Wie funktionieren Funktionen?

- **Speicher** für die Parameter der Funktion wird reserviert.
- Der Aufrufer rechnet die **Argumentwerte** aus,
  die Argumentwerte werden in die Parameter **kopiert**.
- Die aktuelle **Programmstelle im Aufrufer** wird gespeichert.
- Das Programm springt zum **Code der Funktion**.
- **Speicher** für die lokalen Variablen der Funktion wird reserviert.
- Die Funktion wird berechnet.
- Der Speicherplatz der Variablen der Funktion wird **freigegeben**.
- Der **Returnwert** wird zum Aufrufer **kopiert**.
- Das Programm springt **in den Aufrufer zurück**
  (zur zuvor gespeicherten Programmstelle = *"Return-Adresse"*).
- Der Speicherplatz der Parameter der Funktion wird **freigegeben**.
- Der Aufrufer rechnet mit dem Returnwert weiter.

Der “Prototyp” einer Funktion (1)

- … ist die *“Kopfzeile” einer Funktion*
  ohne **{ … }**, aber dafür mit **;** am Ende
- … **deklariert** *die Funktion, ohne den Code dafür anzugeben*
  (d.h. legt **Name, Parameter-Typen und Returntyp** fest)

  ==> *Ab dieser Stelle* ist die Funktion bekannt
  und kann *aufgerufen* werden.

  Die Definition (= der Code der Funktion)
  kann später (hinter den Aufrufen) folgen.

- Bei der nachfolgenden **Definition** der Funktion
  (mit dem Code in **{ … }**)
  müssen Name, Parameter und Returntyp
  noch einmal *genauso* angegeben werden.

Der “Prototyp” einer Funktion (2)

Beispiel:

```
double hoch(double a, int n);
```

sagt dem Compiler:

*“Es gibt eine Funktion, die heißt ’**hoch**’.*
*Es gehen ein **double**- und ein **int**-Wert hinein,*
*und es kommt ein **double**-Ergebnis heraus*

*(aber wie ’**hoch**’ intern funktioniert,*
*steht erst später oder in einem anderen **.c**-File).”*

Hinweis:

Die Parameter-Namen **a** und **n** dürfte man
im Prototyp zwar weglassen,
sollte man aber *wegen der Lesbarkeit immer angeben!*

Der “Prototyp” einer Funktion (3)

- In C, bei *ganz kurzen* Programmen:
  Separate *Prototypen* sind eher *selten*,
  Funktionen werden gleich *vor ihren Aufrufen definiert*.
- In C, bei *längeren* Programmen:
  Für *alle* Funktionen zuerst *Prototypen ganz oben* im File,
  dann Funktions-Definitionen in beliebiger Reihenfolge.
- In C und C++, bei *großen Programmen* mit *mehreren Files*:
  - Die *Funktions-Deklarationen* (Prototypen) in die **.h-Files**
    (das ist Haupt-Zweck und Haupt-Inhalt von .h-Files!).
  - Die *Funktions-Definitionen* (Code) in die **.c-Files**.
- In C++: *Prototypen für alle Funktionen* sind eigentlich *Pflicht*!
  (entweder ganz oben im File oder in einem Header-File).

Funktionen gleichen Namens...

- **... sind in C**

  verboten!

  Eine Funktion *darf nicht gleich heißen*

  - ... wie eine *andere Funktion*
  - ... wie eine *Variable* oder eine *Konstante*
  - ... wie ein *Typ*

- **... sind in C++**

  erlaubt, wenn die Parameter-Typen verschieden sind.

  ==> nächstes Sem.: *"Function overloading"*

`inline`

Sinnvoll vor *ganz kurzen* Funktionen:

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

Muß ***vor*** den Aufrufen stehen!

==> Funktionscode wird im Aufrufer *"vor Ort" eingesetzt* anstatt aufgerufen.

==> Aufruf-Overhead fällt weg, besser optimierbar, ...

==> Programm wird etwas *schneller*, aber ev. *größer* (weil der Funktionscode *an mehreren Stellen* drinsteht).

Wichtig & üblich vor allem in **C++**, Details im 2. Sem. ...

Was kommt noch alles? (1)

- In *allen Funktionen* sichtbare *gemeinsame Variablen* und Variablen, die ihren *Wert zwischen Aufrufen behalten*:
  ==> Demnächst hier...

- *Call by Reference* (**Ändern** der Argumente im Aufrufer):
  a) Mittels *Pointer* (in C und C++ möglich)
     ==> Demnächst hier...
  b) Echte *Referenz-Parameter* (nur in C++ erlaubt)
     ==> 2. Sem.

Was kommt noch alles? (2)

- *Variabel viele* Parameter (wie **`printf`**, **`scanf`** usw.):

  Geht auch für eigene Funktionen, machen wir aber nicht

  ==> Siehe **`stdarg.h`**
- *Default-Werte* für nicht angegebene Parameter:

  Nicht in C, gibt es nur in C++

  ==> 2. Sem.
- Variablen, die auf eine Funktion verweisen:
  Zeiger auf Funktionen

  ==> später hier und in Übungsbeispielen

Zusammenfassung

Funktions-Deklaration (*"Prototyp"*):

returnTyp funktionsName`(`*paramTyp1 paramName1*`, ...);`

Funktions-Definition:

```
returnTyp funktionsName(paramTyp1 paramName1, ...)
{
    ... Variablen der Funktion ...
    ... Code der Funktion ...
    return returnWert;
}
```

Funktions-Aufruf:

... *funktionsName*`(`*argumentWert1*`, ...)` ...

Wozu Funktionen? (1)

Doppelten Code vermeiden!

Doppelter Code ist **wartungsaufwändig und fehleranfällig** (muss bei jeder Änderung an *n* Stellen geändert werden)!

==> Statt gleichem Code an mehreren Stellen im Programm:

- Code *einmal in eine Funktion schreiben*.
- Funktion *von mehreren Stellen aufrufen*.

==> Funktionscode soll möglichst
allgemein und flexibel verwendbar sein,
damit er an möglichst vielen Stellen
wiederverwendbar / aufrufbar ist!

Wozu Funktionen? (2)

Programmcode aufteilen in *überschaubare Portionen!*

Faustregel für die *Codelänge pro Funktion:*

Alles bis 25 Zeilen ist gut,
alles bis 50-100 Zeilen ist zur Not tolerierbar,
alles darüber gehört
in mehrere Funktionen unterteilt!

(Ausnahmen mit Hausverstand,
z.B. bei sehr langen **`switch`** ...)

Wozu Funktionen? (3)

Strukturierung und gedankliche Aufteilung
einer großen Gesamtaufgabe
in *logisch eigenständige Teilprobleme:*

"Each function should do
***one** thing well!"*

Funktionen sind selbständige, in sich abgeschlossene gedankliche & programmtechnische Einheiten,

- die *eine bestimmte Aufgabe erfüllen,*
- *ein eigenständiges Teilproblem lösen, ...*

Beispiele

Typische Aufgaben für *eine* Funktion:

- Sortiere ein Array, suche einen Wert in einem Array, …
- Berechne den größten gemeinsamen Teiler zweier Zahlen.
- Gib eine Fehlermeldung aus und beende das Programm.
- Gib einen Balken mit *x* Mal dem Zeichen *c* aus.
- Berechne den Wochentag zu einem Datum,
  die Sekunden seit Mitternacht zu einer Zeit, …
- Zähle die Anzahl der Worte in einem String.
- Ist eine Zahl prim?
- Enthält ein String eine gültige Zahl?

Siehe auch Datumsbeispiel!

Wozu Funktionen? (4)

Funktionen *unterteilen* nicht nur den Code,
sondern auch die *Daten* in zusammengehörende Teile:

- Jede Funktion hat ihre *eigenen, lokalen Daten*,
  die von anderen Codestücken *nicht beeinflusst*
  werden können.
- Jede Funktion tauscht (fast) nur
  über *Parameter und Returnwert*
  mit dem Rest des Programms Daten aus.

In Programmierer-Deutsch:

Jede Funktion hat eine
wohldefinierte "Schnittstelle" zum Aufrufer.

Wozu Funktionen? (5)

Funktionen machen
(durch ihren Namen)
Code leserlicher
und leichter verständlich.

- 10 Zeilen Code mit Schleifen und **`if`**'s, ...

  ==> Muss erst gelesen / analysiert / verstanden werden:

  Was tut das Ding???

- 1 Funktionsaufruf "**`insert(names, newVal)`**"

  ==> Mit einem Blick naheliegende Bedeutung:

  `newVal` *wird in* **`names`** *eingefügt!*

Wichtig

- Besonders bei Funktionen:

  “Sprechender” Name!

  Standard-Funktionen wie “**`strstr`**”
  sind oft ein *schlechtes* Beispiel! (historisch bedingt …)

- Wichtig auch:

  “Sprechende” Parameter-Namen!

  `strstr(… s1, … s2)` oder

  `strstr(… haystack, … needle)` ?!

  … damit man auf einen Blick weiß,
  was in welcher Reihenfolge übergeben werden muss!

Funktionen helfen! (1)

Eine *gute Aufteilung in Funktionen* erleichtert

- *Entwurf* & *Entwicklung* von Programmen
- das *Testen* von Programmen
  ("Unit-Tests" = jede Funktion für sich testen)

  … und die *Fehlersuche*
  (der Debugger zeigt aktuelle Funktionsaufrufe an)
- das *Lesen und Verstehen* von Programmen
- das *Dokumentieren* von Programmen
  (halb automatisiert: Was macht jede Funktion,
  welche Daten gehen rein, welche kommen raus?)

Funktionen helfen! (2)

Eine *gute Aufteilung in Funktionen* erleichtert

- die *Wartung* & *Erweiterung* von Programmen:

  Auswirkungen von lokalen Änderungen
  sind eingegrenzt und überschaubar

  Schnittstellen-Änderungen
  werden vom Compiler überprüft

- die *Wiederverwendung* von Codeteilen
  für andere Programme

Top-Down-Entwurf (1)

Vorgehen beim Programmieren:

Entwirf dein Programm
von außen nach innen,
vom Groben zum Feinen!

- Fang bei **`main`** an!
- Schreib *zuerst alle groben Schritte* zusammen,

  ignoriere am Anfang die *Detailprobleme*!
- Denk erst dann über jeden Schritt
  einzeln und isoliert von anderen Problemen
  genauer nach.

Top-Down-Entwurf (2)

Immer wenn du auf einen Teilschritt stößt,

- der *zu lang / zu komplex* für direkten Code in der momentanen Funktion ist,
- oder der gedanklich ein *selbständiges Problem* ist,
- oder der *vermutlich noch öfters vorkommt*,

==> ***"erfinde" eine neue Funktion dafür***

==> lege zuerst einmal nur den *Aufruf* (Prototyp) fest

==> und denk *erst später* über das *Innenleben* (die Programmierung) der Funktion nach!

Und wenn die neue Funktion wieder zu lang / kompliziert ist:
Weiter aufteilen!

Funktionen-“Kochrezept” (1)

1) *Was* soll die Funktion überhaupt tun? (*“Problem?”*)

2) *Kopfzeile* gut überlegen:

- *Sprechender Name* für die Funktion?
- *Returntyp* der Funktion (wenn keiner: **`void`**):

  “Welchen Typ hat der Wert,
  den die Funktion als Ergebnis zurückliefern soll?”

- *Argumente* der Funktion (wenn keine: **`(void)`**):

  “Was braucht die Funktion für ihre Arbeit
  vom Aufrufer für Daten?”

  Anzahl und Typ? Sinnvollste *Reihenfolge*?

Funktionen-“Kochrezept” (2)

3) Code schreiben:

- Auf das **`return`** nicht vergessen!
- Passt der Wert im **`return`** zum Returntyp???
- Ev. vor Beginn der eigentlichen Berechnung:

  Parameter auf sinnvolle / zulässige Werte prüfen!
  ==> *Spezieller Fehler-Returnwert?*

Rekursion

- Die Idee und ihre Programmierung
- Die Abarbeitung zur Laufzeit
- Die Speicherung der Daten
- Theoretisches
- Anwendungen

Die Idee

“Löse eine Aufgabe,
indem du sie auf ein oder mehrere
gleichartige, aber “kleinere” Aufgaben
zurückführst,
die du wieder genauso löst,

bis die Aufgaben so klein geworden sind,
dass die Lösung ganz einfach ist.”

==> Das Lösungsverfahren
“verwendet sich selbst” für Teilaufgaben!

… und ihre Programmierung

- Eine Funktion

ruft sich selbst auf

d.h. der Code der Funktion **`recFunc`**
enthält wieder einen Aufruf von **`recFunc(...)`**

==> *"direkte Rekursion"*

- Oder (*"indirekte Rekursion"*):

`funcA` ruft **`funcB`** auf und
`funcB` ruft wieder **`funcA`** auf.

(auch über 3 oder noch mehr Funktionen, *Prototypen notwendig!*)

Beispiel: Die Fakultät

```
int fak(int n)
{
   // Der „ganz einfache“ Fall:
   // „Direkte“ Lösung
   if (n <= 1) return 1;

   // Sonst:
   // Löse das „nächstkleinere“ Problem
   // und berechne daraus die eigene Lösung
   else return n * fak(n - 1);
}
```

Ablauf anschaulich…

"**Die Fakultäts-Brüder**":
Lauter idente "*Klone*" desselben Rechenmeisters!

Vater fragt ältesten Sohn: Fakultät von 4?

Ältester Sohn fragt mittleren Bruder: Fakultät von 3?

Mittlerer fragt seinen kleinen Bruder: Fakultät von 2?

Kleiner Bruder fragt Baby-Bruder: Fakultät von 1?

Baby-Bruder antwortet kleinem Bruder: 1

Kleiner Bruder rechnet 1*2, antwortet mittlerem: 2

Mittlerer Bruder rechnet 2*3, antwortet großem Bruder: 6

Der älteste rechnet 6*4, antwortet seinem Vater: 24

Vater sagt: *Sehr gut!*

… und Ablauf technisch

- *Mehrere Aufrufe derselben Funktion* werden (mit verschiedenen Parametern)

  der Reihe nach gestartet.

- Zu jedem Zeitpunkt

  rechnet nur der aktuell "innerste" Aufruf,

  alle anderen warten auf die Rückkehr des von ihnen gestarteten nächstinneren Aufrufs.

- Die Aufrufe kehren

  in umgekehrter Reihenfolge ihres Aufrufs zurück!

Damit es klappt ... (1)

Ende der Rekursion:

Jede rekursive Funktion **muss**
mindestens *einen **nicht**rekursiven Zweig haben!*

(d.h. ein **`return`** ***ohne*** vorherigen rekursiven Aufruf)

Das ist der "ganz einfache", *direkt lösbare Fall*.

Sonst:

Endlos-Rekursion (endlos viele Aufrufe ohne **`return`**)

bis zum Absturz durch *Speicherüberlauf!*

(Stack voll ==› kein Platz mehr
für die Variablen und Parameter eines weiteren Aufrufs)

Damit es klappt ... (2)

- Das Problem muss mit jedem Aufruf **kleiner** werden!
- Man muss mit jedem Aufruf einen Schritt **näher zum "ganz einfachen" Fall** kommen!

==> Jeder rekursive Aufruf
muss von seinen **Parameter-Werten** her

*"**näher** beim Rekursionsende sein"*

als der Aufrufer:

Kleinere Zahl, weniger Array-Elemente,
weniger noch offene Rechenschritte, ...

Daten in rekursiven Funktionen

Die Daten in allen Aufrufen sind **unabhängig voneinander**:

- Jeder Aufruf hat seine

  eigenen lokalen Variablen und Parameter!

  Ausnahme:
  `static`-Variablen sind *für alle Aufrufe gemeinsam*!

- Jeder Aufruf merkt sich *getrennt* von den anderen,
  wohin er *zurückkehren* muss.

  (Der äußerste nach **`main`** oder in eine andere Funktion,
  die inneren in den vorigen Aufruf derselben Funktion.)

==> Es existieren *mehrere Sätze* von lokalen Variablen,
Parametern und Return-Adressen *gleichzeitig!*
(so viele, wie gerade Aufrufe aktiv sind)

Der Call Stack (1)

Der **Call Stack** ist ein intern vom C-Programm verwalteter, *linearer Speicherbereich* (meist "ganz oben" im RAM).

Er enthält für alle gerade laufenden Funktionen

- die *lokalen Variablen* (außer statischen!)
- die *Parameter*
- die *Return-Adresse* (= Rücksprungadresse in den Aufrufer):

  "Wo muss ich nach der Funktion weitermachen?"

Hat eine Funktion mehrere aktive Aufrufe (*rekursiv*), hat *jeder Aufruf seinen eigenen Satz Daten* am Call Stack.

Bei parallelen Programmen: Jeder *Thread* hat seinen *eigenen Stack!*

Der Call Stack (2)

Das *obere Ende* des Call Stacks ist *fix*
(dort stehen die Daten von **`main`**).

Der Call Stack *wächst bei jedem Funktionsaufruf*
automatisch an seinem *unteren Ende* (neue Variablen, ...),

und er wird bei jedem **`return`** automatisch
am unteren Ende *wieder ein Stückchen kürzer*.

Fachbegriffe:

- "**push**": Etwas am Ende des Stacks *speichern*
  und den Stack *länger machen*.
- "**pop**": Den Wert vom Ende des Stacks *herunterholen*
  und den Stack *kürzer machen*.

Der Call Stack (3)

Im Detail bei jedem Funktionsaufruf:

- Der *Aufrufer* legt am Stack-Ende Platz für die Parameter an und speichert die *Argument-Werte* in den Parametern.
- Der *Call-Befehl* speichert die *Return-Adresse* am Stack-Ende.
- Die *aufgerufene Funktion* macht den Stack um den Platz für ihre *lokalen Variablen* größer.
- ... (die aufgerufene Funktion wird ausgeführt)
- Vor dem **`return`** gibt die *aufgerufene Funktion* den Stack-Bereich für ihre *lokalen Variablen* wieder frei.
- Der *Return-Befehl* holt die *Return-Adresse* vom Stack-Ende.
- Der *Aufrufer* gibt den Platz der *Parameter* wieder frei.

Der Call Stack (4)

Jeder Prozessor "weiß", wo der Stack gerade endet:

Ein *eigenes CPU-Register*, der ***"Stackpointer"***,
zeigt automatisch immer
auf das *untere Stack-Ende*.

Die Funktionen kennen die *absoluten Speicheradressen*
ihrer lokalen Variablen und ihrer Parameter *nicht*
(denn diese Adressen ergeben sich ja erst zur Laufzeit
und sind *bei jedem Aufruf andere*!).

Der Zugriff auf lokale Variablen und Parameter erfolgt

immer relativ zum aktuellen Stackpointer!

Der Call Stack (5)

Beispiele:

- *"Mein Parameter* **n** *liegt 8 Bytes über dem Stackende"*, d.h. liegt auf Speicher-Adresse *(Stackpointer + 8)*.
- *"Mein Array* **`data`** *beginnt 96 Bytes über dem Stackende"*

==> Trifft immer die Daten des *gerade aktuellen Aufrufs!*

- Egal, wo sie liegen und was es an anderen Daten gibt (d.h. unabhängig von der absoluten Adresse der Daten)
- Egal, wie viele Aufrufe / Datensätze (z.B. Parameter **n**) der aktuellen Funktion es gerade am Stack gibt.

==> Die Variablen und Parameter von Aufrufen "weiter oben" sind *nicht direkt ansprechbar* (außer man hat einen expliziten Pointer darauf)!

Theoretische Erkenntnisse

- **`if` + Rekursion + unendlicher Stack**

  reichen als *universeller Rechen-Mechanismus* aus:

  ==> Man braucht *keine Schleifen* und kein **`goto`** !

  ==> Jede Schleife kann *durch Rekursion ersetzt* werden!

- Man kann jede Rekursion *durch eine Schleife ersetzen*,

  benötigt aber ein potentiell *unendlich großes Array*
  zur *Simulation des Stacks*.

(Beide Transformationen sind *automatisiert* möglich,
resultieren aber in ziemlich *unleserlichem* Code.)

Anwendungsbeispiele Rekursion

- Rekursion statt Schleifen
- Rekursiv definierte Datenstrukturen
- Divide and Conquer
- Backtracking,
  Alpha-Beta-Algorithmus (==> 2. / 3. Sem.)
- Compilerbau (==> eventuell 5. / 6. Sem.)
- Rekursive „Spielereien“

Rekursion statt Schleifen (1)

Die *klassischen Lehrbuch-Rekursionsbeispiele*
(siehe folgende Folien: Fakultät, rek. ggT nach Euklid, ...)
sind rekursiv *schöner, aber ineffizienter*:

Sie brauchen (im Vergleich zur Lösung mit Schleife):

- *Konstant mehr Rechenzeit*: Aufwand für Call & Return
- *n-fach mehr Speicher*: 1 Aufruf pro Schleifenumlauf

  ==> Speicherbedarf *wächst* mit der Eingabe-Größe!

  Eine Schleife braucht nur *konstant* viel Speicher!

==> Rekursion ist in diesen Fällen

praktisch nicht sinnvoll!

Die Fakultät

```
int fak(int n) {
  if (n <= 1) return 1;
  return n * fak(n - 1);
}
```

Macht **n** rekursive Aufrufe:

==> Braucht **n** *mal soviel Speicher*
(für **n** Stück "**int n**" und **n** Stück Rückkehradressen)
wie **n** Multiplikation in einer Schleife...

==> daher besser mit Schleife!

ggT nach Euklid

```
int ggT(int a, int b) {
  if (b == 0) return a;
  return ggT(b, a % b);
}
```

Auch mit Schleife etwas schneller
und sparsamer im Speicherverbrauch...

Rekursion statt Schleifen (2)

Beispiele mit **zwei** ***„um 1 kleineren“*** rekursiven Aufrufen

```
int fib(int n) {
  if (n < 2) return 1;
  return fib(n – 1) + fib(n – 2);
}

int binom(int n, int k) {
  if (k > n) return 0;
  if ((k == 0) || (k == n)) return 1;
  return binom(n – 1, k – 1) + binom(n – 1, k);
}
```

… sind *extrem ineffizient* **(~2^n rekursive Aufrufe) !!!**

Viel besser:

“Von unten nach oben” mit *Array und Schleife* rechnen!

==> Zeit ~n statt ~2^n, Speicher konstant statt ~n

Aber …

… fast alle anderen Anwendungen von Rekursion sind

sehr wohl sinnvoll, effizient

und in der Praxis äußerst bedeutend!

Weil (gerade bei 2 oder mehr rekursiven Aufrufen!):

- Die „natürliche“ Lösungsidee ist oft rekursiv,
  eine reine Schleifen-Lösung ist *schwerer zu verstehen*.

  Bei zwei oder mehr rekursiven Aufrufen ist
  der „derekursivierte“ Code meist *völlig unleserlich und viel länger*!

- Es gibt meist *keine Lösungen* ohne Rekursion,
  die *deutlich schneller* sind.

Rekursive Datenstrukturen

Beispiel: **Binärer Baum** (sinngemäß):

„Ein binärer Baum ist entweder *leer*
oder besteht aus einem Wert, an dem
links und rechts wieder ein binärer Baum hängt.“

==> Viele Operationen auf Bäumen
machen einen *rekursiven Aufruf*
für den *linken und/oder rechten Unterbaum*:

```
int nodeCount(tree *p)
{
   if (p == NULL) return 0;
   else return
      1 + nodeCount(p->left) + nodeCount(p->right);
}
```

“Divide and Conquer”

= *“Teile und Herrsche”*

Wesentlicher Unterschied zu „primitiver“ Rekursion:

- Teile das Problem in **mehrere Teile:**

  Meist *2 möglichst gleichgroße Hälften* (n/2)
  statt 1 Unterproblem der Größe n-1 .

- Löse *rekursiv jedes Teilproblem einzeln* für sich

  ==> **mehrere** rekursive Aufrufe!

- Berechne aus den Teillösungen die Gesamtlösung.

==> *Meist schöne & sehr effiziente Lösungen!*

Binäre Suche

```
int binsearch(int a[], int val, int l, int r)
{
  if (l > r) return -1;
  int m = (l + r) / 2;
  if (a[m] < val)
    return binsearch(a, val, m + 1, r);
  else if (a[m] > val)
    return binsearch(a, val, l, m - 1);
  else   /* (a[m] == val): gefunden! */
    return m;
}
```

(im Unterschied zu vielen anderen “Divde-and-Conquer”-Algorithmen ist die binäre Suche *mit Schleife effizienter*!)

Merge-Sort

- Betrachte die linke und die rechte Hälfte des Arrays.
- *Sortiere jede Hälfte getrennt* für sich
  (*rekursiv* wieder mit Mergesort).
- *Vereinige die beiden sortierten Hälften*
  zu einem sortierten Ergebnis
  (“merge”, ähnlich dem “Reißverschlussprinzip”).
- Rekursionsende:
  Wenn eine Hälfte nur mehr Länge 1 hat,
  ist sie schon fertig sortiert.

(hier ist beim Teilen nichts zu tun,
aber das Kombinieren der Teillösungen macht viel Arbeit)

Quicksort

- Such dir ein "*mittelgroßes*" Element.
- Schaufle alle *kleineren* Elemente nach *links*,
  alle *größeren* nach *rechts*.
- *Sortiere den linken und den rechten Teil* getrennt für sich
  (*rekursiv* wieder mit Quicksort)
  ==› Das gesamte Array ist danach schon fertig sortiert!
- Rekursionsende: Ein Teil der Länge 1 ist schon sortiert…

(hier ist das Teilen viel Arbeit,
aber beim Kombinieren der Teillösungen ist nichts zu tun)

==› Rekursionstiefe: *~log(n)*, Zeit: *~n*log(n)*

==› Viel besser als „dummes" Sortieren mit Schleifen
(das rund *~n²* Zeit braucht)

Rekursive „Spielereien“

- **„Towers of Hanoi“:**

  „Um n Scheiben umzulegen,
  lege zuerst n-1 Scheiben …“

- **Raumfüllende Kurven**
  (Drachenkurve, Hilbert-Kurve, Sierpinski-Kurve, …):

  „Ersetze jede Teilstrecke
  durch eine verkleinerte Kopie der ganzen Kurve!“

Variablen-Speicherklassen

- Lokale Variablen und Parameter
- Globale Variablen und Shadowing
- Statische lokale Variablen
- Externe Variablen
- File-statische Variablen
- Dynamische Daten

Lokale Variablen

Bisher hatten wir *nur* lokale Variablen (der "**Normalfall**"):

- Sind *innerhalb der* **{ }** *einer Funktion* deklariert.
- Gelten von ihrer Deklaration bis zum *Ende* der **{ }**.
- Haben mit (gleichnamigen) Variablen anderer Funktionen *nichts* zu tun, sind nur in der Funktion verwendbar.
- Werden bei *jedem Aufruf neu im Speicher angelegt* und zwischen Aufrufen *wieder vergessen*: Sind bei **`return`** weg!
- Falls Initialisierung angegeben: Erfolgt bei *jedem Aufruf neu.*
- Sonst: Inhalt bei jedem Aufruf wieder zufällig / *undefiniert*.
- Jeder rekursive Aufruf hat seine *eigenen lokalen Variablen.*

Parameter

... sind ganz normale lokale Variablen
der aufgerufenen Funktion (nicht des Aufrufers)

==> Verhalten sich wie soeben beschrieben.

==> Gelten innerhalb der **{ }** der Funktion.

==> Haben mit gleichnamigen Variablen oder Parametern in anderen Funktionen *nichts* zu tun (auch nicht mit denen im Aufrufer!).

Einziger Unterschied:

Werden gleich am Anfang der Funktion
mit den Argument-Werten des Aufrufers initialisiert!

Globale Variablen (1)

Globale Variablen sind alle Variablen, deren *Deklaration außerhalb aller* **{ }** *von Funktionen* oder von **`main`** steht.

- Gelten von der Deklaration
  bis zum Ende des Source-Files.
  (bitte *ganz oben im File* deklarieren, nicht mittendrin!!!)
- *Alle Funktionen* können darauf zugreifen und
  sehen und verwenden **dieselbe** *Variable*
  (auch alle rekursiven Aufrufe sehen dieselbe Variable).
- Werden bei Programmstart
  ein einziges Mal fix im Speicher angelegt,
  leben *bis zum Programm-Ende,*
  behalten ihren Wert solange.

Globale Variablen (2)

- *Keine* Initialisierung angegeben:

  Inhalt ist beim Programmstart **automatisch** **`0`**
  (im Unterschied zum *"Blödsinn"* in lokalen Variablen).

- Falls Initialisierung angegeben:

  Erfolgt *einmal beim Programmstart* (*vor* **`main`** !)

- In **C**: Der *Init-Wert* muss *konstant* sein
  (wird nicht ausgerechnet, sondern ist fixer Wert im **`.exe`**-File)

  In **C++**: Der Init-Wert darf auch ein *variabler* Ausdruck sein
  (wird *vor* **`main`** ausgerechnet).

- Bei Arrays: Die *Größe* muss ebenfalls *konstant* sein.

Globale Variablen (3)

Verwendung:

Für zentrale, programmweite Daten,
die von mehreren Funktionen benutzt werden.

Vorteil:

Globale Variablen können
überlange Parameter-Listen ersparen!

Nachteil:

Es ist nicht auf den ersten Blick erkennbar,
welche Funktion wann welche globalen Daten wie ändert
und welche Auswirkungen das auf andere Funktionen hat!

Globale Variablen (4)

==> *Übermäßige Verwendung von globalen Daten führt zu schwer durchschaubarem und fehleranfälligem Code!*

- Globale Variablen *lesen* ist meist tolerierbar
- Globale Variablen *ändern* sollte *auf ein Minimum beschränkt* werden!!!
- Globale Variablen sind *kein Ersatz für Parameter oder Returnwerte*, wenn es nur um den Austausch von Daten *zwischen **einem** Aufrufer und **einem** Aufgerufenen* geht!

“Shadowing”

Eine lokale Variable (oder ein Parameter)
darf ***genauso heißen*** wie eine globale Variable
(trotzdem sind es zwei *getrennte* Variablen!).

Innerhalb der Funktion ist *nur die lokale Variable* sichtbar,
sie “**verdeckt**” oder “**versteckt**” die globale Variable:
Es gibt sie, aber man kann nicht darauf zugreifen.

Das nennt man *“Shadowing”*.

Es ist verwirrend, man sollte gleiche Namen vermeiden!

Für Fortgeschrittene:

In C++ gilt dasselbe: Die lokale Variable “versteckt” die globale Variable.
Mit **`::xxx`** kann man aber auch innerhalb der Funktion
trotz lokalem **`xxx`** ausdrücklich auf die *globale* Variable **`xxx`** zugreifen.

Statische Variablen (1)

... sind *lokale Variablen*
(innerhalb einer Funktion / in **`{ }`** deklariert),
in deren *Deklaration vorne* **`static`** steht.

... sind von der *Sichtbarkeit* her *lokale Variablen:*

- Sind *nur in dieser Funktion verwendbar*,
  gelten bis zum Ende der **`{ }`**.
- Sind in anderen Funktionen unsichtbar.

... aber ...

Statische Variablen (2)

... sind von der *Speicherung* her *wie globale Variablen:*

- Stehen *ein Mal fix im Speicher*.
- Werden *bei Programmstart* angelegt und *einmal* mit dem angegebenen Wert oder **`0`** *initialisiert* (*bevor* **`main`** losrechnet, *nicht* beim ersten Aufruf!), und leben dann *bis zum Programmende*.

  Der *Init-Wert* muss daher wie bei globalen Variablen in C *konstant* und *zur Compilezeit berechenbar* sein!

==> *Behalten ihren Wert* zwischen Aufrufen.

==> Alle rekursiven Aufrufe sehen *dieselbe Variable,* nicht eine neue pro Aufruf.

Statische Variablen (3)

Verwendung:

- Für Werte, die sich die Funktion *zwischen Aufrufen merken* soll.

  Beispiel:

  - Aufrufzähler, Aufruf-Statistiken
  - Initialisierungs-Flag und -Daten (*"Bin ich schon einmal aufgerufen worden?"*)
- Für Daten, die alle *rekursiven Aufrufe einer Funktion gemeinsam* haben sollen.

  Beispiel: Spielfeld / Lösung beim Backtracking.

Externe Variablen (1)

Normale globale oder lokale Variable mit **`extern`** davor:

- Sichtbar wie eine globale oder statische Variable ...
- ... aber es wird *kein eigener Speicher* dafür angelegt ...
- ... sondern die *globale Variable gleichen Namens aus einem anderen Source-File* verwendet:

  ==> In genau einem **`.c`**-File muss die Variable *ohne* **`extern`** deklariert sein!

==> Für globale Variablen, die

von mehreren Source-Files gemeinsam verwendet werden sollen.

Externe Variablen (2)

Achtung:

- Der *Typ* der Variable muss in allen Files *gleich* sein!

  (viele Compiler liefern *keinen* Fehler bei falschem Typ!)

- Die *Größe* (bei Arrays) ist die Größe
  in der nicht-**`extern`**-Deklaration!
  In den **`extern`**-Deklaration stehen oft nur leere **`[ ]`** .

  (viele Compiler liefern *keinen* Fehler bei versch. Größe!)

- Eine *Initialisierung* kann nur
  in der nicht-**`extern`**-Deklaration angegeben werden!

`extern`-Variablen werden bei uns nicht vorkommen...

File-statische Variablen

Globale Variablen mit **`static`** davor:

- Funktionieren genauso wie globale Variablen
- Aber sind *nicht in anderen Files* sichtbar
  (können dort nicht mit **`extern`** angesprochen werden)

==> vermeidet *Namenskollisionen* zwischen Files

==> *bei allen globalen Variablen* **`static`** *davorschreiben,*
wenn sie nur in diesem einen File verwendet werden sollen!

File-statische Variablen werden wir ebenfalls nicht haben...

- **`static`** kann man auch vor ***Funktions***-Definitionen schreiben
  Selber Effekt: Die Funktion ist *nur innerhalb dieses einen* **`.c`***-Files sichtbar*
- **`static`** bildet in C die Sichtbarkeits-Attribute objektorientierter Sprachen nach:
  C's **`static`** entspricht auf File-Ebene dem **`private`** in OO-Sprachen auf Klassen-Ebene

Dynamische Daten

Neben *lokalen Variablen* (bei Aufruf am *Stack*) und *globalen / statischen / externen Variablen* (*fix* im Speicher) gibt es auch **dynamische Daten**:

- Speicher für dynamische Daten wird zur Laufzeit vom Programm *explizit angefordert und freigegeben*:

  In C: **`malloc`** / **`free`** In C++: **`new`** / **`delete`**

- Der dafür benutzte *Speicherbereich* heißt "**Heap**".
- Sie gehören zu *keiner fix deklarierten Variablen* und werden daher *nicht mit Namen angesprochen*.
- Man hat nur einen *Zeiger* (==› kommt gleich) darauf!

==› *Kommt ein bisschen später …*

Zeiger

- Was sind Zeiger?
- Wozu braucht man sie?
- Deklaration und Verwendung
- `void *`
- `NULL`
- Zeiger auf Funktionen
- *“Call by Reference”*

Zeiger (engl.: Pointer)

Gedanklich:

Sind ***Verweise***

==> Zeigen auf andere *(schon existierende!)*
Variablen bzw. Daten

Technisch:

Sind Variablen,
die ***Speicher-Adressen*** enthalten

==> "*Wo* stehen die Daten im Speicher?"

Zeiger sind ein zentrales Konstrukt in C / C++ !

Wozu braucht man Zeiger? (1)

1) "**Call by Reference**" statt "Call by Value" (Kopie):

Die aufgerufene Funktion bekommt einen

Zeiger auf die Daten im Aufrufer

(statt einer Kopie der Werte vom Aufrufer)

==> Die aufgerufene Funktion arbeitet

direkt *auf den Original-Daten im Aufrufer*

und kann *die Daten im Aufrufer* ***ändern!***

==> demnächst hier

Wozu braucht man Zeiger? (2)

2) **Dynamisch angelegte Daten:**

"Reserviere im Speicher jetzt irgendwo
neuen Platz für **`anzahl`** *viele* **`double`***'s*
und gib mir einen Zeiger darauf"

==› etwas später

3) **Verkettete Datenstrukturen** (Liste, Baum, ...):

Datenelemente werden *einzeln dynamisch angelegt*
und *miteinander durch Zeiger verbunden*:

Jedes Element enthält einen (oder mehrere) Zeiger
auf seine Vorgänger / Nachfolger

==› vor allem im 2. Sem. *"Algorithmen & Datenstrukturen"*

Wozu braucht man Zeiger? (3)

4) **Zugriff auf Array-Elemente,**
Durchlaufen von Arrays in Schleifen:

= *Rechnen* mit Pointern bzw. Speicheradressen:

Ersetzt Array+Index, ist (minimal) schneller.

Vor allem für Strings in C allgemein üblich!

==› demnächst hier

5) **Zeiger auf Funktionen:**

Zeiger auf *Code* statt auf Daten.

==› kurz hier & in der Übung

Deklaration von Zeigern (1)

Mit '*' vor dem Variablennamen.

Beispiel:

```
int *p;
```

==> "**p** ist ein (enthält einen) Zeiger auf einen **int**-Wert"

==> "***p** liefert einen **int**-Wert" (siehe *"Dereferenzierung"*)

Achtung: Der * gehört vor *jeden einzelnen* Variablennamen!

- ```
  int *p1, p2;
  ```
  ==> **p2** ist normaler **int**, *kein* Zeiger!
- ```
  int *p1, *p2;
  ```
  ==> 2 Zeiger

Deklaration von Zeigern (2)

Auch komplizierter:

- **`char **words;`**

  `words` ist ein *Zeiger auf einen Zeiger* auf **`char`**'s

  ==> **`*words`** liefert einen Zeiger auf **`char`**'s

  ==> **`**words`** liefert einen **`char`**

Deklaration von Zeigern (3)

- **`double *parr[10];`**

  `parr` ist ein *Array von 10 Zeigern* auf **`double`**'s

  ==> **`parr[i]`** ist ein Zeiger auf einen **`double`**

  ==> **`*parr[i]`** liefert den **`double`**,
  auf den der **`i`**-te Zeiger in **`parr`** zeigt

  Klammerung ist automatisch **`*(parr[i])`** :
  Zuerst **`[]`**, dann **`*`**

- *Analog:* **`const char *argv[]`**

  `argv` ist ein *Array von Zeigern* auf **`const char`**
  (d.h. ein Array von Zeigern auf Texte),
  die Größe des Arrays ist unbekannt

Deklaration von Zeigern (4)

- **`double (*p)[10];`**

  `p` wäre ein *einzelner Pointer auf ein Array* von 10 **`double`**'s

  Zugriff auf einen solchen **`double`** wäre dementsprechend **`(*p)[i]`**

  Aber:

  Wird so praktisch nie verwendet,
  stattdessen einfach **`double *p;`** *!*

  Weil:

  Man unterscheidet in C praktisch *nie* zwischen
  "Pointer auf einen einzelnen Wert vom Typ **`xxx`***"*
  und *"Pointer auf ein Array mit Element-Typ* **`xxx`***"*:

  In beiden Fällen wird **`p`** *einfach als* **`xxx *p;`** *deklariert!*

Deklaration von Zeigern (5)

Mit **`const`** (bei uns nicht so wichtig...):

- **`const char *p;`**
  Änderbarer Zeiger auf *konstante Zeichen*
  ==> Man darf **`p`** *ändern* (auf einen *anderen Text zeigen* lassen)
  ==> Die *Zeichen*, auf die **`p`** zeigt, sind *nicht änderbar*
- **`char const *p;`**
  Genau *dasselbe*.
- **`char *const p = ...;`**
  Konstanter Zeiger auf *änderbare Buchstaben*
  ==> Man darf **`p`** *nicht* ändern (zeigt fix auf dieselben **`char`**'s)
- **`const char *const p = ...;`**
  Beides ist konstant.

Deklaration von Zeigern (6)

In Funktions-Prototypen:

- Funktions-*Parameter* können Pointer sein:

  `int strlen(const char *str);`

  "Die Funktion bekommt einen Pointer als Parameter."

  (Bedeutung und Verwendung: Kommt gleich!)

- Und *Returnwerte* ebenfalls:

  `char *strchr(const char *str, char c);`

  Das sieht zwar komisch aus, aber bedeutet ganz normal:

  "Die Funktion liefert einen Pointer (`char *`) als Ergebnis."

  (das hat *nichts* mit "Pointer auf Funktion" o.ä. zu tun!)

Woher bekommt man Zeiger?

1) Vom ***"Address-of-Operator" &***

   ... der einen *Zeiger auf die angegebene Variable* liefert

2) Von Funktionen, die einen **Zeiger als Returnwert** liefern:

   `malloc`, **String-Funktionen** usw.

3) Von jedem **Array-Namen** und jeder **String-Konstante**

   ... die technisch gesehen *Zeiger-Konstanten* sind
   (den Zeiger auf den Anfang des Arrays / Strings liefern)

4) Von **schon existierenden Zeiger-Variablen**

5) Von Ergebnissen von **Zeiger-Rechnungen**

Das '**&**': Der *"Address-of-Operator"*

Wie bekomme ich einen Zeiger auf eine Variable???

Ein '**&**' vor einer Variable liefert
einen *Zeiger auf diese Variable*
(ihre *Adresse* im Speicher).

- ```
  int i, *p;
  p = &i;
  ```

  "Gib mir einen Zeiger auf **i** *und speichere ihn in* **p***"*

- ```
  double array[100], *p;
  p = &(array[i]);
  ```

  "Speichere einen Zeiger auf das **i***-te Element von* **array** *in* **p***"*

Das '*': Die *"Dereferenzierung"*

Wie komme ich von einem Zeiger zu den Daten???

Ein '*' vor einem Zeiger liefert
den *Wert der Variablen, auf die der Zeiger zeigt*
(den *Inhalt* der Speicherzelle,
deren Adresse im Zeiger steht)

Die Variable, auf die der Zeiger zeigt,
kann *gelesen oder geschrieben* werden:

```
char text[82], *p;
p = &(text[i]);      // p zeigt auf das i-te Zeichen in text
if (*p == '\n') {    // ist das i-te Zeichen ein Zeilenende?
   *p = '\0'; }      // ersetze es durch die Endemarkierung
```

Nicht verwechseln ... (1)

Für einen Zeiger **p** (z.B. für **char *p**) gilt:

- **p *ohne* *:**

  Betrifft ***den Zeiger selbst***,
  nicht den Wert, auf den er zeigt

  ==> Liefert einen *Zeiger-Wert!*

- ***p (*mit* * davor):**

  Betrifft ***den Wert, auf den der Zeiger zeigt***,
  nicht den Zeiger selbst

  ==> Liefert in diesem Beispiel einen **char**-*Wert!*
  (weil **p** als Zeiger auf **char** deklariert ist)

Nicht verwechseln … (2)

Auch bei einer *Zuweisung* gilt (**p** sei wieder ein **char ***):

- **p = … ;** (und **char *p = ...** in der *Deklaration*)

Ändert den Zeiger selbst:
Lässt ihn ***woanders hinzeigen!***

==> Rechts muss ein *Zeiger-Wert* stehen
(eine andere Zeiger-Variable, Zeiger-Rechnung,
Funktionsaufruf mit Zeiger als Returntyp, **&** , …)!

- ***p = … ;** (*außer* **char *p = ...** in der *Deklaration*)

Ändert den Wert, auf den der Zeiger zeigt

==> Rechts muss ein **char**-*Wert* (Zeichen) stehen!

Die Tücke der Zeiger (1)

C "weiß" nicht (und *kann daher auch nicht prüfen!*),

- ob ein Zeiger überhaupt auf *sinnvolle Daten* zeigt

  (oder "irgendwohin"),
- ob die Daten, auf die der Zeiger zeigt, überhaupt *noch existieren*

  (oder inzwischen wieder *freigegeben* wurden),
- und ob die Daten, auf die der Zeiger zeigt, wirklich den *erwarteten Typ* haben

  (oder ganz andere Werte repräsentieren)

==> ***p** ist *gefährlich!*

Die Tücke der Zeiger (2)

Wenn ***p** auf ***"irgendwelche" Speicherbereiche*** zeigt,
die aus Betriebssystem-Sicht *dem Programm gehören*:

Ein Zugriff auf ***p** kann
(fast) beliebige Speicherbereiche des Programms

lesen und überschreiben!

- ... stillschweigend, ohne dass das sofort erkannt wird,
  ohne dass das automatisch eine Fehlermeldung auslöst!
- ... ev. auch den Programm-Code, eine Return-Adresse o.ä.:
  ==> Ausführung von Schadcode?!
  (auf *modernen Systemen* ist der Code-Bereich deshalb oft *schreibgeschützt*)

Die Tücke der Zeiger (3)

Wenn ***p** auf ***"irgendwelche" Speicherbereiche*** zeigt,
die aus Betriebssystem-Sicht ***nicht** dem Programm gehören*:

*Das **Betriebssystem** bricht das Programm*
*beim ersten Zugriff auf ***p** gewaltsam ab:*

Sofortiger "Absturz!"

Es ist *ausdrücklich erlaubt*, einen Zeiger auf das erste Element
vor oder hinter einem Array zu *berechnen*, z.B. **`end = &(a[SIZE]);`**
nur auf ***end** zugreifen* darf man für solche Pointer-Werte *nicht*!

Die Tücke der Zeiger (4)

Ein *frisch deklarierter* Zeiger **p** enthält
(wie jede frisch deklarierte Variable)
am Anfang einen *zufälligen Wert ("Blödsinn")!*

==> **p** zeigt anfangs auf *zufällige* Speicherbereiche!

==> Ein *Zugriff* ***p** *liest / schreibt "irgendwelche" Daten (oder stürzt ab)!*

Daher:

- *Zuerst* den Zeiger auf bestimmte Daten *zeigen lassen* (Initialisierung oder Zuweisung auf den Zeiger)
- Erst *dann Zugriff mit* * auf die Daten, wo er hinzeigt!

Der Pointer-Typ **`void *`**

Obwohl "**`void`**" der "Nichts"-Typ ist:

Ein Zeiger **`p`** vom Typ "**`void *p;`**" ist

- *nicht "Pointer auf nichts"!*
- sondern

"Pointer auf (gültige!) Daten beliebigen / unbekannten Typs"!

==> **`*p`** ist für **`void`**-Pointer *verboten*
(weil der Compiler den Typ des Ergebnisses nicht kennt)

==> Zuerst *Typ-Umwandlung auf "echten" Zeiger-Typ,*
dann erst Zugriff! (z.B. "**`c = *((char *) p);`**")

Der **NULL**-Pointer (1)

- **NULL** ist eine *vordefinierte Zeiger-Konstante*:

  Ein spezieller Pointer-Wert, der anzeigt,
  dass der Zeiger *ungültig ist bzw.*

  absichtlich auf "nichts" zeigt.

- Zugriff auf ***p** wenn **p** gleich **NULL** ist:

  Immer sofort garantierter Absturz!

  (außer bei ganz kleinen 8- und 16-bit-Prozessoren)

  Grund: Der **NULL**-Pointer enthält immer die **Adresse 0**.
  Bei modernen Prozessoren und Betriebssystemen
  ist die **Adresse 0** *immer für das Betriebsysytem* reserviert
  und gehört *nie* zum Speicherbereich normaler Programme.

Der **NULL**-Pointer (2)

Übliche Definition in **`stdio.h`** :

```
#define NULL ((void *) 0)
```

==> **NULL** hat intern den ***Wert 0*** (zeigt auf die Adresse 0)

==> Ein Pointer, der den Wert **NULL** enthält,
gilt in Bedingungen als "**`false`**" (**falsch**)
und jeder andere Pointer-Wert gilt als "**`true`**":

- **`while (p) {`** ... ist dasselbe wie
  `while (p != NULL) {` ...
- **`if (!p) {`** ... ist dasselbe wie
  `if (p == NULL) {` ...

Der **NULL**-Pointer (3)

In **C++** liefern manche Compiler bei Verwendung von **NULL** *Typ-Fehler*.

Grund:

- In C wird in Vergleichen und Zuweisungen *automatisch konvertiert* zwischen **`void *`** (dem Typ von **NULL**) und "echten" Pointer-Typen.
- In C++ nicht: Ein **`void *`** wird in Vergleichen und Zuweisungen *nicht* automatisch an den Typ des anderen Pointers angepasst.

Abhilfe:

- Seit C++11: Neues Sprachkonstrukt **`nullptr`** statt **NULL**
- Zur Not einfach die *Zahl 0 als Pointer* verwenden: **`0`**
  (nur für **`0`** erlaubt, aber nicht schön)
- Oder eine *explizite Typkonvertierung* auf den "echten" Typ machen,
  z.B. **`((char *) 0)`** oder **`((char *) NULL)`**

Seit **C23** gibt es **`nullptr`** *auch in C* (wahlweise statt **NULL**)

Zeiger auf Funktionen (1)

Für Fortgeschrittene:

- *Deklaration* z.B.:
  `int (*fp)(const char *, const char *);`

  ==> Deklariere einen *Zeiger* **`fp`**,
  der *auf eine Funktion* mit Returntyp **`int`**
  und zwei Parametern vom Typ **`const char *`** zeigt.

- *Zuweisung* eines Funktions-Wertes z.B.: **`fp = strcmp;`**

  "Lass **`fp`** *auf die vordefinierte Funktion* **`strcmp`** *zeigen."*

  Wie bei Arraynamen ohne **`[]`** dahinter gilt:
  Schreibt man einen Funktionsnamen *ohne* **`()`** dahinter,
  bekommt man einen Zeiger auf diese Funktion (man braucht *kein* **`&`**).

- *Aufruf* z.B.: **`result = (*fp)(str1, str2);`**

Zeiger auf Funktionen (2)

Wichtigste Anwendung:

Übergeben von Funktionen als Parameter.

Beispiel: Die vordefinierte Sortierfunktion **qsort** bekommt die *Funktion zum Vergleich von 2 Elementen* als Parameter:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Aufruf z.B.: `qsort(lines, SIZE, sizeof(char *), textCmp);`

in diesem Fall für eine selbst geschriebene *Funktion*

```
int textCmp(const void *p1, const void *p2)
{
  ...
}
```

Zeiger auf Funktionen (3)

Oft besser lesbar mit **`typedef`** (lernen wir später):

```
typedef int (*vergl_funp)(const char *, const char *);
```

==> **`vergl_funp`** steht für den Typ
"Pointer auf Funktion mit **`int`***-Returnwert*
und zwei **`const char *`** *Parametern"*

Dann:

```
vergl_funp fp;
```

oder

```
void mein_sort(const char *arr[], int size,
               vergl_funp fp);
```

Call by Reference (1)

Die aufgerufene Funktion bekommt als *Parameter*

statt dem *Wert* des Arguments
(einer *Kopie* des Argumentes)

einen *Zeiger auf den Speicherplatz des Argumentes im Aufrufer*

==> Die aufgerufene Funktion

... "sieht" die Variable im Aufrufer

... und kann sie ändern!

In C muss man das *"händisch" mit Pointern* progammieren
In C++ gibt es dafür ein *eigenes Sprachkonstrukt*: **&**-Parameter

Call by Reference (2)

Beispiel: *"Vertausche den Inhalt von 2* **`int`***-Variablen!"*

==> **`swap`** muss den Speicherplatz
der zu tauschenden Variablen *im Aufrufer* kennen!

```
void swap(int *p1, int *p2)
{
  int t;
  t = *p1; *p1 = *p2; *p2 = t;
}
```

Aufruf: ***Pointer*** *auf die zu tauschenden Werte* übergeben!

- Vertausche den Inhalt von **`i`** und **`n`**: **`swap(&i, &n);`**
- Vertausche den Inhalt von **`a[i]`** und **`a[j]`**:

  `swap(&(a[i]), &(a[j]));`

Call by Reference (3)

Beispiel:

`scanf` (das Gegenstück zu **`printf`**: Formatiertes Einlesen)
muss die eingelesenen Werte *im Aufrufer* speichern!

==> Einzulesende Variablen ***nicht*** ***"by Value"*** übergeben
(**`scanf`** würde die eingelesenen Daten
sonst *nur in seine eigene Kopie* speichern!),

sondern ***"by Reference"***
(= Zeiger auf die *zu befüllende Variable* im Aufrufer)!

Daher:

```
scanf("%d", &anzahl);
scanf("%lf", &(a[i][j]));
```

Call by Reference (4)

Wenn eine Variable,
die *im Aufrufer* schon eine *Pointer-Variable* ist,
"by Reference" übergeben werden soll,
so ist der Parameter der Funktion

ein Pointer auf einen Pointer!

Beispiel: Vordefinierte Funktion **`strtod`** :

```
double strtod(const char *str,
              char **endPtr);
```

(konvertiert wie **`atof`** einen String **`str`** in einen **`double`**
und speichert in **`*endPtr`** zusätzlich einen Pointer
auf das erste nicht konvertierbare Zeichen in **`str`**)

Call by Reference (5)

Beispiel zu `strtod`:

```
char *nextCharP;

// Ein & angewendet auf einen Pointer ...
// ... liefert einen Pointer auf einen Pointer, hier char **
// ==> Übergeben wird die Adresse des Pointers nextCharP
double x = strtod(argv[1], &nextCharP);

// nextCharP zeigt jetzt auf das erste Zeichen
//      nach der verwandelten Zahl
// Fehler wenn dieses Zeichen nicht die Ende-Markierung ist:
//      argv[1] enthielt nicht verwandelbare Zeichen!
if (*nextCharP != '\0') {
   fprintf(stderr, ...);
}
```

Zeiger auf Arrays

- Grundlagen
- Arrays als Funktions-Parameter
- Arrays im Funktions-Aufruf
- Arrays als Funktions-Returnwert
- Mehrdimensionale Arrays
  als Funktions-Parameter
- Zeiger-Vergleich, Zeiger-Rechnungen, Zeiger-Schleifen

Zeiger und Arrays (1)

Ein Arrayname *ohne* **[ ]** dahinter
wird automatisch zu einem
Zeiger auf den Array-Anfang!

(C denkt sich intern ein '**&**' davor)

==> "**arr**" ist dasselbe wie "**&(arr[0])**"
(Adresse des Arrays = Adresse des ersten Elementes)

==> "***arr**" ist dasselbe wie "**arr[0]**"
(Inhalt des 1. Elementes)

==> "**&arr**" ist sinnlos

Zeiger und Arrays (2)

Beispiel:

```
char txt[82];
char *p;
```

`p = txt;`

Das ist eine *Zuweisung eines Zeigers*, *keine Kopie* des Array-Inhaltes!

Sie speichert einen *Zeiger auf den Anfang* von **`txt`** (auf den ersten Buchstaben von **`txt`**) in **`p`**!

Zeiger und Arrays (3)

Eine *Stringkonstante* in **" "** ist ein "**const char**"-Array

==> ihr Wert ist ein *Zeiger auf den ersten Buchstaben*

- **const char *errMsg;**

  ...

  errMsg = "Falsche Eingabe";

  ==> **errMsg** zeigt auf die Stringkonstante (keine Kopie!)

- **const char *monate[] =**
  { "Jan", "Feb", "Mar", ... , "Dez" };

  ==> **monate** ist ein Array von 12 Zeigern auf **const char**

  ==> Jedes Element zeigt auf den entsprechenden Text

Zeiger und Arrays (4)

Umgekehrt:

Man darf hinter jeden Zeiger **`[ ]`** *schreiben,*
als ob er ein Array wäre.

Man braucht dafür keinen **`*`** *!*

Wenn **`p`** *auf ein Array zeigt*, funktioniert das genauso,
wie wenn **`p`** das Array selbst wäre:

"**`p[i]`**" liefert das **`i`**-te Element des Arrays!

(und wenn **`p`** *nicht* auf ein Array zeigt,
greift C *trotzdem* auf den **`i`**-ten Wert hinter **`*p`** zu,
egal, ob das sinnvoll / gültig ist oder nicht!)

Arrays als Parameter

- Arrays können gar *nicht "by Value"* übergeben werden (C kann *keine lokale Kopie* eines Arrays machen)

  ==> *Arrays müssen **immer "by Reference"** (Zeiger darauf) übergeben werden!!!*

  ==> Ein Parameter für ein Array zeigt immer auf das *erste Element des Arrays **im Aufrufer***

  ==> Zuweisungen auf Array-Elemente in der Funktion ***ändern** immer das **Array im Aufrufer**!*

- Wichtigster / häufigster Fall: **String (Text)** als Parameter

  ==> Der Typ ist "**`char *`**" oder "**`const char *`**"
  ==> Der Wert ist ein *Zeiger auf das erste Zeichen*

Arrays als Parameter - Deklaration

Deklaration des Parameters in der aufgerufenen Funktion:

Array-Parameter können mit `*` oder mit `[]` deklariert werden:

- `[]` ist bei Funktionsparametern *genau dasselbe* wie `*`
- `[]` ist *schöner zu lesen*, wird aber *sehr selten verwendet*
  `[]` *sagt: "Das soll ein Pointer auf ein Array sein, nicht auf einen Einzelwert"*

Beispiele:

```
void sort(double array[], int anzahl) { … }
void sort(double *array, int anzahl) { … }
int blankCnt(const char str[]) { … }
int blankCnt(const char *str) { … }
```

Arrays als Parameter - `const` (1)

Da Arrays *immer* "by Reference" übergeben werden:

- Die aufgerufene Funktion kann
  das Array im Aufrufer *immer* ***ändern***
  (auch wenn es gar nicht geändert werden soll!),

  außer es wird als Zeiger auf **`const`** *übergeben!*

- Man sieht einem Array-Parameter einer Funktion
  rein optisch *nicht* an,
  - ob er *nur gelesen* werden soll
    (Input- bzw. Value-Parameter)
  - oder *auch geschrieben/geändert* werden soll
    (Output-Parameter)!

Arrays als Parameter - **`const`** (2)

==> Bitte *nur-Input-Array-Parameter*
(die *gedanklich* ***"by Value"*** sein sollen)

mit* `const` *deklarieren!!!

- Zur **Sicherheit**:

  Der Compiler liefert *Fehlermeldungen*,
  wenn man das Array versehentlich doch ändert!

- Zur **Leserlichkeit**:

  Man *sieht* auf einen Blick:

  Dieses Array geht nur in die Funktion hinein!

Arrays als Parameter - Größe

Achtung:

Die aufgerufene Funktion
kann die Größe des Arrays
***nicht** feststellen!!!*

(sie *weiß nicht einmal,* ob der übergebene Zeiger
auf einen *einzelnen Wert* oder ein *Array* zeigt!)

Daher:

- Bei *Strings,* die *nur gelesen* werden:
  Die Länge erkennt man nur an der *Ende-Markierung* **`'\0'`** !
- Sonst (wenn die Größe *nicht fix* ist):

Die Größe als 2. Parameter getrennt übergeben!!!

Arrays als Parameter - **`sizeof`** ?

`sizeof` hilft ***nicht*** bei Array-Parametern!

Am Beispiel **`void sort(double array[], ... )`** :

- **`sizeof(array)`** liefert die Größe *des Zeigers selbst*

  (*nicht des Arrays*, auf das er zeigt!)

  ==> **4** bei 32 Bit Code, **8** bei 64 Bit Code

- **`sizeof(*array)`** liefert die Größe *des ersten Elementes*

  (*nicht* des gesamten Arrays!)

  ==> **8**, nämlich **`sizeof(double)`**

  (egal, ob **`array`** auf einen Einzelwert
  oder ein ganzes Array zeigt!)

Arrays als Parameter – Zugriff (1)

Zugriff auf Array-Parameter in der aufgerufenen Funktion:

"Ganz normal" mit **[ ]**,
als ob der Zeiger das Array selbst wäre!

```
int blankCnt(const char *str)
{
   int cnt = 0;
   // Schleife bis zur Ende-Markierung von str
   for (int i = 0; str[i] != '\0'; ++i) {
      if (str[i] == ' ') ++cnt;
   }
   return cnt;
}
```

Arrays als Parameter – Zugriff (2)

```
void sort(double array[], int anzahl)
{
  for (int pos = anzahl - 1; pos > 0; --pos) {
    for (int i = 0; i < pos; ++i) {
      if (array[i + 1] < array[i]) {
        double tmp = array[i + 1];
        array[i + 1] = array[i];
        array[i] = tmp;
      }
    }
  }
}
```

Arrays als Parameter – Aufruf (1)

Übergabe eines Arrays im Aufrufer:

- Ein Arrayname ohne **[]**
  ist *automatisch ein Zeiger* auf den Array-Anfang

  ==> einfach den ***Arraynamen*** übergeben

  ==> ***kein*** & *notwendig!*

```
double arr[100];
sort(arr, 100);
char line[82];
if (blankCnt(line) > 0) { …
```

Arrays als Parameter – Aufruf (2)

- Eine String-Konstante in **" "**
  ist automatisch ein Zeiger auf das erste Zeichen

  ==> Auch *kein* **&** notwendig, direkt übergeben!

  Aber: String-Konstanten dürfen *nicht geändert* werden!

  ==> Der Parameter ***muss*** **`const`** deklariert sein!
  (sonst kommt ein Compilerfehler!)

```
int blanks;
blanks = blankCnt("Das ist ein Test!");
```

Arrays als Parameter – Aufruf (3)

Tricks für Fortgeschrittene:

- Man kann auch ein "*Mittelstück*" eines Arrays (Pointer mitten in das Array) übergeben:

  `blanks = blankCnt(&(line[curPos]));`

  ==> **`blankCnt`** "sieht" **`line`** nur ab Element **`curPos`**!

- Man kann auch *eine einzelne Zeile* eines mehrdimensionalen Arrays *als eindimensionales Array* übergeben:

  `double matrix[100][100];`

  `sort(matrix[i], 100);` // sortiert die Zeile **`i`**

Arrays als Returnwert (1)

... sind aus 2 Gründen ***unmöglich***:

- Der Returnwert wird normalerweise
  vom Aufgerufenen in den Aufrufer *zurückkopiert*.

  Aber:

  Ein Array kann *nicht als Ganzes "zurückkopiert"* werden
  (aus der Funktion in ein Array des Aufrufers)

  ==> Daher *kein Array als Returnwert!*

Man müsste in der Funktion stattdessen
einen *Zeiger* auf das Ergebnis-Array
als Returnwert zurückgeben, aber...

Arrays als Returnwert (2)

- Ein *lokales Array* in der aufgerufenen Funktion *verschwindet, sobald die Funktion zurückkehrt:*

  Der Speicher wird beim `return` *freigegeben*!

  ==> Ein zurückgegebener Zeiger darauf

  zeigt nach dem `return` *"ins Leere"*!!!

  ==> Daher *einen Zeiger auf ein lokales Array*
  (oder auf andere lokale Daten)

  ***niemals* als Returnwert zurückgeben!**

  Achtung: Der Compiler meldet *keinen Fehler,*
  wenn man versehentlich einen Zeiger
  auf ein lokales Array als Returnwert zurückgibt!

Arrays als Returnwert (3)

Stattdessen (in 95 % der Fälle die sinnvollste Variante):

- Das Array für das Ergebnis *schon im Aufrufer anlegen!*
- … und beim Aufruf
  als zusätzlichen (meist ersten) Parameter übergeben

==> Die aufgerufene Funktion
speichert das *Ergebnis* gleich
direkt *im Array des Aufrufers*
(einem Parameter!),
statt es als Returnwert zurückzugeben!

Arrays als Returnwert (4)

Häufig:

Ein Zeiger auf das übergebene
Ergebnis-Array des Aufrufers wird
zusätzlich als Returnwert zurückgegeben

==> Praktisch für

ineinander geschachtelte Funktionsaufrufe
(vor allem bei String-Funktionen)!

Der *Returntyp* dafür ist immer ein *Pointer* (mit *).

Man kann beim Returntyp *nicht* **[]** schreiben,
auch wenn der Pointer auf ein Array zeigt!

Arrays als Returnwert (5)

Aber:

- Der Aufrufer muss vorab wissen,
  wie groß das Ergebnis maximal werden kann!
- Die aufgerufene Funktion kann *nicht von selbst feststellen, wieviel Platz* im Array für das Ergebnis ist!

==> Wenn die Größe nicht fix bzw. bekannt ist:

- Der Aufrufer muss
  die ***Größe*** *des Ergebnis-Arrays als 2. Parameter* ***übergeben****!*
- Der Aufgerufene muss ***prüfen***, ob sein Ergebnis hineinpasst!

Klassische Fehlerquelle (Details demnächst):
`strcpy` und **`strcat`** haben leider *keinen* Längen-Parameter
==> *Beide schreiben bei zu langem Input über das Ergebnis-Array hinaus!*

Arrays als Returnwert (6)

Beispiel:

```
// Liefert einen String bestehend aus len Mal Zeichen c
// im Array result (muss min. len+1 Zeichen Platz haben!)
char *makeLine(char result[], char c, int len)
{
   for (int i = 0; i < len; ++i) {
      result[i] = c;
   }
   result[len] = '\0';  // Endemarkierung nicht vergessen!
   return result;       // result zusätzlich als Returnwert
}

char zeile[81];
printf("%d: %s\n", i,
        makeLine(zeile, '=', anzahl[i]));
```

Arrays als Returnwert (7)

Alternative Möglichkeiten (selten, meist historisch):

- Die Funktion liefert als Returnwert

  einen *Pointer* auf ein
  *lokales **statisches** Array* mit dem Ergebnis

  (weil statische Arrays nach dem **`return`** *erhalten* bleiben).

  Kommt bei einigen vordefinierten Funktionen vor!

- Die Funktion speichert das Ergebnis

  in einem ***globalen Array*.**

  ==> Manchmal sinnvoll, meistens unschön
  (weil schwer zu lesen / durchschauen)...

Arrays als Returnwert (8)

Achtung bei beiden Varianten:

- *Der nächste Aufruf*

***überschreibt** das vorige Ergebnis!*

- Derartige Lösungen sind

nicht threadsafe

(nicht für *parallele* Programme geeignet)!

==> *Schlechte Idee, heimtückische **Fehlerquelle!***

==> Don't do that!

Arrays als Returnwert (9)

Moderne Alternative (in C noch *sehr selten*):

- Die Funktion liefert als Returnwert einen Pointer auf ein ***dynamisch angelegtes Array*** mit dem Ergebnis.
  (siehe später)

  ==> Kein Längenproblem!
  (das Array wird vom Aufgerufenen selbst *genau so groß wie für das Ergebnis nötig* angelegt)

  ==> Die schönste und sauberste Lösung!
  ... solange man das *Freigeben* im Aufrufer nicht vergisst:

  Gefahr eines "Memory Leak"!

  ==> ... aber die ***langsamste***!
  (dynamisches Anlegen und Freigeben ist sehr *aufwändig*!)

Mehrdim. Arrays als Parameter

Es gibt *3 Möglichkeiten*
zur Implementierung mehrdimensionaler Arrays:

- ***Echte*** mehrdimensionale Arrays
- ***"Selbst gerechnete"*** mehrdimensionale Arrays

  (Umrechnung von Zeile und Spalte
  auf einen Index in einem *eindimensionalen* Array)
- Ein ***eindimensionales Arrays von Pointern***

  … auf ***mehrere eindimensionale Arrays mit den Daten***

  (für jede Zeile ein eigenes Array)

Echte mehrdimensionale Arrays (1)

- Deklaration z.B.:

```
int brett[8][8];
```

- Übergabe als Parameter:

```
int func(int array[8][8]) { … }

result = func(brett);
```

==> Deklaration des Parameters mit **[]** statt mit *

(es wird trotzdem nur ein *Pointer* übergeben!)

==> Im Unterschied zu eindimensionalen Arrays

müssen die **[]** beim Parameter die ***Größe enthalten!!!***

(nur die *vordersten* **[]** dürfen leer bleiben)

Echte mehrdimensionale Arrays (2)

Unterschied C / C++:

- In C sind ***variabel*** große Arrays seit C99 erlaubt.

  Für Fortgeschrittene: Die variable Größe *muss* ***vor*** *dem Array als eigener Parameter übergeben* werden und wird dann in den `[]` angegeben:

```
int func(int size, int array[size][size])
{ … }

int matrix[n][n];
result = func(n, matrix);
```

  (kommt in der Praxis kaum vor, *einige Compiler sind zu dumm dafür*!)

Echte mehrdimensionale Arrays (3)

- In C++ sind *variabel große Arrays* ***generell verboten***.

  Mehrdimensionale Arrays können daher auch *nicht* direkt als Parameter übergeben werden.

  In den **[ ]** müssen immer **Konstanten** stehen, sowohl bei normalen Deklarationen als auch bei der Deklaration von Parametern.

  Für Fortgeschrittene:

  Auch mit **new** kann man nur Arrays erzeugen, die in *einer* (der *vordersten*) Dimension *variabel* groß sind.

  Die *anderen Dimensionen* müssen auch bei dynamisch angelegten Arrays *fixe Größe* haben.

Mehrdim. Arrays selbst gemacht (1)

Vor allem bei variabel großen mehrdim. Arrays wird daher oft

- stattdessen ein *eindimensionales Array* angelegt:

  ```
  int brett[8 * 8];
  ```

- und die *Indexrechnung überall selbst gemacht*:

  … `brett[zeile * 8 + spalte]` …

Dieses eindimensionale Array kann dann *ganz normal als Parameter an Funktionen übergeben* werden.

Mehrdim. Arrays selbst gemacht (2)

- In C funktioniert das auch mit variabler Größe.
- In C++ braucht man bei variabler Größe zumindest laut Standard ein **`new`** :

  ```
  int *matrix = new int[n * n];
  ```

  (Details im 2. Semester)

  (**`gcc`** und andere moderne C++-Compiler unterstützen *variabel* große Arrays in C++ so wie in C)
- In beiden Fällen wird man der Funktion die

  Zeilen- und Spaltenanzahl als separate Parameter

  übergeben, damit sie den Index ausrechnen kann!

Arrays von Pointern (1)

- In C / C++ macht man statt
  einem zweidim. Array mit **`n`** Zeilen und **`m`** Spalten oft

  `n` *eindim. Arrays mit* **`m`** *Werten* (pro *Zeile* eines)

  und

  ein eindim. Array mit **`n`** *Pointern auf diese Arrays*

  Die Zeilen-Arrays können *auch verschieden lang* sein!
- Beispiel: **`argv`**
  - Je ein eigenes **`char`**-Array pro Befehlszeilen-Wort.
  - Ein **`char *`**-Array mit Zeigern
    auf diese **`char`**-Arrays.

Arrays von Pointern (2)

- Der Zeiger auf den Anfang des Zeiger-Arrays ist ein **Zeiger auf Zeiger**.
- Der Parameter kann mit **`**`** oder **`*[]`** deklariert werden (*nicht* mit **`[][]`** !!!):

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

- *Zugriff:* ***Ganz normal*** mit **`[i][j]`**
  (funktioniert, obwohl kein zweidim. Array da ist)
  - **`argv[i]`** liefert den *Zeiger* auf das **`i`**-te **`char`**-Array
  - **`argv[i][j]`** liefert den **`j`**-ten Buchstaben aus dem Array, auf das dieser Zeiger zeigt

Vergleich von Pointern

- Vergleich **`p1 == p2`** : Sind die Pointer *gleich?*
  D.h.: Zeigen sie auf dieselbe Variable / Speicherstelle?

  `p == array` : Zeigt **`p`** auf den *Anfang* von **`array`** ?

  analog: Vergleich mit **`!=`** : Sind die Pointer *ungleich*?
- Vergleich **`p1 < p2`** bzw. **`p1 <= p2`** :
  Zeigt der Pointer **`p1`** *"vor" den Pointer* **`p2`** *?*

  Nur sinnvoll, wenn **`p1`** und **`p2`**
  auf *Elemente desselben Arrays* zeigen:

  Zeigt **`p1`** *auf ein Element mit kleinerem Index als* **`p2`** *?*

  analog: Vergleich mit **`>`** oder **`>=`**

Rechnen mit Pointern (1)

... ist nur sinnvoll,
wenn die Pointer auf ein *Array* zeigen!

... ist *gefährlich*:

C merkt nicht, wenn ein Pointer
über das Ende des Arrays hinausläuft!

==> Zugriff auf falsche Daten, Absturz!

... ist in C üblich und *weit verbreitet*
(vor allem bei Strings, d.h. **char**-Arrays)

... ist etwas *schneller* als der Zugriff mittels Index
(aber nicht viel!)

Rechnen mit Pointern (2)

... rechnet mit *Einheit Array-Elemente*, nicht mit Bytes!!!
(egal, wie groß die Elemente sind!)

... geht nur mit "echten" Pointern, *nicht* mit **`void *`**
(weil die Element-Größe bei **`void *`** unbekannt ist)

`p + 1` ... Pointer auf das Element *nach* dem Element,
auf das **`p`** gerade zeigt (das "nächste" Element)

`p + i` ... analog: Pointer auf das **`i`**-*te Element hinter* **`p`**

`p - 1` ... Pointer auf das Element *vor* dem Element,
auf das **`p`** gerade zeigt (das "vorige" Element)

Beispiel für **`double *p`** :

`p + 1` rechnet intern **`8`** (die Größe eines **`double`**) zu **`p`** dazu!

++ und −− auf Pointern

++p … *“lass* **p** *auf das nächste Element des Arrays zeigen”*

−−p … *“lass* **p** *auf das vorige Element des Arrays zeigen”*

Achtung:

***(p++)** … “Gib mir zuerst das *alte* Element,
lass dann **p** auf das nächste Element zeigen!”

***(++p)** … “Lass **p** zuerst auf das nächste Element zeigen,
gib mir dann das *neue* Element!”

++(*p) … “Zähl eins zu dem *Wert* dazu, auf den **p** zeigt
(und lass **p** *unverändert*)!”

(*p)++ … analog

Subtraktion zweier Pointer

`p1 - p2`

= *Abstand* zwischen **p1** und **p2** (als Zahl!)

= *Anzahl der Elemente* zwischen **p1** und **p2**
(wenn **p1** *hinter* **p2** zeigt, sonst *negatives* Ergebnis!)

Andere Rechen-Operationen als

- Pointer **+** oder **−** ganze Zahl (liefert einen *Pointer!*)
- Pointer **−** Pointer (liefert eine *Zahl!*)

sind auf Pointern *nicht* erlaubt:

- *kein* Pointer **+** Pointer
- *kein* ***** oder **/**

Schleifen und Pointer-Rechnen (1)

Pointer-Rechnung ist am häufigsten in

Schleifen über Arrays!

- **`p + i`** entspricht **`&(p[i])`** :
  Adresse des **`i`**-ten Elementes
- **`*(p + i)`** entspricht **`p[i]`** :
  Wert des **`i`**-ten Elementes

==> Statt **`p`** fix auf den Array-Anfang zeigen zu lassen
und zu **`i`** jedesmal eins dazuzuzählen

... kann man auch **`p`** selbst ***jedesmal eins weiterschieben***
und **`i`** ganz *weglassen*!

Schleifen und Pointer-Rechnen (2)

Beispiel: *"Lass* **p** *durch das Array* **array** *wandern"*:

```
double array[size], *p;
for (p = array; p < array + size; ++p) {
    printf("%g ", *p);
}
```

Weil:

p zeigt im **i**-ten Durchlauf auf das **i**-te Element des Arrays

==> ***p** ist dasselbe wie **array[i]**

array + size ist dasselbe wie **&(array[size])**

= erste Adresse *hinter* dem Array **array**

==> Wenn **p** dort hinzeigt, ist die Schleife *fertig*

Schleifen und Pointer-Rechnen (3)

Sei **`array`** wie vorher **`double array[size]`** :

Das Array **`array`** selbst
ist eine *Pointer-Konstante!*

==> Man darf **`array`** selbst *nicht* ändern
(nicht woanders hinzeigen lassen)

==> **`array++`** oder **`array = ...`** usw. ist *verboten*!

==> Man muss sich zusätzlich einen Pointer deklarieren!

Schleifen und Pointer-Rechnen (4)

Für Fortgeschrittene:

Wenn man ein *mehrdimensionales* Array hat:

> **`brett[i]`** (mit nur *einer* **`[]`**) liefert einen *Pointer* auf das erste Element der **`i`**-ten *Zeile* des Arrays!

Beispiel: "Gib alle Spalten der **`i`***-ten Zeile von* **`brett`** *aus"*

```
int brett[8][8], *p;

for (p = brett[i]; p < brett[i + 1]; ++p) {
    printf("%d ", *p);
}
```

Strings

- Grundlegendes
- String-Konstanten und Initialisierung
- Arbeiten mit Strings
- Was ist "`char *`" ?
- Vordefinierte String-Funktionen
  `strcpy`, `strcat`, `strcmp`, `strlen`, `strchr`, `strstr`
  Beispiele, `is...`, `to...`
- Strings in C++

Strings in C

Strings (Texte, "Zeichenketten")
sind ganz *normale Arrays* von **`char`** .

==> Man kann mit **`[ ]`**
auf *einzelne Zeichen* zugreifen.

Problem:

- Texte sind *nicht immer gleich lang*!
  ==> Im Array muss die *maximal nötige Länge* Platz haben!
- Aber wo / wie speichert man dann
  die Anzahl der "*tatsächlich gültigen*" Zeichen?

  Wie erkennt man reservierte, aber ungenutzte Zeichen?

Die Ende-Markierung (1)

Daher:

Das Textende muss immer durch '\0' markiert sein!

'\0'

- = Ein Byte mit dem *ASCII-Wert 0*, *nicht* die Ziffer **'0'**!
- heißt ***"Ende-Markierung"***
- *Muss in jedem String unmittelbar hinter dem letzten "gültigen" Zeichen gespeichert sein!*

Bei jedem *zeichenweisen Speichern* eines *neuen Strings*:

Nicht vergessen: '\0' anhängen!!!

Die Ende-Markierung (2)

Die Länge eines Strings wird
nur durch die Ende-Markierung
festgelegt!

- C speichert bei Strings *keine Länge als Zahl* mit:

  Weder die Arraygröße noch die tatsächliche Textlänge!

- Wenn C die tatsächliche Länge des Textes wissen will, *zählt es die Zeichen* bis zum **`'\0'`**
- Die String-Länge ist *durch die Array-Größe beschränkt*
  (und die max. Array-Größe nur durch den Speicherplatz)
- C kann die *Array-Größe zur Laufzeit nicht feststellen*

  … und daher ***nicht*** automatisch auf zu langen Text ***prüfen***!

Die Länge von Strings (1)

Die *Größe des Arrays* in der Deklaration muss daher

- die ***maximale Anzahl*** *der zu speichernden Zeichen*
- … eventuell plus **1** für **`'\n'`**
- … ***plus*** **1** *für* **`'\0'`** sein!

==> Der *Leerstring* **`""`** braucht *auch 1 Zeichen Platz*
(nämlich für das **`'\0'`** !)

Beispiel: **`char line[82];`**

`line` hat Platz für 81 Zeichen (plus **`'\0'`**)
oder 80 Zeichen plus **`'\n'`** (plus **`'\0'`**).

Die Länge von Strings (2)

Unterscheide: Für das **`line`** von der vorigen Folie:

- **`sizeof(line)`** liefert **`82`**

  *Die reservierte **Speichergröße** des* Arrays *laut Deklaration*

  Das ist eine *Konstante* zur *Compile-Zeit*:

  `sizeof` schaut den Inhalt von **`line`** *nicht* an
  ==> ist ***unabhängig** vom Text* in **`line`** !

  `sizeof` funktioniert aber *nur,*

  wenn **`line`** *das Array selbst ist,*
  nicht bei einem ***Pointer*** auf das Array (Array-***Parameter***)!

Die Länge von Strings (3)

- **`strlen(line)`** liefert je nach Text **`0`** bis **`81`**

  *Die tatsächliche Länge des **`Textes`***
  (*nicht* des Arrays!) ***ohne*** **`'\0'`**.

  `strlen` wird zur Laufzeit berechnet

  ==> ist *vom aktuellen Inhalt* von **`line`** *abhängig!*

  ==> *kostet Laufzeit* (je nach tatsächlicher String-Länge)!

  `strlen` funktioniert auch

  mit einem *Pointer* auf den String

String-Konstanten

- ... stehen in **" "**.
- ... haben Typ "**const char []**"
  (bzw. **const char ***)

  ==> sind *nicht änderbar*!

Bei einer String-Konstante über *mehrere Zeilen Programmcode:*
Einfach *mehrere* " ... " *hintereinander* schreiben:

```
printf("Das ist "
       "eine ganz lange "
       "Stringkonstante\n");
```

(Zeilenumbruch innerhalb von **" "** geht *nicht*!)

String-Initialisierung

Deklaration mit Init-Wert und ohne Länge:

```
char txt[] = "Fehler!";
```

txt wird *8 Zeichen* groß gemacht,
'\0' wird *automatisch angehängt*:

txt[0] … **'F'** **txt[1]** … **'e'** usw. bis **txt[6]** … **'!'**
txt[7] … **'\0'**

Bei angegebener Länge:

```
char line[82] = "\n";
```

line hat Platz für **81** Zeichen + **'\0'**

Die ersten beiden Zeichen werden am Anfang
auf **'\n'** und **'\0'** gesetzt (der Rest ist undefiniert).

Unterscheide… (1)

Typ “`char`”:

= ein *einzelnes Zeichen.*

(ohne Ende-Markierung)

Konstanten mit *einfachen* `' '`

Normal (wie einen **`int`**)
zuweisen (**`=`**),
vergleichen (**`<`**, **`==`**), …

Typ “`char [...]`”
oder “`char *`”:

= *String* (Text, Zeichenkette)
= *0 oder mehrere Zeichen*
plus Ende-Markierung **`'\0'`**

Konstanten mit *doppelten* `" "`

Zuweisen, vergleichen usw.
nur mit Stringfunktionen
(**`strcpy`**, **`strcmp`**, …).

Unterscheide… (2)

- **`'a'`** und **`"a"`** sind *ganz verschiedene Dinge*,
  auch wenn nur genau ein Zeichen enthalten ist!
- Man kann sie *nicht mischen*!

  ==> Man kann einen *einzelnen* **`char`** *nicht* direkt
  - mit einem ganzen String vergleichen oder
  - auf einen ganzen String zuweisen oder umgekehrt.
- **Aber**: *Ein Zeichen* eines Strings ist ein *normaler* **`char`** !

  Daher funktioniert

```
line[i] = '\n';
if (argv[1][0] == '-') …
```

Weil Strings Arrays sind ...

- ... hat das erste Zeichen **Index 0** (nicht 1).
- ... kann man sie

  nicht als Ganzes zuweisen, vergleichen, ...

  (aber **`printf`**, **`scanf`**, ... bzw. C++ I/O "**<<**" und "**>>**" funktionieren auf Strings als Ganzes).

==> Entweder selbst

zeichenweise (mit Schleife) verarbeiten

==> oder

String-Funktionen verwenden!

Verarbeitung von Strings (1)

Beispiele: (für `char line[82];`)

Falsch: `line = "";`
Richtig: `line[0] = '\0';`
Oder: `strcpy(line, "");`

Falsch: `line = line + ", ...\n";`
Richtig: `strcat(line, ", ...\n");`

Falsch: `if (line == "Ende") { … }`
Richtig:
```
if (strcmp(line, "Ende") == 0) {
    … }
```

(Details zu Stringfunktionen kommen gleich...)

Verarbeitung von Strings (2)

Die Zeichen eines Strings werden
nur bis zum `'\0'` gelesen / geschrieben!

==> Typische Schleifen über alle Zeichen eines Strings:

```
for (int i = 0; line[i] != '\0'; ++i) {...}
for (char *p = line; *p != '\0'; ++p) {...}
```

==> Der Inhalt der Array-Elemente
nach dem **`'\0'`** bis zum Array-Ende

ist *zufällig / egal / undefiniert!*

Verarbeitung von Strings (3)

Für Fortgeschrittene:

In der Praxis wird das **`!= '\0'`** in der Schleifenbedinung

meist weggelassen!

Weil: **`'\0'`** hat Wert **`0`** und entspricht automatisch **`false`**
Jedes andere Zeichen entspricht **`true`**

Beispiele:

```
for (int i = 0; line[i]; ++i) { … }
```

```
for (p = str1, q = str2; (*p = *q); ++p, ++q) {}
```

Achtung: Die Bedingung in der zweiten Schleife ist eine **Zuweisung** (zum besseren Lesen **`( )`** empfohlen!), **kein Vergleich**:

Nach dem Speichern wird das kopierte Zeichen implizit auf **`'\0'`** *geprüft!*

Und was ist "**char ***" ?

"Zeiger auf **char**"

Verhält sich (fast) genauso wie ein String bzw. Text:

- Ein **char *** ist ein *Zeiger* auf einen String,
  der schon *woanders gespeichert* ist.
- Ein **char *** enthält *nicht die Buchstaben* selbst.

  (ist also *kein Array*
  und reserviert *keinen eigenen Platz* für **char**'s!).
- Ein **char *** sagt nur,
  wo das Array mit den Buchstaben im Speicher liegt.

  (das kann ein *anderswo deklariertes Array*
  oder eine *String-Konstante* sein).

[ ] versus * (1)

Unterscheide z.B.:

```
char txtArr[] = "Ok!";
```

==> Ein Array mit 4 Bytes, Name **txtArr**,
am Anfang initialisiert mit

```
txtArr[0] = 'O'   txtArr[1] = 'k'
txtArr[2] = '!'   txtArr[3] = '\0'
```

```
const char *txtPtr = "Ok!";
```

==> Ein *konstantes* Array mit 4 Bytes, *ohne* Name,
Inhalt fix wie oben...
==> und *zusätzlich* ein Zeiger **txtPtr**,
der anfänglich auf dieses Array zeigt.

[ ] versus ✻ (2)

Funktioniert fast gleich,
aber schaut im Speicher anders aus!

- Bei **`txtArr`** kann man die *Buchstaben*
  (= den Inhalt des Arrays) *beliebig ändern*,

  aber **`txtArr`** ist *immer derselbe Speicherplatz*.

- Bei **`txtPtr`** kann man die *Buchstaben **nicht** ändern*
  (weil **`txtPtr`** auf eine *Text-Konstante* zeigt),

  aber man kann den Zeiger **`txtPtr`** selbst ändern
  und *auf eine andere Speicherstelle*
  mit anderem Inhalt *zeigen lassen*.

[ ] versus * (3)

Ebenso:

```
char mon[][4] = { "Jan", "Feb", "Mar", ... , "Dez"};
```

==> Ein einziges, *zweidimensionales Array* **mon**:
12 Zeilen mal 4 Zeichen
initialisiert mit den angegebenen Texten.

```
const char *mon[] =
      { "Jan", "Feb", "Mar", ... , "Dez"};
```

==> *13 Dinge* im Speicher:
12 einzelne Textkonstanten
plus ein eindimensionales *Array* **mon**,
das *12 Zeiger* auf diese Konstanten enthält.

argv

`const char *argv[]`

- **`argv`** … Name des Arrays
- **`[]`** … **`argv`** ist ein Array,
  aber wir wissen die Größe nicht.
- **`const char *`** … Typ der Elemente des Arrays:
  Zeiger auf Texte (**`char *`**),
  die wir nicht ändern dürfen (**`const`**):
  Jeder Zeiger zeigt auf 1 Wort der Befehlszeile

(Für Fortgeschrittene:
"**`const char **argv`**" funktioniert auch und ist dasselbe)

Wann brauche ich was?

- Wenn die Buchstaben selbst *schon wo gespeichert sind* (z.B. im Aufrufer, in einer Konstante, ...)

  ==> Dann brauche ich **nur einen Pointer** darauf, *kein* neues Array!

- Wenn die Buchstaben *frisch gespeichert* werden sollen (z.B. durch *Einlesen*, *Kopieren*, ...)

  ==> Dann brauche ich **Platz für die Zeichen selbst**, nicht nur einen Pointer!

  ==> ... und zwar entweder ein **Array** für die Zeichen oder einen **Pointer** auf **dyn. Speicher** (**`malloc`**, **`new`**)

Achtung bei String-Pointern

- Wenn man einen Pointer statt einem Array als *Zielstring* (beim Kopieren, Einlesen, ...) angibt:

  Prüfen:

  Zeigt der Pointer schon auf ein Array,
  wo *Platz zum Hinspeichern* der Zeichen ist?

  ==> *Zuerst Pointer initialisieren, dann erst hinkopieren!*

- Wenn *mehrere Pointer auf dasselbe Array* zeigen:

  Änderungen der Zeichen über einen Pointer
  (oder direkt durch Zuweisung auf die Array-Elemente)
  wirken sich auch auf die Zeichen aus,
  die alle anderen Pointer "sehen"!

Vordefinierte **`str`**...-Funktionen

- ... kommen aus **`string.h`** .
- ... arbeiten auf **`\0`**-terminierten Strings
  (**`char`**-Arrays oder Pointer auf **`char`**-Arrays)
- ... sind normalerweise *schneller* als ein Eigenbau!

  (und *viel cleverer* programmiert
  als hier nur als Beispiel gezeigt!)
- ... gibt es noch einige mehr als hier beschrieben.
- Können sollte man:

  `strcpy`, **`strcat`**, **`strcmp`**, **`strlen`**, **`strchr`**, **`strstr`**

  (andere nur zur Info, bei Bedarf nachschauen...)

String-Funktionen: Gefahr!!! (1)

String-Funktionen sind **unsicher**!

- Sie *wissen nicht, wie groß die Arrays sind!*
- Sie arbeiten *ohne Längenprüfung*
  immer bis zum **`'\0'`** !!!

  (Ausnahme: **`strncpy`**, **`strncat`**)

==> Gefahr des *Überlaufes*
(Lesen / Schreiben hinter dem Array-Ende)

- wenn das Ergebnis *länger*
  als das dafür vorgesehene Array ist,
- oder wenn das **`'\0'`**
  am Ende eines Input-Strings überhaupt *fehlt*!

String-Funktionen: Gefahr!!! (2)

==> *Fremde Daten hinter dem Array*
werden gelesen / überschrieben!
==> *Falsche Daten, Absturz! (und Sicherheits-Problem!)*

Beispiel: `strcpy`

- Quelltext mit 100 Zeichen,
  Ziel-Array mit Platz für 80 Zeichen

  ==> **`strcpy`** *überschreibt 20 Bytes hinter dem Ziel-Array!*

- **`'\0'`** im Quelltext *fehlt*

  ==> **`strcpy`** kopiert so lange Zeichen in das Ziel-Array,
  bis zufällig das erste **`'\0'`**
  im Speicher *hinter* dem Quelltext kommt!

Strings kopieren: **`strcpy`**

`char *strcpy(char *ziel, const char *quelle)`

- *Kopiert* **`quelle`** nach **`ziel`** (*"von rechts nach links"*)
- Ergebnis: Pointer auf **`ziel`**
  ==> geschachtelt verwendbar!
- *Achtung: Keine Längenprüfung:*

  `ziel` kürzer als **`quelle`** ==> **Absturz!**
- **`quelle`** kann bestehender String, Konstante, ... sein.
- **`ziel`** muss ein *Array* (oder ein Zeiger auf ein Array) sein, in das man die Buchstaben von **`quelle`** kopieren kann (wo freier *Platz* vorhanden ist)!

Beispiel: `strcpy`

```
char *strcpy(char *dest, const char *src)
{
  // Achtung: '\0' muss noch mitkopiert werden!
  for (int i = 0;
       (dest[i] = src[i]) != '\0';
       ++i) { }
  return dest;
}
```

```
char *strcpy(char *dest, const char *src)
{
  char *result = dest;  // dest als Returnwert aufheben!
  for (; (*dest = *src) != '\0'; ++dest, ++src) { }
  return result;
}
```

Strings kopieren: **`strncpy`** (1)

```
char *strncpy(char *ziel, const char *quelle,
              size_t n)
```

- Wie **`strcpy`**, aber es werden

  max. **n** *Zeichen kopiert*

- **`size_t`** ist irgendein **`int`**-Typ
  (auf den meisten Systemen **`unsigned long`**)

Strings kopieren: **`strncpy`** (2)

Achtung!

`strncpy` schafft fast so viele neue *Probleme*,
wie es lösen sollte!

- Hat **`quelle`** **`n`** oder mehr Zeichen,
  wird **`ziel`** *nicht* **`'\0'`**-terminiert!

  ==> Immer **`'\0'`** *händisch anhängen*,
  sonst Absturz beim nächsten Lesen!

- *Ineffizient*, wenn **`quelle`** viel kürzer als **`n`** ist:

  `ziel` wird *komplett* mit **`'\0'`** auf Länge **`n`** *aufgefüllt*
  ==> Hoher und unnötiger Aufwand!

Besser, aber *nicht* standardisiert: **`strlcpy`**

Strings anhängen: **`strcat`**

`char *strcat(char *ziel, const char *quelle)`

- Wie **`strcpy`**, aber **`quelle`** wird (ohne Zwischenraum!) an den bestehenden Inhalt von **`ziel`** ***angehängt***.
- **`ziel`** muss daher bereits *vorher einen String* (oder zumindest ein **`'\0'`**) enthalten!
- *Achtung:*
  Ebenfalls *Absturz* wenn **`ziel`** *zu kurz* ist!

Strings anhängen: **`strncat`**

`char *strncat(char *ziel, const char *quelle, size_t n)`

- Wie oben, aber es werden maximal **`n`** Zeichen von **`quelle`** an **`ziel`** angehängt.
- *Achtung:*

  `ziel` wird *immer* **`\0`**-terminiert
  ==> es müssen noch mindestens **`n+1`** Zeichen in **`ziel`** *Platz* haben!!!

  `n+1` ist der *noch freie Platz* in **`ziel`**, ***nicht*** die *Gesamtlänge* von **`ziel`**

- Auch hier: **`strlcat`** ist *besser*, aber *nicht* Standard

Strings vergleichen: **strcmp**

int strcmp(const char *s1, const char *s2)

- Vergleicht **s1** und **s2** nach *ASCII-Zeichensatz*

  ==› **A-Z** vor **a-z**
  ==› **Umlaute und ß** weit *hinter* **z**
  ==› Sonderzeichen weit verstreut

- Ergebnis:
  Das Ergebnis von **strcmp** *wird immer mit* **0** *verglichen:*

strcmp(s1, s2) == 0	wenn	**s1 == s2**
strcmp(s1, s2) < 0	wenn	**s1 < s2**
strcmp(s1, s2) > 0	wenn	**s1 > s2**

Beispiel: `strcmp`

```
int strcmp(const char *s1, const char *s2)
{
  for (int i = 0; ; ++i) {
    if (s1[i] != s2[i]) return s1[i] - s2[i];
    // ab hier ist s1[i] gleich s2[i],
    // also entweder beide am Ende oder beide nicht am Ende
    if (s1[i] == '\0') return 0;
  }
}
```

```
int strcmp(const char *s1, const char *s2)
{
  for ( ; ; ++s1, ++s2) {
    if (*s1 != *s2) return *s1 - *s2;
    if (*s1 == '\0') return 0;
  }
}
```

Strings vergleichen: **`str...cmp`**

```
int strncmp(const char *s1, const char *s2,
            size_t n)
```

Vergleicht nur die ersten maximal **`n`** Zeichen.

```
int strcasecmp(const char *s1,
               const char *s2)
```

(und analog **`strncasecmp`**)

Ignoriert beim Vergleich Groß- und Kleinschreibung.

Achtung: Nicht im C-Standard, nur im POSIX-Standard.

```
int strcoll(const char *s1, const char *s2)
```

Vergleicht (hoffentlich) entsprechend *Sortier-Reihenfolge gemäß Ländereinstellungen*.

Stringlänge: **`strlen`**

`size_t strlen(const char *str)`

- Ergebnis (**`size_t`** ist wie **`int`**):

  Die *Länge* von **`str`** (Anzahl der Zeichen)

  ohne das **`'\0'`**.

- *Achtung:*

  Wenn **`strlen(str)`** als Ergebnis **`len`** liefert,
  braucht **`str`** im Speicher **`len+1`** Zeichen Platz!

  (das **`'\0'`** braucht auch Platz!)

Beispiel: `strlen`

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

```
int strlen(const char *str)
{
  const char *p;
  for (p = str; *p; ++p) { }
  return p - str;  // Wie weit ist p vom str-Anfang weg?
}
```

Zeichen suchen: **`strchr`**, **`strrchr`**

`char *strchr(const char *str, char c)`

- *Sucht* das ***einzelne Zeichen*** **`c`** in **`str`** .
- Ergebnis:
  - Zeiger auf das *erste* Vorkommen von **`c`** in **`str`**.
  - **`NULL`** wenn **`c`** in **`str`** *nicht* vorkommt
    (**`NULL`** ist der "nirgendwohin-Zeiger").
- Es ist erlaubt, das **`'\0'`** zu suchen
  (das liefert einen *Zeiger auf die Ende-Markierung*).

`char *strrchr(const char *s, int c)`

Analog, aber liefert Zeiger auf das *letzte* Vorkommen.

Beispiel: `strchr`

```
char *strchr(const char *str, int c)
{
  for (int i = 0; ; ++i) {
    if (str[i] == c) return (char *) &(str[i]);
    if (str[i] == '\0') return NULL;
  }           // Pfui: Diese Typkonvertierung im return ...
}             // ... entfernt das const aus dem Typ von str !

char *strchr(const char *str, int c)
{
  for ( ; ; ++str) {
    if (*str == c) return (char *) str;
    if (*str == '\0') return NULL;
  }
}
```

String suchen: **`strstr`**

```
char *strstr(const char *haystack,
             const char *needle)
```

- Analog, aber *sucht* einen ganzen ***String*** **`needle`** statt einem einzelnen Buchstaben:

  Liefert einen Zeiger auf das erste Vorkommen von **`needle`** in **`haystack`** oder **`NULL`**.

  (Wenn **`needle`** leer ist, wird es gleich ganz vorne gefunden.)

Beispiel: `str...` - Aufrufe (1)

```
char initWert[] = "blabla";
char meinText[40];

// kopiere "blabla" von initWert nach meinText
strcpy(meinText, initWert);

// hänge zweimal etwas an
strcat(strcat(meinText, " und mehr "),
       initWert);

// gib meinText und seine Länge aus
printf("\"%s\" ist %d Zeichen lang.\n",
       meinText, strlen(meinText));
```

Beispiel: **`str...`** - Aufrufe (2)

```
// beende das Programm,
// wenn meinText genau gleich "Ende" ist
if (strcmp(meinText, "Ende") == 0) {
    exit(0);
}

// beende das Programm,
// wenn in meinText irgendwo "Ende" enthält
if (strstr(meinText, "Ende") != NULL) {
    exit(0);
}
```

Zeichen testen: **`is`**...

- **`int isalpha(char c)`** usw.
- Aus **`ctype.h`** (*"character type"*), *nicht* **`string.h`**
- *Testet* ein ***einzelnes Zeichen*** **`c`**,
  Ergebnis **`0`** wenn **falsch**, **ungleich `0`** wenn **wahr**.
- Beispiel: **`if (isspace(c)) { ... }`**
- **`isalpha`** ... Ist **`c`** ein Buchstabe?
  `islower` ... Kleinbuchstabe?
  `isupper` ... Großbuchstabe?
  `isdigit` ... Ziffer?
  `isxdigit` ... Hex-Ziffer?
  `isalnum` ... Buchstabe oder Ziffer?
  `isspace` ... Zwischenraum, Tab, Zeilenvorschub, ...?
  ... und noch ein paar mehr!

Groß-/Klein-Umwandlung: **`to`**...

- **`char toupper(char c)`** und **`char tolower(char c)`**
- Aus **`ctype.h`**, *nicht* **`string.h`**
- *Testet und verwandelt* ein ***einzelnes Zeichen*** **`c`**.
- Returnwert (**`c`** selbst wird ***nicht*** verändert!):
  - Wenn **`c`** ein Buchstabe ist:
    `c` *großgeschrieben* (**`toupper`**)
    bzw. *kleingeschrieben* (**`tolower`**)
  - Sonst:
    `c` *unverändert*.
- Beispiel: **`txt[i] = tolower(txt[i]);`**

Strings in C++

- Auch in C++ können C-Strings (**`char`**-Arrays wie soeben besprochen) verwendet werden.
- Zusätzlich hat C++ einen *eigenen String-Typ*:

  `string`

  Er *funktioniert* intern *anders* und wird mit *anderen Funktionen* verarbeitet.

  Er ist *komfortabler*, aber *langsamer*.
- In der Praxis kommt *beides bunt gemischt* vor…

Zeichensätze

- ASCII und 8-Bit-Zeichensätze
- Unicode
- UTF-8
- UCS-2 und UTF-16
- Andere Unicode-Codierungen

Zeichensätze: ASCII

Umlaute ignorieren wir – bei uns ist alles nur ASCII!

ASCII (seit "ewig": 1963):

- Nur **7 Bit**: 0 … 127 (==> **`signed`** / **`unsigned`** *ist **egal**!*)
- *„**American** Standard Code for Information Interchange"*: ***Keine Umlaute*** oder nationalen Sonderzeichen!
- **0 … 31 & 127:** Steuerzeichen (z.B. **`\n`**, **`\t`**, …)
  32: Zwischenraum
  33 … 126: Sichtbare Zeichen

Zweiter historisch bedeutender Zeichensatz: **EBCDIC**
- Noch früher als ASCII, fast nur in der *IBM-Großrechner-Welt*
- Heute kaum noch verwendet

Achtung: Buchstaben haben in EBCDIC *keine fortlaufenden Code-Nummern*!

8-Bit-Zeichensätze (1)

*Aber: 1 Byte hat **8 Bits**,*
*die **Werte 128-255** sind bei ASCII **unbenutzt!***

==> *Erste Erweiterungen von ASCII:*

"Nationale" 8-Bit-Zeichensätze

- *"Historisch wild gewachsen",*
  ==> Je nach Land, Hersteller, usw. verschieden!
- Haben nur *max. 256 Zeichen!*

  0 ... 127 = *ident* zu ASCII
  128 ... 255 = *Sonderzeichen, Umlaute, grafische Symbole*, ...
  je nach Land und Hersteller

8-Bit-Zeichensätze (2)

Heute:

- *Hersteller*-spezifisch: Nur mehr ***Microsoft-Codepages***

  Westeuropa: **CP 850** (DOS, **cmd**) und **CP 1252** (Windows)

  Umlaute haben in CP 850 und CP 1252 *verschiedene* Codes
  ==> *In Windows getippter Text mit Umlauten*
  wird im DOS-Fenster falsch angezeigt und umgekehrt!

- *Standardisiert*: ***ISO 8859-x***

  Westeuropa: **ISO Latin 1 = ISO 8859-1**
  und **ISO Latin 9 = ISO 8859-15** (= ISO 8859-1 plus '€')

  ISO 8859-1 und CP 1252 sind bezüglich Umlauten *ident*
  (unterscheiden sich nur bei den Codes 128 ... 160:
  In ISO-8859 Steuerzeichen, in CP 1252 sichtbare Sonderzeichen)

8-Bit-Zeichensätze (3)

Vorteile:

- Typ **`char`**, 1 Zeichen = 1 **`char`** *(Platz-effizient!)*
- *"Normaler" Programm-Code wie bisher bei ASCII:*
  bekannte String-Funktionen, **`printf`**, ...

  (einzige Gefahr: Umlaute usw. sind bei **`signed char`** *negativ*!)

Nachteile:

- Gespeicherte Daten sind *nicht portabel*
  auf andere *Zeichensätze* oder in andere *Länder*

  ==› *Es werden "falsche" Zeichen angezeigt!*
- *Maximal 128* (bzw. 95) *zusätzliche* Zeichen, *kein Platz* mehr
- Manche sehr alte Protokolle (z.B. SMTP!) erlauben *nur 7-bit-Daten*

Unicode (1)

- Jedes Zeichen auf der Welt bekommt eine

  weltweit eindeutige Nummer

  Derzeit *über 150000 Codes* definiert, wächst, über 1 Mio. ($17 * 2^{16}$) Codes wären möglich:

  Chinesisch, arabisch, mathemat. Symbole, ...
  Auch Emojis, Musik-Noten, Ligaturen, ...

- *Standardisiert* seit 1993, *hersteller-unabhängig*:

  In ***ISO 10646*** und vom ***Unicode-Konsortium***

- Auch im Unicode: ***0 ... 127 ist ident zu ASCII !***

Unicode (2)

Problem:

Darstellung dieser Codes im Speicher
(haben *nicht in 1 Byte* Platz!)

2 wichtige Standards, konkurrierend:

- **UTF-8**
- UCS-2 bzw. **UTF-16**

==> *Derselbe Zeichen-Code* (z.B. '**€**' = $\mathbf{20AC}_{16}$ = $\mathbf{8364}_{10}$)

wird in UTF-8 und UTF-16
völlig unterschiedlich gespeichert!

UTF-8

Wo:

- Unix & *Linux*, Mac
- *Internet*: UTF-8 = **RFC 3629**
- XML

Wie:

- *Byteweise*, normaler Typ **`char`**
- Aber *variabel lange* Codierung:

  1 Zeichen kann *1 bis 4* **`char`**'s Platz belegen!

UTF-8, Codierung (1)

- 0 ... 127, 7 Bits: **1 Byte 0xxxxxxx**
  ==› Nur-ASCII-Dokumente sind in UTF-8 ident zu "altem" ASCII,
  altes *ASCII ist ohne Konvertierung gültiges UTF-8*
  ==› Alle "wichtigen" Zeichen sind *"platzsparend"* dargestellt
- Alle Codes ›=128 werden ***in mehreren Bytes*** gespeichert:
  - *Alle Bytes* der Darstellung sind ***›=128***
  - Das *erste Byte* legt die ***Länge*** fest:
  - 128 ... 2047 (z.B. Umlaute) = 11 Bits: **2 Bytes 110xxxxx 10xxxxxx**
  - 2048 ... 65535 = 16 Bits: **3 Bytes 1110xxxx 10xxxxxx 10xxxxxx**
  - 65536 ... 1114111 = 21 Bits
    (genug für den gesamten derzeitigen Unicode):
    4 Bytes 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
  - *Fortsetzbar* bis *8 Byte*, 42 Bit Nutzdaten (› 4 Billionen Zeichen)

UTF-8, Codierung (2)

- *Platzbedarf* im Vergleich zu UCS-2 / UTF-16:

  Normaler Text (incl. Umlauten) ist viel *effizienter*,
  Text mit vielen Codes >2048 ist etwas ineffizienter

- UTF-8 ist *byteweise richtig sortierbar*
  (nach aufsteigendem Unicode-Code)

- Der *Anfang* eines Multi-Byte-Zeichens ist
  auch *von hinten* / von mittendrin feststellbar:

  0xxxxxxx ... Single Byte
  10xxxxxx ... Multibyte Folgebyte
  11xxxxxx ... Multibyte Startbyte

  ==> UTF-8 ist auch *von hinten durchsuchbar*

UTF-8, Codierung (3)

Es gibt *ungültige Bytefolgen*:

- *Fehlende* Folgebytes,
  Folgebytes ohne Startbytes
- *Redundante* Codierungen
  (z.B. Codes bis 127 in 2, 3 oder 4 Bytes dargestellt)
- Startbytes für eine *5, 6, 7 oder 8-Byte-Folge*
- Vom Standard *verbotene Codes*, u.a. ***D800–DFFF***
  (verboten wegen ihrer Sonderbedeutung in der UTF-16-Codierung,
  kommen leider fälschlicherweise oft in UTF-8 vor,
  wenn UTF-16 “schlampig” (wie UCS-2) in UTF-8 umgewandelt wird!)

==> *Beim Anzeigen von UTF-8 werden solche Fehler ersetzt durch das Fehlerzeichen* ***FFFD***

UTF-8 versus 8-Bit-Codes

Anzeige mit *falschem* Zeichensatz:

- Als ***UTF-8 gespeichert***,
  als ***8 Bit Code*** (z.B. *ISO-8859)* ***angezeigt***:

  ==> Statt jedem *Umlaut* (= *2-Byte-Unicode-Sequenz*)
  werden *2 seltsame Sonderzeichen* angezeigt

- Als ***8 Bit Code*** (z.B. *ISO-8859)* ***gespeichert***,
  als ***UTF-8 angezeigt:***

  ==> *Jeder* Umlaut (und jedes andere Zeichen 128 ... 255 !)
  ist ein *ungültiger* UTF-8-Code
  und wird daher als *ein "Fehlerzeichen"* angezeigt

UTF-8: Programmierung

- Speicherung und Verarbeitung *"ganz normal"*:

  - *Stringende* in C ist nach wie vor ein **`\0`**-Byte
  - **`strcpy`**, **`strcat`** usw. funktionieren *wie bisher*

- **Änderungen** nur

  - wo **`str[i]`** das **`i`**-te *Zeichen* statt **`i`**-te *Byte* meint

  - wo es auf die *"sichtbare Länge"* ankommt
    (z.B. formatierte Anzeige, Zeichen zählen, ...)

  - bei *Deklarationen (mehr Platz reservieren!!!)*

  *Grund: Die "Länge in Bytes / **`char`**'s bis zum **`\0`**" ist ungleich der "Anzahl der sichtbaren Zeichen"!*

UCS-2

Wo früher:

- *Microsoft* Windows & Office
- *Java, .NET* & viele GUI-Toolkits (z.B. WxWidgets)

Wie:

- *Fix 2 Bytes* (16 Bit) pro Zeichen: *Vorteil fixe Länge!*

  ==> *Anzahl der Doppel-Bytes = Anzahl der Zeichen !*

- *Ende*-Markierung = *Doppel-Null-Byte*
  (denn ein UCS-2-String enthält *viele Einfach-Null-Bytes*!)

Kann nur Zeichen bis Unicode 65535 darstellen

==> *Alt, abgekündigt, ersetzt durch Nachfolger* ***UTF-16***

UTF-16

Codierung ***variabler*** *Länge*:

1 oder 2 Doppel-Bytes *pro Zeichen*

- Code bis 65535 ==> direkt gespeichert
- Code 65536 ... 1114111:
  Zuerst 65536 *abziehen* (==> Rest hat max. 20 Bits), dann

  - *erstes Doppelbyte*: **110110xxxxxxxxxx**
  - *zweites Doppelbyte*: **110111xxxxxxxxxx**

  (das sind genau die Bereiche **D800** bis **DBFF** und **DC00** bis **DFFF**, die im Unicode *ungültige Codes* sind)

UTF-16: Programmierung

- Der *Typ ändert sich* von **`char`** auf

  `wchar_t` *(= "wide character" = 16-Bit-***`char`***)*

  Problem von **`wchar_t`**: Größe ist im Standard *nicht* definiert!
  Kann auch 1 oder 4 Bytes sein (ist unter Unix/Linux meist 4 Bytes!)
  ==> besser **`char16_t`** (seit C11 und C++11) verwenden?!

- Da C *kein Overloading* kennt
  (gleicher Funktionsname für verschiedene Parameter-Typen ist verboten):

  Alle Funktionen müssen umbenannt werden:
  `fwprintf`, **`fgetwc`**, **`wcscpy`**, **`wcscmp`**, **`wcslen`**, ...
  (Header **`wchar.h`**)

- Plus analoge *Änderungen* wie bei UTF-8
  wegen *variabel langer Codierung* der Zeichen!
  (*"sichtbare Länge"* ungleich *"Anzahl der* **`wchar`***'s"*)

UTF-16, Nachteile (1)

- Viel *Platzverschwendung* (fast 50 %)
  bei “normalen” Texten (ASCII):

  Vorderes Byte fast aller Zeichen ist 0!

- *Völlig inkompatibel*
  mit ASCII-, ISO-8859- und UTF-8-Daten:

  Explizite Umwandlungsfunktionen nötig!

Vorteile: Keine???
(seit der Umstellung von UCS-2 auf UTF-16)

UTF-16, Nachteile (2)

Bei externer *Speicherung in einer Datei*,
Übertragung über das Internet, ...:

Endian-abhängig!

Wird das höherwertige Byte ***("big endian")***
oder das niederwertige Byte ***("little endian")***
eines **`wchar_t`***-Wertes zuerst gespeichert / gesendet?*

Abhilfe / Notlösung:

BOM = "Byte Order Mark"
= unsichtbares Zeichen mit Code FEFF
als *erstes Zeichen* speichern / übertragen!

UTF-16, Nachteile (3)

Weil: **FFFE** (**FEFF** *verkehrt*, d.h. Bytes *vertauscht*)
ist *illegaler* Code, darf normal nicht vorkommen

==> wenn **FFFE** als *erstes* Zeichen gelesen wird:
Datei oder Netzwerk-Bytefolge
wurde *"big endian"* geschrieben
und *"little Endian"* gelesen (oder umgekehrt)

==> *Alle Byte-Paare beim Lesen umdrehen!*

- UTF-16-Standard sagt *"Default: Big endian"*,
aber Intel, ARM, ... ist vom System her *immer "little endian"?!*

- Am Internet: Oft *kein* BOM,
sondern *explizite Charset-Angabe* **`utf-16le`** oder **`utf-16be`**

Andere Unicode-Codierungen

- ***Punycode:***

  Darstellung von beliebigen Unicode-Strings *nur* mit den ASCII-Zeichen **`a–z 0–9 –`**

  u.a. für *internationale Domain-Namen*

- ***UCS-4*** bzw. ***UTF-32***: *Fix 4 Bytes* pro Zeichen, enthalten den Unicode-Code als **`int`** (uncodiert)

  Logik ganz *einfach* (fixe Länge), aber noch mehr *Platzverschwendung*: *Fast 75 % Null-Bytes!*

- ***UTF-7***: *Variabel lange* Codierung in *7-Bit*-**`char`**'s

Wiederholung Pointer (1)

Pointer *ohne* * davor

==> Operation *auf dem Pointer selbst*
(*nicht* auf den Daten, auf die er zeigt)

==> Liefert bzw. speichert einen *Pointer-Wert*

Beispiele:

- Pointer *zuweisen*: **`ptr1 = ptr2;`**
- Pointer *vergleichen*: **`while (p < pEnd)`** ...
- Mit Pointer *rechnen*: **`++p;`**

  oder **`pEnd = p + size;`**

Wiederholung Pointer (2)

Pointer *mit* * davor

==> Operation auf den *Daten, auf die der Pointer zeigt*

==> Liefert / berechnet / speichert einen *Daten-Wert*
(**char**, **int**, **double**, ...)

Beispiele:

- Wert speichern, wo der Pointer hinzeigt: `*p = '\n';`
- Den Wert prüfen, auf den der Pointer zeigt:
  `while (*p != '\0')` ...
- Mit dem Wert rechnen, auf den der Pointer zeigt:
  `sum += *p;`

Wiederholung Pointer (3)

Ausnahme: Pointer *mit* * davor ***bei seiner Deklaration***

==> Hier wird der *Pointer* selbst initialisiert
(der * gehört hier *zum Typ*, nicht zur Pointer-Variablen)

==> Rechts vom = muss ein *Pointer-Wert* stehen!

Beispiele:

- ```
  double *p = arr;
  double *end = &(arr[size]);
  ```
- ```
  char *found = NULL;
  ```
- ```
  const char *text = "Hallo!";
  ```
  (eine String-Konstante ist technisch ein *Pointer* auf das erste Zeichen)
  ```

Wiederholung Pointer (4)

Wichtige gedankliche Prüfung
beim Umgang mit Pointern:

Auf *beiden* Seiten einer Zuweisung / eines Vergleiches

- Muss *entweder ein Pointer* stehen
- … *oder ein Datenwert* (**`char`**, **`int`**, **`double`**, …)

Gemischt
(links Pointer / rechts Daten
oder umgekehrt)
ist immer falsch!!!

(“Äpfel und Birnen darf man nicht mischen”)

Wiederholung Pointer (5)

Sonderfall: Pointer zeigt auf ein Array

==> Zugriff auf die Elemente auch mit **`[ ]`** möglich:

`p[0]` ist dasselbe wie **`*p`**

`p[i]` ist dasselbe wie **`*(p + i)`** :

`i`-tes Element des Arrays, auf das `p` zeigt

Beispiel:

```
if (isdigit(p[i])) ...
```

"Ist das **`i`***-te Zeichen in dem String,*
auf den **`p`** *zeigt, eine Ziffer?"*

Wiederholung Pointer (6)

Wie komme ich zu Pointern?

- Einen Pointer auf eine *"einfache" Variable* bekomme ich mit **&**:

  ```
  p = &i;
  ```

- Einen Pointer auf den *Anfang eines Arrays* bekomme ich *nur* mit dem Array-Namen (*ohne* **&** oder **[]** oder sonstwas):

  ```
  p = arr;
  ```

- Für einen Pointer auf ein *Array-Element "mittendrin"* gibt es zwei Möglichkeiten:

  ```
  p = &(arr[i]);
  ```
  oder
  ```
  p = arr + i;
  ```

Wiederholung Funktionen (1)

Schritte bei einer Funktion:

- *Kopfzeile* (Prototyp) gut überlegen:
  - *Sprechender Name* für die Funktion?
  - *Returntyp* der Funktion (wenn keiner: **`void`**):

    "Welchen Typ hat der Wert,
    den die Funktion als Ergebnis zurückliefern soll?"
  - *Argumente: Anzahl, Reihenfolge und Typ?*

    "Welche Daten braucht die Funktion vom Aufrufer?"
    (wenn keine: **`(void)`**) (siehe folgende Folien!)
- *Code* schreiben (passt das **`return`** zum Returntyp???)
- Richtigen *Aufruf* überlegen

Wiederholung Funktionen (2)

"Einfache" Werte als Parameter:

Wichtige Überlegung:

- Geht der Wert *nur in die Funktion hinein*?

  ==> "Call by Value":

  Sie bekommt eine **Kopie** des Wertes

- Oder soll die Funktion
  in dem Parameter auch etwas zurückliefern?
  (den *Wert im Aufrufer speichern / ändern können?*)

  ==> "Call by Reference":

  Sie bekommt einen **Pointer** auf's Original

Wiederholung Funktionen (3)

Call by Value:

- Parameter wird “ganz normal” deklariert:

  `int func(int val)` …

- Zugriff in der Funktion
  wie auf eine normale Variable:

  `if (val < 0)` …

- Aufruf auch mit ganz normalem Wert:

  `result = func(n);`

Wiederholung Funktionen (4)

Call by Reference:

- Parameter wird *als Pointer* deklariert:
  `int func(int *ptr)` …
- Zugriff in der Funktion mit * (weil es ja ein Pointer ist):
  `if (*ptr < 0)` …
- Speichern des Ergebnisses für den Aufrufer auch mit *:
  `*ptr = -1;`
- Aufruf mit einem *Pointer*-Wert
  (*Adresse einer Variablen / eines Array-Elementes*):

```
result = func(&n);
result = func(&(a[i]));
```

Wiederholung Funktionen (5)

Arrays als Parameter:
Immer "by Reference" (Pointer)!
("by value" (Kopie) geht bei Arrays gar nicht!)

- Parameter wird als *Pointer* oder mit **[]** deklariert:

```
int func(int *data) ...
int func(int data[]) ...
```

- Zugriff in der Funktion wie auf normales Array:

```
if (data[i] < 0) ...
```

- Aufruf ganz normal mit Array, **"..."** oder Pointer:

```
result = func(myData);
```

Wiederholung Funktionen (6)

Bei Arrays als Parameter überlegen:

- *Hat das Array eine fixe Größe*
  oder eine Ende-Markierung (wie bei Strings)?

  Wenn nein:

  Größe** eigens als zusätzlichen* **`int`-Parameter übergeben!*

- Soll das Array in der Funktion *nur gelesen*
  (aber nicht geändert) werden?

  Wenn ja:

  Parameter bitte als **`const`** *deklarieren!*

Wiederholung Funktionen (7)

Funktionen mit einem Array als Ergebnis:

- In praktisch allen Fällen (siehe **`strcpy`** usw.!):

  Das Array für das Ergebnis lokal ***im Aufrufer anlegen***
  und ***als zusätzlichen Parameter*** *"zum Befüllen"*
  ("by Reference", als Pointer) an die Funktion übergeben!

- Wieder überlegen:
  Ist ein *separater Parameter*
  für die *Array-Größe* notwendig? (==> *Überlauf* prüfen!)

- Ev. den Pointer auf das übergebene Ergebnis-Array
  zusätzlich als Returnwert zurückgeben
  (nicht notwendig, aber oft praktisch).

Wiederholung Funktionen (8)

Achtung bei Arrays:

- Lokale Arrays kann man *nicht* mit **`return`** *"als Kopie" zurückgeben!*
- Man darf als Returnwert auch *keinen Pointer auf ein lokales Array zurückgeben!*

  Das lokale Array wird bei **`return`** freigegeben

  ==> Der Pointer zeigt *"ins Leere"*!
- Man kann die *Größe von Array-Parametern nicht mit* **`sizeof`** feststellen:

  `sizeof` liefert nur die Größe des Pointers selbst!

`const`-Pointer-Parameter (1)

`const`-Pointer-Parameter:

- Der Wert wird ***"by Reference"*** übergeben,
  d.h. die Funktion *"sieht" das Original* im Aufrufer
- ... aber darf es ***nicht ändern***

==> ***"Read-Only-Referenz-Parameter"***

Anwendung: Vor allem (aber nicht nur) bei Parametern für

- ***Arrays*** (geht nur "by Reference")
- ***Strukturen*** (kommen gleich):

  Aus *Effizienzgründen* meist *"by Reference"* übergeben
  (auch wenn der Parameter *"nur hinein"* in die Funktion geht
  und gar nicht geändert werden soll!)

`const`-Pointer-Parameter (2)

Wenn man

- Einen *Pointer auf eine String-Konstante* **"..."** zuweist oder als Parameter übergibt
- Einen *Pointer* auf eine **`const`**-*Variable* mit **`&const_var`** zuweist oder als Parameter übergibt
- Einen "*Pointer auf* **`const`**" auf einen anderen Pointer zuweist oder als Parameter übergibt

dann ***muss*** der *Ziel-Pointer bzw. der Parameter*
als ***"Pointer auf* `const` ..."** deklariert sein:

`const` *ganz vorne* in der Deklaration, nicht hinter dem *****
==> *"Wert, auf den der Pointer zeigt, ist konstant"*

`const`-Pointer-Parameter (3)

Aber:

- Man darf *nicht-konstante* Werte und Pointer an **`const`**-Pointer-Parameter übergeben
- Man darf normale Pointer auf Pointer-auf-**`const`** zuweisen

Bei Zuweisung, Parameter-Übergabe, Returntyp, ...:

`const` *dürfen dazukommen,*
aber nicht wegfallen!

==> *Man darf kein* **`const`** *"verlieren"!*

(man darf einen erlaubten Schreib-Zugriff verbieten,
aber keinen verbotenen Schreibzugriff erlauben)

`const`-Pointer-Parameter (4)

Bei Verletzung dieser Regel:

- **In C:** *Warning*
- **In C++:** ***Error***

  Wenn wirklich nötig und sinnvoll:

  Explizite Typumwandlung erforderlich!

File-I/O (1)

- **Grundkonzept**
- **Die Standard-Files**
- **I/O-Funktionen:**
  Öffnen, schließen und positionieren
  I/O zeichenweise, zeilenweise, blockweise
  Formatiertes I/O, `sprintf` und `sscanf`
- **Datei-Ende und Fehler**
- **Tücken bei `scanf`**

... (nächste Seite)

File-I/O (2)

- File-I/O unter Windows
- Pufferung von File-I/O
- File-I/O auf System-Ebene
- `mmap`
- File-I/O in C++

Files aus Programmsicht ... (1)

... sind fast alles, was man als

Folge bzw. Strom einzelner Bytes
(deshalb auch ***"Streams"*** genannt)

lesen oder schreiben kann:

- "Echte" *Dateien* (Platte/SSD, Netzlaufwerk, ...)
- *Tastatur*eingabe, *Terminal*ausgabe
- *I/O-Geräte* (Drucker, Scanner, ...)
- *Pipes* (Verbindungen zu anderen Programmen)
- *Netzwerkverbindungen* (TCP/IP-Streams)

Files aus Programmsicht ... (2)

... werden normalerweise

sequentiell verarbeitet,

entweder ***byteweise***
oder in beliebig kleinen / großen
Blöcken von Bytes

==> Jeder Lese- oder Schreibzugriff
liest / schreibt die ***nächsten Bytes***

Die Bytes können ***beliebige binäre Daten***
oder ***Texte (Zeichen)*** darstellen

Files aus Programmsicht … (3)

Bei "echten" Dateien:

Das System führt intern Buch über
die ***aktuelle Schreib- / Lese-Position:***

"Beim wievielten Byte der Datei bin ich gerade?"

Diese Position lässt sich auch *beliebig versetzen*:

"Schreib am Dateiende / am Dateianfang weiter"

"Lies beim nächsten Lesen ab Byte Nummer xxx"

Aber:
- Ist *deutlich langsamer* als rein sequentielles I/O
- Kommt *selten* vor

Möglichkeiten für File-I/O

1.) File-I/O des C-Standards
(implementiert in der *C Library*):

a) ***Zeichen***weise (= byteweise)

b) ***Zeilen***weise

c) ***Formatiert*** (mit Konvertierungen)

d) ***Block***weise, d.h. **n** Bytes (typisch für *binäre* Daten)

2.) File-I/O des Betriebssystems
(*System Calls*, "unter" der C Library)

File-I/O-Syscalls von *Unix* (Linux) sind im *POSIX-Standard!*

3.) `mmap`-Mechanismus o.ä. des Betriebssystems
(*ganz anderes Konzept* für den Zugriff auf File-Daten)

File-I/O-Funktionen des C-Standards

... kommen aus dem Header **`stdio.h`**

... repräsentieren einen File durch einen

Pointer vom Typ **`FILE *`**

`FILE` ist eine (ziemlich große!) *Struktur*
mit *interner Verwaltungsinformation* zur Datei
(z.B. aktuelle Schreib- / Lese-Position)

Das *"Innenleben"* einer **`FILE`**-Struktur
ist *unsichtbar* und für uns *irrelevant*

Die Standard-Files (1)

Es gibt ***drei vordefinierte globale* `FILE` *-Variablen:**

- Für die "normale" Programm*eingabe*
  (normalerweise: Tastatur)

  `stdin` = *"Standard Input"*

- Für die "normale" Programm*ausgabe*
  (normalerweise: Bildschirm bzw. Terminalfenster)

  `stdout` = *"Standard Output"*

- Für die Ausgabe von *Fehlermeldungen*
  (normalerweise: Auch Bildschirm bzw. Terminalfenster)

  `stderr` = *"Standard Error"*

Die Standard-Files (2)

Die **`std`**-Files sind

beim Programmstart *schon geöffnet*,
können sofort gelesen / geschrieben werden

Ausnahme manchmal: Bei *Hintergrundprogrammen & Diensten*
ohne Verbindung zu einem Terminal sind die **`std`**-Files *ev. geschlossen*,
aber selbst dort *sollten* sie (umgeleitet) immer verfügbar sein!

`scanf`, **`getchar`**, ... verwenden implizit **`stdin`**

`printf`, **`putchar`**, ... verwenden implizit **`stdout`**

Man darf die **`std`**-Files für die meisten File-Funktionen
wie normale Files verwenden (auch schließen usw.),

aber man darf *nicht* auf **`stdin`** usw. *zuweisen*

Ein- und Ausgabe-Umleitung

Die **`std`**-Files können ***umgeleitet*** werden:

- Auf der *Befehlszeile* mit **`< > >>`** *in eine Datei*
- Auf der *Befehlszeile* mit **`|`** *in eine Pipe*

  = *Datenverbindung vom linken zum rechten Programm*
- *Im Programm selbst* mit **`freopen`** oder **`dup2`**
- Im *aufrufenden Programm*:

  Ein frisch gestartetes Programm "erbt"
  die *aktuell offenen* **`std`**-Files von seinem "Vater"
  (auch wenn sie *umgeleitet* sind!)

`stderr`

Man sollte ***für Fehlermeldungen*** usw.

immer **`stderr`** verwenden: **`fprintf(stderr, ...);`**

Gründe:

- **`stderr`** wird *trotzdem am Terminal* angezeigt,
  auch wenn **`stdout`** in eine Datei / Pipe *umgeleitet* ist

  Unter *Linux*:
  `stderr` lässt sich mit **`2>`** *getrennt* von **`stdout`** *umleiten*

- **`stderr`** ist *ungepuffert*:
  Jeder Output auf **`stderr`** wird *sofort angezeigt*

  (bei **`stdout`** ist das *nicht* der Fall!)

Der Zugriff auf andere Files

... erfolgt immer in *drei Schritten*:

- ***File öffnen***

  (*vor* dem ersten Lesen/Schreiben)
- ***Lesen / Schreiben*** (+ Positionieren, ...)
- ***File schließen***

  (*nach* dem letzten Lesen/Schreiben)

Das Öffnen

... ist die einzige Funktion,
bei der der ***Dateiname*** angegeben wird

... legt auch die ***I/O-Richtung usw.*** fest
- *Lesen oder Schreiben* oder beides?
- Bei Schreiben: *Neu anlegen* wenn nicht vorhanden?
- Bei Schreiben: *Überschreiben* oder hinten *anhängen*?

==> Das System

... ***sucht die Datei***, legt sie eventuell neu an
... ***prüft die Zugriffsrechte***
... beim Überschreiben: *Leert* die Datei
... *initialisiert* die *interne Verwaltungsinformation* **FILE**
(z.B. File-Position auf *"Ende"* beim Anhängen, sonst auf **0**)

Lesen und Schreiben?

Grundsätzlich kann man einen File

zum *gleichzeitigen Lesen und Schreiben öffnen* und
Lese- und Schreiboperationen beliebig ***mischen***

Aber:

- Das ist *(viel) langsamer* als “nur lesen”
  oder “nur schreiben”
- *Bei jedem Wechsel* zwischen Lesen und Schreiben
  muss eine *Positionier-Funktion* aufgerufen werden
  (auch wenn man die Position gar nicht ändert!)

  Sonst: Effekt *undefiniert* / *implementierungsabhängig!*

==> *Abwechselndes Lesen & Schreiben besser vermeiden!*

fopen

```
FILE *fopen(const char *name,
            const char *mode);
```

- Öffnet den File namens **name**
- Liefert einen *Pointer* auf die *neue* **FILE**-Struktur
  oder **NULL** bei *Fehler (nicht gefunden, keine Rechte, ...)*
  ==> *Returnwert immer sofort auf* **NULL** ***prüfen!!!***

mode: (plus plattformabhängig weitere!)
"r" ... nur *lesen*
"w" ... nur schreiben, *überschreiben*
"a" ... nur schreiben, nur *hinten anhängen* (*"append"*)
"r+", **"w+"**, **"a+"** ... dasselbe mit *lesen & schreiben*
bei **"w"** und **"a"** werden *nicht existierende Files neu angelegt*

Das Schließen

Bei einem *geordneten Programmende* werden
*alle offenen Files **automatisch** geschlossen*,
aber man sollte sie trotzdem

***vorher explizit schließen*!**

- Offene Files belegen (endliche!) *Ressourcen*
  (im Programm und im *Betriebssystem*)
- Geschriebene Daten noch offener Files gehen
  bei Abstürzen meist (teilweise) verloren
- Explizites Schließen kann *auf Fehler geprüft* werden,
  beim impliziten Schließen bleiben Fehler unerkannt

fclose

`int fclose(FILE *file);`

Schließt den File **`file`**:

- Eventuell noch *gepufferte Daten* werden *geschrieben*
- Ressourcen (z.B. die **`FILE`**-Struktur) werden *freigegeben*

Liefert **`0`** bei Erfolg, **`EOF`** bei Fehler

`file` ist nachher *ungültig*

==> nicht mehr darauf zugreifen!

Positions-Funktionen

`long ftell(FILE *file);`

Liefert die *aktuelle File-Position* in Bytes (oder **`-1`** bei Fehler)

`void rewind(FILE *file);`

Setzt die aktuelle File-Position *zurück an den Anfang*

`int fseek(FILE *file, long offset, int pos);`

Positioniert den File **`offset`** Bytes relativ zu **`pos`**:

- **`SEEK_SET`**: **`offset`** ab *File-Anfang*
- **`SEEK_CUR`**: **`offset`** ab *aktueller Position*
- **`SEEK_END`**: **`offset`** ab *File-Ende*

`fgetpos` und **`fsetpos`** sind alternative *Varianten* von **`ftell`** und **`fseek`**

Zeichenweises I/O (1)

`int fgetc(FILE *file);` oder
`int getc(FILE *file);`

liest das *nächste* Byte bzw. Zeichen

Returnwert: Das gelesene Byte (als **`unsigned`**-Wert)
oder **`EOF`** bei *Dateiende oder Fehler*

`int fputc(int c, FILE *file);` oder
`int putc(int c, FILE *file);`

schreibt *ein* Byte bzw. Zeichen

Returnwert: Das geschriebene Byte (als **`unsigned`**-Wert)
oder **`EOF`** bei *Fehler*

Zeichenweises I/O (2)

- **`getc`** und **`putc`** sind normalerweise *keine Funktionen*, sondern *als Makros implementiert* (kommt später)

  ==> minimal *schneller*,
  aber mit ein paar *Einschränkungen*

- **`fgetc`** und **`fputc`** sind *echte Funktionen*

Die Funktionalität ist ident!

Zeichenweises I/O (3)

```
int getchar(void);
int putchar(int c);
```

sind *dasselbe* wie

`getc(stdin)` und **`putc(c, stdout)`**

==> *liest von der Tastatur / schreibt auf den Bildschirm*

```
int ungetc(int c, FILE *file);
```

Anschaulich:

Schiebt ein *"zuviel gelesenes" Zeichen zurück in den File*:

Das Zeichen wird *beim nächsten Lesen* ***nochmals*** gelesen!
(nur für *maximal ein Zeichen* möglich, mehrere **`ungetc`** sind *undefiniert*!)

Zeichenweises I/O (4)

Die ***typische zeichenweise Lese-Schleife:***

```
int c;
while ((c = fgetc(f)) != EOF) { …
```

- **c** muss ***immer*** **int** sein, ***nie*** **char** !!!

  c muss *alle möglichen* **char**-Werte *speichern* können

  … und *zusätzlich* **EOF**

  ==> *Ein Wert mehr*, als in einem **char** Platz hat!

  Die gelesenen **char**'s sind *immer positiv* **0** … **255**, **EOF** ist **–1**
  (auch auf Plattformen wo **char** als **signed char** implementiert ist).

- Die *Klammern* um die Zuweisung sind *notwendig*!

Zeilenweises I/O (1)

```
char *fgets(char *str, int size, FILE *file);
```

Liest ***eine Zeile*** (bis *einschließlich* **\n**),
speichert sie in **str** (incl. dem **\n**)
und hängt ein **\0** an

Returnwert:

str bei *Erfolg*
NULL bei *End-of-file gleich am Anfang* oder *Fehler*

Bei *zu langer Zeile*:

- Es werden **size-1** Zeichen gelesen (nie mehr!)
  und in **str** gespeichert (ohne **\n**, aber mit **\0** dazu)
- Das *nächste Lesen* liest den *Rest der Zeile*

Zeilenweises I/O (2)

Die ***typische zeilenweise Lese-Schleife:***

```
char line[LINE_LEN];
while (fgets(line, sizeof(line), f)) { …
```

oder

```
while (fgets(line, LINE_LEN, f)) { …
```

Implizite Schleifenbedingung:

`fgets( … ) != NULL`

==> Schleife liest *bis End-of-file oder Fehler*

Zeilenweises I/O (3)

`char *gets(char *str);`

Wie **`fgets`**, aber

- liest von **`stdin`** (der *Tastatur*)
- speichert das **`\n`** am Ende *nicht* mit
- macht ***keine** Längenprüfung*
  (schreibt bei zu langem Input über das Ende von **`str`**!!!)

==> *Sicherheitsrisiko!!!*

"Never use this function!"

Stattdessen **`fgets(line, LINE_LEN, stdin)`** verwenden!

Zeilenweises I/O (4)

`int fputs(const char *str, FILE *file);`

schreibt **`str`** in den File (*ohne* das **`\0`**)

... *ohne* ein zusätzliches **`\n`** anzuhängen

Returnwert:

`>=0` bei *Erfolg*, **`EOF`** bei *Fehler*

`int puts(const char *str);`

wie **`fputs`**, aber

- schreibt auf **`stdout`** (den *Bildschirm*)
- gibt nach **`str`** ein *zusätzliches* **`\n`** aus

Blockweises (binäres) I/O

```
size_t fread(void arr[], size_t e_size,
             size_t e_cnt, FILE *file);

size_t fwrite(const void arr[], size_t e_size,
              size_t e_cnt, FILE *file);
```

Liest oder schreibt eine *fixe Anzahl* von **`e_cnt`** *Elementen*
(Größe *eines* Elementes: **`e_size`** Bytes)
in ein / aus einem *Array* **`arr`**

Die Array-Daten werden ***1:1 kopiert***: *Binär*, *unformatiert*, *ohne* Berücksichtigung von Zeilenvorschüben, **`\0`** usw.

Returnwert:
Anzahl der gelesenen / geschriebenen *Elemente* (nicht Bytes!)
Bei *Fehler* oder *End-of-File*: *Weniger* als gewünscht, ev. **`0`**

Formatiertes I/O

```
int fscanf(FILE *file, const char *format, ...);
int fprintf(FILE *file, const char *format, ...);
```

Wie **`printf`** und **`scanf`**,
aber liest *aus einer Datei* / schreibt *in eine Datei*

Returnwert bei **`fprintf`**:

Anzahl der *geschriebenen Zeichen* oder **<0** bei *Fehler*.

Returnwert bei **`fscanf`**:

- *Anzahl* der *gespeicherten Werte*
  = Anzahl der *erfolgreich* gelesenen und konvertierten **%**...
  Können *weniger* sein als **%**... vorhanden sind!
- **EOF** bei *I/O-Fehler* oder wenn *nichts konvertiert* wurde:
  Bei *End-of-file* oder *Mismatch* ***vor*** dem *ersten* **%**...

sscanf und **sprintf**

```
int sscanf(const char *str, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size,
             const char *format, ...);
```

Wie **fscanf** / **fprintf**,
aber *liest aus einem String* statt einem File
bzw. *schreibt in einen String* statt einen File

snprintf *begrenzt den Output* auf **size** Zeichen (incl. **\0**)
==> *Sicherer*, ***immer*** statt **sprintf** verwenden!

Anwendung z.B.:

- Universelle ***Konvertierungen*** Text <==> Zahl
- *Formatiertes* ***Einlesen mit Fehlerbehandlung*** (==> später)

Datei-Ende usw. (1)

Lesende File-Funktionen enden
mit einem *Fehler-Returnwert*, wenn

- ***wirklich ein Fehler*** aufgetreten ist
- ***oder wegen Dateiende*** nichts mehr gelesen wurde

Anhand des Returnwertes der Lese-Funktion

kann man das ***nicht unterscheiden!***

==> *Nach* Returnwert "Fehler" beim Lesen:

Mit **`feof`** oder **`ferror`** *prüfen*
um *echte Fehler* vom *Dateiende* zu *unterscheiden*!

Datei-Ende usw. (2)

`int feof(FILE *file);`

liefert **`true`**, wenn das *vorangehende* Lesen über das *Dateiende* hinaus lesen wollte

`int ferror(FILE *file);`

liefert **`true`**, wenn bei einer Dateioperation ein *"echter" Fehler* aufgetreten ist

Beide *bleiben gesetzt*, bis man sie explizit löscht:

`void clearerr(FILE *file);`

(**`fseek`** löscht **`feof`** ebenfalls)

Datei-Ende usw. (3)

Achtung:

*Man prüft **nie vorbeugend*** **`feof`** *!*

`feof` wird ***nicht*** *"vorausschauend"* gesetzt
==> meldet *nicht* *"das nächste Lesen wird scheitern"*

... sondern ist *erst dann* **`true`**,
nachdem eine Lese-Funktion versucht hat,
über das Dateiende hinaus zu lesen!

==> Auch wenn **`feof`** *vor dem Lesen* noch **`false`** war,
kann das *nächste Lesen* wegen End-of-File *scheitern*!

(z.B. wenn das Lesen davor
genau bis zum letzten Byte gelesen hat)

Datei-Ende usw. (4)

Achtung:

Man *prüft* für den *Lese-Erfolg* / das *Schleifenende*
immer den ***Returnwert der Lese-Funktion***, nie **`feof`** !

`feof` wird ***nur nach*** *"Fehler" beim Lesen* geprüft!

`feof` sagt nur: *"Das File-Ende wurde erreicht"*
Aber es sagt *nicht* aus, ob beim Lese-Aufruf davor

noch erfolgreich Daten gelesen wurden oder nicht!

Wenn noch erfolgreich der *letzte Rest Daten gelesen* wurde
(zumindest teilweise), und *dabei das File-Ende erreicht* wurde:
Lese-Funktion liefert "**ok**", **`feof`** ist danach trotzdem **`true`**

Nur bei *"Fehler"* der Lesefunktion wurde *nichts mehr gelesen*!

Fehler beim I/O

Jeder Aufruf einer I/O-Funktion
ist ***auf Fehler zu prüfen***

Strenggenommen *auch* das Lesen / Schreiben
von **`stdin`** / **`stdout`**
(könnte ja *auf eine Datei umgeleitet* sein!)

Pragmatisch geduldete Ausnahmen:

- *Schreiben* auf **`stderr`** (dann hilft nichts mehr…)
- *Schließen* von nur-*Lese*-Files (hat praktisch nie Fehler)

Bei Schreib-Files kann das Schließen scheitern!!!

Korrekte Fehlerbehandlung: Nächstes Kapitel!

Fehler bei **scanf** (1)

scanf / **fscanf** erfordert

umfangreiche Fehlerbehandlung,
wenn es *"deppensicher"* sein soll!

==> *Ungeprüftes* **scanf** wie in Übungsbeispielen
ist in produktivem Code

strikt verboten!

Weil:

scanf in Schleifen führt bei *falscher Eingabe*
meist zu *Endlosschleifen*!

Fehler bei **scanf** (2)

scanf liest den Input *nur so weit*,
wie er *erfolgreich konvertiert* werden kann:

Was nicht konvertiert wird,
bleibt im Input stehen!

==> *Fehlerhafter* Input bleibt nach dem **scanf**
unverändert als *nächster zu lesender Input* erhalten
(Text statt Zahl, "**,**" statt "**.**" in Kommazahl, ...)

==> Das *nächste* **scanf** *scheitert wieder*
(weil es wieder den *alten, falschen Input* bekommt
und wieder nicht verarbeiten kann)

==> *Endlosschleife*, Input kommt nicht mehr voran

Richtiges **`scanf`** (1)

Meine Empfehlung für *zeilenorientierten* Input:

In einer *Schleife* ...

- Eine *ganze Zeile* mit **`fgets`**
  vom Input *in einen String* lesen
- Den *Inhalt dieses Zeilen-Strings*
  mit **`sscanf`** konvertieren

  (oder "zu Fuß" mit Stringfunktionen, **`atoi`**, ...)

... beides wiederholen,
bis die Konvertierung *erfolgreich* war

Richtiges **`scanf`** (2)

Oder in einer *Schleife*:

- **`scanf`** machen, Erfolg *prüfen*
  (Returnwert von **`scanf`** = Anzahl der gewünschten **%**)
- Bei *Erfolg*: Schleife *verlassen*
- Mit zweiter Schleife *Input zeichenweise überlesen*
  (mit **`getchar`** bzw. **`fgetc`**)

  … bis zum nächsten *"sicheren Ausgangspunkt"*:
  Meist bis zu **`\n`** oder *"White space"*

  (in C++: Eigene Funktion **`ignore`**)

Richtiges **scanf** (3)

```
int wert, anzahl, zeichen;

for (;;) {
    printf("Eine Zahl bitte: ");
    anzahl = scanf("%d", &wert);
    if (anzahl == 1) break;
    if (anzahl == EOF) { … }
    do {
        zeichen = getchar();
    } while ((zeichen != '\n') &&
             (zeichen != EOF));
}
```

Richtiges **`scanf`** (4)

Beim Lesen von **`stdin`** rufen viele Programme **`fflush(stdin)`** zum Überlesen von falschem Input auf

Nicht nachmachen!

- De facto auf zahlreichen Plattformen:

  `fflush` *verwirft vorhandenen Tastatur-Input*
  (genau das will man haben)

- **Laut Standard:**

  `fflush` auf Input-Files
  ist *nur für "echte" Files (Disk) definiert*,
  nicht für **`stdin`**, Pipes usw.!

Weitere **scanf**-Probleme (1)

scanf liest bei "einfachen" Formaten (nur **%**)

- pro **%** *genau **ein** "Wort"*
- getrennt durch *beliebigen* ***"White space"***
  (Zwischenraum, Tab, Zeilenvorschub, ...)
- und zwar *beliebig viel* ***"White space"***

==> Man kann

- *nicht "Zwischenraum"* und *"Enter" unterscheiden*
- *nicht* gezielt *genau eine ganze Zeile* lesen

Beispiel: Ein **scanf** mit *drei* **%d** akzeptiert

- drei Zahlen mit Zwischenraum in einer *einzigen Zeile*
- drei Zahlen untereinander in *drei Zeilen*
- *zwei* Zahlen mit Tab + fünf *Leerzeilen* + *eine* Zahl usw.

Weitere **`scanf`**-Probleme (2)

Das ist

- bei *freier Eingabe* vom Terminal oft *praktisch*
- aber beim Lesen *Tabellen-formatierter Files* ***gefährlich:***

  Enthält eine Zeile einen Wert zuviel / zuwenig, werden die Werte in allen folgenden Zeilen um eine Spalte verschoben eingelesen!

Weitere **`scanf`**-Probleme (3)

`scanf` liest den *White Space*
hinter dem letzten "Wort" *nicht mehr*,
der bleibt im Input

==> Das nächste Lesen bekommt ihn!

- Das ist *kein Problem* für das nächste **`scanf`**:
  Es *überliest* diesen White Space
- Aber beim *Mischen* von **`scanf`**
  mit **`getchar`**, **`fgets`**, ...:

  Das *nächste* **`getchar`**, **`fgets`**, ... bekommt
  nur *das "Enter" vom Ende* des vorigen **`scanf`** !

Formatierter Input in C++

Das Lesen mit **>>** in C++
hat *genau die gleichen Probleme*
wie das Lesen mit **`scanf`** in C !!!

==> Gleiche Gegenmaßnahmen erforderlich!

- *Schleife* mit Prüfung, *Überlesen* des Inputs, ...

  (in C++ zusätzlich: *Fehlerstatus* zurücksetzen!)

- Oder *zeilenweises Lesen*,
  Zeile in *Stringstream* verwandeln,
  Stringstream mit **>>** lesen

Die Tücken von Windows (1)

Zeilenende in Textfiles:

- Unter Unix / Linux, in C: Nur **`\n`**
- In *Windows-Textfiles*:
  Traditionell **`\r\n`** *(2 Zeichen!)*

Beim Lesen:

`\r` gilt zwar auch als *"White Space"*,
aber verwirrt ev. Längenberechnungen usw.

Beim Schreiben:

Textfiles nur mit **`\n`** wurden z.B.
in **`notepad`** als *eine lange Zeile* angezeigt...

Die Tücken von Windows (2)

==> Viele Windows-Implementierungen
machen intern *automatisch* eine *Konvertierung*:

`\r\n` wird als **`\n`** gelesen
`\n` wird als **`\r\n`** geschrieben

Achtung:

- *Binäre Daten* werden dadurch *zerstört*!
- Die *Position / Filegröße* laut **`ftell`** usw.
  stimmt eventuell nicht mehr überein
  mit der *tatsächlichen Anzahl*
  gelesener / geschriebener Zeichen!

Die Tücken von Windows (3)

Für Binärfiles:

Im **`fopen`** an den **`mode`** ein **`b`** anhängen:

`"rb"`, **`"wb"`**, **`"ab"`**, **`"rb+"`**, …

- Unter Windows:
  Schaltet die **`\r\n`**-*Konvertierung aus*
- Auf allen anderen Plattformen:
  Wird laut Standard *akzeptiert & ignoriert*

Die Tücken von Windows (4)

Das **\n** am *Ende der **letzten** Zeile* eines Textfiles

- … ist unter Unix / Linux
  praktisch immer vorhanden

  (auch wenn das kein Standard garantiert!)
- … ***fehlt unter Windows sehr oft!***

==> In Programmen *nicht darauf verlassen*,
dass die letzte Zeile mit **\n** endet!

(z.B. in Prüfungen,
ob eine von **`fgets`** gelesene Zeile *zu lang* war)

Die Tücken von Windows (5)

Das **Verzeichnis-Trennzeichen in Filenamen** (z.B. im **`fopen`**):

- In Unix / Linux, in C: **/**
- In Windows: \

Die meisten Windows-Implementierungen *konvertieren automatisch*

==> Wenn möglich:

Auch unter Windows / verwenden!

Die Tücken von Windows (6)

Wenn \ verwendet werden muss:

Der \ muss in String- und **`char`**-Konstanten ***doppelt geschrieben*** werden

(da \ ja Steuerzeichen wie **`\n`** einleitet!)

Beispiel:

```
"C:\\tmp\\workfile.txt"
```

Output-Pufferung (1)

Geschriebene Daten werden
auf dem Weg vom **`printf`** usw.
bis zur Platte / zum Bildschirm / ...

auf 3 Ebenen zwischengespeichert
("gepuffert"):

- In der **C-Library**
- Im **Betriebsystem**
- In der **Hardware**
  (RAID-Karte, Platten-Controller)

Output-Pufferung (2)

Grund: ***Effizienz / Geschwindigkeit***

- Möglichst *wenige Aufrufe des Betriebsystems* (hoher Overhead!)
- Möglichst *wenige* Platten-Operationen, möglichst *große* Platten-Operationen, möglichst *gut sortierte* Platten-Operationen
- Bei *Notebooks*: Möglichst *langes Power-Down* der Platte
- Platten lesen/schreiben ***immer nur blockweise*** (früher 512 Bytes, heute 4 KB)

C-Library-Pufferung (1)

In der C-Library gibt es *3 Varianten der Pufferung*:

- ***Ungepuffert:***

  Geschriebene Daten gehen *sofort* an das Betriebsystem

- ***Zeilen-gepuffert:***

  Geschriebene Daten gehen erst *am Ende jeder Zeile* (sobald ein **\n** geschrieben wird) an das Betriebsystem

- ***Block-gepuffert:***

  Geschriebene Daten werden *intern zwischengespeichert*, bis ein *Puffer fixer Größe* (meist 4 KB) voll ist, gehen erst dann *in einem Rutsch* an das Betriebssystem

C-Library-Pufferung (2)

Per Default gilt:

- **`stderr`** ist *ungepuffert*
- **`stdout`** ist *Zeilen-gepuffert*,
  wenn es ***am Terminal*** angezeigt wird

  (aber *nicht bei Umleitung* in einen File!)
- ***alle anderen*** Ausgaben sind *Block-gepuffert*

Änderbar mit **`setvbuf`** usw.

Zwangsweises sofortiges Ausgeben
des *aktuellen Puffer-Inhaltes* mit **`fflush`** *("Puffer leeren")*
(sinnvoll z.B. nach *jedem* Schreiben von *Logfile-Einträgen*)

Schreiben von Input-Prompts

Aufforderungen wie **`"Zahl eingeben: "`**
haben meist ***kein*** **`\n`** am Ende,
da die *Eingabe gleich dahinter* erfolgen soll

==> Werden bei der Ausgabe laut Standard *gepuffert*,
nicht sofort ausgegeben!

==> Werden erst *nach der Eingabe angezeigt*
(gemeinsam mit der nächsten Ausgabe)

==> Vorsichtshalber *immer* **`fflush(stdout)`** verwenden!
(gleich nach dem **`printf`**, *vor* dem **`scanf`**)

Aber zum Glück:
Viele C-Libraries machen *bei jedem Lesen* von **`stdin`**
automatisch zuerst einmal ein **`fflush(stdout)`** !

Folgen des C-Library-Puffers

- Das *letzte tatsächliche Schreiben* (des letzten Puffers) erfolgt erst beim **`fclose`**

  ==> Beim `fclose` können Schreibfehler auftreten!!!
  (Platte voll, Netz weg, ...)

- Bei einem *Absturz des Programms* können *bis zu 4 KB schon "geschriebene" Daten* im Puffer

  verloren gehen!

- Bei *parallelem Schreiben* in *denselben* File:
  - Daten landen *nicht in zeitlicher Reihenfolge* der Schreib-Aufrufe im File
  - Das *Aneinanderhängen* bzw. Zusammenmischen der Daten erfolgt *blockweise*, *nicht zeilenweise*!

Folgen des System- / HW-Puffers (1)

Daten stehen auch
nach erfolgreichem **`fflush`** oder **`fclose`**
und sogar *nach dem Programmende*
***nicht* sicher auf dem Speichermedium!**

Z.B. bei Linux:
Pufferung typisch *max. 1 Minute*, bei Notebooks *bis zu 10 Minuten*!

- Daten können *bei System-Absturz trotzdem weg* sein!
- Das Schreiben ist zwar *logisch garantiert ok*
  (bezüglich Platz, Rechten, ...),

  aber kann immer noch *physikalisch scheitern*
  (Platte plötzlich defekt, Netzverbindung weg, ...)

Folgen des System- / HW-Puffers (2)

Eine Kontrolle der System- / HW-Puffer
ist auf Ebene des C-Library-File-I/O
***nicht* direkt möglich!**

Nur *auf Ebene des System-File-I/O* (kommt gleich):

- Aufruf von **`fsync`**

  Auf Unix / Linux und kompatiblen Systemen,
  nur POSIX-Standard, nicht C-Standard:

  ```
  fsync(fileno(file));
  ```

- Oder Flag **`O_SYNC`** beim **`open`**

Linux-Schmutztrick bei Programmen mit **`root`**-Rechten:
`sync();` oder **`system("sync");`**
(Ist Overkill: Schreibt systemweit *alle* Puffer auf Disk)

System-File-I/O (1)

- Das bisher besprochene File-I/O
  ist *im C-Standard definiert* (==> portabel)
  und *in der C-Library implementiert*

  ... und zwar unter Nutzung
  plattform-abhängiger *Betriebssystem-Aufrufe*
- Man kann auch unter *Umgehung der C-Library direkt auf die Betriebssystem-Aufrufe* zugreifen
  (und muss es für einige Dinge auch)
- Die System-I/O-Aufrufe von Unix / Linux stehen *nicht im C-Standard*, sondern *im POSIX-Standard*,
  sind aber *auf fast allen Plattformen verfügbar*

System-File-I/O (2)

- *Lesen & schreiben* mit System-File-I/O ist *viel primitiver*:

  Es gibt nur **`read`** und **`write`**
  für eine *fixe Anzahl von Bytes*:
  Binär und ungepuffert

  (keine Konvertierung / Formatierung,
  kein zeilenweises I/O, ...)

- *Offene Files* werden

  durch* `int`*'s repräsentiert,
  nicht durch **`FILE`**-Strukturen

  `stdin`, **`stdout`** und **`stderr`** haben die Nummern **`0`**, **`1`** und **`2`**
  (bzw. **`STDIN_FILENO`**, **`STDOUT_FILENO`** und **`STDERR_FILENO`**)

System-File-I/O (3)

- Die meisten System-File-I/O-*Funktionen* haben *kein* **`f`** im Namen:

  `open`, **`close`**, **`read`**, **`write`**, **`lseek`**, ...

  Es gibt aber zahlreiche *Ausnahmen* mit **`f`**:
  `fsync`, **`ftruncate`**, ...

- Die *Header* heißen **`unistd.h`** und **`fcntl.h`**, für manche Funktionen auch **`sys/stat.h`** usw.

System-File-I/O (4)

Achtung:

C-Library-I/O und System-I/O
sind *zwei Zugriffs-Ebenen* auf *dieselbe Funktionalität*

Im Allgemeinen:

Nicht beides für denselben File *mischen*!
Vor allem *nicht* auf *demselben* File *gleichzeitig*
mit C-Library-I/O-Funktionen und System-I/O-Funktionen
lesen und schreiben: *Entweder - oder!*

Für manche Zwecke ist es sinnvoll / nötig,
für einen File der C-Library-Ebene auf die System-Ebene zu gehen:

- **`fileno(file)`** liefert die **`int`**-*Filenummer auf Systemebene*
  zu einem offenen File auf C-Library-Ebene
- **`fdopen(nr, ...)`** liefert einen neuen **`FILE`** zum offenen System-File **`nr`**

mmap (1)

I/O auf Unix-Systemebene bietet neben **read** und **write** noch einen *konzeptionell ganz anderen Mechanismus*, um File-Daten *zu lesen und zu schreiben*:

"memory mapped I/O"

Funktion **mmap**

- Nur *für "echte" Files*, nicht für Terminal, Pipes usw.
- Verwendet die *virtuelle Speicherverwaltung* bzw. Hardware-Unterstützung durch die *MMU*
- Wird unter Linux z.B. statt **read** genutzt, um beim Programmstart *Shared Libraries* zu laden

mmap (2)

Der **mmap**-Aufruf

- *reserviert* einen bisher freien *Speicherbereich* im RAM in der *Größe des angegebenen Files*
  (unabhängig von Stack, Heap und globalen/statischen Daten!)
- blendet die *Daten des Files 1:1* in diesen Bereich ein
- liefert einen *Pointer auf den Anfang* dieses Bereiches

==> Über diesen Pointer kann man

***direkt** auf die **Daten des Files** zugreifen,*
*als ob diese **ein riesiges Array im Speicher** wären!*

(die Details des **mmap**-Aufrufes sind sehr *komplex*...)

mmap (3)

Aber:

Beim **mmap**-Aufruf selbst werden
*noch **keine Daten** vom File ins RAM geladen!*

mmap stellt nur im Betriebssystem *die logische Verbindung*
zwischen RAM-Bereich und Plattenplatz der Datei her

Die Daten werden *erst beim ersten echten Zugriff*
auf den Speicher in *beliebiger Reihenfolge*

***"on Demand" geladen** (und ev. wieder freigegeben)*

und zwar *in Blöcken von 4 KB*

==> Nur die *tatsächlich benutzten Blöcke* der File-Daten
stehen *wirklich im RAM* und belegen RAM-Platz!

mmap (4)

Auch beim *Ändern der Daten* im RAM

- werden die Daten ***nicht sofort*** in den File *geschrieben*
- sondern nur in der MMU *als "verändert" markiert*

Die *veränderten 4-KB-Blöcke*
(*nur* die wirklich veränderten!)
werden *irgendwann später asynchron*
(unabhängig vom Programm)
zurück in den File kopiert!

(genauso wie die *File-Write-Buffer* des Betriebssystems)

Mit **msync** kann man das Rückschreiben *gezielt* anstoßen

mmap (5)

- Mit **mmap** kann man die *bestehenden Daten* eines Files lesen und schreiben, aber einen File

  weder neu anlegen noch vergrößern!

- Wenn *mehrere Programme gleichzeitig denselben File* mit **mmap** einblenden, kann dies *"privat" oder "geshared"* geschehen:

  Bei "geshared" sehen alle Programme
  denselben RAM-Bereich,
  d.h. sie sehen *sofort alle Änderungen der Daten*,
  die ein anderes Programm macht!

==> **mmap** wird in der *Parallel-Programmierung* häufig für *Shared Memory* genutzt! (auch *ohne* File möglich)

C++ File-I/O (1)

Die *Konzepte und Fähigkeiten* von C++ File-I/O
sind *sehr ähnlich zu C*

Die Typen, Funktionen usw. *heißen komplett anders*:

- Header **`iostream`** für Terminal, **`fstream`** für echte Files
- Statt **`FILE`**: **`ifstream`**, **`ofstream`**, …
- Statt **`stdin`**, …: **`cin`**, **`cout`**, **`cerr`** + **`clog`**
- Ähnlich wie C: **`open`**, **`close`**, **`flush`**, **`eof`**, …
  (aber **`open`** *meist implizit beim Anlegen* des **`stream`**-Objektes)
- I/O "roh", zeichenweise und zeilenweise ähnlich C
- Statt **`printf`** und **`scanf`**: **`<<`** und **`>>`**
- *Fehlerbehandlung* ist *deutlich komplexer* / aufwändiger!

C++ File-I/O (2)

Normales C-I/O *funktioniert auch in C++* wie bisher,
ist daher in C++-Programmen *statt C++-I/O verwendbar*.

Aber zumindest früher galt
(z.B. für große Programme mit C- und C++-Codeteilen):

***Niemals** im selben Programm auf denselben File*
***gleichzeitig mit C und C++ I/O** zugreifen!!!*

Grund: Die C- und die C++-Library
führten meist *unabhängig voneinander*

- ihre eigenen *Read- / Write-Puffer*
- ihre eigene *File-Position und -Länge*, den **`eof`**-Status usw.

==> Folge oft: *Massive Daten-Korruption!*

C++ File-I/O (3)

Daher historisch:

Viele C++-Programme verwenden *ausschließlich C-I/O*

Stand heute:

- Verhalten laut *Standard* ist immer noch *undefiniert*

  ==> verboten!

- Praktsich alle aktuellen C++-Implementierungen *machen es richtig*

  ==> Mischen kein Problem!

==> *“Politische” Entscheidung,*
Programmierrichtlinien der Firma usw. beachten!

Fehlerbehandlung

- Grundprinzipien
- Fehlermeldungen
- `errno`
- `strerror` usw.
- `exit`
- `setjmp` & `longjmp`
- `abort` & `assert`

Fehlerursachen

Haupt-Fehlerquellen:

- ***Probleme im Umfeld:***

  Platte voll, File fehlt, Netzwerk weg, Rechte fehlen, ...

- ***Probleme mit Daten:***

  Falscher / zu großer / bösartiger
  Benutzer-Input, File-Inhalt, Programm-Aufruf, ...

Erwartung:

"Geordnet" und "hilfreich" reagieren!

(ein Absturz ist weder geordnet noch hilfreich...)

Prinzip

“Vertrauen ist gut, Kontrolle ist besser!”

Wir prüfen:

- Alle ***Funktionsaufrufe***,
  die laut Doku einen Fehler liefern könnten
  (insbesondere jedes I/O, Syscalls, **`malloc`**, …)
- Alle ***“Daten von außen”***:
  - Benutzer-*Input*
  - *File-Daten*, Netzwerk-Daten, …
  - *Programm-Aufruf*, Environment, …

Erfahrungen

“If you made it fool-proof,
someone will invent a better fool!”

“Zwei Dinge sind unendlich,
das Universum und die menschliche Dummheit,
aber beim Universum bin ich mir
noch nicht ganz sicher.”

“Das Böse ist kreativ!”

Beliebtes Hobby der Qualitätssicherung:
“Wir füttern das Programm einmal
mit **/dev/urandom** *…”*

Fehlerbehandlung

- In *Spielzeug*-Programmen:

  Fehlerbehandlung *lokal*, meist lokales **`exit`**

- In *größerer Software*,
  vor allem in *universell verwendbaren Funktionen* meist:
  - Eher *keine eigenmächtige Fehlermeldungs-Ausgabe*
  - Kein **`exit`** !

  ==> Fehler z.B. *per Returnwert zurückmelden*,
  Fehlerausgabe usw. ist *Sache des Aufrufers*!

  Grund: Oft *komplexe, zentrale Fehlerbehandlung*:
  Aufräumen, Files & DB schließen, Logs schreiben, ...

- In *moderneren Sprachen* als C (z.B. C++): ***Exceptions!***

Fehlermeldungs-Ausgabe

Grundsätzlich ***immer*** auf **`stderr`** bzw. **`cerr`**,
nie auf **`stdout`** !

Übliche Form:

Programmname: Operation: Fehlergrund

- Bei *Syntax-Fehlern* in Compilern,
  beim Lesen *formatierter Input-Files*:

  Inputfilename:Zeilennummer:Spaltennummer: Fehlertext

- Bei Fehlern im *Programm-Aufruf*:

  Nach der eigentlichen Fehlermeldung:

  Hilfe / korrekten Aufruf */ Liste der Optionen anzeigen!*

Fixe Meldungsteile

- ***Programmname:*** **`argv[0]`**
  Damit *für alle Programmteile* verfügbar: **`argv[0]`** *in globale Variable* speichern!
  Unter Linux schon *vordefiniert*: **`program_invocation_name`** (nicht Standard)

  Weil: Skript mit *mehreren* Programm-Aufrufen,
  Zentraler Log-Dienst für mehrere Dienste, ...:

  "Von welchem Programm kam diese Meldung???"

- ***Operation:*** *"Was wollte das Programm tun, als der Fehler auftrat"*

  Falls vorhanden: ***Betroffenes Objekt*** *mit ausgeben!!!*
  (Filename, Netzwerk-Hostname, ...)

- ***Fehlergrund:*** Falls **`errno`**-bezogener Fehler (siehe unten)

  Immer den ***standardisierten Fehlertext des Systems****!*

Beispiele

```
more: Cannot open test.txt:
                No such file or directory

ping: Cannot resolve www.heise.de:
                Timed out
```

Optionale Meldungsteile (1)

- Bei **Servern, Diensten, langlaufenden Programmen, …**:

  Ganz vorne ***Datum & Uhrzeit!***

  Hilft bei zeitlicher Einordnung,
  Zuordnung zu anderen Ereignissen,
  Korrelation mit Meldungen anderer Programme:

  “Fehler trat auf, kurz bevor anderes Programm xyz abgestürzt ist”
  (==› Mögliche Ursache für dessen Absturz?)

  “Fehler treten jeden Arbeitstag um 9:30 Uhr herum auf”
  (==› Überlast-Problem?)

  “Fehler trat kurz nach Sterben des zentralen Netz-Switches auf”
  (==› wohl Folge-Fehler, weitere Analyse nicht nötig?)

Optionale Meldungsteile (2)

- ***Fehlerklasse:*** Üblich sind

  FATAL: *Fehler*, der das Programm *beendet*

  ERROR: *Fehler* (z.B. im Input), aber das Programm *macht weiter*

  WARNING: *Kein* Fehler (Operation wurde erfolgreich ausgeführt)
  aber Hinweis auf *mögliches / zukünftiges Problem*
  ==> *Kontrolle* durch den Benutzer / Administrator angebracht!

  INFO: *"Nur zur Info"*, alles gut (z.B. Versionsausgabe)
  ==> *Keine Schritte* des Benutzers / Administrators nötig

  DEBUG: *Interne Information* nur für den Entwickler, ignorierbar

  ==> Hilft dem Administrator bei *schneller optischer Klassifikation:*
  "Wo brennt es am dringendsten?"

  ==> Hilft automatischen Log-Analyse-Systemen, z.B.:
  Bei **FATAL** automatisch Administrator-Handy anrufen!

`errno` (1)

Grundprinzip aller Library-Funktionen mit Fehler-Fällen im C-Standard:

- Der ***Returnwert*** (**`NULL`**, **`-1`**, **`EOF`**, …) sagt nur, dass *irgendetwas schiefgegangen* ist, ***aber nicht was / warum!***
- Die ***eigentliche Fehlerursache*** steht *nach dem Aufruf* in **`errno`**

`errno` (aus **`errno.h`**) ist (gedanklich)

eine *vordefinierte* ***globale*** **`int`**-***Variable***

(de facto ist **`errno`** heute meist eine *thread-lokale Variable* oder ein *Makro*, damit sich die **`errno`**'s paralleler Threads *nicht gegenseitig beeinflussen*, aber die Vorstellung *"globale Variable"* ist in etwa zutreffend und hilfreich)

errno (2)

Werte von `errno`:

- **`0`** heißt *"kein Fehler"*

  Jeder andere Wert
  steht für eine *bestimmte Fehler-Ursache*

- Für alle Werte definiert **`errno.h`** *Konstanten*
  (siehe **`man errno`**):

  `EACCES`, **`ETIMEDOUT`**, **`ENOMEM`**, ...

- Die *konkreten Zahlen* sind *auf jeder Plattform andere*

  ==> Nur die Konstanten verwenden, keine Zahlen!

- Für *jede Funktion* steht in ihrer **`man`**-Page aufgelistet,
  welche **`errno`**'s bei dieser Funktion auftreten können

Falle bei **`errno`** (1)

Achtung:

`errno` ist nach einer Funktion *nur dann definiert*,
wenn der *Returnwert* ***"Fehler"*** angezeigt hat!

Nach *erfolgreichem Aufruf* ist **`errno`** *undefiniert*:

- Kann **`0`** sein
  (obwohl alle vordefinierten Funktionen laut Standard
  `errno` *nie* gezielt auf **`0`** setzen sollten!)
- Kann *unverändert*
  den *Wert vor dem Aufruf* haben
- Kann (trotz Erfolg!) einen
  beliebigen neuen Fehlerwert haben

Falle bei **`errno`** (2)

Konsequenz 1:

Um den *Erfolg eines Aufrufes* zu prüfen,
reicht es *nicht*, *nur* die **`errno`** mit **`0`** zu vergleichen:

Man muss *zuerst den Returnwert*
und *dann erst* die **`errno`** prüfen!

Falle bei **`errno`** (3)

Konsequenz 2:

Braucht man den aktuellen (bisherigen) **`errno`**-Wert *nach dem Aufruf einer anderen Funktion* noch
(z.B. nach einem **`printf`** !),

so muss man ihn *vor dem Aufruf in eine Hilfsvariable sichern*, auch dann,
wenn der folgende Aufruf sicher erfolgreich ist!

Insbesondere bei der *Ausgabe von Fehlermeldungen*:

Nach dem Aufruf des ersten **`printf`** o.ä.
enthält **`errno`** *nicht mehr*
den *ursprünglichen Fehlercode*!

Meldungs-Ausgabe-Funktionen

Zur Ausgabe auf **`stderr`** in üblicher Form:

- Am *schönsten* wären die **`err`**-Funktionen (BSD Unix) oder die **`error`**-Funktionen (GNU / Linux)

  ==> *nicht Standard, nicht portabel, eher vermeiden!*

- **`perror(...)`** ist Standard,
  aber erfordert *davor* ein separates **`sprintf`** o.ä.,
  um einen schönen Fehlertext zusammenzubasteln

  ==> *auch nicht gut!*

- **Ich empfehle daher:**

  Normales **`fprintf(stderr, ...)`** mit **`strerror(errno)`** als *letzten Teil* des Textes

strerror(int errno)

... ist eine Funktion des C-Standards (aus **string.h**),
die zur Fehlernummer im übergebenen **errno** den

standardisierten Fehlertext *liefert*

Also z.B.

EACCES ==> **"Permission denied"**

ETIMEDOUT ==> **"Connection timed out"**

Mit Glück berücksichtigt **strerror**
sogar die *aktuelle Ländereinstellung*
(auf Windows/mingw leider nicht)

`exit(int status)`

- *Beendet das Programm* normal
- **`status`** wird *dem aufrufenden Programm*
  bzw. *der Shell* zurückgemeldet,
  kann dort abgefragt werden
- Mit **`atexit`** kann man vorher *Funktionen registrieren*,
  die dann bei **`exit`** *automatisch ausgeführt* werden

  ==> Für *"Aufräum-Funktionen"*,
  sogenannte *"Exit Handler"*
- *Files* werden *automatisch geschlossen* usw.

Der Exit-Status

- Laut *Standard* nur:

  EXIT_SUCCESS für *"ok"*
  EXIT_FAILURE für *"Fehler"*

  Entspricht auf fast allen Plattformen **0** und **1**

- Unter *Linux*: Auch **2** … **125** *frei nutzbar*, ab **126** *reserviert*

  Zur Unterscheidung *verschiedener Fehler-Arten*,
  z.B. Aufruf-Fehler, falscher Input, "harter" Fehler

  Oder zur *Anzeige des Programm-Ergebnisses*, z.B.
  z.B. bei Text suchen:
  0 … gefunden, 1 … nicht gefunden, 2 … echter Fehler
  oder bei Files vergleichen:
  0 … gleich, 1 … verschieden, 2 … echter Fehler

`setjmp` & `longjmp`

Bieten eine Art **`goto`** *über Funktionsgrenzen hinweg*

... aber nur nach "weiter oben"
in den aktuell aktiven Funktionsaufrufen

Beispiel:

Von einem *tief geschachtelten Aufruf*
direkt zurück zur *zentralen Fehlerbehandlung* in **`main`**

==> *"C's primitiver Ersatz für Exceptions"*

Schmutztrick von Freaks, schwer nachzuvollziehen

==> *wenn möglich vermeiden*

Verboten, wenn das Programm auch *C++-Teile* enthält!

`abort()`

- ***Harter Programmabbruch***

  (schickt sich selbst ein **`kill SIGABRT`**:
  Technisch in etwa wie ein **`NULL`**-Pointer-Zugriff)

- Für *schwere interne Fehler*,
  nicht für normale Benutzerfehler

- *Kein Aufräumen* mehr,
  File-Write-Buffer werden nicht mehr geschrieben, …

- Unter *Linux*:

  Schreibt *Core Dump* (Speicherabzug) des Programms
  (zum späteren Debuggen der Leiche)

`assert(`*`condition`*`)`

aus **`assert.h`** (technisch: Ist ein Makro, keine Funktion!)

Prüft ***condition***

- Wenn **`true`**, d.h. **`!= 0`**: Tut *nichts*, wird *ignoriert*
- Wenn **`false`**, d.h. **`0`**:
  - Schreibt ***Fehlermeldung***: **`"Assertion failed..."`**
    mit *Quellcode-Filenamen* und *-Zeilennummer*,
    mit *Funktionsnamen* und mit der **`condition`**
  - ***Beendet das Programm*** hart mittels **`abort()`**

`assert` kann mit **`NDEBUG`** beim Kompilieren *komplett entfernt* werden
==> **`condition`** wird dann *gar nicht ausgerechnet*,
daher *keinen logisch notwendigen Code* ins **`assert`** schreiben!

Verwendung von **`assert`** (1)

Nicht zum Prüfen
von *normalen Fehlerbedingungen*,
Benutzerinput usw.

==> *Der Benutzer sollte **nie***
eine **`assert`***-Fehlermeldung sehen!*

Sondern ***nur für den Entwickler***

- zum *Testen & Fehlersuchen*
- zur *Absicherung gegen Denkfehler*
- in großen Projekten:
  Zum *Schutz gegen Mißverständnisse*
  (falsche interne Funktionsaufrufe usw.)

Verwendung von **`assert`** (2)

Eine solche **`condition`** (Bedingung) heißt

"Assertion"

= *eine Bedingung, die an dieser Stelle*
eigentlich immer erfüllt sein muss,
wenn das Programm keinen Denkfehler enthält

Typische Beispiele:

```
assert(p != NULL);
```
"**`p`** *darf an dieser Stelle nie* **`NULL`** *sein"*

```
assert(feof(myfile));
```
"hierher darf ich nur kommen,
wenn der File komplett gelesen wurde"

Strukturen

- Konzept
- Deklaration & Beispiele, Variablen
- Zugriff, Zuweisung und Vergleich
- Initialisierung, “Compound literals”
- Strukturen und Funktionen
- `typedef`
- Namensräume

Was sind Strukturen? (1)

Strukturen sind (so wie Arrays)

zusammengesetzte Datentypen:

Sie fassen *mehrere Werte* / Elemente
in einer Variable / unter einem Namen zusammen.

Beispiele:

- Daten einer Person (Name, Adresse, Geb-Datum, ...)
- Datum & Zeit (Tag, Monat, Jahr, Wochentag, Std, Min, Sek)
- Textposition (Filename, Zeile, Spalte)
- Bildschirm-Fenster (Position, Größe, Farbe, ...)

Was sind Strukturen? (2)

Aber:

- Jeder Strukturtyp ist ein *eigener Typ*
  mit einem *eigenen Typ-Namen*
  (Arrays sind kein eigens benannter Typ).
- Die Elemente können *verschiedene Typen* haben
  (bei Arrays haben alle Elemente gleichen Typ).
- Jedes Element hat einen *eigenen Namen*,
  mit dem auf das Element zugegriffen wird
  (beim Array sind die Elemente nummeriert,
  Zugriff auf ein Element mittels Index, d.h. seiner Nummer).
- Die *Anzahl* der Elemente in jedem Struktur-Typ ist *fix*
  (Arrays können variabel groß sein).

Bezeichnungen

- Die *Elemente* einer Struktur heißen im Englischen

  "Members" oder *"Fields"*, ev. auch *"Components"*
  (aber ***nicht*** "Elements").

- Im Deutschen sagt man auch oft

  "Strukturfelder",

  aber das hat *nichts* mit dem Begriff "Feld"
  für "Array" zu tun!

- Der *Name eines Strukturtyps* wird im Englischen
  "Struct Tag" genannt

  ("Tag" = "Kennzeichnung", "Aufkleber", ...).

Deklaration eines **`struct`**-Typs (1)

```
struct name {
    MemberDeklaration1;
    MemberDeklaration2;
    ...
};    // Achtung: ';' nicht vergessen!!!
```

name ... Name des Strukturtyps ("Tag")
(kann man sich beliebig aussuchen)

MemberDeklaration ... Deklaration von Struktur-Membern:
Schauen aus wie normale Variablen-Deklarationen,
die Namen der Member kann man sich aussuchen.

Keine Initialisierung **in der Deklaration möglich!!!**

Deklaration eines **`struct`**-Typs (2)

Nach einer solchen Deklaration gibt es

einen neuen Typ

"**`struct`** *name*",

den man wie **`int`**, **`double`**, **`char *`**, ... verwenden kann, um Variablen, Parameter, Pointer usw. zu deklarieren.

Nach einer solchen Deklaration gibt es aber

noch keine Variablen,

die diesen Typ haben!

Deklaration eines **`struct`**-Typs (3)

"**`struct xxx { ... };`**" legt nur fest,
wie Strukturen vom Typ ***xxx intern aussehen***,
aber legt ***noch keinen Speicher*** dafür an!

Anders gesprochen:

"**`struct xxx { ... };`**" ist nur der ***Bauplan!***

Erst ***danach*** werden

"Häuser nach diesem Plan gebaut"
(= Variablen mit dieser Struktur angelegt)

(keines, eines oder beliebig viele...)

Beispiele (1)

```
struct Datum {
   int tag, monat, jahr;
};

struct Addr {
   char strasse[32];
   int plz;
};

struct Person {
   char vorName[32], nachName[32];
   struct Datum gebDatum;
   struct Addr *adresse;
   long long int versNummer;
   char mw;
};
```

Beispiele (2)

Aus **<sys/time.h>**, siehe "**man gettimeofday**":

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

Aus **<grp.h>**, siehe "**man getgrent**":

```
struct group {
    char   *gr_name;        /* group name */
    char   *gr_passwd;      /* group password */
    gid_t   gr_gid;         /* group ID */
    char  **gr_mem;         /* group members */
};
```

... und auch hinter **FILE** steckt intern eine **struct** .

Achtung: Wenn *vordefinierte Funktionen* **struct**'s verwenden: **struct**-Deklaration kommt *aus dem Header*, *nicht* abtippen!

Anmerkungen … (1)

- *Arrays* als Member müssen *fixe* Größe haben!

  (es gibt in C einen Schmutztrick,
  aber der sollte sparsamst verwendet werden)

- Eine Struktur kann *Pointer* als Member enthalten,
  auch *Pointer auf den **eigenen** Struktur-Typ*.

  Das ist in der Praxis einer der *wichtigsten* Fälle:

  "Verkettete Datenstrukturen"
  (Listen, Bäume, …)!

- Eine Struktur darf *andere Strukturen*
  als Member enthalten, aber *nicht sich selbst*!

Anmerkungen … (2)

- Der neue **`struct`**-Typ ist erst *nach der Deklaration* bekannt, nicht davor!
- Deklarationen von **`struct`**-Typen (und anderen Typen) macht man fast *immer global* (*ganz oben* im Code): Man braucht sie ja meist in mehreren Funktionen…
- Auch wenn zwei **`struct`**-Typ-Deklarationen *genau gleich* ausschauen (selbe Member mit selben Namen & Typen):

  Es sind dennoch zwei *verschiedene Typen*

  ==> Variablen sind *nicht* aufeinander zuweisbar usw.!

“Incomplete **`struct`**’s”

Nur für Fortgeschrittene:

Man darf *Pointer* (und **`typedef`**’s) auf eine Struktur deklarieren, *bevor* der **`struct`**-Typ selbst deklariert wurde.

`struct xxx *xxxPtr;` // *vor* **`struct xxx { ... };`**
==> “**`xxxPtr`** ist ein Pointer auf eine **`xxx-struct`**,
aber das Innenleben von **`xxx`** ist (noch) unbekannt.”

Solange das Innenleben von **`struct xxx`** nicht deklariert wurde, heißt sie “*Incomplete* **`struct`**”
(und man kann *nichts* damit machen außer Pointer deklarieren!)

Wichtige Anwendungen:

- **`struct A`** enthält Pointer auf **`struct B`** *und umgekehrt*.
- **`struct`**’s mit *geheimem Innenleben*, z.B. **`FILE`** .

Die Größe von Strukturen (1)

Die **`struct`**–Member stehen

der Reihe nach im Speicher,

aber der Compiler darf

***Löcher** dazwischen / am Ende* lassen!

==> *Die Größe einer ganzen* **`struct`**

ist größergleich der Summe der Einzelgrößen!

==> **`sizeof`** verwenden, *nicht* selbst zusammenzählen!

Für Fortgeschrittene:

`offsetof(struct`*-Typ, membername*`)` aus **`stddef.h`**
liefert den *Abstand* des Members vom **`struct`**-Anfang (in Bytes).

Die Größe von Strukturen (2)

Grund für die Löcher: *Optimierung "Alignment"*:

Der Compiler legt **`int`**'s, **`double`**'s, Pointer, ...
immer auf *durch die eigene Größe teilbare Adressen*

(z.B. **`double`**-Werte auf durch 8 teilbare Adressen)

(und ganze **`struct`**'s auch oft auf durch $2^?$ teilbare Adressen)

==> Die Hardware kann dadurch *schneller* zugreifen

Bei manchen Compilern: **`packed struct`**,
wenn die Member *"dicht auf dicht"* gespeichert werden sollen

Deklaration von **`struct`**-Variablen

- … mit "**`struct`** ***name***" als *Typ* in der Deklaration (statt **`int`**, **`double`**, …):

  ```
  struct Datum heute;
  ```

- Auch Pointer auf **`struct`**, Array von **`struct`**'s, …:

  ```
  struct Person mitarbeiter[1000], *persPtr;
  struct Datum *datePtr, *pGebTage[1000];
  ```

- Auch als Parameter / Returnwert (siehe später!):

  ```
  struct Datum addDays(struct Datum datum, int tage);
  void sortByAge(struct Person members[], int size);
  ```

Deklaration in einem …

`struct`-*Typ*-Deklaration (hier: **`struct errorPos`**)

und **`struct`**-*Variablen*-Deklaration (hier: **`errors`**, **`errPtr`**)

kann man *kombinieren*:

```
struct errorPos {
   char *fileName;
   int lineNr, colNr;
} errors[MAX_ERR_CNT], *errPtr;
```

Wenn man den **`struct`**-Typ *nur in dieser einen Deklaration* braucht, darf man den **`struct`**-Typ-Namen (hier: **`errorPos`**) auch *weglassen* (keine gute Idee, der Name hilft beim Lesen!).

Zugriff auf Member (1)

- Wenn man die **Strukturvariable selbst** hat:

  structVar`.`*memberName*

  Beispiel: **`heute.jahr`**

- Wenn man einen **Pointer** auf eine **`struct`** hat:

  structPtr`->`*memberName*

  Beispiel: **`datePtr->tag`**

  Daher: **`datePtr->tag`**
  ist dasselbe wie **`(*datePtr).tag`**

Zugriff auf Member (2)

- Zugriff *beliebig geschachtelt*:

```
if (mitarbeiter[i].gebDatum.jahr < 1980)
putchar(mitarbeiter[i].nachName[j]);
postCode = mitarbeiter[i].adresse->plz;
persPtr->gebDatum.jahr = -1;
persPtr->vorName[0] = '\0';
persPtr->adresse->strasse[len] = '\0';
pGebTage[i]->jahr = j;
```

- *Adressen* kann man von ganzen **struct**'s oder von *einzelnen Membern* berechnen:

```
datePtr = &(mitarbeiter[i].gebDatum);
```

“Zugriffszeichen” (1)

Ich habe	***Ich will***	
Array oder Pointer auf Array	*1 Element* des Arrays	**`a[i]`**
Pointer auf Irgendetwas (außer Array)	Den *Wert / Inhalt* von “Irgendetwas” *als Ganzes*	**`*p`**
Struktur, Union oder Objekt	*1 Member* davon	**`s.m`**
Pointer auf Struktur, Union oder Objekt	*1 Member* davon	**`p->m`**
Irgendetwas (außer Array)	*Pointer* auf dieses Irgendetwas	**`&x`**

“Zugriffszeichen” (2)

Schritt für Schritt von *links nach rechts*
bzw. von *außen nach innen!*

Beispiele:

- *Array von Strukturen:*
  Zuerst `[]`, dann `.` **`stArray[i].member`**
- *Struktur mit Array oder Array-Pointer als Member:*
  Zuerst `.`, dann `[]` **`myStruct.arrMember[i]`**
- *Array von Struktur-Pointern:*
  Zuerst `[]`, dann `->` **`ptrArr[i]->member`**
- *Struktur mit Struktur-Pointer als Member:*
  Zuerst `.`, dann `->` **`myStruct.stPtr->member`**

Zuweisung von **`struct`**'s

Man kann **`struct`**'s *als Ganzes zuweisen*:

```
struct Datum heute, fertig;
fertig = heute;
```

Das ist vor allem bei großen **`struct`**'s aber *ineffizient*:

Alle Member-Werte werden *kopiert*!

==> Wenn möglich macht man nur eine Pointer-Zuweisung.

Achtung:

Beide Seiten müssen vom *selben* **`struct`**-Typ sein,
verschiedene **`struct`**-Typen sind *nicht zuweisbar*,
auch wenn sie intern dieselben Member haben!

Vergleich von **`struct`**'s

Man kann **`struct`**'s *nicht* als Ganzes *vergleichen*!

```
if (fertig == heute) { …
```

liefert eine Compiler-*Fehlermeldung*!

Für Fortgeschrittene:

```
if (memcmp(&fertig, &heute, sizeof(heute)) == 0) { …
```

Initialisierung von **`struct`**'s (1)

Muss in der **`struct`**-*Variablen-Deklaration* erfolgen (pro Variable!), *nicht* in der **`struct`**-Typ-Deklaration:

- Die **`struct`**-Typ-Deklaration beschreibt nur, wie eine Struktur dieses Typs *intern aufgebaut* ist!

  ==> Dabei wird noch **kein Speicher** angelegt!

  ==> Daher kann auch noch **nichts initialisiert** werden!

  ==> ... **`= `*`wert`*`;`** in einem **`struct`** ist **immer falsch**!

- Erst die Variablen-Deklaration legt *Speicher* dafür an!

  ==> Erst an dieser Stelle kann man angeben, wie dieser Speicher initialisiert werden soll!

Initialisierung von **`struct`**'s (2)

Alte Variante:

```
struct Datum gebTag = { 15, 7, 1965 };
struct Addr dhge = { "Weg d. F. 4", 07546 };
```

- Werte werden *der Reihe nach* angegeben
  (Reihenfolge der Member in der **`struct`**-Deklaration)
- Bei geschachtelten **`struct`**'s, geschachtelten Arrays, ...:
  Geschachtelte **`{ }`** !

```
struct Person test = {
   "Max", "Muster", { 31, 12, 1999 },
   &dhge, 9999999999, 'm' };
```

- Wie bei Arrays: *Leere* **`{ }`** initialisieren alles mit **`0`**

Initialisierung von **`struct`**'s (3)

Neue Variante (ab C99-Standard):

```
struct Datum gebTag =
   { .jahr = 1965, .monat = 7, .tag = 15 };
struct Addr dhge =
   { .strasse = "Weg d. F. 4", .plz = 07546 };
```

- Darf durcheinander sein und Lücken enthalten (nicht klug: *verwirrend*!)
- Bei *geschachtelten* **`struct`**'s und Arrays:

  Mit zusammengesetzten Feldbezeichnern und **`[ ]`** :
  `.gebDatum.jahr =` ...

(der **`gcc`** kennt noch eine 3. Variante und diverse Erweiterungen)

Initialisierung von **`struct`**'s (4)

Wie bei Arrays gilt:

Speichern mehrerer Werte in eine **`struct`**
mit … **`= { … };`**
ist *nur als Initialisierung*
(beim *Anlegen* in der *Deklaration*) möglich:

Man kann mit … **`= { … };`** *nichts*
in eine schon *bestehende* **`struct`** speichern,

d.h. … **`= { … };`** darf *nicht*
auf der rechten Seite einer *Zuweisung* stehen!

Aber …

Compound Literals (1)

"Compound Literals"

= *"Zusammengesetzte Konstanten"*

= Konstanten, die ***mehrere Werte*** enthalten

= *Struktur- oder Array-Konstanten*

Sie bestehen aus

Werten in { }

(wie bei der *Initialisierung*, beide Schreibweisen erlaubt)

mit dem ***Typ in ( ) davor***

Compound Literals (2)

`struct`-Literals kann man *zuweisen*:

```
// für heute und datePtr wie zuvor deklariert
heute = (struct Datum) { 24, 12, 2020 };
// oder
heute = (struct Datum)
   { .jahr = 2020, .monat = 12, .tag = 24 };

// & liefert einen Pointer auf das Compound Literal
datePtr = &((struct Datum) { 1, 1, 2021 });
```

Compound Literals (3)

Achtung:

Compound Literals existieren im Speicher

nur bis zum Ende des umgebenden **{ }** *-* ***Blockes****,*

nicht *während der gesamten Programmlaufzeit*

==> *nach* den **{ }** zeigen Pointer darauf *"ins Leere"*

==> Aufpassen, ganz gemeiner Fehler!

==> *Pointer auf Compound Literals ...*

- nicht als Returnwert zurückgeben

- nicht in globale Datenstrukturen speichern

Compound Literals (4)

Bei *Arrays*: Der *Array-Inhalt* wird *nicht kopiert*, es wird nur ein *Pointer auf den Anfang des Array-Literals* zugewiesen

==> Links vom **=** muss ein *Pointer* stehen, *kein Array*!

```
int *primes;
// lass primes auf den Anfang
// eines namenlosen Arrays mit den 6 int's zeigen
primes = (int []) {2, 3, 5, 7, 11, 13};
```

`(int [])`	ist erlaubt (Länge = Anzahl der Werte)
`(int [6])`	fixe Größe ist auch erlaubt
`(int [len])`	ist für *variables* **len** *nicht* erlaubt

`primes[0] = 1;` ist *erlaubt* & *ändert* den Wert im Array, *außer* man schreibt `(const int [])` statt `(int [])`

`struct`'s als Parameter (1)

Man kann **`struct`**'s (im Unterschied zu Arrays) *wahlweise* sowohl *by Value* (***Kopie*** **der ganzen Struktur**)

- als **Argument** übergeben
- und als **Returnwert** zurückgeben

als auch *by Reference* (***Pointer*** **auf das Original**):

```
struct Datum addDays(struct Datum d, int t);
fertig = addDays(heute, 27);

struct Datum *incrDate(struct Datum *dPtr);
datePtr = incrDate(&fertig);
```

(**&** nicht vergessen, ist *nicht* wie bei Arrays automatisch!)

`struct`'s als Parameter (2)

Aber:

Kopieren kostet, vor allem bei großen Strukturen:
Zeit zum Umkopieren und **Platz** für die Kopie

Daher:

- In >90 % der Fälle werden `struct`'s ***"by Reference"*** (mittels Pointer) übergeben und zurückgegeben
- `struct`-Übergabe und -Return *"by Value"* ohne Notwendigkeit gilt als schlechter Stil

Wenn ein Pointer auf eine `struct` übergeben wird, die *nicht geändert* werden soll: Bitte `const` deklarieren!

`struct`'s als Parameter (3)

Compound Literals kann man

als Parameter verwenden!

Sowohl *"by value"*:

```
fertig = addDays(
   (struct Datum) { 1, 1, 2021 }, laufzeit);
```

Als auch *"by Reference"*:

```
// für int cmpDate(struct Datum *d1, struct Datum *d2);

if (cmpDate(&fertig,
            &((struct Datum) { 1, 1, 2022 })
           ) < 0) { ...
```

`struct`'s als Returnwert (1)

Bei einem **`struct`**-**Ergebnis** gilt *wie bei Arrays:*

Einen *Pointer auf eine lokale* **`struct`**
als Returnwert zurückzugeben

ist ein *schwerer Fehler!*

Die lokale **`struct`** ist nach dem **`return`** weg

==> Der *Pointer zeigt ins Leere!*

`struct`'s als Returnwert (2)

1) Beste und häufigste Methode (wie bei Arrays):

- Die **`struct`** für das Ergebnis *schon im Aufrufer* anlegen
- … und der Funktion als *"by-Reference"*-Parameter (= Pointer darauf) *"zum Ausfüllen"* übergeben.

2) Wenn keine Struktur für das Ergebnis übergeben wird und die Struktur *nicht allzu groß* ist:

- Eine lokale Struktur mit dem Ergebnis *by Value* zurückgeben (==> wird in den Aufrufer kopiert).

  Häufig/notwendig bei C++ Operatoren wie **+** usw.

`struct`'s als Returnwert (3)

3) Notlösung, in einigen *vordefinierten Funktionen*:

   Es wird ein Pointer auf eine lokale *statische* **`struct`** mit dem Ergebnis zurückgeben.

   Problem dabei:
   Der nächste Aufruf (oder ein parallel laufender Thread) *überschreibt* das vorige Ergebnis!

4) Weitere Variante, für Fortgeschrittene (oft in C++):

   Es wird ein Pointer auf eine *dynamisch angelegte* **`struct`** zurückgegeben.

   (siehe demnächst)

Das lästige "**`struct`**"

- ***Nur in C++*** sind **`struct`**-Typ-Namen automatisch

  "echte" Typ-Namen (wie **`int`**, ...)

  ==> Man darf das "**`struct`**"
  in Variablen- und Parameterdeklarationen
  einfach *weglassen*:

  ```
  Person mitarbeiter[1000], *persPtr;
  ```

  ist genau dasselbe wie

  ```
  struct Person mitarbeiter[1000], *persPtr;
  ```

- In C leider *nicht* ...

`typedef` (1)

Daher wird in C für **`struct`**'s oft **`typedef`** verwendet:

`typedef` deklariert keine Variable,

sondern führt einen *neuen Typnamen*

*als **Abkürzung** für einen bestehenden Typ ein.*

Der neue Typ kann dann
wie **`int`**, **`double`**, ... verwendet werden
(in Variablen-Deklarationen, Prototypen usw.).

`typedef` (2)

Eine **`typedef`**-Deklaration sieht aus

- wie eine normale Variablen-Deklaration
- mit **`typedef`** davor.

Das, was bei einer normalen Deklaration
der *Variablenname* gewesen wäre,

ist bei einer **`typedef`**-Deklaration
der *Name des neuen Typs*.

Der *neue Typ* ist genau der Typ,
den die Variable gehabt hätte.

Eine **`typedef`**-Deklaration legt *keinen Speicher* an.

typedef (3)

Beispiele:

```
typedef unsigned long long int uint64;
uint64 nanoSecs;

typedef struct Person Person, *PersPtr_t;
Person mitarbeiter[1000];
PersPtr_t persPtr;   // PersPtr_t ist eher verwirrend:
                     // Besser Person * verwenden!

typedef char zeile[82];   // zeile ist ein Array-Typ!
zeile input, output;      // Don't, eher verwirrend!!
```

Im C-Standard *vordefinierte* **typedef**-Typen
enden oft auf "**_t**": **time_t**, **size_t**, ...
(auch der Typ **FILE** für Dateien ist ein **typedef**-Typ)

`typedef` und **`struct`** (1)

Ein **`typedef`**-Typ und der ursprüngliche Typ sind **ident**

==› in Zuweisungen & Vergleichen,
als Parameter usw.
beliebig mischbar!

Aber:

Der **`typedef`**-Typ existiert erst

ab dem Ende der **`typedef`**-Deklaration (ab dem **`;`**)

und kann daher innerhalb der Deklaration selbst

noch nicht verwendet werden!

`typedef` und `struct` (2)

Falsch:

```
typedef struct {
    int value;
    elem *next;
} elem;
```

Richtig:

```
typedef struct elem {
    int value;
    struct elem *next;
} elem;
```

oder:

```
typedef struct elem elem;

struct elem {
    int value;
    elem *next;
};
```

Namensräume (1)

- *Innerhalb* eines "Namensraumes"
  darf *kein Name doppelt* sein!

  (außer in C++:
  Funktionen mit *verschiedenen* Parameter-Typen
  dürfen gleich heißen)

- Verschiedene Namensräume
  dürfen *denselben* Namen enthalten:

  Die zwei Namen existieren gleichzeitig,
  sind gleichzeitig benutzbar und bezeichnen
  zwei voneinander unabhängige Dinge!

  (sie sind typischerweise durch die Stelle ihrer Verwendung
  eindeutig unterscheidbar)

Namensräume (2)

Je ein Namensraum sind:

- **“Normale” Namen**: Alle ...

  - Variablennamen
  - Funktionsnamen
  - Typnamen (auch **`typedef`**-Namen)
  - **`enum`**-Konstanten, ...

- In C: Alle **`struct`**- / **`union`**- / **`enum`**-**Tags**

  (nicht in C++: Dort sind das auch *“normale Namen”!*)

  ==> **`struct`**-Tag und **`typedef`**-Name dürfen in C *gleich* sein!

- In C++: Explizit **selbst definierte** Namensräume (==> 2. Sem.)

Namensräume (3)

Je ein eigener Namensraum sind weiters:

- Die **Member-Namen** jeder **`struct`**:

  *Ein **separater** Namensraum pro* **`struct`** !

  ==> Member in zwei verschiedenen `struct`'s dürfen gleich heißen!

  ==> Member dürfen gleich wie Variablen heißen!

- Die **`goto`-Labels**

Namensräume (4)

Bei *doppelten* Namen in *einem* Namensraum:

- Entweder (bei zwei Deklarationen auf *gleicher* Ebene, z.B. zwei gleichen globalen Namen)

  Fehlermeldung vom Compiler

- Oder (bei zwei Deklarationen in *geschachtelten* **`{}`** -Ebenen)

  "Innerer versteckt äußeren" *(Shadowing)*

  ==> *Lokale Variable "versteckt" gleichnamige globale Variable (im Bereich der lokalen Variablen ist die globale unsichtbar)*

  ==> *In* **`for`***-Schleife deklariertes* **`i`** *versteckt umgebendes* **`i`**

  Verwirrt, vermeiden!!!

Dynamische Daten

- Prinzip
- `malloc` usw.
- Gefahren
- `struct`'s variabler Größe
- `alloca`
- Dynamische Datenstrukturen: Listen und Bäume

Arten von Speicher

Bei der Programmausführung gibt es

3 Bereiche für Daten im RAM

- **Data:** ***Globale & statische Variablen****, Konstanten*

  fixe Größe, von Programmstart bis Ende *dieselben Daten*
- **Stack:** ***Lokale Variablen****, Parameter, Return-Adressen*

  wächst & schrumpft automatisch bei *Funktionsaufrufen*
- **Heap:** *"**Dynamisch angelegte" Daten***

  wächst und schrumpft durch
  explizites Reservieren und Freigeben von Speicherplatz

Dyn. Daten: Lebensdauer

Dynamische Daten “leben” (existieren)
vom *Moment des expliziten Anlegens*

“Reserviere mir jetzt 100 Bytes”

… bis zum *Moment des expliziten Freigebens*

“Diese Daten brauche ich ab jetzt nicht mehr”

… oder (wenn keine Freigabe) bis zum *Programmende*

Ganz *unabhängig von Funktionsaufrufen* oder **{ }**,
auch über Funktionsgrenzen hinweg!

Dyn. Daten: Sichtbarkeit

Dynamische Daten sind *keine benannten Variablen*,
sondern *"anonyme" Daten:*

Es gibt keine Deklaration dafür!

Daher haben sie auch ***keinen Variablennamen***,
der in bestimmten Programmbereichen gilt!

Beim Anlegen bekommt man einen

Pointer auf die Daten,

der ***Zugriff*** ist ***nur über diesen Pointer*** möglich!

==> *Jeder, der den Pointer hat bzw. kennt, kann zugreifen!*

(auch über alle Funktionsgrenzen / Filegrenzen hinweg)

Dyn. Daten: Der Pointer

Man kann / muss den *Pointer* beim Anlegen *speichern*

- in einer eigenen *Pointer-Variablen* (auch global möglich)
- oder in einem *Pointer-Member einer Struktur*
  (oft einer selbst wieder dynamisch angelegten Struktur!)
- oder in einem Array von Pointern usw.

Man kann ihn *kopieren*,
als *Parameter/Returnwert* übergeben, …

Der Typ der Daten wird nur festgelegt
durch den Typ des Pointers,
mit dem man auf die Daten zugreift!

Dyn. Daten: Aufräumen? (1)

"Verliert" man den letzten Pointer auf die Daten
(ohne die Daten freizugeben)

- so *existieren die Daten trotzdem noch* im Speicher
- aber man kann auf die Daten *nicht mehr zugreifen*
- und man kann die Daten auch *nicht mehr freigeben*!

C / C++ hat ***keinen "Garbage Collector"***:

"Unerreichbare" Daten belegen sinnlos Speicher,
werden nicht automatisch freigegeben!

(erst nach Programmende vom Betriebssystem)

Dyn. Daten: Aufräumen? (2)

Praktisch **alle anderen Programmiersprachen**

- kennen *keine explizite Freigabe* dynamischer Daten
- haben stattdessen einen *"Garbage Collector"*

*... der jeden **"unerreichbar"** gewordenen dynamischen Speicher*
automatisch erkennt und freigibt

Vorteil:

- Viel *komfortabler zu programmieren*
- Viel deppensicherer, *weniger fehleranfällig*

Nachteil: Komplex, *sehr ineffizient*, ev. *"Aufräum-Pausen"*

`malloc` (1)

Anlegen von dynamischem Speicher:

```
void *malloc(size_t n);
```

- *Reserviert* **`n`** ***Bytes*** dynamischen Speicher
- Liefert einen ***Pointer auf dessen Anfang***

`size_t` ist irgendein **`int`**-Typ, meist **`unsigned long`**

Das Ergebnis ist **`void *`**, weil **`malloc`** ja nicht weiß, auf welchen Typ von Daten der Pointer zeigt

`malloc` usw. kommt aus **`stdlib.h`**

malloc (2)

Achtung:

- **malloc** liefert **NULL**,
  - wenn es ***keinen Speicher mehr*** findet
  - oder wenn **n** gleich **0** war

  ==> Das Ergebnis von **malloc**
  immer auf **NULL** ***prüfen!***

- Die von **malloc** gelieferten Bytes sind *uninitialisiert*
  (enthalten zufälligen *Bit-Müll*)

- In vielen Programmier-Richtlinien von Firmen
  (in C nicht zwingend notwendig, in C++ schon):

  Ergebnis von **malloc** explizit *auf den richtigen Pointer-Typ casten*

`malloc` (3)

Argument von **`malloc`**:

- Fast immer ein **`sizeof`**, vor allem bei *Strukturen*
- Bei *Arrays*:
  `sizeof(element_typ) * anzahl_elemente`
- Bei *Strings*: **`strlen(str) + 1`**
  (Platz für die Ende-Markierung!)

Beispiel:

```
typedef struct Person Person;
Person *persArr =
  (Person *) (malloc(sizeof(Person) * persAnz));
if (persArr == NULL) ...
```

`calloc`

Alternativ:

```
void *calloc(size_t elems, size_t esize);
```

- Reserviert dynamischen Speicher für ein *Array* von **`elems`** *Elementen* mit **`esize`** *Bytes pro Element*, d.h. insgesamt **`elems * esize`** Bytes

  (wenn kein Array gefragt ist: Eines der beiden auf **`1`** setzen)

- *Initialisiert* die angelegten Bytes mit **`0`**

Sonst wie **`malloc`**

`strdup`

Alternativ für ***Strings***:

```
char *strdup(const char *str);
```

- Liefert eine *dynamisch angelegte Kopie* von **`str`**
  (in der für den Inhalt von **`str`** nötigen Länge)
- Macht intern im Wesentlichen
  `strlen`, **`malloc`** und **`strcpy`**
- Ergebnis so wie bei **`malloc`** auf **`NULL`** *prüfen*!

realloc (1)

Größe ändern:

```
void *realloc(void *p, size_t n_neu);
```

- *Ändert* die Größe des *bestehenden* dyn. Blockes, auf den **p** zeigt, auf **n_neu** Bytes.
- Kann einen Block *größer oder kleiner* machen.
- Die im Block vorhandenen *Daten bleiben erhalten* (eventuell gekürzt)
- Neu dazukommende Bytes sind *uninitialisiert*
- **p** gleich **NULL**: Es wird ein *neuer Block* angelegt
- **n_neu** gleich **0**: Der bestehende Block wird *freigegeben*

`realloc` (2)

Returnwert von **`realloc`**:

- Bei *kleiner* machen, oder wenn *größer machen "vor Ort"* möglich war (wenn dahinter freier Platz war):

  `p` (der *bestehende* Block)

- Bei *größer* machen, wenn ***nicht*** *"vor Ort"* möglich:

  *Pointer auf **neuen** Block!*

  Daten werden *umkopiert*, alter Block wird *freigegeben*:
  ==> Altes **`p`** ist danach *ungültig*!
  ==> *"Größer machen"* kann sehr *aufwändig* sein!

- Bei *größer* machen, wenn *kein Platz*: **NULL**
  (der bestehende Block "lebt" in diesem Fall noch!)

free

Freigeben:

void free(void *p);

Gibt den Block, auf den **p** zeigt, *frei*

- **p** *muss* auf den *Anfang* eines Blockes zeigen, der von **malloc** / **calloc** / **realloc** / **strdup** angelegt wurde (sonst Absturz)
- **p** gleich **NULL** ist erlaubt, tut *nichts*

Der freigegebene Platz wird irgendwann von **malloc** für neue Daten *wiederverwendet*

==> Nach* free *nicht mehr auf* *p *zugreifen!!!

new und **delete**

malloc, **free** usw. sind

konzeptionell für Objekte ungeeignet!

==> C++ hat *eigene Konstrukte* für dynamischen Speicher

new statt **malloc**

delete statt **free**

(**new** und **delete** sind spezielle Sprach-Konstrukte von C++,
keine Library-Funktionen wie **malloc** usw.)

Aber:
Für Daten, die ***keine** Objekte sind oder enthalten*,
darf **malloc**, **strdup** usw. *auch in C++* verwendet werden

Gefahren (1)

Dynamische Daten bieten

wenig bis keine Sicherheitsprüfungen

Falsche Zugriffe werden *nicht abgefangen*,
sondern können zu *"beliebigen Effekten"* führen:

- Falsche / *korrupte Daten*
- *Abstürze* (sofort oder *irgendwann später*)

Zur **`malloc`**-Fehlersuche gibt es *Spezial-Software*
(z.B. *"Address Sanitizer"* oder *"Valgrind"*),
diese ist aber für den Regelbetrieb *viel zu langsam*
und auch von der Nutzung her *nichttrivial*.

Gefahren (2)

Typische Fehlerquellen:

- Zugriff *nach* einem **`free`**
  (trifft inzwischen *neue* Daten oder *interne* Daten von **`malloc`**)
- *Doppeltes* **`free`** desselben Blockes
- **`free`** mit einem Pointer,
  der *nicht auf den Anfang* eines **`malloc`**-Blocks zeigt
- *Fehlende* **`NULL`**-*Prüfung* bei **`malloc`**
- Zugriff *hinter das Ende* eines **`malloc`**-Blocks
  (oder noch gefährlicher: *Vor den Anfang*!)
- Zugriff auf dynamische Daten mit *falschem Typ*

“Memory Leaks”

Der frei verfügbare Speicher *“rinnt aus”*,
d.h. wird *im Laufe der Zeit immer weniger*

==> Irgendwann findet **`malloc`** keinen Platz mehr,
liefert **`NULL`** (Programm *endet mit Fehlermeldung*)

Grund:

Vergessene* `free` *bzw. “verlorene” Daten

Es wird *immer wieder Speicher* mit **`malloc`** *angefordert*,
aber *nicht mehr freigegeben*,
obwohl man ihn *nicht mehr braucht*
(und oft *gar keinen Pointer* mehr darauf hat)

“Memory Fragmentation”

Einmal angelegte **`malloc`**-Blöcke können
nachträglich nicht mehr verschoben werden!

Wenn man
- sehr *viele Blöcke*
- in *vielen verschiedenen Größen*
- in *bunt gemischter Reihenfolge*

anlegt, freigibt und wieder anlegt,

dann bleiben zwischen den Blöcken
immer mehr kleine, nicht nutzbare Lücken als “Verschnitt”!

==> Obwohl in Summe *viele Megabytes frei* sind,
findet **`malloc`** *keinen zusammenhängenden,*
freien Speicherbereich brauchbarer Größe mehr!

`struct`'s variabler Größe (1)

Das ***letzte*** *Member* einer **`struct`** darf
ein *Array* ***unbekannter*** *Größe* sein

(deklariert mit *leeren* **`[]`**)

Dieses Member wird beim **`sizeof`** *nicht* mitgezählt:
Das **`sizeof`** liefert die Größe der **`struct`** *ohne das Array*

Mit solchen **`struct`**'s darf man
keine *lokalen oder globalen* **`struct`**-Variablen anlegen:

Man kann sie *nur* ***dynamisch*** *anlegen*

Dabei muss man die *gewünschte Array-Größe*
händisch zum **`struct`**-**`sizeof`** *dazuzählen*

`struct`'s variabler Größe (2)

Beispiel:

```
typedef struct {
    unsigned int len;
    char text[];
} meinstring;


unsigned int in_len = strlen(input);
meinstring *str_p = malloc(
  sizeof(meinstring) + in_len + 1);
str_p->len = in_len;
strcpy(str_p->text, input);
```

`alloca` (1)

Parameter & Returnwert wie **`malloc`**, aber

*der Platz wird **am Stack** reserviert*

==> Platz wird *bei Verlassen der Funktion* (**`return`**)

***automatisch** wieder **freigegeben**!*
(wie bei lokalen Variablen)

==> **Nicht** mit **`free`** freigeben!!! *(Absturz!)*

Vorteile:

- Viel *schneller / effizienter* als **`malloc`** / **`free`**
- Keine Gefahr des Vergessens von **`free`**

`alloca` (2)

Aber:

- **`alloca`** ist *nicht standardisiert*!
  (wird von einigen Compilern nicht unterstützt)
- **`alloca`** ist *in vielen Programmierrichtlinien verboten*

Historischer Grund:

`alloca` war / ist ein *Hack für variabel große lokale Arrays*

Variabel große lokale Arrays sind ...
- in C erst seit C99 erlaubt, davor nicht
- in einigen Compilern (MSVC !) bis heute nicht unterstützt
- in C++ bis heute offiziell nicht erlaubt

==> *Einzige Alternative* **`alloca`**

Dynamische Datenstrukturen (1)

... sind Datenstrukturen, die

- eine *beliebige Anzahl einzelner Elemente* aufnehmen können
- je *nach Bedarf* *"dynamisch"* *wachsen* und *wieder schrumpfen*
- *nur so viel Platz* belegen, wie wirklich gerade *Elemente benötigt* werden
- *keine fixe Obergrenze* für die *Anzahl* der Elemente haben
  (Begrenzung *nur* durch verfügbaren **`malloc`**-Speicher)

Dynamische Datenstrukturen (2)

Jedes einzelne Element

... steckt in seiner *eigenen Struktur*

... die *bei Bedarf* mit **`malloc`** *angelegt*
und wieder *gelöscht* wird

... und neben dem Element selbst (den *"Nutzdaten"*)

Pointer auf weitere Elemente
bzw. Strukturen enthält

==> *Eine der* ***Haupt-Anwendungen*** *von* **`malloc`**

Dynamische Datenstrukturen (3)

Von außen *"sieht"* man nur *ein oder zwei Elemente*
(d.h. *hat einen direkten Pointer* auf sie)

Die restlichen Elemente erreicht man,
indem man den ***Pointern ("Verkettungen")***
zwischen den Elementen folgt!

Vorteil (neben variabler Größe):

Es lassen sich (im Unterschied zum Array)
jederzeit "mittendrin"
Elemente einfügen und löschen
(ohne Umkopieren / Verschieben von Elementen)

Dynamische Datenstrukturen (4)

Die *Form der Verkettung* hängt ab
vom geplanten *Einsatzzweck*
bzw. den geforderten *Zugriffs-Operationen*

==> Wichtigstes Thema von
"Algorithmen und Datenstrukturen"
(nächstes Semester)

Wichtigste Formen dynamischer Datenstrukturen:

- ***Verkettete Liste*** (auch: *Stack, Queue, ...*)
- ***Binärer Baum***

Verkettete Listen

- ***Lineare Anordnung*** von Elementen
- *Jedes Element* enthält
  - immer einen **Pointer** auf seinen **Nachfolger:**
    Einfach *verkettete Liste*
  - eventuell einen zweiten **Pointer** auf seinen **Vorgänger:**
    Doppelt *verkettete Liste*

  Nachfolger des letzten und Vorgänger des ersten Elementes: **NULL**

- *Von außen* hat man
  - immer einen **Pointer** auf das **erste Element:** ***"head"***
  - eventuell einen zweiten **Pointer** auf das **letzte Element:**
    "tail"

  Bei *leerer* Liste: **Head** ist **NULL** (und Tail auch)

Die einfach verkettete Liste (1)

- *Durchlaufen* ist *nur vorwärts* möglich!
  Für vorwärts & rückwärts: *Doppelt* verkettete Liste!

  Beispiel für das *Durchlaufen*:

  ```
  for (p = head; p; p = p->next) { … }
  ```

- Die einfachsten und schnellsten Operationen sind
  "vorne anhängen" und *"erstes Element lesen oder löschen"*
- Einen **Tail-Pointer** braucht man nur, wenn man
  - *schnell* (ohne Durchlaufen) *hinten anhängen* will
  - oder *schnell* auf das *letzte Element* zugreifen will.
- Auch mit Tail-Pointer kann man
  ***nicht** schnell hinten löschen* (immer Durchlaufen nötig!).

Die einfach verkettete Liste (2)

- Zum *Einfügen und Löschen eines Elementes* ***mittendrin*** muss man sich beim Durchlaufen den ***Pointer auf den Vorgänger merken!***

  (oder eine *doppelt* verkettete Liste verwenden)

  Ein Pointer auf das zu löschende Element *reicht nicht!*

- Dabei *vorne* einfügen/löschen erfordert einen *Sonderfall!*

  Vermeidung des Sonderfalls eventuell mit ****-Trick**:

  Schleife mit ***"Pointer auf den zu ändernden Pointer"***:

  - Zeigt zu ***Schleifenbeginn*** auf den ***Head-Pointer***
  - … und ***dann*** auf den ***Next-Pointer des Vorgängers***

Bäume (hier "Binäre Suchbäume")

- *Jedes Element* enthält ***zwei Pointer***
  auf seine ***"Söhne"*** (linker und rechter Sohn),
  NULL wenn kein Sohn an dieser Stelle existiert
- ***Links*** hängen die Elemente mit ***kleineren*** Werten,
  rechts hängen die Elemente mit ***größeren*** Werten
- *Von außen* hat man einen ***Pointer***
  auf das ***oberste Element:*** Die ***"Wurzel"***
  (= einziges Element, das von niemandem Sohn ist)

Beispiel für das *Durchsuchen nach Wert* **`val`**:

```
for (p = root;
     p && (p->data != val);
     p = (p->data < val) ? p->right : p->left) { }
```

Bit-Operationen

- Die Bit-Operatoren und ihre Tücken
- Typische Bitmuster
- Bits ausschneiden, löschen, setzen, ...
  Bits testen
  Kombinierte Bit-Operationen
- Bit-Operationen für Rechnungen
- Schleifen über Bits, Beispiel
- Bitfield-`struct`'s

Die Bit-Operatoren

- **`a & b`**: Bitweises **UND** (1 wenn *beide* 1)
- **`a | b`**: Bitweises **ODER** (1 wenn *mindestens einer* 1)
- **`a ^ b`**: Bitweises **XOR** (exklusives ODER: *genau einer* 1)
- **`~a`**: Bitweises **NICHT** (*invertiert*: 1 wird 0, 0 wird 1)
- **`a << i`**: **`a`** um **`i`** Bits **nach links geschoben**
- **`a >> i`**: **`a`** um **`i`** Bits **nach rechts geschoben**

  (Heißen *"Bit Shift Operations"*)
- Zahl beginnend mit **`0x`**... : *Hexadezimalzahl*
  (z.B. **`0xff`** = ...**`000 1111 1111`** = **`255`**)
  (weiters: **`0`**... : *Oktalzahl*, **`0b`**... : *Binärzahl* (nur mit **`gcc`**!))

Bit-Operationen: Funktion

Verarbeiten die Bits *einzeln* und *unabhängig voneinander*
(von **`int`**, **`char`**, **`long`**, ..., aber *nicht* von **`double`**, Pointern, ...):

Beispiel: **`c = a & b;`**

1. Bit von **`c`** = (1. Bit von **`a`**) UND (1. Bit von **`b`**)
2. Bit von **`c`** = (2. Bit von **`a`**) UND (2. Bit von **`b`**)

...

32. Bit von **`c`** = (32. Bit von **`a`**) UND (32. Bit von **`b`**)

Beispiel: **`a = ~a;`**

==> *Jedes einzelne Bit* von **`a`** wird invertiert

Beispiel: **`a <<= 8;`**

==> Jedes einzelne Bit von **`a`** rutscht *um 8 Plätze nach vor*

==> Die *vordersten* 8 Bits von **`a`** *gehen verloren*

==> *Hinten* kommen 8 *neue* **`0`**-*Bits dazu*

Bit-Operationen: Achtung! (1)

- *Nicht mit logischen Operationen verwechseln!*

  `&&  ||  !` funktionieren *ganz anders!*

- **`&  |  ^`** haben *falschen Vorrang:*
  - *Unter* den Vergleichen **`<  <=  >  >=  ==  !=`**
  - *Unter* den Bit Shifts **`<<  >>`**

  ==> *Immer einklammern!!!*

  Sonst z.B.:

`a & 1 == 0`	wird zu	**`a & (1 == 0)`**
`a != b & 0xff`	wird zu	**`(a != b) & 0xff`**
`a & 0xff << 16`	wird zu	**`a & (0xff << 16)`**

Bit-Operationen: Achtung! (2)

- **<<** schiebt rechts immer **0**-Bits nach ==› kein Problem...
- Was **>>** nachschiebt, hängt vom *Typ der Operanden* ab!
  - Operanden sind **unsigned**
    ==› links kommen immer **0**-*Bits* nach
  - Operanden sind normale **int** (**signed**)
    ==› links kommen ***Kopien des vordersten Bits*** nach!

    Weil: Das Bit ganz links ist das *Vorzeichen-Bit, wird erhalten*:

    Ursprüngliche Zahl **<0** ==› Bit ganz links ist **1**
    ==› es kommen **1**-Bits nach ==› Ergebnis ist auch **<0**

    Ursprüngliche Zahl **>=0** ==› Bit ganz links ist **0**
    ==› es kommen **0**-Bits nach ==› Ergebnis ist auch **>=0**

Bit-Operationen: Achtung! (3)

Daher bei **<<** und **>>** fast immer notwendig und sinnvoll, wenn es um Bits und nicht um Vorzeichen-Zahlen geht:

- *Variablen* für Bit-Operationen **unsigned** *deklarieren*
- oder Operanden explizit *auf* **unsigned** *umwandeln*

  Beispiel: `a |= ((unsigned int) i) >> 8;`
- oder **unsigned**-Konstanten verwenden (**U** hinten dran)

  Beispiel: `a = 0xffffffffU >> i;`
  Besser: `a = -1U >> i;` *oder* `a = ~0U >> i;`
  -1 und **~0** bestehen beide aus *lauter* **1**-*Bits*
  -1U schaut zwar blöd aus, ist aber zulässig

Typische Bitmuster

Sei im Folgenden *Bit 0 ganz rechts*, *Bit 31 ganz links*:

- `1 << i` `i`-tes Bit ist `1`, sonst alle `0`
- `~(1 << i)` `i`-tes Bit ist `0`, sonst alle `1`
- `(1 << i) - 1` oder `~(-1 << i)` oder `~(~0 << i)`
  `i` viele Bits hinten sind `1`, davor alle `0`
- `~((1 << i) - 1)` oder `-1 << i` oder `~0 << i`
  `i` viele Bits hinten sind `0`, davor alle `1`
- `-1U >> i` oder `~0U >> i`
  `i` viele Bits vorne sind `0`, dahinter alle `1`
- `~(-1U >> i)` oder `~(~0U >> i)`
  `i` viele Bits vorne sind `1`, dahinter alle `0`

Bits ausschneiden

Bits ausschneiden mit **&**:

```
neu = alt & maske;
```

- Dort wo **maske** ein **1**-Bit enthält:

  neu enthält *das ursprüngliche Bit* aus **alt**.

- Dort wo **maske** ein **0**-Bit enthält:

  neu enthält **0**, egal, welches Bit in **alt** war.

Beispiele:

```
b = a & 0xff;  // b enthält die letzten 8 Bits von a, sonst 0
b = a & (1 << i);   // b enthält nur das i-te Bit von a
```

Bits löschen

Analog bzw. genau invertiert: **Bits löschen** mit **&** und **~**:

```
neu = alt & ~maske;
```

- Dort wo **maske** ein **1**-Bit enthält:

  neu enthält **0**, egal, welches Bit in **alt** war.

- Dort wo **maske** ein **0**-Bit enthält:

  neu enthält *das ursprüngliche Bit* aus **alt**.

Beispiele:

```
a &= ~0xff; // löscht die letzten 8 Bits, Rest bleibt gleich
a &= ~(1 << i);  // löscht das i-te Bit, Rest bleibt gleich
```

Bits setzen

Bits setzen mit **|**:

```
neu = alt | maske;
```

- Dort wo **maske** ein **1**-Bit enthält:

  neu enthält **1**, egal, welches Bit in **alt** war.

- Dort wo **maske** ein **0**-Bit enthält:

  neu enthält *das ursprüngliche Bit* aus **alt**.

Beispiele:

```
a |= 0xff;   // setzt die letzten 8 Bits, Rest bleibt gleich
a |= 1 << i;   // setzt das i-te Bit, Rest bleibt gleich
```

Teile zusammensetzen

Analog: **Bit-Werte zusammensetzen** mit **|**:

```
neu = teil1 | teil2;
```

- **neu** enthält die **1**-Bits aus **teil1** und aus **teil2**.

Beispiele:

```
addr = (pageNum << 12) | offset;
dblVal = signBit | exponent | baseVal;
```

Bits umdrehen

Bits umdrehen mit **^**:

```
neu = alt ^ maske;
```

- Dort wo **maske** ein **1**-Bit enthält:

  neu enthält *das invertierte Bit* aus **alt**.

- Dort wo **maske** ein **0**-Bit enthält:

  neu enthält *das ursprüngliche Bit* aus **alt**.

Beispiele:

```
a ^= 0xff; // invertiert die letzten 8 Bits, Rest bleibt gleich
a ^= 1 << i;   // invertiert das i-te Bit, Rest bleibt gleich
```

Prüfung von Bits

`if ((a & 1) != 0)`	Ist das *letzte* Bit von **a** gleich **1** ?
`if ((a & (1 << i)) != 0)`	Ist das **i**-te Bit von **a** gleich **1** ?
`if ((a & (1 << i)) == 0)`	Ist das **i**-te Bit von **a** gleich **0** ?
`if ((a & 0xff00) == 0)`	Ist das *2. Byte* von **a** gleich **0** ? (Bit 8 bis Bit 15 alle gleich **0** ?)
`if ((a & 0xff) == 0xff)`	Hat das *letzte Byte* von **a** nur **1** ? (Bit 0 bis Bit 7 alle gleich **1** ?)
`if ((a >> 24U) == 0xff)`	Hat das Byte vorne in **a** nur **1** ?

In der Praxis auch:

`if (a == -1)` ...	Sind *alle* Bits von **a** gleich **1** ?
`if (a < 0)` ...	Ist das *vorderste* Bit von **a** gleich **1** ? (das vorderste Bit ist das *Vorzeichen*!)
	Achtung: Klappt *nicht* für **unsigned a** !!!

Kombinierte Bit-Operationen

- **Eine Zahl ausschneiden** mit **&** und **>>** :

```
b = (a & 0x00fff00) >> 8U;
b = (a >> 8U) & 0x0000ffff;
```

  Beides: **b** enthält die Zahl in den *mittleren 16 Bits* von **a**

- **Eine Zahl einsetzen** mit **<<** und **|** :

```
a |= b << 8;
```

  wenn die Bits in **a** schon **0** sind, sonst:

```
a = (a & 0xff0000ff) | (b << 8);
```

  Fügt **b** (max. 16 Bit) ab dem 8. Bit in **a** ein

- Ausschneiden und wieder zusammenbauen:

```
reverse = (original << 24U) |
          ((original & 0x0000ff00) << 8U) |
          ((original & 0x00ff0000) >> 8U) |
          (original >> 24U);
```

Bitrotationen (1)

C / C++ kennt (im Unterschied zu Assembler)

keine Bitrotationen

Man muss sie aus 3 Schritten zusammensetzen:

- *Vorderen* Teil nach *hinten* schieben
- *Hinteren* Teil nach *vorne* schieben
- Beide Ergebnisse mit | *zusammensetzen*

Für Fortgeschrittene:
Die meisten Compiler bieten *vordefinierte Funktionen*,
die direkt in Rotations-Maschinenbefehle übersetzt werden.

Bitrotationen (2)

Beispiel:

"Rotiere **a** um **i** Stellen nach links"
(**a** ist ein 32 Bit **unsigned int**)

```
a = (a << i) | (a >> (32 - i));
```

Trickreiche Bitoperationen

- **`-a & a`**

  liefert einen Wert,
  der *nur das hinterste* **`1`**-Bit von **`a`** enthält (sonst alles **`0`**)

- **`a = a & (a - 1);`**

  setzt das *hinterste* **`1`**-Bit von **`a`** *auf* **`0`**
  und lässt den Rest von **`a`** unverändert

- **`if ((a & (a - 1)) == 0)`**

  prüft daher, ob **`a`** *genau eine Zweierpotenz* (oder **`0`**) ist,
  d.h. ob *kein oder genau ein Bit* in **`a`** *gesetzt* ist

Bitoperationen für Rechnungen (1)

<< und **>>** werden oft verwendet,
um *mit Zweierpotenzen zu multiplizieren / dividieren:*

Grund: **<<** und **>>** sind *schneller* als ***** und **/** ,
und C / C++ kennt ja keinen *"hoch"*-Operator für **`int`**.

- **`a << i`** entspricht "**a** mal (**2** hoch **i**)"
  `a << 1` entspricht daher "**a** mal **2**"
  `1 << i` entspricht daher "**2** hoch **i**"
- **`a >> i`** entspricht "**a** geteilt durch (**2** hoch **i**)"
  `a >> 1` entspricht daher "**a** geteilt durch **2**"

Deshalb auch das seltsame "vorzeichenerhaltende" **>>**-Verhalten
für **`signed`**-Werte: **`a >> i`** dividiert auch *negative* **a** richtig!

Bitoperationen für Rechnungen (2)

- **`a & ((1 << i) - 1)`**

  liefert "**a** modulo (**2** hoch **i**)"
  (*Rest* bei Division durch (**2** hoch **i**))

- **`a & 1`** liefert daher **1** wenn **a** *ungerade*, **0** wenn **a** *gerade*

- **`a & ~((1 << i) - 1)`**
  (oder z.B. **`a & ~0xff`**)

  schneidet **a** auf das *nächstkleinere Vielfache* von (**2** hoch **i**) *ab*

- **`(a + ((1 << i) - 1)) & ~((1 << i) - 1)`**
  (oder z.B. **`(a + 0xff) & ~0xff`**)

  rundet **a** auf das *nächste Vielfache* von (**2** hoch **i**) *auf*

Typische Bit-Schleifen

- Schiebe eine *Ein-Bit-Maske* der Reihe nach *über alle Bits*:

  ```
  for (m = 1; m; m <<= 1) { ... }
  ```

  Die Schleife *endet*, wenn das eine Bit in **m** *"vorne rausfällt"*
  (==> **m** wird dadurch **0**)

- Bring alle Bits von **n** der Reihe nach an *die letzte Stelle*,
  bis alle **1**-Bits von **n** hinten waren und verarbeitet wurden:

  ```
  while (n > 0) {
      ...  // verarbeite das hinterste Bit von n
      n >>= 1;
  }
  ```

C23 Bitfunktionen

Seit **C23** gibt es **`stdbit.h`**
mit praktischen ***Bit-Funktionen***,
die häufige Schleifen für Bits ersetzen:

- 1- oder 0-Bits insgesamt *zählen*
- *Führende / abschließende* 1- oder 0-Bits *zählen*,
  Position des ersten / letzten 1- oder 0-Bits
- *Prüfen* auf *Zweierpotenz*,
  auf- oder abrunden auf Zweierpotenz

Je nach Prozessor:
Mit *speziellen Assembler-Befehlen* implementiert!

Beispiel - Aufgabe

“Schreib eine Funktion,

die einen **`int`**-Wert **`n`** bekommt

und die ***Parität dieses Wertes*** zurückliefert:

`0` … wenn **`n`** eine ***gerade Anzahl*** von **`1`**-Bits
(oder *gar keine* **`1`**-Bits) enthält

`1` … wenn **`n`** eine ***ungerade Anzahl*** von **`1`**-Bits
enthält”

Beispiel - Lösungsidee 1

- Das ***Xor*** einer *geraden* Anzahl von **1**-Bits liefert **0**,
  das Xor einer *ungeraden* Anzahl von **1**-Bits liefert **1**,
  ein Xor mit einem **0**-Bit *ändert nichts* am Ergebnis.

  ==> **Das Ergebnis ist das *Xor* aller Bits von n**
  (wobei man **0**-Bits *weglassen* darf)

- Für die Verknüpfung mit Xor
  brauchen wir alle Bits von **n** *an derselben Stelle*.

  ==> Wir berechnen das ***Xor des jeweils hintersten Bits***
  und ***schieben alle Bits*** *der Reihe nach* ***nach hinten***.

- Wir können *aufhören*, wenn keine **1**-Bits mehr da sind
  (d.h. wenn **n** gleich **0** geworden ist).

Beispiel - Lösung 1

```
// unsigned, damit >> vorne sicher 0-Bits nachschiebt!
int parity(unsigned int n) {
   int x = 0;  // Das Xor-Ergebnis, muß am Anfang 0 sein
   while (n != 0) {  // solange n noch 1-Bits enthält:
      // (n & 1) liefert nur das hinterste Bit von n
      // ^= Xor-verknüpft dieses Bit mit x
      x ^= (n & 1);
      // schiebt n um ein Bit nach rechts:
      // das soeben verknüpfte hinterste Bit fällt weg
      // und das nächste Bit kommt an die hinterste Stelle
      n >>= 1;
   }
   return x;
}
```

Beispiel - Lösungsidee 2

- Die Parität ist der ***Rest*** von

  "Anzahl der **1***-Bits in* **n** *dividiert durch* **2***"*

  ==> Wir müssen die **1**-Bits in **n** ***zählen***

  ==> Dazu ***schieben*** wir eine ***Maske*** **m** ***mit einem*** **1***-Bit*
  der Reihe nach *über alle Bits* von **n**
  und prüfen jedes Bit von **n** *einzeln*

- Die Schleife *endet*,
  wenn das **1**-Bit aus der Maske *herausfällt*
  und die Maske dadurch **0** wird

- *"Rest von* **anzahl** *dividiert durch* **2***"*
  ist dasselbe wie das *letzte Bit* von **anzahl** .

Beispiel - Lösung 2

```
int parity(int n) {
   int anzahl = 0;
   // In m ist genau ein Bit gesetzt: Am Anfang das letzte Bit
   // Dieses 1-Bit wandert nach vor, bis es vorne rausfällt
   for (int m = 1; m != 0; m <<= 1) {
      // Betrachtet von n nur das eine Bit an der Stelle,
      // an der gerade das 1-Bit von m ist,
      // und prüft dieses Bit auf ungleich 0
      if ((n & m) != 0) { ++anzahl; }
   }
   // (anzahl & 1) liefert nur das letzte Bit von anzahl
   return (anzahl & 1);
}
```

Beispiel - Lösung 3

```
int parity(int n) {
   int anzahl = 0;
   // dieselbe Idee, aber ohne Masken-Variable:
   // i ist die Position des Bits, das geprüft werden soll
   for (int i = 0; i < 8 * sizeof(int); ++i) {
      // Betrachtet von n nur das eine Bit an der Stelle i
      // ( (1 << i) liefert ein einzelnes 1-Bit an Stelle i)
      // und prüft dieses Bit auf ungleich 0
      if ((n & (1 << i)) != 0) { ++anzahl; }
   }
   // (anzahl & 1) liefert nur das letzte Bit von anzahl
   return (anzahl & 1);
}
```

Bitfield **`struct`**'s (1)

Bitfield **`struct`** = Eine **`struct`** bei der

- *alle* Member **`int`** oder **`unsigned int`** sind
- hinter *jedem* Member ***seine Breite in Bits*** steht

==> Member liegen *bitweise nebeneinander* in 32 / 64 bit

Anwendung:

- Abbilden der *Inhalte von Hardware-Registern*
  (die **`struct`**-Variable wird dann mit Pointer-Tricks *auf einen I/O-Baustein* statt ins RAM gelegt)
- *"Verpacken" mehrerer* kurzer **`int`**'s
  in *einen* 32- oder 64-bit-Speicherplatz

Bitfield **`struct`**'s (2)

Beispiel:

```
struct sensorData {
    unsigned int sec : 6;
    unsigned int min : 6;
    unsigned int hour : 5;
    int : 5;    // namenloses "Platz-Füll-Member":
                // 5 unbenutzte Bits
    int valid : 1;
    int error : 1;
    int data: 8;
};
```

Bitfield **`struct`**'s (3)

Achtung:

Bitfield **`struct`**'s sind standardisiert,
aber ***nicht portabel!***

Grund:

Der Standard überlässt wesentliche Dinge
dem Compiler!

- *Anordung* der Member *von links* oder *von rechts*???
- Gesamt-Größe, *Alignment*, Breite des Basis-Typs, Behandlung Alignment-überschreitender Member, ...
- Ist ein "nur **`int`**"-Bitfield **`signed`** oder **`unsigned`** ?

Bitfield **`struct`**'s (4)

Daher:

Wenn die Anordnung der Bits
exakt fixiert sein muss
(z.B. für I/O-Bausteine)

... und der Code portabel
(***plattformunabhängig***) sein soll:

Besser die Bits selbst mit **`&  |  <<  >>`**
einpacken und auspacken

und *keine* Bitfield **`struct`** verwenden!

Der Präprozessor

- Prinzip
- `#include`
- `#define` und `#undef`,
  `#define` mit Parametern und Tricks
- `#if`
- `#error`
- Vordefinierte Makros

Der Präprozessor (1)

- … läuft in den meisten Implementierungen
  beim Kompilieren als *eigenständiges Programm* (**cpp**)
  vor dem eigentlichen Compiler
- … macht ***"Textverarbeitung"*** auf dem Programmtext:
  - Files *einfügen*
  - Texte *ersetzen*
  - Codeteile *bedingt weglassen*
- … erzeugt *wieder C-Code* (lesbar!),
  der dann vom eigentlichen Compiler kompiliert wird
- … wird gesteuert von *allen Zeilen mit* **#** *am Anfang*

Achtung: Präprozessor-Befehle sind *keine C-Befehle*,
daher *kein* **;** am Ende von #-Zeilen!

Der Präprozessor (2)

- … verarbeitet den Programmtext als *"strukturlose" Folge von Namen, Konstanten und Sonderzeichen,* ***ohne*** deren tiefere Bedeutung in C zu kennen

Achtung: Der Präprozessor hat ***keine Ahnung von C*** (von C-Befehlen, Funktionen, Rechnungen, …)

==> Man kann auch *ganz andere Dinge als C* mit dem Präprozessor verarbeiten (wird auch hin und wieder gemacht)

==> Der Präprozessor hat keine Hemmungen, ohne Fehlermeldung *beliebig falschen oder schwachsinnigen C-Code* zu erzeugen, wenn man ihm das anschafft!

`#include` (1)

Siehe Kapitel “Grundbausteine”:

Die **`#include`**-Zeile
wird *durch den Inhalt* des angegebenen Files
ersetzt

Siehe nächstes Kapitel:

Dient zum Inkludieren von *Header-Files* (**`.h`**)

Das Inkludieren von **`.c`**-Files
ist meistens eine *ganz schlechte Idee*!

#include (2)

2 Formen:

- **#include** <*filename*>

  Filename relativ zum *System-Header-Verzeichnis*
  (für *vordefinierte* Header-Files)

- **#include** "*filename*"

  Filename relativ zum *aktuellen Verzeichnis*
  (für Header-Files des *eigenen Projektes*)

Pfadnamen (mit Unterverzeichnissen) sind erlaubt:

```
#include <sys/stat.h>
#include "../includes/myhdr.h"
```

`#define` (1)

`#define` definiert ein ***"Makro"*** = eine ***"Abkürzung"***

Alle Vorkommen des Makros im C-Programm
nach dem **`#define`** werden ***textuell ersetzt***
durch die rechte Seite des **`#define`**
(bis zum Zeilenende)

Stilistisch üblich (aber nicht Pflicht):

Makro-Namen werden
komplett groß geschrieben
(mit **_** zur Wort-Trennung)

#define (2)

2 Formen:

- Makro-***Konstanten***: **#define** ***name ersatztext***

  Beispiel: **#define LINE_LEN (80 + 2)**

  Achtung: Kein **=** zwischen ***name*** und ***ersatztext***!

- Makros mit *Parametern* (“***Funktions-Makros***”):

  #define ***name*(*parameter*) *ersatztext***

  Beispiel:
  #define MAX(a, b) (((a) > (b)) ? (a) : (b))

Befehlszeilen-Makros

Fast alle Compiler bieten die Möglichkeit,

Makros *beim Compiler-Aufruf*
auf der *Befehlszeile* zu definieren

Dadurch kann man

denselben Code unterschiedlich compilieren,
ohne ihn zu ändern

Beispiel:

```
gcc ... -D DEBUG_LEVEL=2 ...
```

#undef

Die Anweisung

#undef ***name***

löscht ein bestehendes **#define** ***name*** **…**

- Ab dieser Stelle wird ***name*** *nicht mehr ersetzt*
  bzw. ist ***name*** *unbekannt*
- Verwendung u.a., wenn man *danach*
  ein *neues (anderes)* **#define** für ***name*** machen will

  (denn einfach *“drüberdefinieren”* ist *verboten*!)

Funktions-Makros (1)

Funktionsmakros sind meist *etwas schneller*
als echte Funktionen

Grund:

- Der *Aufruf-Overhead fällt weg*
  Der "Aufruf" wird ja "vor Ort" durch den Code ersetzt
- Der Compiler kann den Code *besser optimieren*

Aber (bei *langer rechter Seite* und oftmaliger Verwendung):

Der Programmcode wird *etwas größer*

Grund:

Die rechte Seite wird *an vielen Stellen eingesetzt*
es wird *jedesmal wieder Code* dafür erzeugt

Funktions-Makros (2)

Die ***Argumente** im Aufruf* eines Funktions-Makros
werden Wort für Wort
rein textuell und "ohne Hirn"
statt den Parametern *im Ersatztext **eingesetzt***

==> Argument-Werte werden *nicht zuerst ausgerechnet*
und dann als Wert eingesetzt,
sondern (ev. mehrmals!) ***unausgerechnet** eingesetzt*

==> Es findet *keinerlei Typ-Prüfung* o.ä. statt!
(deshalb gibt es auch keine Typ-Deklaration der Parameter)

==> Funktions-Makros haben

viele ganz gemeine Fehlerquellen!!!

Seiteneffekte in Makros (1)

Enthält der Ersatztext einen Parameter ***mehrmals***,
wird bei Argumenten mit Seiteneffekten
(Funktionsaufrufe, vor allem aber **++** und **--**)
der Seiteneffekt ***entsprechend oft ausgeführt***!

Beispiel (für unser **MAX** oben):

```
max = MAX(max, i++);
```

ergibt

```
max = (((max) > (i++)) ? (max) : (i++));
```

Seiteneffekte in Makros (2)

Effekt:

- **`i`** wird je nach Ausgang des Vergleiches
  um 1 ***oder um 2*** *erhöht*
- Auf **`max`** wird der *erhöhte* Wert von **`i`** zugewiesen
  (trotz **`i++`**, d.h. *"nimm noch den alten Wert von* **`i`***"*)!

==> Makros mit *mehrfachen Vorkommen eines Parameters*
wenn möglich überhaupt *vermeiden!!!*
(sonst zumindest unübersehbar *dokumentieren*!)

==> *Seiteneffekte in Makro-Argumenten*
(**`++`**, **`--`**, Funktionsaufrufe)
wenn möglich *vermeiden!*

Die äußeren ( ) (1)

Ist der Ersatztext nicht
als Ganzes komplett geklammert,
kann er auf nicht geplante Art
mit der Aufrufsstelle zusammenwirken!

Beispiel:

```
#define DIGIT_VAL(c)  (c) - '0'

n = 10 * DIGIT_VAL(str[i]);
```

ergibt

```
n = 10 * (str[i]) - '0';
```

Die äußeren ( ) (2)

Effekt (auf Grund der Vorrang-Regeln):

```
n = (10 * str[i]) - '0';
```

statt wie gewünscht

```
n = 10 * (str[i] - '0');
```

==> *Makro-Ersatztext* (rechte Seite) immer

komplett in ( ) schreiben!!!

Richtig:

```
#define DIGIT_VAL(c)  ((c) - '0')
```

Die inneren ( ) (1)

Fehlende Klammern im Ersatztext
können *bei mehrteiligen Argumenten*
zu *unerwarteten Berechnungs-Reihenfolgen*
im entstehenden Code führen:

Beispiel:

```
#define SQR(x) (x * x)


q = SQR(i + 1);
```

ergibt

```
q = (i + 1 * i + 1);
```

Die inneren ( ) (2)

Effekt:

```
q = 2 * i + 1;
```

statt wie gewünscht

```
q = (i + 1) * (i + 1);
```

==> Jedes einzelne *Vorkommen eines Parameters* im Makro-Ersatztext immer

einzeln in ( ) einschließen!!!

Richtig:

```
#define SQR(x) ((x) * (x))
```

Mehrzeilige Makros

Wenn der *Ersatztext* eines **#define**
zu lang für den Rest der Zeile ist:

Mit \ unmittelbar gefolgt vom *Zeilenvorschub*
kann man den Makro-Text
in die nächste Zeile *fortsetzen*

Beispiel:

```
#define LONG_MACRO(a, b) \
  (someFunctionCall(a) + \
   someFunctionCall(b))
```

Variabel viele Parameter

Der C99-Standard definiert auch Makros mit

`...` am *Ende der Parameter-Liste*
(für *variabel viele* Parameter)

`__VA_ARGS__` im *Ersatztext*
wird durch *alle* **`...`**-Argumente ersetzt

Anwendung: z.B. für Makros, die

mit beliebig vielen Parametern aufgerufen werden

und

intern ein **`printf`** *mit allen diesen Parametern* aufrufen

Geschachtelte Makros

Nach der Ersetzung eines Makros
wird der eingesetzte Ersatztext *nicht sofort ausgegeben*,
sondern *wieder nach Makros abgesucht*

Diese werden *wieder ersetzt* usw.,
bis kein Makro mehr im eingesetzten Text vorkommt

==> Makros können *geschachtelt* verwendet
bzw. aufgerufen werden:

Sowohl *im Ersatztext*
als auch *in den Aufrufs-Argumenten*
dürfen *wieder Makros* vorkommen

Aber: *Rekursion* ist *keine* gute Idee...

Das String-#

Ein **#** im Ersatztext *vor einem Parameter-Namen*
bewirkt, dass der Argument-Text *nicht als Programm-Code*,
sondern *als String-Konstante* in **"..."** eingesetzt wird

(so setzt **`assert`** beispielsweise
die Bedingung als Ausgabe-Text in ein **`printf`** ein)

Notwendigkeit für dieses Konstrukt:

Makro-Parameter innerhalb von **" "** *werden nicht ersetzt!*

Funktioniert *nicht* (**par** in **"** ... **"** wird unverändert ausgegeben):

```
#define PAR_TEST(par) puts("Aufruf mit par.")
```

Funktioniert:

```
#define PAR_TEST(par) puts("Aufruf mit " #par ".")
```

(weil mehrere **"..."** automatisch *zu einem String verbunden* werden)

Das Konkatenations-##

Ein ## im Ersatztext bewirkt,
dass die *Namen vor und nach* dem ##
zu einem *einzigen* Namen *verschmolzen* werden

(z.B. zur Erzeugung neuer Variablennamen)

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] = {
  COMMAND(quit),
  COMMAND(help), …
};
```

liefert

```
struct command commands[] = {
  { "quit", quit_command },
  { "help", help_command }, …
};
```

`inline` statt Makros

In modernem Code sollten *statt Funktions-Makros*

immer **`inline`**-*Funktionen verwendet werden!*

- *Genauso schnell* (wird auch “vor Ort” eingesetzt)
- *Typgeprüft*
- *Besser lesbar*
- Die drei genannten Fehler können *nicht auftreten*

Problem: **`inline`** ist

sehr spät *nachträglich* zum C-Standard dazugekommen

- Alte / kleine Compiler haben *teilweise Probleme* damit
- Mehrere *leicht inkompatible Implementierungen* in Umlauf…

`#if` Grundform

- Allgemeine Form:

  `#if` ***bedingung***

  ...

  `#endif`

  Die ***bedingung*** kann eine *beliebige Bedingung* sein,
  die ***zur Compile-Zeit*** *berechenbar* ist

  (d.h. im Wesentlichen aus *Makros*, *Konstanten*
  sowie *Rechenzeichen* und *Vergleichen* besteht)

- Es gibt auch

  `#elif` ***bedingung***

  `#else`

`#if` Sonderformen

- Speziell für **`#if`**-Bedingungen gibt es **`defined(`*name*`)`**

  Liefert genau dann **`true`**, wenn ***name*** zuvor
  mit **`#define`** *als* ***Makro*** *definiert* wurde
  (egal mit welchem Wert bzw. Ersatztext)

- In der Praxis kommen statt **`#if`**
  fast nur die **`#if`**-***Kurzformen*** vor:

  - **`#ifdef`** ***name*** entspricht **`#if defined(`*name*`)`**

    *"Ist **name** ein definiertes Makro?"*

  - **`#ifndef`** ***name*** entspricht **`#if !defined(`*name*`)`**

    Gegenteil davon: *"Ist **name** undefiniert?"*

`#if` Wirkung

- Ist die ***Bedingung erfüllt***,

  bleiben die *Zeilen* zwischen **`#if`** und **`#endif`** *erhalten*
  (bzw. die Zeilen zwischen **`#if`** und **`#elif`** oder **`#else`**)

  … und die Zeilen *nach* **`#else`** werden *entfernt*

- Ist die ***Bedingung falsch***,

  werden die *Zeilen* zwischen **`#if`** und **`#endif`** *entfernt*
  (völlig *ignoriert*, *übersprungen*)

  Auch Präprozesser-Anweisungen (**`#include`**, **`#define`**)
  in den übersprungenen Zeilen werden *ignoriert*

  Falls vorhanden,
  bleiben *stattdessen* die **`#else`**-Zeilen *erhalten*

`#if` Beispiel & Hinweise

```
#ifdef DEBUG_LEVEL
   printf("Schleife i = %d\n", i);
#endif
```

- In **`#if`**-Bedingungen gilt wie üblich
  wenn eine *Zahl als Bedingung* verwendet wird:

  `0` ist ***falsch***, *alles andere* gilt als ***wahr***

- Mehrere **`#if`** usw. ... **`#endif`**
  können *ineinander geschachtelt* werden

  (*Einrückung* bevorzugt zwischen **#** und **`if`**)

`#if 0` zum Auskommentieren

Sehr häufiger Trick:

Statt `/* ... */` werden *Codestücke auskommentiert* mit

```
#if 0
...
#endif
```

`0` ist *immer falsch* ==> Die ...-Zeilen werden *immer entfernt*

Vorteil gegenüber `/* ... */`:

- Funktioniert auch, wenn ... bereits `/* ... */` *enthält*, denn `/* ... */` kann man *nicht ineinander schachteln*!
- *Optisch* leichter zu finden?!

#error

#error *text*

gibt beim Kompilieren eine Fehlermeldung aus,
die u.a. ***text*** enthält.

Beispiel:

```
#if defined(BIG_ENDIAN)
...
#elif defined(LITTLE_ENDIAN)
...
#else
#error BIG_ENDIAN or LITTLE_ENDIAN must be set
#endif
```

Seit **C23** auch: **#warning *text***

Vordefinierte Makros (1)

Es gibt viele *(systemspezifische) Makros*,
die der Compiler automatisch *vorab definiert*, u.a. für

- den unterstützten *C-Standard*: **`__STDC_VERSION__`**
- den *Compiler* und die *System-Plattform*:
  z.B. **`__GNUC__`** und **`__linux__`**
- die *HW-Plattform* und *Prozessor-Features*:
  z.B. **`__x86_64__`** oder **`__SSE2__`**
- Eigenschaften von *Datentypen*:
  z.B. **`__LP64__`**: *"`long`'s and Pointers are 64 Bit"*

Zweck: Zur Prüfung mit **`#if`**
(Plattform-*Unterscheidung* in universellem Code)

Vordefinierte Makros (2)

Pseudo-Makros mit *variablem* Wert:

- **__DATE__** und **__TIME__**:
  Aktuelle Zeit beim Kompilieren (als String)
- **__FILE__**: Aktueller *Filenamen* (als String)
- **__LINE__**: Aktuelle *Zeilennummer* (als Zahl)

Das Makro **assert(...)** verwendet z.B. **__FILE__** und **__LINE__** um die Position des fehlgeschlagenen **assert** im Code auszugeben

U.a. für solche Makros:

Alle Namen *beginnend* mit **_** (besonders **__**)
sind *vom Standard reserviert*!

`.c`-Files und `.h`-Files

- Motivation
- Aufteilung des Codes
- Inhalt von `.h` und `.c`
- Wer inkludiert was?
- Schutz vor doppeltem Include

Mehrere Files

Der Code *größerer C-Projekte* ist meist

auf ***mehrere*** **`.c`**-Files verteilt

Bei wirklich großen Projekten (z.B. Linux-Kernel):

Einige 10000 Files!

Offensichtlicher Vorteil:

Die *einzelnen Files* behalten eine *überschaubare Größe*
(sinnvoll: einige *hundert*, ev. wenige *tausend* Zeilen),

trotz Gesamt-Projekt-Größen
von vielen Millionen Zeilen

Vorteile (1)

- ***Verteilung auf mehrere / viele Programmierer:***

  Ein File kann *nicht gleichzeitig*
  von mehreren Programmierern *editiert* werden

  ==> Jeder Programmierer bekommt ein paar Files

  ==> Jeder File wird (zu einem Zeitpunkt)
  nur von einem Programmierer bearbeitet

  ==> Viele Programmierer können trotzdem
  gleichzeitig ohne Kollisionen arbeiten

Vorteile (2)

- ***Modulare, strukturierte Entwicklung:***

  Jede Komponente kann *für sich entworfen*, *implementiert* und teilweise auch getestet werden.

- ***Klare Schnittstelle:***

  Bei jedem File gibt es eine *klare Trennung*

  zwischen ***öffentlicher Schnittstelle***
  (was aus dem Files ist
  für andere sichtbar / verwendbar?)

  und ***geheimem Innenleben***
  (vor direktem Zugriff aus anderen Files geschützt!)

Vorteile (3)

- ***Compilezeit-Ersparnis:***
  - Bei Änderungen müssen *nur die geänderten Files frisch kompiliert* werden, nicht der gesamte Source (nur ein paar hundert statt Millionen Zeilen!)
  - Beim Kompilieren des gesamten Projektes können *viele Compiler-Aufrufe gleichzeitig* (parallel) laufen
- ***Automatische Dokumentation:***

  Wird *pro File generiert* / aktualisiert
  und ist automatisch *nach Files strukturiert*

  ==> Weniger Generierungs-Aufwand bei Änderungen

  ==> Aussagekräftiges Erzeugungs- / Änderungsdatum

Vorteile (4)

- ***Leichtere Fehlersuche:***

  Wenn am Tag X noch alles funktioniert hat:

  Neue Fehler können *nur in Files* stecken,
  die *seit Tag X geändert* wurden

  ==> Wesentliche *Eingrenzung des Suchbereiches*!

- ***Sinnvollere Versionsverwaltung:***
  - *Versionsstand und Änderungsgeschichte* sind *für jeden File separat* dokumentiert
  - *Einzelne Files* lassen sich im Notfall *auf einen älteren Stand* zurücksetzen, ohne dabei Änderungen in anderen Files zu verlieren

Aufteilung in Files

In *objektorientierten Sprachen*
meist einfach bzw. implizit vorgegeben:

Ein File pro Klasse

(ev. incl. deren interne Hilfs-Klassen)

In C: Aufteilung ***frei wählbar***!

Sinnvoll: Funktionale Aufteilung ähnlich wie C++ usw.:

Ein File pro "Komponente"

= ***Logisch zusammengehörige*** *Funktionen & Deklarationen*
(z.B. eine Datenstruktur, ein Lösungsschritt, ...)

"Erfahrung" und "Gefühl" helfen!

Zusammenhang **.c** & **.h** (1)

Es gibt auch "alleinstehende" **.c**- und **.h**-Files,
aber meistens kommen **.c** und **.h** ***paarweise*** vor:

Gedanklich bilden der **.c**-File
und sein zugehöriger **.h**-File
gemeinsam ein Modul
bzw. *eine Komponente*
des Gesamtprojektes

Beide Files haben typischerweise *denselben Filenamen*,
nur einmal mit **.c** und einmal mit **.h**

Beispiel:

pers_liste.c und **pers_liste.h**

Zusammenhang **.c** & **.h** (2)

Jeder **.h**-File sagt *anderen* **.c**- und **.h**-Files,
was sie von *"seinem"* **.c**-File sehen
bzw. im Code *verwenden* dürfen bzw. können:

Ein **.h**-File definiert die
öffentliche "Schnittstelle" (das *"was"*)
seines **.c**-Files bzw. seines Moduls
zum Rest der Welt,
d.h. ***die von "außen" sichtbaren Dinge!***

Der **.c**-File enthält
die ***"Implementierung"*** (das *"wie"*),
d.h. das ***"Innenleben"*** des Moduls
(geht andere **.c**- und **.h**-Files *nichts an*!)

Zusammenhang `.c` & `.h` (3)

==> Zu (fast) jedem **`.c`**-File gibt es einen **`.h`**-File,
und zu einem Großteil der **`.h`**-Files gibt es einen **`.c`**-File

Alleinstehende `.c`-Files (*ohne* entsprechendem **`.h`**-File)

= Code, der *von niemand anderem aufgerufen* wird

= meist **`main`** und *"`main`-interne" Funktionen*

Alleinstehende `.h`-Files:

= *Allgemeine*, projektweite *Typen, Konstanten, Makros, ...,*
ohne separate Implementierung in einem **`.c`**-File

= Dinge, die *keinen "echten" Code*
und keine Daten darstellen

Zusammenhang **.c** & **.h** (4)

Umgekehrt:

Ein Entwickler braucht
zum *Schreiben eines Moduls*
nur die* .h-*Files der Module, die es *verwendet*,
aber *nicht* deren **.c**-Files

- Auch *Kompilieren* kann man ein Modul
  ohne die "fremden" **.c**-Files
- *Nur zum Linken* des **.exe**-Files
  braucht man die Kompilate
  (**.o**-Files oder Libraries) *aller* Module

.h-Files (1)

.h-Files werden *immer nur inkludiert*
(*nie* für sich allein *kompiliert*!),
und zwar meist von *vielen* anderen **.c**- und **.h**-Files

==> **.h**-Files enthalten *Deklarationen*,
die zum Kompilieren von **.c**-Files nötig sind:

- ***Typen*** (**`struct`**, **`typedef`**, **`enum`**, ...)
- ***Konstanten, Makros*** (**`#define`**), **`inline`**-*Funktionen*
- **`extern`**-***Prototypen für alle Funktionen*** und
  `extern`-***Deklarationen für alle globalen Variablen***
  "ihres" **.c**-Files

... *die* ***"von außen"*** *(= für andere* **.c***-Files)* ***sichtbar sein sollen***

.h-Files (2)

Änderungen in einem **.h**-File erfordern

- logisch meist

  manuelle Änderungen im Quellcode
  aller **.c**-Files, die diesen **.h**-File *direkt inkludieren*

  (weil sich ja Strukturtypen, Funktionsaufrufe usw. geändert haben)

- build-technisch jedenfalls

  die *Neu-Kompilierung* aller **.c**-Files,
  die diesen **.h**-File
  direkt oder indirekt inkludieren

.h-Files (3)

==> Änderungen in **.h**-Files müssen

mit allen Team-Mitgliedern abgestimmt werden!

(*Nicht* "einfach heimlich ändern"!)

==> Der Inhalt von **.h**-Files wird in großen Projekten

meist ***möglichst frühzeitig festgelegt***

und ab dann ***möglichst selten geändert***!

Kein “echter” Code in **.h** ! (1)

… außer **inline**-Funktionen und **#define**-Makros

Alles, was in einem **.h**-File steht,
darf *keinen ausführbaren Code erzeugen*
und auch *keinen Speicherplatz anlegen*
(also *nichts* enthalten, was in den **.exe**-File geht)

Grund: Ein **.h**-File wird

in ***mehreren*** **.c**-Files inkludiert & *mitkompiliert*

==> Der **.o**-File von *jedem* dieser **.c**-Files
würde *den Code & die globalen Daten* dieses **.h**-Files
mit demselben Namen *enthalten*

==> ***Fehler beim Linken: Doppelte Funktionen / Daten!***

Kein “echter” Code in **.h** ! (2)

Deshalb müssen *alle Variablen-Deklarationen*
in **.h**-Files **extern** sein

==> Sind nur ein Verweis auf die “echte” Deklaration
der entsprechenden globalen Variable im **.c**-File,
kein Anlegen von Speicher dafür im **.h**-File

Aus demselben Grund
sind *alle Funktions-Prototype* im **.h**-File **extern**

.c-Files (1)

.c-Files werden *kompiliert, nie inkludiert*

==> **.c**-Files *erzeugen Code & Daten* für den **.exe**-File

==> **.c**-Files enthalten

- Die ***Implementierung*** (den Code) von Funktionen
- Die nicht-**`extern`**-Deklarationen & Initialisierungen ***globaler Variablen***
- Alle **`#include`**, **`#define`**,
  Typen, Konstanten, Prototypen, usw. ...

  ... die *nur in diesem* **.c**-*File benötigt* werden
  und daher *nicht nach "außen" sichtbar* sein sollen

.c-Files (2)

Änderungen in einem **.c**-File

- haben daher *keine Auswirkung auf den Quellcode* anderer **.c**- oder **.h**-Files
- erfordern nur eine *Neu-Kompilierung des eigenen Files* und ein *Neu-Linken des* **.exe**-*Files*

==> erfordern *keine* Information anderer Team-Mitglieder

==> sind technisch & organisatorisch *"billig"*

Was soll File-**`static`** sein?

- *Funktionen & globale Variablen* eines **`.c`**-Files,
  die ***nicht*** im entsprechenden **`.h`**-File stehen,
  sollten ***immer*** **`static`** deklariert werden:

  Was nicht im **`.h`***-File steht,*
  ist nicht zur Verwendung von außen gedacht

  ==> Sichtbarkeit / Verwendbarkeit des Namens
  mit **`static`** auf den eigenen **`.c`**-File *begrenzen*!

- **Umgekehrt:**
  - Wenn etwas aus gutem Grund *nicht* **`static`** ist,
    gehört es wohl *auch in den* **`.h`**-*File*?
  - Wenn etwas im **`.h`**-Files steht,
    darf es *nicht* **`static`** sein!

`#include` (1)

`#include`'s dokumentieren
die ***Abhängigkeiten zwischen Modulen***

Grundsätzlich gilt:

Nicht mehr `#include`'s als nötig!
(nichts "auf Verdacht" inkludieren)

==> *Unnötige Abhängigkeiten vermeiden!*

Ermöglicht u.a. effizienteres Compilieren des Projektes:

Änderungen in einem **`.h`**-File erfordern
Änderungen und Neukompilierung aller **`.c`**-*Files*,
die diesen **`.h`**-File direkt oder indirekt inkludieren

==> *Je weniger `.c`-Files das sind, umso besser!*

`#include` (2)

Jeder File inkludiert nur,
was er selbst direkt benötigt / verwendet!

Für **`#include`**'s,
die *nur indirekt* für inkludierten Code nötig sind,
ist *deren Header zuständig*, nicht der eigene File

Beispiel:

- **`.c`**-File **`a.c`** braucht Header **`x.h`**
- Header **`x.h`** enthält Dinge, für die Header **`y.h`** nötig ist
- **`.c`**-File **`a.c`** verwendet *nichts* aus Header **`y.h`** direkt

==> **`y.h`** wird *nur* in **`x.h`** inkludiert

==> **`a.c`** enthält *kein* direktes **`#include`** für **`y.h`**

`#include` (3)

==> Man braucht *nicht* wissen,
was andere Header intern für **`#include`**'s benötigen,
denn sie machen die nötigen **`#include`**'s *ohnehin selbst*

(wenn nicht: Schlecht programmiert, Header korrigieren!)

Aber umgekehrt:

Wenn man *selbst* Dinge aus **`xyz.h`** *verwendet*,
dann soll man **`xyz.h`** jedenfalls *selbst direkt inkludieren*,

auch dann, wenn man genau weiß,
dass andere inkludierte Header
`xyz.h` ohnehin schon indirekt inkludieren!

`#include` (4)

Bei jedem **`#include`** überlegen:

Benötige ich dieses **`#include`**
nur intern in meiner Implementierung,
d.h. nur für meinem **`.c`**-File?

==> *Wenn ja:* ***Nur im `.c`-File inkludieren, nicht im `.h`-File***
Was ich intern brauche, geht die "Außenwelt" nichts an!

Oder sind Dinge aus diesem **`#include`**
in meiner Schnittstelle sichtbar,
d.h. in meinem **`.h`**-File
(Beispiel: Typen, die *in meinen Prototypen vorkommen*)

==> ***Im `.h`-File inkludieren, nicht im `.c`-File!***

`#include` (5)

Jeder **`.c`**-File inkludiert

immer "seinen eigenen" `.h`-File

... damit die *eigenen Prototypen*
und die nach außen sichtbaren *Typen, Konstanten usw.*
auch im eigenen Code verfügbar sind

... damit *Inkonsistenzen* zwischen **`.c`** und **`.h`**
vom Compiler erkannt werden

Schutz vor doppeltem Include (1)

Sehr häufige Situation, einfachster Fall:

- **`a.c`** includiert **`y.h`** und **`z.h`**
- **`z.h`** inkludiert **`y.h`** ebenfalls

==> Beim *Kompilieren* von **`a.c`** wird

`y.h` *doppelt inkludiert*
(einmal *direkt*, einmal *indirekt* über **`z.h`**)

==> Im Präprozessor-Output von **`a.c`**
sind alle Zeilen von **`y.h`** *zwei Mal enthalten*

==> Der Compiler liefert *viele Fehler* wegen
doppelter Deklarationen usw. (vor allem in C++)

Schutz vor doppeltem Include (2)

Jeder **`.h`***-File muss folgenden*
***Schutz gegen doppeltes Include** enthalten:*

Ganz am Anfang:

```
#ifndef xxx
#define xxx 1
```

Ganz am Ende:

```
#endif
```

xxx ... *irgendein Makroname* nur für diesen Zweck,
und zwar für *jeden* **`.h`**-File *ein anderer!*

Meist: **`_FILENAME_H`** (z.B. in **`myfile.h`** : **`_MYFILE_H`**)

Schutz vor doppeltem Include (3)

Effekt (am Beispiel `myfile.h`):

- Beim ***ersten*** Inkludieren von **`myfile.h`**:

  `_MYFILE_H` ist *noch undefiniert*

  ==> Das **`#ifndef`** ist *wahr*

  ==> **`_MYFILE_H`** wird *definiert*

  ==> Der gesamte Inhalt von **`myfile.h`** wird

  ganz normal inkludiert

Schutz vor doppeltem Include (4)

- Bei ***jedem weiteren*** Inkludieren von **`myfile.h`** im selben Präprozessor-Lauf:

  `_MYFILE_H` ist *schon definiert*
  (vom ersten Mal)

  ==> Das **`#ifndef`** ist *falsch*

  ==> Alles bis zum **`#endif`** wird *ignoriert*

  ==> Es wird nur mehr ein

  komplett leerer File inkludiert

union, enum usw.

union:

- Grundkonzept
- Anwendungen

enum:

- Grundkonzept
- Werte von enum-Konstanten
- Unterschiede C/C++

volatile

`union` (1)

Eine Union enthält wie eine **`struct`**

benannte Member

mit (meistens) verschiedenen Typen,

aber zu jedem Zeitpunkt
immer nur eines davon

==> eine Union enthält

wahlweise bzw. abwechselnd eine
von *mehreren verschiedenen Möglichkeiten!*

`union` (2)

Deklaration, Zugriff, ...:

... ganz analog zu **`struct`**,
nur mit "**`union`**" statt "**`struct`**".

Initialisierung:

- Es kann nur für *ein* Member
  ein Initialwert angegeben werden.
- Bei "Oldstyle"-Initialisierung
  (ohne Member-Namen in den **`{ }`**)
  muss der Wert des *ersten Members* angegeben werden.

`union` (3)

Member:

… können einfache Werte, Pointer,
aber auch Arrays, **`struct`**'s und weitere **`union`**'s sein

Speicherung:

- Der Compiler reserviert für eine **`union`**
  soviel Platz, wie das *größte Member* braucht.
- Alle Member beginnen *"ganz vorne"*
  in der **`union`**…
- … und liegen sozusagen *"übereinander"*
  statt nacheinander.

Gefahr bei **`union`**'s! (1)

Eine **`union`** "weiß" *nicht*,
welches Member sie gerade enthält!

Man kann es auch *nicht herausfinden*!

==> *keine Prüfung oder Fehlermeldung*
bei "falschem" Zugriff!

Technisch:

Je nachdem, mit welchem Member zugegriffen wird,

werden *dieselben Bits im Speicher*
unterschiedlich interpretiert

(als **`int`**, als **`double`**, als Pointer, als mehrere **`char`**'s, ...)

Gefahr bei **`union`**'s! (2)

Beispiel:

Eine Union aus **`double`**-Member und *Pointer*-Member enthält gerade einen **`double`**-Wert.

Bei Zugriff auf das Pointer-Member:

==> Die Bits des **`double`**
werden als Pointer-Wert verwendet!

==> Keine gültige Adresse, *Absturz*!

Daher:

Immer woanders merken,
was gerade in der **`union`** *enthalten ist!*

1. Anwendung von **`union`**'s (1)

Früher ganz wichtig:

Unions als **Member von `struct`'s**
für ***variable*** **Teile der `struct`**.

Beispiel:
`struct` für Personendaten in einer Firma.

Wenn die Person *Kunde* ist:
    Union enthält Jahresumsatz (**`double`**)
Wenn die Person *Mitarbeiter* ist:
    Union enthält Abteilungsname (**`char`**-Array)
Wenn die Person *Ex-Mitarbeiter* ist:
    Union enthält Austrittsdatum (**`time_t`**)

1. Anwendung von **`union`**'s (2)

Heute in C++ kaum noch verwendet:

Abgelöst durch *objektorientierte Programmierung*
und das Konzept *abgeleiteter Klassen*
(siehe 2. Semester!).

Noch häufig in der *Systemprogrammierung*
(weil die verwendet noch C statt C++ ...)

Beispiele:

- Netzadresse (**`union`** für IPv4- oder IPv6-Adresse).
- I/O-Geräte-Adresse (SATA, USB, ...)

2. Anwendung von **`union`**'s

`union`'s, auf die ***absichtlich "falsch"*** zugegriffen wird (oft zur Konvertierung zwischen interner Darstellung und einzelnen Bytes).

Beispiel 1 (häufig!):

`union` aus einem **`int`** und einem **`char[4]`**-Array:

Erlaubt Zugriff auf die *einzelnen Bytes*,
mit denen der **`int`** intern gespeichert wird.

Beispiel 2:

`union` aus einem **`long long int`** und einem **`double`**:

Erlaubt *Bit-Operationen* auf den Bits der **`double`**-Zahl.

`enum`-Typen

`enum` =

“Enumeration” =

Aufzählungstyp =

Ein Typ, der ***einen*** *Wert*
aus einer Menge ***benannter Konstanten***
speichern kann

Bekanntestes Beispiel Typ **`bool`**:

Besteht aus 2 Konstanten **`true`**, **`false`**

enum-Beispiel

Deklaration eines **enum**-*Typs*:

```
enum wochentag { montag, dienstag, …, sonntag };
```

Deklaration einer **enum**-*Variable*:

```
enum wochentag tag_heute;

struct mydatum {
   …
   enum wochentag w_tag;
   … };
```

Verwendung:

```
tag_heute = montag;
if (tag_heute == sonntag) { … }
```

Was bewirkt **`enum`** ?

- "**`montag`**" usw. werden als *neue Konstanten* deklariert
- "**`enum wochentag`**" ist ein *neuer Typ*,
  lässt sich wie **`int`**, … bzw. wie "**`struct Person`**"
  in Deklarationen, als Returntyp usw. verwenden
- Wenn man das **`enum`** weglassen will:
  Wie bei **`struct`** ein **`typedef`** verwenden!
- Wenn man ohnehin ein **`typedef`** macht
  oder *nur die Konstanten* aber *nicht den Typ* braucht,
  kann man das Tag (den Typnamen) nach dem **`enum`**
  auch weglassen.

  Siehe Beispiele ein paar Folien weiter…

`enum`-Werte

`enum`-Werte werden intern durch *Zahlen* dargestellt
(das sind *keine Strings*, daher auch *keine* **`"..."`** !):

- Wenn man nichts angibt, wird *durchnummeriert*:
  - Die erste Konstante bekommt den Wert **`0`**
  - Jede weitere bekommt **`(`*`Wert der vorigen`*` + 1)`**
- Man kann aber für einzelne oder alle Konstanten *beliebige* **`int`**-*Werte* in *beliebiger Reihenfolge* festlegen

  (es sind auch Löcher und gleiche Werte
  und sogar negative Werte erlaubt)

enum-Beispiele mit Werten (1)

```
typedef enum {
   black = 0x000000,
   red = 0xff0000,
   yellow = 0xffff00,
   green = 0x00ff00,
   cyan = 0x00ffff,
   blue = 0x0000ff,
   magenta = 0xff00ff,
   white = 0xffffff,
   alpha = 0xff000000
} Color;
```

enum-Beispiele mit Werten (2)

```
enum noten {
   sgt = 1, gut, befr, ausr, mangelh, ungen,
   befreit = 0 };

enum http_retcode {
   ok = 200,
   not_found = 404, forbidden = 403, error = 500,
   unknown = -1 };
```

Entspricht nicht der tatsächlichen Definition,
aber könnte so definiert werden:

```
typedef enum { true = 1, false = 0 } bool;
```

Speicherung von **`enum`**'s

Achtung:

Die Größe (**`sizeof`**) eines **`enum`**'s wird
vom Compiler *je nach Wertebereich* festgelegt!

==> Die Größe ist nicht immer **`sizeof(int)`**,
sondern kann alles von **`sizeof(char)`**
bis **`sizeof(long long)`** sein!

==> Das **`sizeof`** zweier verschiedener **`enum`**-Typen
kann *verschieden* sein!

==> **`sizeof(bool)`** ist meist *nicht* **`sizeof(int)`** !

Der Compiler darf auch frei wählen,
ob ein **`enum`** intern **`signed`** oder **`unsigned`** ist!

Seit **C23** kann man den verwendeten *Typ optional händisch festlegen*

enum und **int** in C

In C: **enum**'s sind **int**-kompatibel, auch vom Typ her

==> Man kann **int** und **enum** aufeinander zuweisen,
in Rechnungen *mischen*, miteinander vergleichen, ...

==> Überall, wo ein **int** erlaubt ist,
ist auch ein **enum** erlaubt
(Rechnungen und Bit-Operationen, Array-Index, ...)

==> **enum**'s werden oft

für *benannte* **int**-*Konstanten* verwendet

(statt **#define** *name wert*
oder **const int** *name* **=** *wert***;**)

`enum` und **`int`** in C++

In C++:

`enum`'s sind *eigene Typen*,
vom Typ her formal *nicht* **`int`**-kompatibel

(obwohl es technisch trotzdem **`int`**'s sind)

==> Man kann **`int`** und **`enum`** nur aufeinander zuweisen
bzw. einen **`enum`** nur dann als **`int`** verwenden
oder umgekehrt (in Rechnungen usw.),
wenn man eine

explizite Typumwandlung

macht.

`volatile` (1)

`volatile` ist ein *Typ-Zusatz* wie **`const`**
==> darf überall dort stehen, wo **`const`** erlaubt ist

`volatile` (wörtlich ***"flüchtig"***)
ist das *"Gegenteil"* von **`const`**:

*Der Wert in der Variablen kann sich **jederzeit***
*ohne Wissen bzw. Zutun des Programmes **ändern***

Effekt: Der Compiler darf **`volatile`**-Zugriffe *nicht optimieren!*

==> *Jeder Zugriff* auf einen **`volatile`**-Wert *im C-Code* muss 1:1
in *genau einem Speicher-Zugriff zur Laufzeit* resultieren
(selbe Reihenfolge, selbe Anzahl, selbe Stelle im Code)

(nicht in Register legen, nicht aus Schleifen herausoptimieren,
Ausdrücke nicht vorausberechnen, ...)

`volatile` (2)

`volatile` ist für Variablen oder Pointer auf Werte, die z.B.

- *von außen geändert* werden
  (von anderen Programmen oder *parallel laufendem Code*)
- nicht im normalen RAM liegen,
  sondern *Hardware-Register repräsentieren*

  ==> *Änderung durch die Hardware jederzeit möglich!*

Zugriff auf Hardware-Register mit normalem C-Code
ist u.a. möglich durch ...

- ... einen *Pointer auf* **`volatile`** ...
- ... initialisiert mit der *Adresse des I/O-Bausteins*
  (z.B. als Hex-Konstante)

Die C Library

- Auflistung der C-Standard-Header
- Beispiele für POSIX-Header
- Andere Libraries

Die C Standard Library

Definiert im *Standard der Sprache C* selbst:

24 Header *(seit C11 29 Header)*

mit exakt definierten Funktionen usw.

==> Müssen *Plattform-unabhängig*
in jeder Implementierung von C vorhanden sein,
die sich *"Standard-konform"* nennt

(nicht jede C-Implementierung ist Standard-konform,
z.B. auf 8-Bit-Mikroprozessoren)

Systemseitig *"Kleinster gemeinsamer Nenner"*,
wenig Betriebssystem-bezogene Funktionen:
- Keine Parallelität (kam erst mit C11 dazu)
- Kein Netzwerk

C Std Header (1)

- **`assert.h`**: **`assert(...)`**
- **`complex.h`**: Komplexe Zahlen
- **`ctype.h`**: *"Character Type"*: **`is...`** und **`to...`**
- **`errno.h`**: **`errno`** und **`E...`**-Konstanten
- **`fenv.h`**: Einstellungen für **`float`** / **`double`**
  (Rundungs-Modus, Fehler-Behandlung, ...)
- **`float.h`**: Konstanten für Limits usw. von **`float`** und **`double`**
- **`inttypes.h`**: Makros für **`printf`**-Formate für **`int`**'s
- **`iso646.h`**: Makros für Sonderzeichen
- **`limits.h`**: Konstanten für System-Limits usw. und für **`int`** usw.
- **`locale.h`**: Ländereinstellungen

C Std Header (2)

- **`math.h`**: Mathematische Funktionen und Konstanten
- **`setjmp.h`**: Goto-ähnliches Konstrukt zwischen Funktionen
- **`signal.h`**: Signal Handling (ähnlich wie Interrupts)

  Achtung: *In POSIX inkompatibel redefiniert / erweitert!*
- **`stdarg.h`**: Variabel viele Funktions-Argumente
- **`stdbool.h`**: Typ **`bool`**
- **`stddef.h`**: **`size_t`**, **`ptrdiff_t`**, **`NULL`**, **`offsetof`**, ...
- **`stdint.h`**: **`int`**-Typen fixer Länge (8 / 16 / 32 / 64 Bits)
  sowie **`intptr_t`** und **`intmax_t`**, Konstanten dafür
- **`stdio.h`**: File-I/O, File-Funktionen (**`remove`**, **`rename`**, ...), **`perror`**

C Std Header (3)

- **`stdlib.h`**: Alles, was sonst nirgends dazugehört:

  `exit` / **`abort`**, **`atoi`** / **`strtol`** / ..., **`abs`**, **`rand`** / **`srand`**

  `malloc` usw.

  `getenv`, **`system`**, **`bsearch`**, **`qsort`**, ...

  einige **`wchar`**- und Multibyte-Funktionen
- **`string.h`**: String- und **`mem`**-Funktionen, **`strerror`**
- **`tgmath.h`**: Gemeinsame Funktionen für **`double`** und **`complex`**
- **`time.h`**: Datum und Zeit
- **`wchar.h`**, **`wctype.h`**: **`wchar`**-Funktionen und -Makros, **`wchar`**-I/O

C Std Header (4)

Seit C11:

- **`stdalign.h`**:
  Makros für die *Ausrichtung* von Variablen und **`struct`**-Membern auf bestimmte *Speichergrenzen* (z.B. *"durch 16 teilbar"*)
- **`stdatomic.h`**: Typen & Funktionen für *"Atomare"* Speicherzugriffe (nötig für Parallelprogrammierung)
- **`stdnoreturn.h`**:
  `noreturn` … *"diese Funktion kehrt nicht zurück"*

  - Nützliche Information für den Leser des Programmes
  - Hilft dem Compiler beim Optimieren und Fehlersuchen
- **`threads.h`**: Low-Level-Konstrukte
  für *parallele Threads*, Thread-Synchronisation usw.
- **`uchar.h`**: Typen und Funktionen für UTF-16 und UTF-32 Zeichen

Die POSIX Library

POSIX = Standard aller "unixoiden" Betriebssysteme
(auch ein "formaler", hersteller-unabhängiger Standard!)

- Alle *kommerziellen Unixe*
- *macOS*
- *Linux* & *BSD*-Derivate (beide nicht offiziell zertifiziert), …
- Auch *viele Echtzeit-Systeme* (VxWorks, QNX, …)
- Windows nur *teilweise*! (WSL ist nicht zertifiziert)

Rund 80 Header:
- *Übermenge* des C-Standards (fast)
- Zugriff auf *viele Betriebssystem-Funktionalitäten*
- Vor allem auch: *Parallel-Programmierung*
- *Netzwerk* (TCP/IP), …
- …

Beispiele POSIX Header

- **`unistd.h`**, **`fcntl.h`** und **`sys/stat.h`**:
  Allgemeine Unix-Systemfunktionen
  - Low-Level-File-I/O (**`read`**, **`write`**, ...)
  - Unix-Prozess-Handling (**`fork`**, **`exec`**, ...)
  - Kleinkram: **`sleep`**, ...
- **`dirent.h`**: Directory-Zugriffe
- **`sys/*.h`**:
  spezifische Unix-Systemfunktionalitäten
  - **`mmap`**
  - Interprozess-Kommunikation & Synchronisation
- **`pthread.h`**: Parallele Threads

Andere Libraries

Viele hundert für fast jeden Zweck!

In weiten Teilen *Open Source*

Aber: *Nicht* standardisiert, “wilde” Entwicklung!

Beispiele:

- GUI (z.B. GTK)
- Regular Expressions
- Verschlüsselung
- Multimedia, Bildverarbeitung, …
- “Höhere” Netzwerk-Kommunikation

Ausblick

- **Die wesentlichen Konzepte von C++:**
  Objektorientierung & Vererbung
- **Grundidee der GUI-Programmierung**

Was kommt im 2. Sem.?

C++

==> viele Erweiterungen gegenüber C!

Vor allem "*objektorientierte Programmierung*":

Objekte & Vererbung

Weiters u.a.:

- Funktionen: Referenz-Parameter, Default-Werte, ...
- Selbstdefinierte Rechenzeichen **+**, **<=**, ...
- Neues Fehlerbehandlungs-Konzept ("Exceptions")
- Programmstücke mit Typ-Parametern ("Templates")

Objekte

Klasse =
Strukturtyp plus Funktionen
für Werte dieses Typs (“***Methoden***”)

Objekt =
Variable bzw. Wert,
dessen *Typ eine Klasse* ist

Beispiel:

```
class Student { … };  // Klassendeklaration
Student pi1[32];  // Array von Objekten der Klasse
```

Methoden

Methoden …

- … werden *für ein bestimmtes Objekt aufgerufen*:

  `myObj.myMeth(…)` oder **`myOPtr->myMeth(…)`**

- … haben intern automatisch einen
  Pointer auf dieses ("das eigene") Objekt: **`this`**
- … können auf die *Struktur-Member* des eigenen Objektes
  wie auf *lokale Variablen* zugreifen

Jede Klasse hat zwei *Spezial-Methoden*:

- Der "**Konstruktor**" wird automatisch
  bei *jedem Anlegen eines Objektes* aufgerufen (Initialisierung)
- … und der "**Destruktor**" analog beim Vernichten

Vererbung (Ableitung)

Eine *“abgeleitete Klasse”* …

- hat (“erbt”) *alle Member &*
  kann *alle Methoden der “Basisklasse”*
- kann *neue Member & neue Methoden dazudefinieren*
- kann *anderen Code für geerbte Methoden* definieren
  (Methoden *“überschreiben”*)

Beispiele:

- **`Button`**, **`Textfeld`**, **`ChoiceBox`**, …:
  Abgeleitet von **`DialogElement`**
- **`IconButton`**: Abgeleitet von **`Button`**
- **`Motorrad`**, **`PKW`**, **`LKW`**, …: Abgeleitet von **`Kraftfahrzeug`**

Was kommt nicht?

GUI-Programmierung:

Grafische Benutzeroberflächen
mit Fenstern, Menüs, Dialogen, …

Ist *kein* standardisierter Bestandteil von C++,
sondern geschieht über verschiedene
Erweiterungen (“*Qt*”, “*wxWidgets*”, …)
mit *hunderten / tausenden vordefinierten Klassen*
(Dialog, Button, Textfeld, Menü, Menüeintrag, …)!

GUI-Programmierung (1)

Programm-Struktur:

`main` ist fix:

Zuerst die **Initialisierung**,
dann die "**Event Loop**"
(bis zum Programmende)

"Event Loop":

- *Wartet* auf eine *Benutzer-Aktion*
  (*"Event"* = Taste, Mausklick, ... ev. auch Timer-Ablauf)
- Führt die *dazugehörige Funktion* (*"Event Handler"*) aus
- Wartet auf die nächste Aktion ...

GUI-Programmierung (2)

Klassische Programmier-Methodik
(*ohne* "Gui-Designer" usw.):

- *Ableiten* von vordefinierten Rahmen-Klassen
  (leeres Hauptprogramm, leerer Dialog, ...),
  diese erweitern
- *Event-Handler-Funktionen* schreiben
- *Initialisierung* schreiben
  (meist im Konstruktor):
  - Alle GUI-Elemente, Menüs, ... anlegen & positionieren
  - Events mit Events-Handlern verbinden