Notizen Programmieren 1

Klaus Kusche, 2010 / 2011

1. Kapitel: Allgemeines

Programmieren =

Programme schreiben =

dem Computer sagen, was er tun soll

Notwendige *Kenntnisse* dazu:

- *Programmiersprache* (lernen wir hier)
- *Standard-Libraries* bzw. Klassenbibliotheken
  (schon vorhandener Code, den man verwenden kann)
  (lernen wir auch hier)
- *Algorithmen & Datenstrukturen*
  (lernen wir im 3. Sem. auch hier)
- *Funktionsweise des Computers* (Datendarstellung, Files, ...)
  (siehe Rechnerarchitektur & Betriebssysteme)
- Kenntnis der *Programmierwerkzeuge* (Editor, Compiler/Linker, Debugger, ...)
  (die einfachen Dinge: Hier selber beibringen. Fortgeschrittenes: 4. Sem.)
- Bei größeren Projekten: Kenntnisse des *Software Engineering*
  (4. Sem. & höhere Semester)

Ein Programm ist wie ein Kochrezept:

1. Beschreibung der ***Daten***:

   Deklarationen (Beschreibung der Zutaten, mit denen gearbeitet wird)

2. Beschreibung des ***Ablaufes***:

   Anweisungen (Befehle, die Schritt für Schritt nacheinander ausgeführt werden)

Daten:

Konstanten und ***Variablen***

==> Details später

Anweisungen:

- **"Rechnen"** (arith. und log. Ausdrücke)
- **"Speichern"** (Zuweisungen)
- **Funktionsaufruf** (Ausführung eines anderen Codestückes)
- Zusammengesetzte Anweisungen:

- **Bedingungen** (**`if`**) und Fallunterscheidungen (**`switch`**)
  "wenn … gilt, mach dieses, sonst jenes"
- **Schleifen** (**`while`**, **`for`**)
  "wiederhole etwas x mal", "wiederhole etwas bis … gilt"
  ==> Details später

Unterscheide:

- *Quellcode* (**`.c`**, **`.cpp`**, …):

  In einer Programmiersprache, vom Menschen geschrieben, portabel

  Aus der Sicht des Computers: Irgendein Text

- *Binär-Programm* (**`.exe`**):

  Maschinenbefehle, vom Prozessor ausführbar
  Prozessor- und Betriebssystem-spezifisch

  Für den Menschen (de facto) unlesbar (irgendwelche Bitmuster)!

Vom Quellcode zum Binärprogramm: *Compiler + Linker*
(Alternativen: Interpreter, JIT)

Einteilung Programmiersprachen:

- *Maschinennahe* (Assembler)
- *Prozedurale* (C, Cobol, Fortran, Ada, …)
- *Objektorientierte* (C++, Java, C#, Smalltalk, Visual Basic, viele Scriptsprachen)
- *Funktionale* (Lisp, …) und *logische* (Prolog, …)

Geschichte von C:

1970, Bell Labs (Kernighan, Ritchie), gemeinsam mit *Unix*

Standards:

- *K&R C* (heute nicht mehr verwendet)
- *ANSI C* / ISO C / C 89 (heute Mindest-Standard)
- *C99* (setzt sich weitgehend durch)
- *C11* (noch praktisch unbekannt)
- *Gnu C* (bei Open Source SW, nutzt Spracherweiterungen des gcc Compilers)

Was ist in C geschrieben?

- Der Linux-Kernel (Windows auch?)
- Ein Großteil der Linux-Software
- Viele Hardware-nahe oder Performance-kritische Programme, …

C/C++ ist die *älteste* heute noch allgemein benutzte Programmiersprache
und die derzeit *am weitesten verbreitete*.
C/C++ war auch Vorlage für Java, C#, viele Scriptsprachen, …

Vorteile von C:

- Effizient
- Direkter Zugriff auf die Hardware möglich
- Schult das Computer-Verständnis

Nachteile von C:

- Wenig komfortabel
- Viele Fehlermöglichkeiten (nicht "deppensicher")

C++:

... ist die Erweiterung von C um Objekte

==› C ist im Wesentlichen eine *Untermenge* von C++

Unterschiede zwischen C und C++:

- *Ein- und Ausgabe:*

  C: **`printf`**, **`fgets`**, ...

  C++: **`cin`**, **`cout`**, **<<**, **>>**
- *Bool'sche Werte* (wahr/falsch):

  C: Als Zahl (**`int`**), erst ab C99 eigener Datentyp **`bool`** aus dem Header **`stdbool.h`**

  C++: Eigener Typ **`bool`** fix in der Sprache (ohne Header)
- *Dyn. Speicherverwaltung*:

  C: **`malloc`** / **`free`**

  C++: **`new`** / **`delete`**
- Viele kleine Details... (siehe 2. Semester)

Schritte beim Programmieren:

- *Problem analysieren*, Aufgabe klar & vollständig formulieren
  (Eingabe, Ausgabe, Randbedingungen, ...)
- *Lösung entwerfen* (Welche Daten, welche Programmstruktur / Schritte?)
  Trennung Lösungsidee ("Algorithmus") / Programmiersprache!
  Algorithmus ist unabhängig von Programmiersprache!
- Lösungsidee in *C-Quelltext* verwandeln, eintippen
- *Compilieren* (genauer: Präprozessor, Compiler, Linker)
- *Ausführen, testen*
- Alle Schritte begleitend: Dokumentieren!

2. Kapitel: Variablen und Konstanten

Konstanten können sein:

- *Direkt angegebene Werte:*

  Zahlen (Limits, Naturkonstanten, Umrechnungsfaktoren), Texte, …

  ==› Besprechung bei den einzelnen Typen

- *Benannte Konstanten:*

  3 Möglichkeiten:

  - *Präprozessor-Konstanten:*

    `#define` ***NAME Wert***

    Bewirkt textuelle Ersetzung von ***NAME*** durch ***Wert*** durch den Präprozessor

    Kein "**`=`**" und kein "**`;`**" am Ende!

  - **`enum`***-Konstanten:*

    ==› Viel später (bei Enum-Typen)

  - **`const`***-Variablen:*

    Bei Variablen-Deklaration **`const`** davor und Initialisierung dahinter

Konstanten haben

- Einen *Typ* (**`int`**, **`double`**, **`const char *`**, …)
- Einen *Wert*

Variablen sind…

benannte Speicherplätze für Werte,
die sich während des Programmablaufes ändern.

Vor erster Benutzung **deklarieren**!

Variablen haben:

- Einen *Namen*:
  Vom Benutzer wählbar, durch Deklaration festgelegt

  (Regeln für Namen: siehe unten!)

  Sinnvolle Variablennamen sind wesentliches Element der Lesbarkeit!!!

- Einen *Typ* (**`int`**, **`double`**, **`const char *`**, …):
  Durch Deklaration festgelegt (bestimmt Art der Werte und Platzbedarf)

- Einen (aktuellen) *Wert*:
  Durch Initialisierung oder Zuweisung (**`=`**) (und durch Parameter-Übergabe)

- Einen *Speicherplatz* (Adresse & Größe):
  Vom Compiler, Größe je nach Typ, siehe später…

- Einen *Scope* (Sichtbarkeitsbereich) und eine *Lebensdauer:*
  Siehe später…

Zulässige Namen:

- Buchstaben und Ziffern, beginnend mit Buchstabe
- Groß- und Kleinschreibung wird unterschieden
- **_** zählt als Buchstabe
- sonst keine Sonderzeichen!
- Keine Umlaute und ß
- Keine Schlüsselworte, **__**… ist auch reserviert

Üblich:

- Entweder *"CamelCase"* (Anfangsbuchstaben der Teilworte groß): **`LangerVarName`**
- oder alles klein und Teilworte mit _ getrennt: **`langer_var_name`**
- "Nur Großbuchstaben" ist üblich für **`#define`**-Namen, sonst nicht!

Schlüsselworte:

`for`, **`while`**, **`if`**, **`int`**, … (immer klein!)

Einschub: Stoff in der Übung

Whitespace (Zwischenraum, Tab, Zeilenvorschübe):
(Fast) nach Belieben, zur besseren Lesbarkeit, vom Compiler ignoriert

==› Code *systematisch einrücken* (pro **`{}`**)

==› *1 Befehl pro Zeile*

==› Üblichen Stil betreffend Leerzeichen nachahmen

==› Gedankliche Blöcke durch Leerzeilen trennen

==› Professionell: Code Formatter verwenden!

==› Bitte ***keine*** Tabs! (durch entsprechend viele Zwischenräume ersetzen!)

Kommentare:

- **`/*`** … **`*/`** (auch über mehrere Zeilen)
- **`//`** … (bis zum Zeilenende, erst in C++ und ab C99)

Wird ignoriert, Anmerkungen für den Leser des Programms
==› Reichlich, systematisch und *sinnvoll* verwenden (Lösungsidee beschreiben!)

Achtung: **`/* */`** schachtelt nicht!

(*Problem beim Auskommentieren von Codeteilen*, die schon **`/* */`**-Kommentare enthalten, stattdessen **`#if 0`** -Trick)!

Präprozessor-Befehle:

- Beginnen mit "**#**", bis zum Ende der Zeile (kein "**`;`**" !)
- **`#include <...>`**, **`#include "..."`**, **`#define`** …, Details später

Zu jeder vordefinierten Funktion das entsprechende **`#include`** verwenden,
siehe **`man`**-Page / Hilfe!

Programm:

Folge von Präprozessor-Befehlen und Definitionen bzw. Deklarationen
(Variablen, Typen, Funktionen, ...).

Hinter jeder Deklaration und hinter jeder Anweisung steht ein "**;**"

Ausnahme: Hinter zusammengefassten Anweisungen **{...}** nicht !
(unmittelbar vor **}** schon!)

Funktionsdeklaration:

*Typ_des_Ergebnisses Name_der_Funktion***(***erster_Wert_hinein***,** *zweiter_Wert_hinein***,** ...**)**
{
*Variablen_Deklaration***;**
...
*Anweisung***;**
...
*Anweisung***;**
}

Sonderfall Funktion "`main`":

- Wird beim Programmstart aufgerufen
- Ende von **`main`** = Programmende
- Argumente: Commandline-Argumente als **`argc`** und **`argv`**
  (immer zumindest eines: Der Programmname selbst)
- Returnwert von **`main`**: In Shellscript / Batchfile abfragbar
  (0...ok, 1-125...Fehler, 0 ist Default, wenn man unten aus **`main`** rausfällt)
  (besser: **`exit(0);`** usw. verwenden)

3. Kapitel: Ganze Zahlen

Typen

- Normale ganze Zahl: Typ **`int`**
- Mit Präfix **`short`**, **`long`**, (**`long long`**) (dann kann man **`int`** auch weglassen)
- Mit Präfix **`signed`** oder **`unsigned`**

  `int`: default signed (mit Vorzeichen)

  `char`: default je nach Plattform verschieden!!!

  Auf x86 Prozessoren bei gcc: default **`signed char`**
  ==› Umlaute usw. sind ‹ 0 !
  ==› Vorsicht bei Verwendung von **`char`** als Array-Index!
- **`char`** (Zeichen) und **`wchar`** (für UCS-2 Unicode Zeichen) sind auch "ganze Zahlen"
- **`size_t`** (Ganze Zahl für Längen, File-Größen usw., meist gleich **`unsigned long`**)
- **`enum`**-Typen (später) sind de facto auch **`int`**-Typen (ganzzahlig)

Wertebereich

bei n Bits: unsigned $0...2^n-1$, signed $-2^{n-1}...2^{n-1}-1$

(die kleinste negative Zahl -2^{n-1} hat *kein positives Gegenstück!*)

Länge in Bit (`char` / `short` / `int` / `long` / `long long`):

- 8- und 16-Bit-Prozessoren (Mikrokontroller, Ur-x86): 8 / 16 (oder 8) / 16 / 32 / ---
- x86 heute im 32-Bit-Mode: 8 / 16 / 32 / 32 / 64
- x86 heute im 64 Bit Mode: 8 / 16 / 32 / 64 / 64
- Einzige Garantie:
  - **`char`** ist immer 1 Byte (8 Bits)
  - Bereich **`char`** <= Bereich **`short`** <= Bereich **`int`** <= Bereich **`long`**

Mit `char` kann man ganz normal rechnen:

Wert eines Buchstabens ist sein Code (ASCII).

Beispiele:

`c - '0'` (wenn **`c`** eine Ziffer ist: numerischer Wert der Ziffer **`c`**)

`n + '0'` (Zifferndarstellung der einstelligen Zahl **`n`**)

`(n - 10) + 'a'` (Hexziffer-Darstellung eines Wertes **`n`** zwischen 10 und 15)

Wahrheitswerte *("wahr" / "falsch"):*

Klassisches C hat *keinen* Typ "**`bool`**" (optional in **`stdbool.h`** ab C99) , nur C++:

Jeder `int`-Wert (oder **`double`** oder **`char`**-Wert oder Pointer-Wert)
kann als Wahrheitswert interpretiert werden:

0 ... "falsch"
alles andere ... "wahr"

Vergleiche, boolsche Operationen usw. liefern als Ergebnis Typ **`int`**, Wert 0 oder 1.

Konstanten:

- normale Dezimalzahlen, mit **-** (vorne), ohne Tausender-Gruppierung
- Suffix **`L`**, **`U`** (**`long`**- oder **`unsigned`**-Konstante)
- Führende **`0x`** ... Hexzahl
- Führende **`0`** ... Oktalzahl
- Nur gcc: **`0b`** ... Binärzahl

Zu große Konstanten werden meist stillschweigend mod 2^n genommen!

Char-Konstanten:

Ein Zeichen in ***einfachen*** Anführungszeichen: **`'*'`**, **`'A'`**, ...

Unterscheide: Doppelte Anführungszeichen: String, d.h. Zeichenkette

Sonderwerte mit Backslash:

- **\n** … *Zeilenvorschub*
- **\r** … Wagenrücklauf
- **\t** … Tabulator
- **\0** … Zeichen mit Wert 0 (nicht Ziffer 0),
  nnn … Zeichen mit Oktal-Wert nnn
- **\'** … das einfache Anführungszeichen (analog für das doppelte: **\"**)
- \\ … Backslash (z.B. in Windows-Filenamen)
- und ein paar mehr…

Beachte:

- Zeilenvorschub Unix / Linux / Mac OS X: **\n**
- Zeilenvorschub DOS / Windows: **\r\n**
- Zeilenvorschub Mac vor OS X: **\r**

Operationen

- **+ – * /**
- unäres **–**
- *Kein* "hoch" !
- **a % b** … Rest von **a / b**
- Die üblichen Vorrangregeln, **()** gelten wie üblich (im Zweifelsfall zu viele setzen!)

Achtung: **int/int** liefert **int** (nicht **double**), *abgeschnitten!*

Achtung: **/** und **%** sind bei *negativem Divisor* in C89 gar nicht, in C99 "falsch" definiert
(**%** ist bei negativen Zahlen *nicht* "mod", sondern negativ)
==› *am besten nicht verwenden!*

Fehlerbehandlung:

- **/ 0**, **% 0** … fataler Fehler
- sonst: **Überlauf ignoriert, Rechnung mod 2^n !**

"Gemischte" Rechnungen:
Für jede einzelne Operation: Implizite Konvertierung

- … auf längeren Typ der beiden Operanden (**char** ==› **int**, **int** ==› **long**),
- … auf **unsigned** wenn ein Operand **unsigned** ist

(genaue Regeln sind sehr kompliziert!),
dann erst Ausführen der Rechenoperation.

Als Funktionsargument wird üblicherweise immer **int** oder **long** übergeben,
nie **char** oder **short** (automatische Verlängerung!).

Explizite Konvertierung mit "Typecast":

((*type***) (***ausdruck***))**

Weitere Operatoren:

- *Vergleiche:* **<**, **<=**, **>**, **>=**, **==**, **!=** (Unterscheide **=** und **==** !!!!!)
- *Logische Operatoren:*

  && (logisches "und") und **||** (logisches "oder")
  ! ist logisches unäres "not"

  Abgekürzte Auswertung bei **&&** und **||** :
  Wenn das Ergebnis nach der linken Seite schon feststeht, wird die rechte Seite *nicht* ausgerechnet (Beispiel: **`if ((n == 0) || (a % n == 0))`** … ist ok)
- *Bedingter Operator:* *bedingung* **?** *Wahr-Wert* **:** *Falsch-Wert*

  Beispiel: **`sign = (n < 0) ? -1 : 1`**

  Es wird nur *einer* der beiden Werte ausgerechnet!

  Beispiel: **`b = (n == 0) ? a : (a / n)`**
- *Bitweise Operatoren:*

  & (bitweises "und"), **|** (bitweises "oder"), **^** (bitweises "xor")

  ~ (unäres bitweises "not")

  << und **>>** (bitweise nach links bzw. rechts schieben)

  Hinweise:

  - Nicht mit logischen Operatoren verwechseln!
  - Haben teilweise *falsche Priorität* (*unter* den Vergleichen) ==> Klammern!
  - **<<** und **>>** werden oft für Multiplikation und Division mit Zweierpotenzen verwendet.
  - **>>** hat unterschiedliches Verhalten bei **`signed`** und **`unsigned:`**
    `signed`: Vorne kommt eine *Kopie* des ursprünglichen ersten Bits herein
    `unsigned`: Vorne kommt 0 herein
- *Increment und Decrement* ("1 dazuzählen / wegzählen"):

  ++ und **--**

  Achtung: Unterscheide **`++i`** und **`i++`** !!!

  Beide erhöhen `i` um eins,
  aber **`++i`** hat als Ergebnis den *neuen* Wert (zuerst dazuzählen),
  und **`i++`** hat als Ergebnis den *alten* Wert von **`i`** (danach dazuzählen) !

  Die Reihenfolge von Seiteneffekten ist undefiniert!
  Daher: Wenn in einem Ausdruck **`++i`** oder **`i++`** vorkommt,
  darf **`i`** *kein zweites Mal* vorkommen!
  Negatives Beispiel: **`a[i] = b[i++];`** oder **`x[i++] = 1 / i;`**
- *Zuweisungen:* **=**

  Abgekürzte Zuweisungen: **+=**, **-=**, ***=**, **/=**, **<<=**, **&=** usw.
  `n += i;` ist dasselbe wie **`n = n + i;`**

  Zuweisungen haben einen *Wert als Ergebnis* (nämlich den zugewiesenen Wert)

- ==› Mehrfach-Zuweisung: **`x = y = z = 1;`** (von rechts nach links)
- ==› Zuweisung in Ausdrücken:

  `while ((c = getchar()) != EOF)` …

  (deshalb wird **`if (i = 10)`** … statt **`if (i == 10)`** …
  nicht als Fehler angezeigt: Es ist legal!)

Reihenfolge von Zuweisungen in Ausdrücken ist wie bei **`++i`** *undefiniert*!
(Compiler darf Kommutativ- und Assoziativgesetz anwenden …)

- *Beistrich-Operator:* …**`,`** …
  Rechnet zuerst die linke, dann die rechte Seite aus.
  Ergebnis ist der Wert der rechten Seite.

  Verwendungen:
  `while (c = getchar(), c != EOF)` …
  `for (i = 1, j = n; i < j; ++i, --j)` …

`printf`-Formate

Details siehe **`man`**-page!

- **`%d`** … **`int`** als *Dezimalzahl* (mit Vorzeichen) ausgeben (**`%i`** ist dasselbe)
- **`%u`** … **`int`** als *Dezimalzahl ohne Vorzeichen* (“unsigned”)
- **`%x`** … als *Hexzahl* ausgeben
- **`%o`** … als Oktalzahl ausgeben
- **`%c`** … als einzelnes *Zeichen* ausgeben
- **`%%`** … ein **`%`** ausgeben

Erweiterungen:

- **`%6d`** … Feldbreite mindestens 6 Zeichen
- **`%06d`** … dasselbe, mit führenden 0 statt Leerzeichen
- **`%+d`** … auch **`+`** explizit ausgeben, nicht nur **`-`**
- **`%ld`** … der Wert ist ein **`long`**, kein normaler **`int`**
- und vieles mehr…

4. Kapitel: Gleitkomma-Zahlen

Typen:

- **`float`** (Bereich 10^{+-38}, Genauigkeit rund 6 Stellen, 4 Bytes)
- **`double`** (Bereich 10^{+-308}, Genauigkeit rund 16 Stellen, 8 Bytes)
  ==› Nur **`double`** ist praktisch sinnvoll!

Gleitkommazahlen sind

- … exakt für ganze Zahlen bis zur Genauigkeit (also bis 16 Stellen),
- … sonst (Zahlen mit Komma, ganze Zahlen › 16 Stellen):
  Mit Konvertierungs- und Rundungsfehlern behaftet!

Auch *endliche Dezimal-Kommazahlen* können eine *unendliche (periodische) Binär-Kommadarstellung* haben und sind daher mit Rundungsfehlern behaftet (z.B. 0,1 !)!

- … und grundsätzlich *langsamer* als ganze Zahlen!
  ==› Nur verwenden, wo notwendig und sinnvoll!

Gerechnet wird *immer* in **`double`**, automatsiche Konvertierung von **`float`** beim Lesen / Speichern. An Funktionen wird normalerweise **`double`** (und nicht **`float`**) übergeben, und es kommt auch ein **`double`** zurück.

Konstanten:

Entweder mit **`.`** (nicht **`,`** !) oder mit **`e`***nn* oder **`e-`***nn* (groß oder klein) oder mit beidem.

Beispiele: **`1e-15  1.23 27e9 6.66E+222`**

Alle Gleitkomma-Konstanten sind **`double`**.

Operationen:

Grundrechnungsarten **`+ - * /`**, unäres **`-`**, Kurzformen **`+=`** usw., *kein* **`%`**, *keine* Bit-Op's, …

Implizierte Konvertierung nach **`double`** beim Mischen von **`int`** und **`double`**.
(*pro Operator*, nicht für die ganze Rechnung auf einmal!)

Explizierte Konvertierung von **`int`** mittels Typecast: **`((double) i)`**
(besonders bei Division zweier **`int`**!)

Achtung:

Konvertierung in Gegenrichtung (von **`double`** nach **`int`**)
rundet nicht, sondern *schneidet ab*!

(egal ob *explizit* mittels Type-Cast oder *implizit* bei Zuweisung von **`double`** auf **`int`**)

Es gibt zahlreiche *vordefinierte Rundungs-Funktionen*,
für den typischen Fall (kaufmännische Rundung) ist **`i = lround(x);`** die Richtige.

Funktionen in **`math.h`**:

- Winkelfunktionen
- Wurzel, Potenzieren, Logarithmus, e^x
- Konstanten für Pi, e, …
- Absolutbetrag, explizites Runden, …
- Testfunktionen für **`Inf`** und **`Nan`**, Fehler- und Rundungsmodus (siehe unten)
- Und vieles andere mehr!

Vergleiche `< <= > >= == !=`

Achtung bei **`==`** wegen Rundungsfehlern!!! "Trifft" nicht unbedingt genau:
Statt **`x == c`** besser **`fabs(x - c) < epsilon`** wenn wirklich "gleich" gemeint ist
und **`x <= c`** oder **`x >= c`** wenn man sich von einer Seite nähert!

Rundungsmodus: Default "round to nearest", 4 Modi einstellbar.

Overflow, Fehlerverhalten: Einstellbar:

- Entweder: *Weiterrechnen* mit **`+-Inf`** ("infinity") oder **`NaN`** ("not a number")
  (Defaultverhalten, auch bei **`/0`**, negativer Wurzel, …).

- Oder: Abbruch bei Overflow und Fehler.

`printf`-Formate:

- **`%f`** … Fixkomma-Darstellung
  (**`%15.10f`** … 15 Stellen insgesamt, 10 nach dem **`.`**)
- **`%e`** … Exponenten-Darstellung
- **`%g`** … je nach Wert Fixkomma- oder Exponentendarstellung

Anmerkung: Strings

… kommen später im Detail!

Für uns derzeit nur:

Typ **`char[]`** bzw. **`char *`**

"Array von **`char`***"* bzw. *"Pointer auf* **`char`***"*

Zugriff auf einzelne Zeichen mit **`[…]`**, **Index beginnt bei 0!**

Ende-Markierung ist ein 0-**`char '\0'`** (ein Byte mit Wert numerisch 0, nicht Ziffer **`'0'`**!)
(die Länge wird **nicht** explizit mitgespeichert!)

==> Ein String braucht 1 Zeichen mehr Platz, als man Text darin speichern will!!!

Der Leerstring **`""`** belegt daher *ein* Zeichen!

Konstanten:

`"blabla"` in **doppelten** Anführungszeichen (Typ **`const char[]`**)

Nicht verwechseln: Einfache Anführungszeichen sind für einzelne **`char`***-Konstanten!*

Mehrzeilige Konstanten:

- *Alt:*
  `"blabla\`*`newline`*
  `mehr blabla"`
- *Neu:*
  `"blabla"` *newline*
  `"mehr blabla"`

`printf`-Ausgabe mit **`%s`**, bei Feldbreite mit **`-`**: **`%-30s`** … linksbündig statt rechtsbündig

5. Kapitel: Befehle

Siehe Schüler-Folien oder altes Skript…

Besonders wichtig:

- **`break`** bei **`switch`** !
- *Einschränkungen* bei **`switch`** :
  - Der *Typ* des Wertes im **`switch`** muss **`int`**, **`char`** oder ein **`enum`** sein,
    kein **`double`** und kein Pointer (und damit auch *kein String*!).
  - Die Werte nach **`case`** müssen *Konstanten* sein,
    keine Variablen und keine nicht-konstanten Berechnungen.

- *Kein* **;** nach den **()** von **for**, **if**, **while** !
- *Unterschied* **while** und **do {...} while** !

Empfehlungen:

- Bei **for**, **if**, … mit *einem Befehl in zwei Zeilen*: Immer **{...}**, nicht **;**
- Bei *leerem* **for**, **if**, …: Immer **{}**, nie **;** (**;** fällt optisch zu wenig auf!)
- Bei **switch** *immer* **default:** (zumindest eine *Fehlermeldung* ausgeben!)

Schleifenparade:

- *Endlosschleife*: **for (;;) {...}** oder **while (1) {...}**

  (verlassen mit **break** oder **return**)
- Die klassische *Einlese-Schleife*: **while ((c = getchar()) != EOF) {...}**

  (die *Klammern* um die Zuweisung sind *notwendig*,
  denn im Vorrang steht die Zuweisung *unter* dem Vergleich!)
- Dasselbe, *zeilenweise*: **while (fgets(line, sizeof (line), f)) {...}**
- Schleife mit anderem *Weiterzählen* als **++** oder **--**, z.B.:
  1, 10, 100, …: **for (i = 1; i < n; i *= 10) {...}**
  alle einzelnen Bits: **for (b = 1; b; b <<= 1) {...}**
- *2 Variablen*:
  ```
  for (i = 0, j = n; i < j; ++i, --j) {...}
  for (i = 0, b = 1; !(b & n); ++i, b <<=1) {}
  ```
- Durchlaufen einer *Pointerkette* (verkettete Liste):
  ```
  for (p = head; p; p = p->next) {...}
  ```
  Oder ganz böse (Suche im Binärbaum):
  ```
  for (p = root;
       p && (p->data != val);
       p = (p->data < val) ? p->right : p->left) {}
  ```
- Schleifen bis *Stringende*: Zeichen als Bedingung: Implizit **... != '\0'**

  Ermitteln der *Stringlänge*: **for (l = 0; str[l]; ++l) {}**

  Pointer auf *Stringende*: **for (p = str; *p; ++p) {}**

  Stringkopie: **for (i = 0; (dest[i] = src[i]); ++i) {}**
  oder **for (p = dest, q = src; (*p = *q); ++p, ++q) {}**

  Die Bedingungen sind *Zuweisungen, keine Tests*!
  Es ist üblich, zur *optischen Hervorhebung* von Bedingungen,
  die nur aus einer Zuweisung bestehen, *zusätzliche* **( )** zu machen.

6. Kapitel: Funktionen

Funktion = Benannter Programmteil (Befehle + interne Daten),
der von anderen Programmteilen verwendet werden kann.

Alter Name: *"Unterprogramm"*

Bisher bei uns:

- Eine Funktion haben wir *selbst geschrieben*: **`main`**
- Einige vorhandene Funktionen haben wir *verwendet*: **`printf`**, **`sqrt`**, **`atoi`**, …

Zweck von Funktionen:

- Ursprünglich: Arbeitsersparnis
  Mehrmals benutzten Code nur einmal schreiben, Code mehrfach verwenden.
- Dann: Übersichtlichkeit
  Langen Code in überschaubare Einheiten unterteilen.
- Aufteilung in Funktionen heute meist **nach funktionalen Kriterien**,
  um Programme leichter entwickeln / lesen / testen / erweitern zu können:

  Funktionen sind abgeschlossene gedankliche & programmtechnische Einheiten,
  die eine bestimmte logisch zusammenhängende Aufgabe erfüllen / abstrahieren,
  ein bestimmtes Teilproblem lösen, …:

  "Each function should do one thing well!"
- Funktionen strukturieren nicht nur den Code, sondern auch die Daten:
  Jede Funktion hat ihre *eigenen Variablen* (haben *nichts* mit den Variablen in anderen Funktionen zu tun, auch nicht bei Namensgleichheit!).
- Funktionen haben eine wohldefinierte **Schnittstelle zum Rest des Programms**:
  Die **Parameter** werden beim Aufruf hineinkopiert,
  ein **Returnwert** kommt am Ende zurück.

Beispiele: (siehe auch Datumsbeispiel!)

- Berechne den ggT zweier Zahlen
- Sortiere ein Array
- Gib eine Fehlermeldung aus und beende das Programm
- Suche die Internet-Adresse zu einen Hostnamen

Regeln für Funktionen:

- **Sprechende Namen** für Funktionen!
  (C-Standard-Funktionen sind ein *schlechtes* Beispiel!)
- … und ihre Parameter!
  (Klassiker **`strstr`**: "**`s1`**" und "**`s2`**" oder "**`haystack`**" und "**`needle`**"?)
- Faustregel für **Codelänge** pro Funktion:
  Alles bis 20 Zeilen ist gut,
  alles bis 50-100 Zeilen ist zur Not tolerierbar,
  alles darüber gehört in mehrere Funktionen unterteilt!
- **Datenaustausch** mit der Außenwelt primär mittels Parameter und Returnwert, ev. Lesezugriff auf globale Daten, *nur in Ausnahmefällen Änderung globaler Daten.*

Top-Down-Entwurf:

- ***Entwirf dein Programm von außen nach innen, vom Groben zum Feinen:***
  Zuerst **`main`**, dann die dabei eingeführten Toplevel-Funktionen, dann deren Unterfunktionen, …
- Wenn immer du auf einen Teilschritt stößt,
  - der *zu komplex* für direkten Code in der momentanen Funktion ist,
  - oder *gedanklich ein selbständiges Problem* ist,
  - oder *vermutlich öfters vorkommt*,

  ==› ***mach eine Funktion dafür*** und denk erst später über ihre Realisierung nach!

Funktions-Deklaration (***"Prototyp"*** der Funktion genannt):

*Returntyp Funktionsname***`(`***Parameter-Typ*$_1$ *Parameter-Name*$_1$ **`, …);`**

Kündigt an:

- dass es eine Funktion dieses *Namens* gibt,
- *wie viele Parameter von welchem Typ* in die Funktion hineingehen,
- und *welchen Typ ihr Ergebnis* hat.

==› Ab der Deklaration kann die Funktion im Code aufgerufen werden,
davor ist sie unbekannt!

Parameternamen könnte man im Prototyp weglassen,
soll man aber wegen der Leserlichkeit **immer mit angeben**!

Die *Headerfiles* enthalten *vor allem Funktions-Deklarationen!*

Funktions-Definition:

- Wie die Funktions-Deklaration **ohne `;`** und mit **`{`** *code* **`}`** hinten dran.
- Definiert die Implementierung der Funktion (ihren *Code*).
- Name, Parameter-Typen und Returntyp müssen *ident zur Deklaration* sein.
- Darf *nach* den Aufrufen stehen.
- Wenn die Definition einer Funktion *vor allen Aufrufen* steht,
  darf eine separate Deklaration (Prototyp) entfallen.

  Sinnvoll ist eine separate Deklaration aus Stilgründen aber immer,
  notwendig ist sie bei indirekt rekursiven Funktionen.

Sonderfälle:

- Kein Returnwert: **`void func(…);`**
- Keine Parameter: … **`func(void);`**

  In C: ***Nicht*** leerlassen: Kein … **`func()`** !!!
  (leer heißt Anzahl und Typ der Parameter *unbekannt*: Keine Prüfung, alles erlaubt!)

  In C++ ist **`func()`** und **`func(void)`** ident.

  Der *Aufruf* erfolgt immer mit leeren **`()`** , nicht mit **`(void)`** , z.B. **`rand()`** .

Im Code von Funktionen mit **echtem Returntyp**:

- **`return`** immer mit typmäßig passendem ***Wert***.
- *"Unten hinausfallen"* ohne **`return`** ist ***verboten!***

Im Code von Funktionen mit Returntyp **`void`**:

- Vor der **`}`** ist *kein* **`return`** *notwendig,* man darf unten aus der Funktion rausfallen.
- **`return`** immer **ohne** Wert.

Begriffsverwirrung "Parameter" / "Argument": ???
Heute meist:

- ***Parameter*** sind die Variablen in **`()`** in der Funktionsdefinition.
- ***Argumente*** sind die Werte, die zur Laufzeit im Aufruf dafür übergeben werden.

Was fehlt uns von Funktionen noch?

- *Call by Reference*: Wenn wir Pointer durchgenommen haben…
  (echte Referenz-Parameter gibt es nur in C++, nicht in C)
- *Function Pointer*: Siehe z.B. Deklaration **`qsort`**. Kommt nach Pointern kurz…
- *Variabel viele Argumente* (wie bei **`printf`**):
  Lassen wir aus. Bei Bedarf: Siehe **`stdarg.h`** !
- **`inline`**: In C++ häufig, in C weniger wichtig:
  Funktionscode wird nicht aufgerufen, sondern im Aufrufer eingesetzt
  ==› Programm wird etwas schneller (kein Aufruf-Overhead, Optimierungen), aber größer (mehrfacher Code).
  Wirkt nur für nachfolgende Aufrufe, nicht für davor liegende!
  Details bitte selber anschauen…
- *Default-Parameter*: Gibt es in C nicht, nur in C++.

7. Kapitel: Speicherklassen von Variablen

Je nach **Ort der Deklaration** und **Schlüsselwort vor dem Typ**:

- Am Anfang von **`{ }`**: **"lokale Variablen"**
  - Liegen am **Stack**:
    Speicher wird beim Betreten der **`{}`** angelegt,
    beim Verlassen wieder *freigegeben*.
    - ==› Verwendbar nur *innerhalb* der Funktion / der **`{}`**.
    - ==› Behalten ihren Wert ***nicht*** von Aufruf zu Aufruf.
    - ==› Sind bei rekursiven Funktionen *für jeden aktiven Aufruf je einmal* unabhängig voneinander vorhanden (d.h. insgesamt mehrmals!).

    ==› Hineinspringen in **`{}`** mit Variablen ist verboten (Problem Anlegen!).
  - Sind uninitialisiert! (außer bei expliziter Initialisierung)
    ==› zufälliger Inhalt! (und zwar bei *jedem* Aufruf, nicht nur beim ersten!)
    Bei expliziter Initialisierung: Werden bei *jedem* Aufruf neu initialisiert.

    - Sind außerhalb der **{}** unsichtbar!
      ==› *Zwei lokale Variablen gleichen Namens in verschiedenen Funktionen sind erlaubt und haben **nichts** miteinander zu tun!*

- Innerhalb der **()** im Funktionskopf: **"Parameter"**
    - Sind *lokale Variablen*: Nur innerhalb der Funktion bekannt, angelegt bei Funktionsaufruf / vernichtet bei Funktions-Return, eine Instanz pro rekursivem Aufruf, …

      ==› *Ein Parameter hat **nichts** mit einer Variable gleichen Namens im Aufrufer zu tun, weder speichermäßig noch wertmäßig!*
    - Aber: Werden bei jedem Aufruf mit dem Wert, den der Aufrufer übergibt, **initialisiert** (wie durch eine **Zuweisung**)!

      ==› Enthalten zu Funktionsbeginn eine ***Kopie*** des Wertes im Aufrufer, ***nicht das Original!***

      ==› Dürfen in der Funktion wie lokale Variablen geändert werden, aber die ***Änderungen wirken sich nicht auf die Werte im Aufrufer aus***!

      Das heißt

      "Call by Value"

      (Gegenteil: "Call by Reference":
      Pointer-Parameter in C, Referenz-Parameter in C++)

- *Außerhalb von Funktionen: **"globale Variablen"***
    - Bekommen bei Programmstart einen **fixen Platz im Speicher**:
        - ==› Leben und behalten ihren Inhalt, bis das Programm endet.
        - ==› Sind nur <u>*einmal*</u> (für alle Funktionen gemeinsam) vorhanden.
    - Sind automatisch **auf 0 initialisiert**, können bei expliziter Initialisierung nur mit <u>*Konstanten*</u> initialisiert werden (Initialisierung geschieht nicht durch Zuweisung, sondern durch Laden der Init-Werte aus dem .exe-File!).

      (Unterschied C++: In C++ können globale und statische Variablen auch mit Berechnungen initialisiert werden.)
    - Sind in allen Funktionen hinter der Deklaration bis zum File-Ende sicht- und verwendbar (globale Deklarationen stehen normalerweise vor allen Funktionen!), und in allen anderen Files, die mit **`extern`** darauf zugreifen.

- *Am Anfang von **{}**, mit **`static`**: **"statische Variablen"***
    - Sind von der Sichtbarkeit her *lokale Variablen*
      ==› sicht- und verwendbar *nur in dieser Funktion* bis zum Ende der **{}**.
    - Aber sind von der Speicherung her *globale Variablen*:
        - Ein Mal fix im Speicher.
        - Ein Mal zu Programmstart (*nicht bei jedem Aufruf!*)
          mit 0 oder explizit initialisiert.
        - **Behalten** ihren Wert zwischen Aufrufen!

        - Alle rekursiven Aufrufe einer Funktion verwenden *dieselbe* statische Variable, nicht eine neue pro Aufruf!
    - Verwendungsbeispiele:
        - Aufruf-Zähler, Initialisierungs-Merker für den ersten Aufruf, …
        - Gemeinsame Daten für rekursive Aufrufe.
- *Innerhalb oder außerhalb von Funktionen mit* **`extern`**: ***"externe Variablen"***
    - Sichtbarkeit wie lokale Variablen wenn innerhalb von **`{}`** (bis Ende **`{}`**) oder wie globale Variablen wenn außerhalb von **`{}`** (bis File-Ende).
    - Aber: Es wird *kein eigener Speicher* angelegt, sondern auf die globale Variable gleichen Namens *in einem anderen File* zugegriffen!

      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 der nicht-**`extern`**-Deklaration! In den `extern`-Deklaration stehen oft nur leere **`[ ]`** . (viele Compiler liefern *keinen* Fehler bei verschiedener Größe!)
        - Eine *Initialisierung* kann nur in der nicht-**`extern`**-Deklaration angegeben werden!
- *Außerhalb von Funktionen, mit* **`static`**: **File-globale (File-statische) Variablen**.

  Wie globale Variablen,
  aber andere Files können *nicht* mit **`extern`** darauf zugreifen.
  ==› Es kann auch keine Namenskollision mit Variablen in anderen Files geben.

Konzept des **Stacks** (Tafelbild!):

- *Wächst und schrumpft dynamisch bei der Programm-Ausführung*
- Enthält für *jeden* gerade aktiven Funktions-Aufruf:
    - Die *Parameter*
    - Die *lokalen Variablen*
    - Die *Rückkehr-Adresse* (*"wo im Aufrufer muss ich nach dem Return weitermachen?"*)

"Shadowing":

Eine lokale / statische Variable oder ein Parameter *"verstecken"*
in ihrem Sichtbarkeits-Bereich eine globale Variable *gleichen* Namens
==› die globale Variable ist in diesem Bereich nicht sicht- und zugreifbar!
(Ausnahme: In C++ mit **`::`**)

Verwirrend, fehleranfällig ==› vermeiden!!!

8. Kapitel: Arrays

Array (deutsch: "Feld") =

Eine Variable für eine fixe Anzahl von Werten (= "Elementen") gleichen Typs
(z.B. 100 **`char`**'s oder 10 **`double`**'s).

Die Elemente sind durchnummeriert,
der Zugriff erfolgt mittels Nummer (= "Index", "Subscript").
Die einzelnen Werte stehen im Speicher "dicht auf dicht" nacheinander.

Deklaration:

z.B.: **`int a[10];`** … Array **`a`** mit 10 **`int`**-Werten

- "Klassisches" C (und C++):
  Die Array-Größe (Element-Anzahl) muss immer eine **Konstante** sein!
  Sonst: **`malloc`** oder **`alloca`** (siehe unten) verwenden.
- Seit C99 (nicht in C++!):
  Lokale Arrays dürfen nicht-konstante Größe haben (z.B. **`int data[argv];`**),
  globale Arrays müssen konstante Größe haben.

Deklaration *ohne Größe* (mit *leeren* **`[]`**):

- Bei *externen* Arrays (aus anderem File) unbekannter Größe.
- Bei *Funktionsparametern*
  (die Größe von Array-Parametern ist *immer unbekannt*, siehe unten!).
- Wenn die Größe durch die *Initialisierung* festgelegt wird
  (Größe = Anzahl der angegebenen Init-Werte).

Zugriff:

z.B. **`sum += a[i];`** oder **`str[0] = '\0';`** oder **`if (x[i] < 0)`** …

wobei der Index bei *size* Elementen von **`0`** bis ***size – 1*** geht!
(es gibt ***kein*** **`a[size]`** !)

C prüft ***nicht***, ob der Index innerhalb der Array-Grenzen liegt!!!
"Fremde" Daten werden gelesen / überschrieben
==› "Zufällige" Programmfehler, Absturz!

Weil: *Die Größe eines Arrays ist zur Laufzeit **unbekannt**! (wird **nicht** mitgespeichert!).*

Der Index-Ausdruck muss sowohl bei der Deklaration als auch beim Zugriff ein *ganzzahliger* Typ (**`int`** oder **`char`**) sein, **`double`** ist unzulässig.

Arrays können ***nicht*** als Ganzes

- mit **=** zugewiesen werden: **`a = b`** für zwei Arrays ist illegal
- mit **==** usw. verglichen werden: **`a == b`** für zwei Arrays ist erlaubt, aber stets 0
- mit Operatoren **`+ – *`** usw. verknüpft werden
- als Ganzes ausgegeben (**`printf`**) oder eingelesen werden
- "By Value" (als Kopie) als Parameter oder Returnwert übergeben werden
  (siehe unten!)

==> *Elementweise* mittels **for-Schleife** machen!

Mehrdimensionale Arrays:

Deklaration und Zugriff auf Elemente mit **[i][j]**, nicht mit **[i, j]**
(mehrdimensionale Arrays sind gedanklich und technisch *"Arrays of Arrays"*).

Initialisierung:

```
int a[] = {1, 2, 4, 8, 16};
int a[][2] = {{1, -1}, {-1, 1}};   // geschachtelte Init-Werte
const char msg[] = "Fehler!";
```
(**msg** ist der String selbst, **msg** ist konstant!)
```
const char *p = "Fehler!";
```
(**p** ist ein (änderbarer!) Pointer, der anfangs auf den angegebenen String zeigt)
```
char *mon[] = {"Jan", "Feb", ... , "Dez"};
```
(**mon** ist ein Array von 12 Pointern, die auf die angegebenen Strings zeigen)

Wenn eine Initialisierung *und* eine Feldgröße angegeben sind, und die Initialisierung weniger Werte enthält als das Array laut Feldgröße hat, so werden die ersten Werte gemäß Initialisierung initialisiert *und die restlichen auf 0 gesetzt.*

Praktischer Trick daher: **int a[256] = { 0 };** setzt *alle* Elemente des Arrays auf 0 (klappt *nicht* für Arrays, deren Größe keine Konstante ist).

sizeof

sizeof kann

- entweder mit einem *Typ* aufgerufen werden (z.B. **sizeof(int)**),
- oder mit einem *Wert* (z.B. **sizeof(a[0])** , auch ohne **()** zulässig).

In beiden Fällen liefert es die **Anzahl der Bytes**, die der Typ bzw. Wert belegt.
Auch wenn **sizeof** nach Funktion aussieht: Es ist ein *eigenes Sprachkonstrukt*,
das *vom Compiler berechnet* wird (nicht zur Laufzeit!).

sizeof(char) ist per Definition 1.

Trick für *Anzahl der Elemente* eines Arrays **a**: **sizeof(a) / sizeof(a[0])**

Funktioniert *nur* bei lokal oder global deklarierten Arrays,
nicht bei externen oder als Parameter übergebenen!
(**sizeof** eines mit **[ ]** deklarierten externen Arrays ist undefiniert,
sizeof eines Array-Parameters ist 4 bzw. 8,
weil – siehe unten – ein *Pointer* auf das Array übergeben wird
und **sizeof** die *Größe des Pointers* liefert!)

Achtung:

Unterscheide bei Strings sizeof und strlen!

- **sizeof** ist die fixe, bei der Deklaration angegebene, *im Speicher reservierte Größe* des Arrays (d.h. die Anzahl der Zeichen *incl.* **\0**, die *maximal Platz haben*),
  ganz unabhängig vom String, der gerade im Array gespeichert ist.
- **strlen** liefert die *aktuelle Länge* des im Array gespeicherten Strings bis zum **\0**, d.h. die momentane Anzahl der Buchstaben *excl.* **\0**,
  ganz unabhängig davon, wie groß das Array ist.

9. Kapitel: Pointer

Pointer =

- **technisch**: *Hauptspeicher-Adresse*
- **logisch**: *Zeiger (Verweis) auf (schon existierende!) Daten*

Operationen:

- "*****" (*"Dereferenzierung"*):
  ***p**... der *Wert*, auf den der Pointer **p** zeigt

  Kann gelesen und geschrieben werden: **`c = *p;`** und **`*p = '\0';`**
  Auch: **`if (*p == '\n') {...}`**
- "**&**" (*"Address-Of-Operator"*):
  &i ... liefert einen *Pointer* auf die Variable `i`
  &(a[n]) ... liefert einen Pointer auf das **n**-te Element von **a**:
  Beispiel: **`end = &(a[size]);`**
  (Pointer eins vor oder eins hinter das Array sind zulässig!)

Deklaration:

`int *p;` ... "**p** ist ein Pointer auf **`int`**-Werte", gleichbedeutend mit "***p** ist ein **`int`**"

Auch: **`char **p;`** ... "**p** ist ein Pointer auf einen **`char`**-Pointer"

Achtung: Der ***** gehört vor *jede einzelne* Variable!
Falsch: **`int *p1, p2;`** // *ein* Pointer und ein **`int`**
Richtig: **`int *p1, *p2;`** // *zwei* Pointer

Achtung:
"Neue" Pointer enthalten so wie neue Variablen zufällige Werte (*zeigen "irgendwohin"*)!

Achtung:

- **`int *p[10];`** ... **p** ist ein Array von 10 Pointern auf **`int`**
  `char *argv[]` ... **argv** ist ein Array unbekannter Größe von Pointern auf **`char`**

  Implizite Klammerung wie beim Zugriff:
  `*(p[i])` ... der **`int`**, auf den das **i**-te **p** zeigt.
  Zuerst die Indizierung **[]**, und dann den Pointer, der da rauskommt, mit ***** dereferenzieren.
- **`int (*p)[10];`** ... **p** ist ein (einzelner!) Pointer auf ein Array von 10 **`int`**'s

  Analog zum Zugriff:
  `(*p)[i]` liefert den **i**-ten **`int`** aus dem Array, auf das **p** zeigt:
  Zuerst die Dereferenzierung *****, dann die Indizierung **[]**

  Wird in der Praxis *nie verwendet*: Man unterscheidet nicht zwischen "Pointer auf einen einzelnen Wert" und "Pointer auf ein Array", sondern verwendet für Arrays "Pointer auf den ersten Wert des Arrays".

Anwendungen von Pointern:

- *Call by Reference:*

  Die aufgerufene Funktion bekommt einen *Pointer auf Daten im Aufrufer* übergeben

==> Die aufgerufene Funktion kann die Daten im Aufrufer *ändern!*

Z.B.: **`scanf("%d", &input);`** oder **`scanf("%lf", &(a[i][j]));`**

Call by Reference, Beispiel:

```
void swap(int *a, int *b)
{
  int t;
  t = *a;
  *a = *b;
  *b = t;
}

swap(&(a[i]), &x);
```

- *Durchlaufen von Arrays* (vor allem Strings), Ersatz für Array + Index: Siehe unten.
- *Dynamisch angelegter Speicher* (**`malloc`**): Demnächst...
- *Dynamische Datenstrukturen* (verkettete Listen, Bäume, ...): Später (3. Sem)!

Der Null-Pointer: `NULL`

- Spezieller vordefinierter Pointer-Wert, zeigt "auf ***nichts***".

  Zugriff auf **`*p`** wenn **`p`** **`NULL`** ist: Garantierter ***Absturz!***
- Hat den Wert **`0`** (**`false`**), kann direkt in **`if`**, **`while`** usw. geprüft werden:

  `while (p)` ist dasselbe wie **`while (p != NULL)`**
  `if (!p)` ist dasselbe wie **`if (p == NULL)`**

  Übliche Definition von **`NULL`** (in **`stdio.h`**): **`#define NULL ((void *) 0)`**

  Achtung: In C++ sind die Dinge komplizierter, dort verwendet man besser direkt `0`.

Sonderfall `void *`

Nicht "Pointer auf nichts", sondern *"Pointer auf Daten unbekannten / beliebigen Typs"*.

`void *` -Pointer kann man ohne vorherigen Typecast nicht dereferenzieren!
(weil der Compiler ja nicht wissen kann, welchen Typ das Ergebnis von **`*p`** hat)

`const` und Pointer:

- **`const char *p`** : **`p`** ist ein (nicht-konstanter) Pointer auf *konstante* **`char`**'s
  ==> **`p`** kann geändert werden, aber auf **`*p`** darf nicht zugewiesen werden.

  `char const *p` ist dasselbe.
- **`char *const p`** : **`p`** ist ein *konstanter Pointer* auf (nicht-konstante) **`char`**'s
  ==> **`p`** selbst darf nicht geändert werden, aber **`*p`** schon.
- **`const char *const p`** : Weder **`p`** noch **`*p`** dürfen geändert werden.

`sizeof` *und Pointer:*

- **`sizeof(p)`** ist immer 4 (auf 32-bit-Systemen) bzw. 8 (auf 64-bit-Systemen).
- **`sizeof(*p)`** ist die Größe *eines* Wertes (auch wenn **`p`** auf ein Array zeigt!).

Sonderfall "Zeiger auf Funktion":

- Deklaration z.B. **`int (*fp)(char *, char *);`**

  ==› **`fp`** zeigt auf eine Funktion
  mit Returntyp **`int`** und zwei Argumenten vom Typ **`char *`**
- Aufruf z.B. **`result = (*fp)(str1, str2);`**
- Vorkommen z.B. als Compare-Funktions-Parameter von **`qsort`**.

Pointerarithmetik:

- Bei Pointern, die auf *Elemente eines Arrays* zeigen.
- Rechnet in Einheit "Elementen", nicht in Bytes!
  (d.h. wird je nach Elementgröße mit 2, 4, ... skaliert!)

Erlaubt sind:

- **`p + i`** und **`p - i`** : Ein Pointer auf das **`i`**-te Element hinter / vor **`p`**.

  Daher: **`++p`** und **`--p`** : Lässt **`p`** auf das *nächste / vorige Element* zeigen.

  Achtung:
  `*(p++)` ... returniert den Wert, auf den **`p`** zeigt, und lässt **`p`** auf das Nächste zeigen.
  `(*p)++` ... zählt 1 zu dem Wert, auf den **`p`** zeigt (und lässt **`p`** gleich).
- **`p - q`** : Abstand (Anzahl der Elemente) zwischen **`p`** und **`q`** (ein **`int`**, kein Pointer!)
- **`p < q`** : Zeigt **`p`** auf ein Element vor **`q`** ? (analog **`== != <= > >=`**)

Arithmetik auf **`void *`**-Pointern ist verboten (weil die Elementgröße unbekannt ist)!

Arrays und Pointer:

- Ein *Array ohne* **`[ ]`** wird automatisch zu einem *Pointer auf den Array-Anfang*:

  ==› **`a`** ist dasselbe wie **`&(a[0])`**

  ==› **`a[i]`** ist dasselbe wie **`*(a + i)`** (**`*a`** ist daher **`a[0]`**)

  ==› **`&(a[i])`** ist dasselbe wie **`a + i`**

  ==› **`&a`** ist sinnlos

  Man darf **`a`** aber nicht ändern: **`a = ...`** oder **`++a`** ist für Arrays *verboten*!
- Daher: Operationen mit Arrays und Pointern:
  - **`p = a`** : In **`p`** wird ein Pointer auf den Anfang von **`a`** gespeichert.
  - **`p == a`** : Zeigt **`p`** auf den Anfang von **`a`** ?
- **Bei mehrdimensionalen Arrays:**

  Eine teilweise Indizierung (**`a[i]`** bei zweidim. Array)
  liefert einen *Pointer auf das* **`i`***-te Zeilen-Array*.
- Umgekehrt: Alle Pointer dürfen auch mit **`[ ]`** *indiziert* werden:

  `p[i]` ist dasselbe wie **`*(p+i)`**

Arrays (bzw. Strings) als Funktionsparameter:

==› Die Funktion bekommt einen *Pointer* auf den Array-Anfang übergeben.

==› Das Array wird *"by Reference"* übergeben
(da man Arrays ja nicht "by Value" übergeben kann).

==› Die Funktion bekommt *keine Kopie* des Arrays,
sondern greift direkt auf das ursprüngliche Array *im Aufrufer* zu.

==› Die Funktion kann die Werte des Arrays im Aufrufer *ändern*!

==› Im Aufruf das "Mittelstück" eines Arrays zu übergeben ist kein Problem:

```
n = myfunc(&(a[offset]));
```

Deklaration des Array-Parameters in der Praxis üblicherweise immer als Pointer, nicht als Array:

- Meist **`char *strfunc(char *s1, char *s2)`**
- … und nicht **`char *strfunc(char s1[], char s2[])`** (leere **`[]`** !),

  obwohl das bei Parametern genauso erlaubt ist und dasselbe bedeutet,
  und das zweite eigentlich *viel besser zu lesen* ist!
  ("ich will hier einen Pointer auf ein Array übergeben,
  und nicht einen Pointer auf irgendeinen einzelnen Wert!")

Üblich und sinnvoll bei Array-Parametern,
die *nur gelesen und nicht geändert* werden sollen: **`const`**

- Verhindert versehentliches Ändern der Werte im Aufrufer (==› Compilerfehler)!
- Erleichtert das Lesen: *"Dieser Parameter geht nur in die Funktion hinein!"*

Beispiel für eine Funktion mit Array-Parametern:

```
char *strcpy(char *dest, const char *src)
{
  char *result = dest;
  for ( ; (*dest = *src); ++dest, ++src) {}
          // das = ist eine Zuweisung, kein Vergleich!
  return result;
}

char initWert[] = "foobar";
char meinText[40];
strcpy(meinText, initWert);
```

Achtung!!!

C "weiß" weder zur Compile-Zeit noch zur Laufzeit,

- ob ein Pointer auf einen *einzelnen Wert* oder auf ein *Array* von Werten zeigt,
- wie *groß* das Array ist, auf das der Pointer zeigt,
- ob ein Pointer überhaupt auf einen *gültigen Speicherbereich* zeigt
  (oder ob die Daten z.B. längst schon wieder freigegeben wurden),
- und ob dort wirklich Daten des erwarteten *Typs* liegen (oder ganz andere).

==› Keine Fehlerprüfung / Fehlererkennung!

==› Wenn ein Array keinen "Ende-Wert" (wie **`'\0'`** bei Strings) hat:

Größe als 2. Parameter mit-übergeben!!!
(sonst hat die Funktion keine Chance,
die Größe des übergebenen Arrays herauszufinden!)

Aber (siehe Schüler-Folien!):

Mehrdimensionale `[][]` als Parameter sind nicht möglich!

- Nur die erste darf **`[]`** sein, die anderen **`[]`** müssen eine *fixe Größe* haben!
- Oder: Das Array *eindimensional* übergeben und die Indexrechnung selber machen!
- In C statt mehrdimensionalen Arrays oft: **Array von Pointern auf Arrays**, z.B. **`argv`**

  ==› Der Zugriff schaut genauso aus, trotz *ganz anderer Darstellung im Speicher*!

  ==› Die einzelnen Zeilen können *verschieden lang* sein!

Arrays als Returnwert:

Gibt's nicht!

Weil:

- Der Returnwert kann nicht ein Array als Wert sein, denn es ist nicht möglich, den gesamten Inhalt eines Arrays so wie einen einzelnen Wert in den Aufrufer zurückzukopieren und dort in einem Array zu speichern
  (Arrays können nicht als Ganzes kopiert werden).
- Der Returnwert könnte ein Pointer auf ein in der Funktion deklariertes Array sein, in dem das Ergebnis gespeichert ist, aber ein solches lokal deklariertes Array wird beim Return automatisch freigegeben!

  ==› *der Pointer zeigt nach dem Return auf **undefinierte** Daten*
  ("ins Leere" bzw. auf freigegebenen Speicher)!

  Achtung: Der Compiler erkennt ***keinen*** Fehler!

==› Niemals Pointer auf lokale Variablen oder Arrays
als `return`-Wert zurückgeben!

4 Varianten zur Rückgabe eines Arrays aus einer Funktion:

- ***Das Ergebnis-Array zuvor im Aufrufer anlegen***
  und "by Reference" als Parameter übergeben (wie bei **`strcpy`**)

  ==› Die aufgerufene Funktion schreibt ihr Ergebnis direkt in den Aufrufer.

  Das ist die häufigste und meist auch beste Variante, *aber*
  die aufgerufene Funktion ist an die vom Aufrufer angelegte Arraylänge gebunden, d.h. der Aufrufer muss wissen, wie lang das Ergebnis (maximal) werden kann.

  Wenn die Array-Größe keine Konstante ist:
  Sinnvollerweise die Größe des Ergebnis-Arrays *als 2. Parameter ebenfalls übergeben!*
  (sonst weiß der Aufgerufene nicht, wie viele Daten im Ergebnis-Array Platz haben!)
  (deshalb: **`strcpy`** *ist böse!*)

- ***Returnwert = Pointer auf ein statisch in der Funktion angelegtes Ergebnis-Array.***

  Das wird historisch bedingt bei einigen C-Standard-Funktionen gemacht,
  ist aber ganz schlecht:
  Der vorige Returnwert wird beim nächsten Aufruf der Funktion *überschrieben*!
  (oder von einem parallel laufenden Aufruf der Funktion!)

  ==› Ungeeignet für Programme mit mehreren parallelen Threads!
  ==› *Don't do that!!!*

- ***Das Ergebnis in einem globalen Array ablegen.***

  Je nach Situation vertretbar oder nicht…
  Hat dasselbe *Überschreib-Problem* wie ein statisches Ergebnis-Array!

- ***Das Ergebnis-Array dynamisch mit*** **`malloc`** (siehe unten) ***anlegen und einen Pointer darauf returnieren.***

  Das ist die *flexibelste* (ohne vorab-Längenbeschränkung),
  aber auch die "*teuerste*" Variante (**`malloc`** kostet Zeit!).

  Der Aufrufer darf nicht vergessen, den Returnwert mit **`free`** wieder freizugeben,
  wenn er ihn nicht mehr braucht! (sonst: Memory Leak!).

10. Kapitel: Strings, Mem- und String-Funktionen

Unterscheide:

- **`mem`**-Funktionen:

  Arbeiten immer auf *angegebener Anzahl von Bytes*, *unabhängig* von **`\0`**-Bytes.

- **`str`**-Funktionen:

  Arbeiten *bis zum* **`\0`**-*Byte*, *wissen nicht, wie lang das Array ist!*

  - ==› *Laufen über die Array-Grenze*, wenn das **`\0`**-Byte in einem String *fehlt*!
  - ==› *Überschreiben fremde Daten* hinter dem String,
    wenn das *Ziel-Array zu kurz* für das Ergebnis ist!
  - ==› Bytes im Array hinter dem **`\0`** bleiben *unberührt*
    (undefiniert, zufälliger Inhalt!).

Liste der Funktionen: Siehe C-Funktions-Spickzettel.

Achtung bei **`strcpy`**, **`strcat`**, …:

Das Ziel-Array muss *genug Platz* haben,
sonst werden Daten hinter dem Array *überschrieben*!

`strncpy` ist **nur teilweise besser**:

- Es schreibt zwar nie zu weit,
  aber es returniert bei zu langem Input einen String *ohne* **`\0`**-Terminierung
  ==› Das nächste Lesen geht zu weit!
  ==› Trotzdem Länge händisch prüfen und ev. letztes Byte auf **`\0`** setzen!!!
- Es ist ineffizient: Es schreibt nicht nur ein **`\0`**,
  sondern es *füllt den Zielstring* immer bis zur angegebenen Länge mit lauter **`\0`** !

Tipp:
`strchr` kann auch nach dem **`\0`** suchen ==› Liefert einen *Pointer auf das String-Ende*.

Typische Schleifen beim selbst Programmieren:

```
for (i = 0; str[i]; ++i) {...}
for (p = str; *p; ++p) {...}
```

Überlegen beim Definieren von String-Variablen:

Brauche ich ein Array oder einen Pointer?

- Wenn die Buchstaben *schon wo gespeichert* sind (z.B. im Aufrufer), brauche ich *nur einen Pointer* darauf.
- Wenn ich *Platz für die Buchstaben selbst* brauche, brauche ich ein *Array* (z.B. zum Einlesen von einem File, zum Umkopieren, …) oder einen Pointer und ein **`malloc`** .
- Wenn ich einen *Pointer* statt einem Array *als Zielstring* in **`strcpy`**, **`strcat`**, **`fgets`** usw. angebe, muss ich ihn *zuerst auf ein Array* zeigen lassen, wo Platz ist (Zuweisung eines fixen Arrays oder **`malloc`**).
- Achtung bei *mehreren String-Pointern auf dasselbe Array*:

  Eine Änderung im Array ändert das, was alle Pointer sehen!
  (d.h. es ändern sich alle Strings!)

National Language Support

Strings wie hier besprochen funktionieren nur mit **8-bit-Zeichensätzen**, d.h. ein Zeichen entspricht genau einem **`char`**.

Wir werden uns auch auf derartige Zeichensätze beschränken.

Problem: Solche Zeichensätze sind *auf 256 Zeichen beschränkt.*
De facto:

- **ASCII-Teil** (< 128) ist *fix*: 33 Steuerzeichen, 95 sichtbare Zeichen
- "Obere Hälfte" (128…255) ist *je nach Land / Zeichensatz unterschiedlich*: Umlaute usw.

Beispiele: ASCII, ISO Latin 1 = ISO 8859-1 (Westeuropa), ISO Latin 9 = ISO 8859-15 (dasselbe mit Euro-Zeichen), Microsoft CP 850 (DOS) oder CP 1252 (Windows).

Wenn zur Codierung von Zeichen **Unicode** verwendet wird, ist eine andere String-Darstellung erforderlich, weil Unicode ja nicht auf Zeichenwerte zwischen 0 und 255 beschränkt ist, sondern Zeichenwerte zwischen 0 und rund 100000 kennt.

Es gibt *2 verschiedene Darstellungen* von Unicode-Strings:

- **UCS-2** (bzw. der Nachfolger UTF-16) war der erste Ansatz zur Unicode-Darstellung und wird heute vor allem in der **Microsoft-Welt** und in **Java** verwendet (und z.B. auch in **WxWidgets**). Jedes Zeichen belegt *fix 2 Bytes* (16 Bits), das Stringende wird daher durch einen *2-Byte-Nullwert* angezeigt (einzelne Nullbytes kommen in UCS-2-Strings massenhaft vor: Das obere Byte jedes ASCII-Zeichens ist 0).

C hat für solche Zeichen einen eigenen Typ "**wchar**" ("**wide character**"), ein UCS-2-String ist daher ein **wchar *** oder **wchar []**. Für die Verarbeitung sowie die Ein- und Ausgabe solcher Strings braucht man einen komplett *eigenen Satz von Funktionen* (z.B. **wcscpy** statt **strcpy**, **wprintf** statt **printf** usw.).

UCS-2 hat mehrere **Nachteile**:

- Es braucht *doppelt so viel Platz* wie normale Strings.
- Es erfordert *umfangreiche Programmänderungen*.
- Es ist *Big-Endian / Little-Endian-abhängig* (kommt das höherwertige oder das niederwertige Byte zuerst im Speicher / im File?), d.h. UCS-2-Daten sind plattform-spezifisch.
- 2 Bytes *reichen nicht* zur Darstellung aller Unicode-Zeichen.

- Am weitesten verbreitet ist heute **UTF-8** (ISO 10646, RFC 3629). Es ist in der **Linux/Unix-Welt** üblich und als Unicode-Darstellung für das **Internet** (Webseiten) standardisiert.

  In UTF-8 belegt 1 Zeichen je nach seinem Codewert *zwischen 1 und 4 Bytes*, wobei die ASCII-Zeichen alle *nur 1 Byte* brauchen (Umlaute: 2 Bytes, chinesische & arabische Schriftzeichen, ...: 4 Bytes). Solange keine Umlaute vorkommen, ist die UTF-8-Darstellung eines Strings daher *ident zur bisherigen ASCII-Darstellung*. Erhalten bleibt auch das *\0-Byte als Endmarkierung*, innerhalb eines UTF-8-Strings kommt kein 0-Byte vor.

  UTF-8 hat den Vorteil, dass der Typ derselbe bleibt (**char**), und auch die bisherigen Stringfunktionen und Codestücke größtenteils unverändert weiter funktionieren, sogar der Vergleich von Strings (**strcmp**) liefert korrekte Ergebnisse.

  Der einzige Unterschied ist, dass die *Stringlänge in Bytes im Speicher nicht mehr wie bisher der Stringlänge in dargestellten Zeichen am Bildschirm oder Papier entspricht* (d.h. **strlen** liefert *nicht* die Anzahl der sichtbaren Zeichen im String): Ausgabe-Formatierungen und Längenberechnungen müssen daher modifiziert werden.

11. Kapitel: Dynamische Speicherverwaltung

Es gibt neben dem **Stack** (für die *lokalen Variablen*) und dem **fixen Datenbereich** für *globale und statische Variablen* noch einen dritten Datenbereich:
Den **Heap** für globale, zur Laufzeit *explizit angelegte / freigegebene* ("dynamische") Daten.

Im Unterschied zu den uns bisher bekannten Daten sind Daten am Heap *"anonym"*:

- Da es keine Deklaration dafür gibt, gibt es auch **keinen Variablennamen** dafür.
- Der Zugriff ist **nur über den Pointer** möglich, den man beim Anlegen der Daten bekommt.
- Die Daten existieren, bis man sie explizit wieder freigibt, und jeder, der den Pointer kennt, kann darauf zugreifen: *Lebensdauer und Sichtbarkeit* sind weder an **{ }**-Grenzen noch an Funktionen noch an File-Grenzen gebunden.
- **"Verliert"** man den Pointer (ohne vorheriges Freigeben der Daten),
  - so sind die Daten zwar *noch vorhanden* (d.h. belegen Speicher!),

- aber man kann *weder* darauf *zugreifen*,
- *noch* kann man sie jemals wieder *freigeben*.

Funktionen zur dynamischen Speicherverwaltung

1. **Anlegen** mit **`malloc(`***bytes***`)`**: *bytes* ist Anzahl der *Bytes*, die man haben möchte.

   Returniert einen **`void *`**-*Pointer* auf frisch reservierten Platz mit *bytes* Bytes oder **NULL**, wenn *kein Platz* mehr frei ist.
   ==› Auf **NULL** prüfen!
   ==› Auf richtigen Pointer-Typ casten!

   Die neuen Bytes sind *uninitialisiert* (enthalten *beliebige* Werte)!

2. Alternative Anlege-Funktion: **`calloc(`***elems***`, `***size***`)`**:
   Reserviert Platz für *elems* Elemente mit je *size* Bytes,
   initialisiert den Platz auf 0, sonst wie **`malloc`**.

3. Alternative Anlege-Funktion für **Strings**: **`strdup(`***str***`)`**:
   Returniert einen Pointer (**`char *`**, nicht **`void *`**)
   auf eine *dynamisch angelegte Kopie des Strings* *str*:
   Macht intern im Wesentlichen ein **`strlen`**, ein **`malloc`** und ein **`strcpy`**.

4. **Freigeben** mit **`free(`***p***`)`**.
   Nachher *nicht mehr* auf ***`p`*** *zugreifen*!!!
   (der freigegebene Platz wird von **`malloc`** wiederverwendet!)

5. **Größe ändern** mit **`realloc(`***p***`, `***bytes***`)`**
   ==› Ändert die Größe des Platzes, auf den *p* zeigt, auf *bytes* Bytes,
   liefert einen Pointer auf den geänderten Block.
   ==› Die im alten Block gespeicherten *Daten* bleiben *erhalten*.
   Bei Vergrößerung: Wenn möglich "vor Ort", sonst wird der Block *umkopiert*
   ==› kann das **alte** *p*, einen **neuen Pointer** oder **NULL** liefern!

Beispiel:

```
int *p;
p = (int *) (malloc(num_elems * sizeof(int)));
```

Die Tücken von `malloc` und `free`

Fallen bei `malloc` und `free`:

- "**Memory Leak**": Man vergisst auf das **`free`**:
  Je länger das Programm läuft, umso mehr wird der Speicher durch Daten belegt, die nicht mehr gebraucht werden und nicht mehr erreichbar / freigebbar sind
  ==› irgendwann ist der Speicher voll…
- "**Access after Free**": Man greift nach dem **`free`** noch auf den Speicherbereich zu, auf den der Pointer zeigt
  ==› beim Lesen kommt "irgendetwas", beim Schreiben meist ein Absturz!
- "**Double free**": Man ruft für denselben Datenblock zwei Mal **`free`** auf: Absturz!

- **free** eines Pointers, der gar nicht von einem **malloc** kam: Absturz!
  (z.B. **free** lokaler oder globaler Variablen, oder **free** eines Pointers,
  der mitten in einen **malloc**-Block statt auf dessen Anfang zeigt).
- **Speicherüberschreiber:**

  Auch bei **malloc**-Daten *prüft C nicht,*
  ob ein Zugriff innerhalb des allokierten Speicherblocks liegt
  oder über dessen Anfang / Ende hinausgeht

  ==› es werden "zufällige" Daten gelesen bzw. überschrieben
  (entweder von anderen **malloc**-Blöcken oder interne **malloc**-Verwaltungsdaten)
  ==› bei letzterem: Absturz!

Das Gemeine an diesen Abstürzen:
Sie treten nicht sofort auf, sondern irgendwo und irgendwann später...

Im 4. Semester lernen wir speicher-prüfende **malloc**-Implementierungen
und speicher-prüfende Compiler/Interpreter für C kennen.

Achtung:
Der Heap kann im Laufe der Zeit **"fragmentieren"**: Es gibt zwar noch freien Speicher,
aber er ist auf viele unbrauchbar kleine Blöcke aufgesplittert.
==› **malloc** eines großen Blockes schlägt trotz freiem Speicher fehl!

Achtung:
Unix- und Linux-Systeme sind meist standardmäßig auf **"Overcommit memory"**
konfiguriert (Windows nicht):

Speicheranforderungen
(nicht nur Heap-Anforderungen, sondern alle Arten von Speicherbelegungen)
werden auch dann *erfolgreich* durchgeführt,
wenn das System den *Speicher gar nicht mehr hat*
(in der Hoffnung, dass der Speicher nie wirklich in vollem Umfang genutzt wird).

Das spart viel Swapspace und ist (wegen der Systemkonzepte und Programmier-Modelle
von Unix) meist eine sinnvolle und sehr effiziente Strategie (weil tatsächlich im Schnitt
über 80 % des angeforderten bzw. theoretisch belegten Speichers ungenutzt bleiben!),
bewirkt mitunter aber, dass Programme nachträglich vom System gekillt werden, wenn
sie das erste Mal auf Speicher zugreifen, den sie zuvor schon erfolgreich allokiert haben,
weil das System für den schon zugesagten virtuellen Speicher keinen freien realen
Speicher oder Swapspace mehr findet.

Dynamische Stack-Allokation: alloca

Wie **malloc**, aber die Daten werden *am Stack allokiert*.

==› Die Daten werden beim Return der Funktion *automatisch wieder freigegeben*!
(*nicht* mit **free** freigeben!!!)

alloca ist nicht standardisiert und in vielen Programmierrichtlinien verboten,
aber in manchen Situationen durchaus wertvoll und *viel effizienter* als **malloc** / **free**.

Es stammt aus der Zeit vor C99, als C noch keine variabel großen lokalen Arrays kannte.
Da variabel große lokale Arrays in C++ noch immer nicht offiziell unterstützt werden,
hat **alloca** in C++ durchaus noch eine Daseinsberechtigung.

12. Kapitel: File-I/O

Der Zugriff auf Files (*Dateien, aber auch das Terminalfenster, Pipes zwischen Programmen, Netzwerkverbindungen, I/O-Geräte, …*) erfolgt durch den Aufruf vordefinierter Funktionen aus **`stdio.h`**:

- Ein File muss zuerst **geöffnet** werden. Dabei werden der **Dateiname**, die **Richtung** (lesen / schreiben / beides, wobei beides nur in Ausnahmefällen sinnvoll ist) und im Falle des Schreibens zusätzliche Flags (Überschreiben oder anhängen?) angegeben. Die häufigste Funktion dafür ist **`fopen`** .

  Nicht existierende Files, die man zum Schreiben öffnet, werden *neu angelegt*.

- Als Ergebnis liefert das **`fopen`** einen **Pointer** auf ein Library-internes **File-Objekt** (**`FILE *`**), das die Verwaltungsinformationen zu diesem File enthält.

  Alle weiteren Zugriffe auf den File erfolgen *über diesen Pointer*.

  Das "Innenleben" des File-Objektes, auf die ein **`FILE *`** zeigt, ist irrelevant.

- Der Zugriff auf Files erfolgt normalerweise **sequentiell**, als ob der File eine **Folge von Bytes bzw. Buchstaben** wäre: Jeder Aufruf einer I/O-Funktion auf den File liest oder schreibt die *nächsten* paar Bytes.

  Die Library bzw. das Betriebssystem merken sich intern die **aktuelle Lese- bzw. Schreibposition** (man kann sie mit **`ftell`** abfragen).

  Es gibt zwar auch Funktionen zum *Umpositionieren* (**`fseek`** oder **`rewind`**), aber vor allem **`fseek`** ist sehr *langsam*!

- Lesende Funktionen melden einen Fehler, sobald das **Ende der Datei** erreicht ist. Die Funktionen **`feof`** und **`ferror`** erlauben die Unterscheidung zwischen Dateiende und echtem Fehler.

  Achtung: **`feof`** liefert erst **`true`**, **nachdem** eine Lese-Funktion versucht hat, über das Fileende hinaus zu lesen, und mit Fehler-Returnwert zurückgekehrt ist, und nicht schon "vorbeugend", wenn die nächste Lese-Operation wegen des Fileendes scheitern würde!.

- Wenn man ihn nicht mehr braucht, sollte man einen File **schließen** (implizit geschieht das automatisch am Programm-Ende, aber bei einem Programm-Absturz gehen Daten ungeschlossener Files verloren).

  Das **`fclose`** gibt auch den Platz für das File-Objekt (einige KB) wieder frei.

Die wichtigsten **File-I/O-Funktionen** sind auf meinem *C-Funktionen-Spickzettel* aufgelistet:

- *Einzelne Zeichen* lesen/schreiben: **`fgetc`**, **`getchar`**, **`fputc`**, **`putchar`**, …
- *Zeilenweise* lesen, einen String schreiben: **`fgets`**, **`fputs`**, …
  (**`gets`** ist aus Sicherheitsgründen *verboten*!)
- *Formatiertes* Lesen / Schreiben: **`fprintf`**, **`fscanf`**, …

  Achtung: Auf die *Längen-Modifier* im Format bei **`printf`** / **`scanf`** nicht vergessen! (für **`long int`** usw., beim Einlesen auch für **`double`** !!!)

- Unformatiertes Lesen / Schreiben einer *fixen Anzahl von Bytes*: **`fread`**, **`fwrite`**, …

Alle File-I/O-Funktionen (mit Ausnahme des Schreibens von **`stderr`**, des **`fclose`** nur gelesener Files, und ev. des Lesens / Schreibens am Terminal) sind **auf Fehler** (angezeigt durch den *Returnwert*) zu **prüfen**!

(Korrektes Ausgeben von Fehlermeldungen: Siehe folgendes Kapitel!)

Klassische I/O-Schleifen:

- **Zeichenweise:**

  ```
  while ((c = getchar()) != EOF) { ...
  ```

  Achtung:

  - **`c` muss `int` sein**, nicht **`char`**, denn der Returnwert muss alle möglichen **`char`**-Werte und *zusätzlich* **`EOF`** darstellen können! **`char`**-Werte werden von **`getchar`**, **`fgetc`** usw. – egal ob **`char`** auf der betreffenden Plattform **`signed`** oder **`unsigned`** ist – immer als *positive* **`int`**-Werte zurückgegeben, d.h. zuerst auf **`unsigned char`** und dann auf **`int`** verwandelt.
    `EOF` hingegen ist *negativ*.
  - Die *Klammern* rund um die Zuweisung sind *notwendig*, weil das **`!=`** höhere Priorität als das **`=`** hat: Ohne Klammern würde er zuerst den Returnwert von **`getchar()`** mit EOF vergleichen und dann das Ergebnis des Vergleiches (also **`1`** für wahr oder **`0`** für falsch) in **`c`** speichern!

    Generell ist es aus Gründen der Leserlichkeit allgemein üblich, um *Zuweisungen in Bedingungen* (**`if`**, **`while`**, ...) **zusätzliche `( )`** zu machen, auch wenn kein Vergleich folgt, damit sich die Zuweisung optisch deutlich von einem Vergleich unterscheidet.

- **Zeilenweise:**

  ```
  while (fgets(line, sizeof (line), f)) { ...
  ```

  Achtung: **`gets`** ist unsicher (keine Längenprüfung) ==> Nur **`fgets`** verwenden!

Die Standard-Files

3 vordefinierte Files sind beim Programmstart normalerweise *schon geöffnet* (d.h. das sind **`FILE *`** - *Konstanten*, für die das **`fopen`** schon gemacht wurde), sie können aber beim Programmaufruf auf der Befehlszeile (oder auch innerhalb des Programms) auf andere Dateien *umgeleitet* werden:

- **`stdin`** enthält den *Input* vom Terminal (was man eintippt).
- **`stdout`** und **`stderr`** werden beide am Terminal angezeigt.

  `stdout` ist für die *normale Programm-Ausgabe* gedacht,
  `stderr` für *Fehlermeldungen*.

  Die Trennung ist sinnvoll, damit man normalen Output und Fehlermeldungen *getrennt in Dateien umleiten* kann.

Da diese Files besonders häufig verwendet werden, gibt es von vielen File-Funktionen eine eigene Variante, die implizit auf **`stdin`** bzw. **`stdout`** arbeitet.

Die Tücken von Windows...

C-File-I/O unter Windows macht mehrere Probleme:

- C verwendet **`\n`** für das **Zeilenende**, Windows **`\r\n`**.

  Wenn man Windows-kompatible Textfiles lesen / schreiben will, muss man konvertieren. Die C-Library erledigt das, aber man muss unter Windows beim **`fopen`** in den Optionen angeben, ob konvertiert werden soll (Modus "**`t`**" = "Text") oder nicht (Modus "**`b`**" = "binary").

  Achtung: *Bei konvertierten Files stimmt die Anzahl der gelesenen / geschriebenen Zeichen (konvertiert) nicht mehr mit der Filegröße (unkonvertiert) überein!*

- Unter Unix/Linux hat auch die letzte Zeile eines Textfiles ein **`\n`** am Ende.
  Unter Windows *fehlt* das **`\n`** am *Ende der letzten Zeile* hingegen meist.

  Bei zeilenweisem Lesen liefert **`fgets`** normalerweise Strings mit **`\n`** am Ende, für die letzte Zeile aber unter Windows einen String ohne abschließendes **`\n`** !

- Windows benötigt beim **`fopen`** im Dateinamen '\' statt '/' als **Verzeichnis-Trennzeichen** in Pfaden.

  Nicht vergessen: '\' muss in Strings in C *doppelt* geschrieben werden:
  `"C:\\tmp\\workfile.txt"`
  (weil ein einfacher '\' ja ein Steuerzeichen einleitet)

  Je nach C-Library-Variante werden '/' auch unter Windows als Verzeichnis-Trennzeichen erkannt (automatisch in '\' umgewandelt) oder nicht...

File-I/O auf Systemebene

Jegliche Konvertierung, Formatierung, Aufteilung in Zeilen usw. erfolgt *innerhalb der C-Library* (nicht im Betriebssystem, für das jeder File nur eine Ansammlung von Bytes ohne tiefere Bedeutung ist!), ebenso die Pufferung (siehe folgendes Kapitel).

Zwischen Library und Betriebssystem gibt es eine zweite Ebene von File-I/O-Funktionen, die rein binär und ungepuffert einfach eine fixe Anzahl von Bytes transferieren.

Files werden auf dieser Ebene durch Nummern identifiziert (0 = **`stdin`**, 1 = **`stdout`**, 2 = **`stderr`**, dann selbst geöffnete Files). Details siehe mein *alter* C-Funktionen-Spickzettel.

Diese Funktionen sind auf mehrere Header verteilt (**`sys/stat.h`**, **`fcntl.h`**, **`unistd.h`**) und können auch direkt aufgerufen werden. Man darf allerdings auf einem File *nicht Aufrufe beider Ebenen mischen*.

Pufferung des File-I/O's

Pufferung auf Library-Ebene

File-I/O in C ist aus Effizienzgründen grundsätzlich gepuffert:

- Die C-Library verwaltet intern pro File einen Pufferspeicher (meist 4 KByte).
- Dieser dient dazu, mehrere I/O-Aufrufe des Programms zusammenzufassen, ohne jedesmal einen Betriebssystem-Aufruf machen zu müssen: Auf Betriebssystem-Ebene wird wenn möglich immer ein ganzer Puffer voll gelesen oder geschrieben:

- Beim Lesen wird vom Betriebssystem immer der nächste Block von 4 KB gelesen und dann stückchenweise von den Lese-Funktionen an das Programm weitergegeben.
- Beim Schreiben werden die geschriebenen Daten zusammengesammelt, bis ein 4-KB-Block voll ist. Dieser wird dann auf einmal geschrieben.

Wichtigster Effekt: *Nicht alles, was man schreibt, landet sofort im File!*
Im Normalfall landen geschriebene Daten vorläufig einmal nur im Puffer der Library (und sind daher ev. bei einem späteren Absturz verloren!).

==> Mit der Funktion **`fflush`** kann man sofortiges Schreiben des Puffers erzwingen (sinnvoll z.B. bei Logfiles!).

==> Das letzte Schreiben des restlichen Pufferinhaltes erfolgt erst beim **`close`**, d.h. eventuelle Schreibfehler dabei werden erst vom **`close`** gemeldet!

==> Mit den Funktionen **`setbuf`** usw. kann man die Pufferung umstellen (auf *unbuffered* oder *line-buffered*).

Ausnahmen:

- **`stdout`** am *Terminal* (und nur dort, bei Umleitung auf Files nicht!) ist per Default *line-buffered*, nicht block-buffered (d.h. wirklich geschrieben wird erst, sobald ein '**`\n`**' ausgegeben wird).

  Will man früher schreiben, braucht man **`fflush`** oder **`setbuf`**.

  (Die meisten Libraries machen implizit ein **`fflush`** auf **`stdout`**, sobald man von **`stdin`** liest, damit der Input-Prompt *vor* dem Input angezeigt wird, aber der C-Standard schreibt das *nicht* vor: Eigentlich müsste man nach dem Schreiben eines jeden Prompts explizit **`fflush`** aufrufen, bevor man von **`stdin`** liest.)
- **`stderr`** ist (sowohl auf Files als auch am Terminal) per Default *unbuffered*, d.h. alles wird *sofort* geschrieben.

`fflush` auf Input-Files:

Auch auf Input-Files bewirkt **`fflush`**, dass die bereits in den Puffer geladenen, aber noch nicht gelesenen Input-Daten verworfen werden. Sinnvoll ist das vor allem bei **`stdin`**:

- Mit einem **`fflush`** unmittelbar *vor* einer Lesefunktion kann man sicherstellen, dass *vorausgetippter Input ignoriert* und nur frisch getippter Input gelesen wird.
- Mit einem **`fflush`** *nach* einem **`scanf`**, bei dem der Input nicht zu den Konvertierungsangaben im Formatstring gepasst hat, kann der *fehlerhafte Input entsorgt* werden, damit eine Wiederholung des **`scanf`** neu getippten Input liest (anstatt wieder an den alten, falschen Input-Daten zu scheitern).

Pufferung im Betriebssystem

Neben dieser Pufferung auf Library-Ebene gibt es noch eine *Pufferung beim File Schreiben* auf **Betriebssystem-Ebene**: Unter Linux beispielsweise kann es typischerweise *60 Sekunden* (auf Laptops zum Stromsparen teilweise bis zu 10 Minuten, damit die Platte nicht so oft anläuft) dauern, bis die Daten (und die dazugehörigen Verzeichnis-Änderungen) nach einem **`fflush`** oder **`fclose`** aus dem Betriebssystem wirklich auf die Platte geschrieben werden.

Das hat zwei fatale Konsequenzen:

- Bei einem *Absturz* des ganzen Systems in dieser Zeit können trotz erfolgreichem **fclose** und Programmende *noch Daten verlorengehen*.
- *Echte I/O-Fehler* (Hardware-Probleme) können auch erst nach dem **fclose**, ja sogar erst *nach dem Programmende, auftreten* (und können dann natürlich nicht mehr an das Programm zurückgemeldet werden). Ein erfolgreiches **fclose** ist daher *keine Garantie* dafür, dass die Daten wirklich geschrieben wurden bzw. werden können.

Auf C-Library-File-I/O-Ebene gibt es dagegen *kein Mittel*, nur auf System-File-I/O-Ebene: **fsync** oder **O_SYNC** beim **open**. Diese sind vor allem bei Datenbanken und ähnlichen Anwendungen wichtig, bei denen es auf hohe Datensicherheit und eine bestimmte Schreib-Reihenfolge zur Sicherstellung der Daten-Konsistenz ankommt.

File-I/O in C++

Die Konzepte und Fähigkeiten des File-I/O sind in C und C++ fast ident, aber die Typen, Funktionsnamen usw. unterscheiden sich völlig:

- Der Header heißt **iostream** für Terminal-I/O (auf den Standard-Files) und **fstream** für File-I/O.
- Die Klassen heißen **ifstream** (nur lesen), **ofstream** (nur schreiben) oder **fstream** (beide Richtungen).
- Die Standard-Files heißen **cin**, **cout** und **cerr** (und **clog** als gepufferte Variante von **cerr**).
- Es gibt ein explizites **open**, aber das Öffnen erfolgt *meist implizit* durch Angabe des Filenamens gleich bei der Deklaration des File-Objektes.
- Es gibt ein **eof**, ein **close** und ein **flush**,
  sowie die Funktionen **seekg** und **seekp** zum Positionieren.
- Formatierte Ein- und Ausgabe (mit Konvertierung) erfolgt mit **<<** und **>>**.
- Daneben gibt es ein **read** und ein **write** für binäre Ein- und Ausgabe, ein **get** und ein **put** für zeichenweise Ein- und Ausgabe, und ein **getline** zum Lesen einer ganzen Zeile (denn **>>** eines Strings liest nur wortweise, bis zum nächsten Zwischenraum, nicht zeilenweise, ebenso wie **scanf** mit **%s** nur einen String bis zum nächsten Zwischenraum liest).
- Die Behandlung von Fehlern und Datei-Ende ist deutlich aufwändiger als in C.
- Die Pufferung funktioniert ähnlich wie in C.

Formatierter Input

Formatierter Input (**scanf** in C bzw. **>>** in C++) erfordert **besondere Vorsicht**:
Wenn einmal ein Fehler auftritt (weil die Daten am Inputfile bzw. Terminal nicht zum angegebenen Format bzw. zum erwarteten Typ passen), so sind im günstigsten Fall alle nachfolgenden Inputs um ein Feld verschoben, und im ungünstigsten Fall läuft das Programm in eine Endlosschleife, weil das Lesen nie über die falschen Daten hinwegkommt.

Naiver formatierter Input, wie man ihn oft in Übungsbeispielen sieht, ist daher in produktivem Code strikt zu vermeiden!

Es gibt *zwei Möglichkeiten*, es richtig zu machen:

- Man **prüft** jeden formatierten Input, ob er erfolgreich war.

  `scanf` returniert die *Anzahl der erfolgreich konvertierten Felder*, und auf C++-Streams gibt es eine Funktion **`fail`** für diesen Zweck (außerdem zeigt der Returnwert von **`>>`** Erfolg oder Mißerfolg an).

  Bei Mißerfolg **überliest** man zeichenweise den fehlerhaften Input, bis man wieder an einer Stelle im Input ist, ab der zukünftige Leseoperationen korrekt arbeiten (z.B. einem **`\n`**). C++ bietet zum Überlesen eine eigene Funktion **`ignore`**.

  In C++ muss man zusätzlich den **Fehlerstatus zurücksetzen**.

- Man liest grundsätzlich nur **zeilenweise in einen String** (mit **`fgets`** bzw. **`getline`**, *ohne* **`scanf`** bzw. **`>>`**) und **zerlegt und konvertiert dann den String** ("zu Fuß" mit Stringfunktionen und **`atoi`** usw. oder automatisch mit **`sscanf`** bzw. in C++ mit Stringstreams).

  Tritt ein Fehler auf, ist nur der String im Speicher betroffen, aber nicht das Lesen vom File: Man setzt einfach ganz normal mit dem Lesen der nächsten Zeile fort.

Alternativen zu File-I/O

Files können auch mit **`mmap`** gelesen und geschrieben werden.

Das ist ein grundsätzlich anderes Konzept, das auf **virtueller Speicherverwaltung und Paging** beruht:

- Der gesamte File (oder der angegebene Teil) wird **virtuell in den Speicherbereich des Prozesses eingeblendet** (ohne dass der File tatsächlich sofort gelesen wird, und ohne dass der Speicher sofort tatsächlich belegt wird).
- `mmap` liefert einen *Pointer auf den Anfang des Files im Speicher*, man kann darauf **wie auf ein Array of `char`** (oder sonst irgendeine Datenstruktur) **zugreifen** (lesend oder schreibend).
- Das eigentliche I/O erfolgt "on demand", ohne dass man es merkt oder sich darum kümmern muss: Der File wird stückweise (in Pages, d.h. meist 4-KB-Blöcken) beim ersten Zugriff auf einen 4-KB-Block gelesen (nicht unbedingt sequentiell, sondern eben genau die Page, auf die man gerade zugreift), und die veränderten Pages werden in die Datei zurückgeschrieben, wenn das Betriebssystem den Speicherplatz für andere Zwecke braucht und die Pages wieder freigibt.

Das Laden von Shared Libraries erfolgt unter Unix / Linux beispielsweise mittels **`mmap`**, nicht mit normalem File-I/O.

13. Kapitel: Fehlerbehandlung

"Vertrauen ist gut – Kontrolle ist besser!"
("If you made it idiot-proof, someone will invent a better idiot!")

Programme sollen auf *Probleme im Umfeld* (Platte/Speicher voll, File geht nicht auf, …) und auf *beliebig dummen, falschen oder großen Aufruf, Input oder File-Inhalt*

"geordnet" reagieren!

==› Wir **prüfen**

- Alle **Aufrufe von I/O- und Library-Funktionen** (u.a. **`malloc`**), die einen Fehler liefern können (siehe Dokumentation!).
- Alle **eingegebenen / eingelesenen Daten**.
- Alles, was von der **Befehlszeile** und aus dem **Environment** kommt.

Klassischer Test: Input eines Programms mit Zufallsgenerator füttern ==› Absturz?

Nicht jede Fehlerprüfung resultiert sofort in einer Ausgabe auf **`stderr`** und einem **`exit`**: Im Gegenteil, es ist in "allgemein verwendbaren" Funktionen üblich, *Fehler nur (mittels Returnwert o.ä.) zum Aufrufer zu melden*.

Die eigentliche *Fehlerbehandlung* (anwendungsabhängig alles schließen, aufräumen und abspeichern, Log-Eintrag schreiben, …) geschieht dann meist zentral und möglichst weit oben in der Programmlogik oder in einer eigenen Funktion.

`errno`

- *Grundprinzip* von vordefinierten Funktionen mit Fehlerfällen in C:
  - Der **Returnwert** sagt nur, ob der Aufruf *gut- oder schiefgegangen* ist (je nach Funktion mit **`NULL`**, **`-1`**, **`EOF`**, …) *(aber sagt nicht, warum!)*.
  - Wenn der Aufruf laut Returnwert schiefgegangen ist: **`errno`** sagt, *warum*.
- **`errno`** ist eine *externe globale* **`int`**-Variable (oder verhält sich zumindest so – es kann auch ein Makro sein!).
- **`errno`** wird durch Inkludieren von **`errno.h`** deklariert.
- **`0`** steht für "kein Fehler", andere Werte für bestimmte Fehler-Gründe.
- Für alle Fehler-Gründe sind *numerische Konstanten* **`E...`** *als Makros* definiert, siehe unter Linux "**`man errno`**".
  Achtung: Nur diese **Makros verwenden**:
  Die numerischen Werte sind von System zu System unterschiedlich! (und unleserlich)
- **`errno`** wird von allen I/O-Funktionen und einigen anderen Library-Funktionen (nicht allen, siehe jeweilige Man-Page!) *im Fehlerfall gesetzt*.
  - Achtung: **`errno`** ist **nur** *im Fehlerfall definiert!*

    Wenn der Returnwert einer Funktion *keinen Fehler* anzeigt, ist **`errno`** *undefiniert* (alter Wert, beliebig veränderter neuer Wert, 0, …).

Im Besonderen kann **printf** usw. auch im Ok-Fall **errno** beliebig ändern. Bei einer Fehlerausgabe daher nicht zuerst **printf** machen und danach **errno** anschauen (könnte verändert worden sein!), sondern den Fehlercode in **errno** vor dem **printf** wegsichern!

- Achtung: **errno** wird von *keiner* Library-Funktion im Ok-Fall garantiert auf **0** gesetzt (im Gegenteil: **errno** sollte von vordefinierten Funktionen eigentlich *nie auf 0 gesetzt werden*, nur halten sich nicht alle Funktionen daran).

  Eine Prüfung von **errno** auf **0** für "kein Fehler" funktioniert daher *nicht*!

- **errno** ist Thread-lokal (zumindest in modernen C-Implementierungen).

Fehlermeldungen

Wie sollen Fehlermeldungen aussehen?

- Fehlermeldungen *müssen immer auf* **stderr** *(in C++:* **cerr***) ausgegeben werden!*
- Übliche Form: ***"Programmname: Operation: Fehlergrund"***

  Die Operation sollte dabei *auch das betroffene Objekt* enthalten (Filename, Netzadresse, ...)!

  Beispiele:
  "**more: Cannot open test.txt: No such file or directory**"
  "**ping: Cannot resolve www.heise.de: Timed out**"

- Für den Fehlergrund sollten so weit wie möglich die *vordefinierten, einheitlichen Texte* verwendet werden (siehe unten!).
- Für die Ausgabe des Programmnames: **argv[0]** in globale Variable kopieren! (unter Linux, *nicht* Standard: Globale Variable **program_invocation_name**)
- Bei Server-Programmen und anderen "Langläufern" zusätzlich vorne: *Datum und Uhrzeit des Fehlers!!!* (damit man den Fehler mit externen Ereignissen, gleichzeitigen Fehlermeldungen in anderen Programmen usw. in Zusammenhang bringen kann)
- Eventuell: *Fehlerklasse* (z.B. fatal / error / warning / info / debug) zur raschen Unterscheidung wichtiger von unwichtigen Meldungen.
- Bei Fehlern in *Eingabe-Files* (Syntax-Fehlern, Beispiel gcc-Fehlermeldungen): "*Input-Filename:Zeilennummer: Fehler*" oder "*Input-Filename:Zeilennummer:Spaltennummer: Fehler*"
- Bei *Aufrufs-Fehlern* in der Befehlszeile: *Korrekten Aufruf bzw. Hilfe anzeigen!*

Funktionen für Fehlermeldungen:

- Optimal zur Ausgabe und Regel-konform wären **err(...)** usw. und **error(...)** usw., aber beide sind nicht standardisiert (nur Linux bzw. nur BSD Unix).
- **perror(...)** ist standardisiert, aber unpraktisch (man braucht vorher ein **sprintf**, um Fehler regelkonform auszugeben).
- Daher:**strerror(errno)** (standardisiert) liefert den **Fehlertext zu errno**, den Rest erledigt man "zu Fuß" mit einem **fprintf** auf **stderr**.

exit(int status)

- *Beendet* das Programm normal.
- Der **status** wird an das Betriebssystem (bzw. die Shell / das aufrufende Programm) weitergegeben.
- Im Standard: **EXIT_SUCCESS** (meist 0) … ok, **EXIT_FAILURE** (meist 1) … Fehler

  Unix/Linux: auch 2-125 sind frei nutzbare Fehlercodes, Werte ab 126 sind reserviert.
  Ev. zur Anzeige unterschiedlicher Fehler, z.B.: 1 … Aufruf-Fehler, 2 … I/O-Fehler, usw.

  Manchmal auch zur *Anzeige des Programm-Ergebnisses*,
  z.B. bei Text suchen: 0 … gefunden, 1 … nicht gefunden, 2 … echter Fehler
  oder bei Files vergleichen: 0 … gleich, 1 … verschieden, 2 … echter Fehler
- Funktionen, die mit **atexit** registriert wurden, werden ausgeführt (Aufräum-Funktionen, sogenannte "Exit Handler").
- Files werden geschlossen.

abort(void)

- *Programmabbruch bei schwerem internem Fehler* (**nicht** für den Normalbetrieb!)
- Unter Linux: Schreibt einen *Core Dump* zum Debuggen.

assert(int condition)

- *Makro zum Programm Testen*:
  Argument ist eine Bedingung, die an dieser Stelle *immer gelten muss*.
  Wenn sie **true** ist: **assert** tut *nichts*.
  Wenn sie **false** ist: **assert** schreibt eine *Fehlermeldung*
  "Assertion failed … (mit File, Zeile, Bedingung)"
  und beendet Programm mit **abort** (incl. Core Dump zum Debuggen).
- **assert** ist *nur für den Entwickler zum Testen und Fehlersuchen* gedacht
  und **nicht** dazu da, Fehlermeldungen für den normalen Benutzer auszugeben!
- **assert** wird *völlig ignoriert*,
  wenn beim Compilieren das Makro **NDEBUG** definiert ist
  ==› Die Bedingung sollte *keinen echten Code und keine Seiteneffekte* enthalten,
  wirklich nur Testcode (weil bei **NDEBUG** sonst echter Code wegfällt!)

14. Kapitel: Die C Library

Unterscheide:

- Funktionen im **C-Standard** ==› müssen überall vorhanden sein!
- Funktionen im **POSIX-Standard** ==› sind standardisiert und fast überall vorhanden (meist – mit Einschränkungen – auch unter Windows), obwohl sehr Unix-lastig.
  Beispiel: Netzwerk-Funktionen, Unix-Prozesse und Prozess-Kommunikation, …

  Der **XOPEN-Standard** ist noch Unix-lastiger und weniger portabel als POSIX.
- Andere: *Hunderte Libraries* mit tausenden Funktionen für jeden Zweck,
  aber *nicht standardisiert / nicht portabel*!

Im C Standard (vollständig):

- **assert.h**: **assert**
- **complex.h**: Komplexe Zahlen
- **ctype.h**: **is**… und **to**…
- **errno.h**: **errno**
- **fenv.h**: Einstellungen für **float** / **double** (Rundungs-Mode, Fehler-Handling, …)
- **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 Limits usw. von **int** usw.
- **locale.h**: Ländereinstellungen
- **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), **intptr_t**, **intmax_t**
  Limit-Konstanten dafür
- **stdio.h**: File-I/O, File-Funktionen (**remove**, **rename**, …), **perror**
- **stdlib.h**: **exit** / **abort**, **atoi** / **strtol** / …, **abs**, **rand** / **srand**, **malloc** / …, **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

In POSIX (Auswahl aus Dutzenden!):

- **unistd.h**, **fcntl.h**, **sys/stat.h**: Unix-Funktionen
  z.B. Low-Level-File-I/O, Unix-Prozess-Handling, sleep, …
- **dirent.h**: Directory-Zugriffe
- **sys/*.h**: Unix-Systemfunktionalitäten, z.B. **mmap**, Interprozess-Kommunikation

15. Kapitel: Strukturen

Strukturen sind ***aus einzelnen Elementen zusammengesetzte Datentypen***,
die sich in wesentlichen Punkten von Arrays unterscheiden:

- *Strukturtypen* sind *eigene Typen* und haben einen *Namen*.

  Achtung: In C++ kann man den *Namen allein* als Typ verwenden,
  In C muss man immer und überall **`struct`** *vor den Strukturtyp-Namen schreiben*
  (oder **`typedef`** verwenden)!

- Die *Komponenten* einer Struktur sind nicht wie beim Array durchnummeriert,
  sondern haben einen *Namen*, mit dem sie angesprochen werden.

- Die Komponenten einer Struktur haben normalerweise *verschiedene Typen*.

Terminologie:

- Im Englischen heißen die Komponenten "**member**" der Struktur (oder "**fields**").

- Im Deutschen werden die Komponenten oft auch "Felder" der Struktur genannt,
  was aber im Hinblick auf die ganz andere Bedeutung von "Feld" bei Arrays
  verwirrt und daher unterlassen werden sollte!

Deklaration eines Struktur-Typs:

```
struct name {
  Memberdeklaration1;
  Memberdeklaration2;
  ...
};
```

Die Memberdeklarationen schauen genauso aus wie normale Deklarationen,
aber sie deklarieren die Komponenten der Struktur, nicht normale Variablen.

Der *Name* eines Strukturtyps ist frei wählbar (im Englischen wird der Name nach dem **`struct`** auch gelegentlich "struct tag" genannt).

Oft wird für **`struct`**-Typen **`typedef`** verwendet, siehe nächstes Kapitel.

Achtung: Zwei in getrennten **`struct`**'s deklarierte Strukturen sind *verschiedene* Typen (Fehlermeldung beim Zuweisen usw.), auch wenn sie intern denselben Aufbau und dieselben Member-Namen haben!

Achtung: Eine Strukturtyp-Deklaration beschreibt *nur den Aufbau* des Typs
(z.B. "Wie schaut der Typ 'Person' intern aus?"). Es werden noch *keine Variablen*
und *kein Speicherplatz* angelegt, daher sind auch *keine Initialwerte*
in Member-Deklarationen möglich!

Deklaration von Variablen eines Struktur-Typs:

```
struct struct_name myStructVar, *ptrToStruct, ArrOfStruct[MAX_SIZE];
```

Man kann die Deklaration des Strukturtyps und der Variablen auch zusammenfassen:
Die Variablenliste kommt nach der **`{ }`** vor dem **`;`** . Wenn man einen Strukturtyp nur in einer einzigen Deklaration braucht, kann man den Namen nach **`struct`** auch weglassen (nicht klug: Dem Leser hilft der Name, auch wenn er nicht notwendig ist).

Zugriff auf Struktur-Member:

Unterscheide:

- Habe ich eine **Struktur selbst** ==› Zugriff mit "**.**"
- Habe ich einen **Pointer auf eine Struktur** ==› Zugriff mit "**->**"
  Daher: `(*p).f` ist dasselbe wie `p->f`

Beispiel:

```
struct errorPos {
  char *fileName;
  int lineNr, colNr;
} errors[MAX_ERR_CNT], *errPtr;

errors[nextErr].lineNr = curLine;

printf("%s:%d:%d: %s\n", errPtr->fileName, errPtr->lineNr,
                         errPtr->colNr, errText);
```

Weitere klassische Kandidaten für Struktur-Typen:

- Person (Name, Adresse, Geburtsdatum, …)
- Datum & Zeit (Sekunde, Minute, Stunde, Tag, Monat, Jahr, Wochentag)
- **FILE** (Puffer, aktuelle Position, Fehlerstatus, Richtung)

Geschachtelte Datentypen:

Strukturen können als Komponenten *andere Strukturen, Pointer, Arrays usw.* enthalten (dann kommen beim Zugriff eben entsprechend viele **.** und **->** sowie **[]**).

Arrays in Strukturen müssen allerdings fixe Größe haben, da ja alle Komponenten fixe Plätze (Anfangsadressen) innerhalb der Struktur bekommen müssen und auch die Struktur als Ganzes im Normalfall fixe Größe hat (nur die letzte Komponente könnte mit Tricks ein variabel großes Array sein).

Ein Strukturtyp kann als Komponente *nicht eine Struktur desselben Typs* enthalten, aber sehr wohl (häufig!) *Pointer auf Strukturen desselben Typs* (siehe unten).

Umgekehrt sind Arrays von Strukturen möglich.

Operationen:

Folgende Operationen sind *mit ganzen Strukturen* möglich:

- *Zuweisung* (der Strukturinhalt wird komplett kopiert)
- *Übergabe* als By-Value-Argument und als Returnwert.

Vor allem das zweite ist aber ***völlig unüblich***: Es wird (außer bei sehr *kleinen* Strukturen) *fast immer ein Pointer übergeben und zurückgegeben*, weil das viel schneller als das Kopieren der Struktur selbst ist. Aus demselben Grund sind Struktur-Zuweisungen wenn möglich zu vermeiden bzw. durch Pointer-Zuweisungen zu ersetzen.

Nicht möglich ist hingegen der *Vergleich* zweier ganzer Strukturen.

Initialisierung:

Hier gibt es historisch bedingt 2 Varianten:

- "Oldstyle": **`struct errorPos dummyErr = {NULL, -1, -1};`**
  (Werte in der *Reihenfolge der Komponenten*!)
  (bei geschachtelten Strukturen und Arrays: geschachtelte **`{ }`** !)
- C99: **`struct errorPos dummyErr = {`**
  `.fileName = NULL, .lineNr = -1, .colNr = -1};`
  (kann auch *durcheinander* sein und *Lücken* haben!)
  (kann geschachtelte Komponentenzugriffe, **`[ ]`** für Arrayindices usw. enthalten)
- *Achtung:* gcc kennt auch eine andere, alte Syntax für Initialisierungen mit expliziten Membernamen und implementiert einige Erweiterungen gegenüber C99!

Größe von Strukturen:

Strukturen haben normalerweise eine **fixe Größe** (Ausnahme: Strukturen, deren letzte Komponente ein variabel großes Array ist – selten und kompliziert!).

Achtung:

- Strukturen können ***Lücken*** (unbenutze Bytes) oder ***Füllbytes am Ende*** enthalten ("Padding"), wenn es das Alignment der Elemente erfordert (Ausrichtung auf Adressen, die Vielfache von 4, 8 usw. sind).
- Die Größe einer Struktur kann daher *mehr* als die Summe der Größen der Komponenten sein (**`sizeof`** liefert stets den tatsächlichen Wert ==› verwenden!).

Es gibt ein Makro **`offsetof`**, das den Abstand im Speicher (in Bytes) einer Komponente vom Anfang der Struktur liefert.

"incomplete struct":

Wenn eine **Struktur-Variable** angelegt wird, muss die Größe und damit das *Innenleben der Struktur vollständig bekannt* sein. Die Verwendung von Strukturtypen in Variablen-Deklarationen (oder z.B. als Member anderer Strukturen) ist daher erst *nach* dem **`struct ... { }`** möglich.

Für die Deklaration von **Pointern auf Strukturen** muss hingegen *weder Größe noch Innenleben der Struktur bekannt* sein. **`struct unknownStruct *myStrPtr;`** ist daher zulässig, ohne bzw. bevor der Strukturtyp **`struct unknownStruct`** deklariert wird. Der wichtigste Vertreter eines Pointers auf eine *unbekannte Struktur* ist **`FILE *`**. Solange das Innenleben nicht bekannt ist, kann man darauf auch nicht mit **`->`** zugreifen.

Daher ist auch die Deklaration *mehrerer Strukturtypen, die zyklisch Pointer aufeinander enthalten*, kein Problem:
In der Deklaration von **`struct A`** darf **`struct B *p;`** als Member-Deklaration vorkommen, auch wenn die Deklaration von **`struct B`** erst später folgt.

Solche Strukturtypen, von denen nur die Existenz, aber nicht das Innenleben bekannt ist, heißen "incomplete struct".

Vordefinierte Strukturen:

Zahlreiche Funktionen aus dem C- und POSIX-Standard haben Strukturen (bzw. Pointer darauf) als Parameter oder Returnwert. Diese vordefinierten Strukturen sind üblicherweise in der Dokumentation (**man**-Page) der Funktion beschrieben.
Man muss die Strukturdefinition aber nicht abtippen: Der *Header*, der den Prototyp für die Funktion enthält, enthält auch die *Deklaration der benötigten Strukturtypen*.

Strukturen als Returnwert:

Obwohl Strukturen "by Value" als Returnwert zurückgegeben werden können, geschieht das sehr selten (nur bei sehr kleinen Strukturen). Bei der Rückgabe eines Pointers auf eine Struktur ergibt sich aber dasselbe Problem wie bei der Rückgabe von Arrays: *Es darf kein Pointer auf eine lokale Struktur der aufgerufenen Funktion zurückgegeben werden*, denn die lokale Struktur wird im Moment des Returns freigegeben, und der Pointer würde auf undefinierte Daten zeigen.

Auch die Lösungsmöglichkeiten des Problems sind dieselben wie bei Arrays:

1.) Der *Aufrufer übergibt der aufgerufenen Funktion eine "leere" Struktur als "by Reference"-Parameter* und die aufgerufene Funktion speichert ihr Ergebnis in diese Struktur des Aufrufers.

2.) Die aufgerufene Funktion returniert einen Pointer auf eine *interne statische Struktur* (mit dem Problem, dass das Ergebnis durch den nächsten Aufruf überschrieben wird!).

3.) Die aufgerufene Funktion returniert einen Pointer auf eine mit **`malloc`** *dynamisch angelegte Struktur*, die der Aufrufer mit **`free`** freigeben muss, wenn er sie nicht mehr braucht.

Namensräume

Alle Namen, die wir bisher kannten, liegen im *selben* Namenraum:
Variablennamen, Funktionsnamen, (später auch: **`enum`**-Konstanten, **`typedef`**-Typen), …

Doppelte Namen führen daher entweder zu einer **Fehlermeldung**, oder
die weiter innenliegende Deklaration "versteckt" die außenliegende Deklaration
(englischer Ausdruck: "Shadowing").

C unterteilt Namen in mehrere Namensräume:

- Die eben erwähnten **"normalen" Namen**.
- Die **Labels** von **`goto`**'s.
- Die **Typ-Namen aller `struct`-Typen** (und **`union`**- sowie **`enum`**-Typen):
  Diese Namen treten nur unmittelbar nach dem **`struct`** / **`union`** / **`enum`** auf.

  (In C++ ist das *kein* eigener Namensraum, weil man das **`struct`** weglassen darf!)
- Die **Namen der Komponenten von `struct`** (und **`union`**) bilden **pro `struct`** einen eigenen Namensraum: Sie treten nur nach **`->`** und **`.`** auf, und der Ausdruck vor dem **`.`** bzw. **`->`** legt fest, um Komponenten welches Strukturtyps es sich handelt.
- **In C++** (und nur dort!) kann man zusätzlich auch beliebige **eigene Namensräume** definieren.

Namen verschiedener Namensräume *kollidieren nicht*, gleiche Namen in verschiedenen Namensräumen koexistieren unabhängig voneinander: Es ist daher möglich, Strukturtypen gleich wie normale Variablen (oder – was häufig vorkommt – gleich wie **`typedef`**-Typen) zu nennen, oder *in mehreren verschiedenen Strukturtypen gleiche Komponentennamen zu vergeben*.

Dynamische Datenstrukturen

Am häufigsten werden Strukturen für *Elemente dynamischer Datenstrukturen* benutzt: Jede Struktur enthält neben den eigentlichen Daten eines Elementes *Pointer auf andere Elemente*, die einzelnen Elemente sind "verkettet", d.h. ausgehend von einem Element findet man die weiteren Elemente, indem man den Pointern folgt.

Die beiden wichtigsten dynamischen Datenstrukturen sind

- (einfach oder doppelt) verkettete **Listen**
- und (meist binäre) **Bäume**.

==> Beispiel siehe Tafel!
==> Hauptinhalt des 3. Semesters!

Diese Art der Datenspeicherung ist zwar **komplexer** als in Arrays,
hat aber mehrere **Vorteile**:

- Es werden nur *so viele Elemente* im Speicher angelegt, *wie wirklich benötigt* werden.
- Es lassen sich durch Änderung der Verkettung "mittendrin" Elemente *einfügen und löschen*, *ohne Elemente im Speicher verschieben* zu müssen.

16. Kapitel: `typedef`

`typedef` führt einen *neuen Typnamen als Abkürzung für einen bestehenden Typ* ein.
Der neue Name kann dann wie die vordefinierten Typ-Namen **`int`**, **`double`** usw. in Deklarationen, als Parameter- und Returnwert-Typ, in **`sizeof`**, als **`struct`**-Member usw. verwendet werden, auch in Kombination mit **`*`**, **`[ ]`** usw..

`typedef` wird einfach *vor eine normale Deklaration* gestellt: Das, was bei einer normalen Deklaration die *zu deklarierende Variable wäre*, ist bei einer **`typedef`**-Deklaration der *neue Typname*. Eine **`typedef`**-Deklaration legt *keinen Speicher* an.

Die Sichtbarkeit des **`typedef`**-Namens ist wie bei einer normalen Deklaration (üblicherweise werden **`typedef`**'s *global* und nicht lokal deklariert und gelten daher bis zum Ende des Files).

Oft werden **`typedef`**-Namen mit **`_t`** am Ende erkennbar gemacht, das ist aber nur Konvention, nicht Pflicht.

`typedef`-Typen sind im Sinne der Typprüfung *keine eigenen, neuen Typen*, sondern nur Abkürzungen für bestehende Typen: Eine mit einem **`typedef`**-Typ deklarierte Variable und eine mit dem ursprünglichen Typ deklarierte Variable haben *denselben Typ*, sie können in beide Richtungen aufeinander zugewiesen oder miteinander verglichen und an denselben Stellen im Programm verwendet werden.

Beispiele:

`typedef long long unsigned int ui64, *ui64p;`
(**`ui64`** ist eine Abkürzung für **`long long unsigned int`**,
`ui64p` ist der Typ "Pointer auf **`long long unsigned int`**")

`typedef char line_t[LINE_LEN + 2];`
(der Typ **`line_t`** ist ein Array of **`char`** mit der oben angegebenen Anzahl von Elementen,
Anwendung beispielsweise **`line_t inLine, outLine;`**)

```
typedef struct {
  double x, y, z;
} coord, *coord_p;
```

(der Typ **`coord`** ist ein Strukturtyp mit den 3 Membern **`x`**, **`y`**, **`z`**;
der Typ **`coord_p`** ist der Typ von Pointern auf solche Strukturen)

Achtung:

Der **`typedef`**-Typ existiert erst ab dem *Ende* der **`typedef`**-Deklaration und kann daher in der Deklaration selbst noch *nicht* verwendet werden.

Falsch:

```
typedef struct {
  int value;
  elemPtr next;
} elem, *elemPtr;
```

Richtig entweder

```
typedef struct elem {
  int value;
  struct elem *next;
} elem, *elemPtr;
```

(das Member **`next`** ist trotzdem typkompatibel mit Werten vom Typ **`elemPtr`**)

oder

```
typedef struct elem elem, *elemPtr;

struct elem {
  int value;
  elemPtr next;
};
```

17. Kapitel: Der C Präprozessor

Der C Präprozessor läuft beim Compilieren als *eigener Schritt* **vor** dem Compiler.
Er ist in vielen Fällen ein separates, vom Compiler *unabhängiges Programm*.

Der Präprozessor macht *textuelle Veränderungen am Programmtext*,
bevor der eigentliche Compiler den Programmtext sieht und verarbeitet.

Im Wesentlichen hat der Präprozessor folgende Fähigkeiten:

- *Inkludieren* von Files
- *Textersetzung* ("Makros")

- *Bedingtes Überspringen* von Text

Achtung:

Präprozessor-Befehle (**#**...) sind *keine C-Konstrukte* und haben daher generell **keinen ;** am Ende!

Generell hat der Präprozessor **keine** tiefere Kenntnis der Programmiersprache C:
Er betrachtet den Programmtext einfach als *Folge von Zahl- und Stringkonstanten, Namen und Sonderzeichen*, *ohne deren Bedeutung* zu kennen (der C Präprozessor kann auch als eigenständiges Programm ohne nachfolgenden Compiler aufgerufen werden und wird demgemäß gelegentlich zur Bearbeitung ganz anderer Texte als C-Programme eingesetzt).

#include

Eine **#include**-Zeile wird *durch den Inhalt des angegebenen Files ersetzt*.
Bei den inkludierten Files handelt es sich üblicherweise um Header-Files (Endung **.h**, siehe nächstes Kapitel).

Es gibt 2 Formen:

- **#include <*filename*>**
  Inkludiert *System-Headerfiles* (Suche in vordefinierten System-Verzeichnissen).
- **#include "*filename*"**
  Inkludiert *Headerfiles des eigenen Projektes* (Suche im aktuellen Verzeichnis).

Als *filename* können auch Pfadnamen *mit Verzeichnis* (relativ zum System-Include-Verzeichnis bzw. lokalen Verzeichnis) angegeben werden, viele Standard-Unix-Header liegen z.B. im Unterverzeichnis **sys** und werden mit **#include <sys/...h>** inkludiert.

#define

#define definiert ein **Makro** (eine Abkürzung): Alle Vorkommen des Makros nach dem **#define** werden *textuell* durch die Definition des Makros *ersetzt*.

Es gibt 2 Formen:

- **#define *name ersatztext***
  z.B. **#define LINE_LEN (80 + 2)**
  Für **Konstanten** (**Achtung: Kein =** zwischen *name* und *ersatztext*!)
- **#define *name*(*parameter*) *ersatztext***
  z.B. **#define MAX(a, b) (((a) > (b)) ? (a) : (b))**
  Für **Makros mit Parametern** (Funktions-Makros bzw. Inline-Code).

Makro-Namen werden meist *komplett groß* und mit **_** zur Worttrennung geschrieben.

Makros – vor allem Funktions-Makro's! – sollten *durch modernere Konstrukte abgelöst* werden:

- Konstanten durch **const-Deklarationen** (das klappt in C++, weil **const**-Variablen in C++ wie echte Konstanten (Zahlen) verwendet werden können und auch genauso effizient sind, aber nur eingeschränkt in C, weil **const**-Variablen in C nur "nicht veränderbare Variablen" sind, die nicht an Stellen verwendet werden können, wo ausdrücklich Konstanten gefordert sind, und im Unterschied zu C++ auch Speicher belegen) oder durch **enum-Konstanten** (siehe übernächstes Kapitel).

- Funktions-Makros durch **`inline`-Funktionen** (lernen wir bei C++, gibt es aber inzwischen auch im C-Standard): Funktionsmakros wurden ursprünglich eingeführt, um den Laufzeit-Overhead eines Funktionsaufrufes einzusparen, aber **`inline`**-Funktionen haben auch keinen Overhead, weil sie ebenfalls "vor Ort" eingesetzt werden, allerdings nicht textuell wie bei Makros, sondern als fertig compilierter Code wie bei echten Funktionen.

Achtung:

Auch Funktions-Makros werden bei jedem Vorkommen rein **textuell eingesetzt**. Die Parameter werden ebenfalls nicht fertig ausgerechnet eingesetzt, sondern rein textuell:

Jedes Vorkommen eines Parameter-Namens in der Makro-Definition wird Zeichen für Zeichen durch den Argument-Text im Aufruf ersetzt.

Das erzeugt 3 wesentliche Fehlerquellen:

1.) Enthält die Makro-Definition ein **Argument mehrmals**, wird bei Argumenten mit Seiteneffekten (Funktionsaufrufe, vor allem aber **++** und **−−**) der **Seiteneffekt entsprechend oft ausgeführt**!

Beispiel (mit **`MAX`** wie oben definiert):

```
max = MAX(max, i++);
```
ergibt `max = (((max) > (i++)) ? (max) : (i++));`

`i` wird je nach Ausgang des Vergleiches um 1 oder **um 2 erhöht**,
und auf **`max`** wird bei fast allen Compilern der **erhöhte** Wert von **`i`** zugewiesen!

==> **Makros mit mehrfachen Vorkommen eines Argumentes wenn möglich überhaupt vermeiden!!!** (sonst zumindest *unübersehbar dokumentieren*!)

==> **Seiteneffekte (++, −−, Funktionsaufrufe) in Makro-Argumenten wenn möglich vermeiden!**

2.) Ist der Ersatztext **nicht als Ganzes komplett geklammert**,
kann er auf unerwünschte Art **mit der Aufrufsstelle verschmelzen**:

Beispiel:

```
#define DIGIT_VAL(c)  (c) - '0'
```

`n = 10 * DIGIT_VAL(s[i]);` ergibt `n = 10 * (s[i]) - '0';`
und das ist auf Grund der *Vorrangregeln* `n = (10 * s[i]) - '0';`

==> **Makro-Ersatztext (rechte Seite) immer komplett in ( ) schreiben!!!**

3.) Makro-Argumente, die aus mehr als einer Konstante oder Variable bestehen, können zu **unerwarteten Berechnungen des fertig ersetzten Ausdruckes** führen, wenn die Parameter im Ersatztext nicht alle einzeln für sich geklammert sind:

Beispiel:

```
#define SQR(x) (x * x)
```

```
q = SQR(i + 1);
```
ergibt `q = (i + 1 * i + 1);` d.h. ausgerechnet `q = (2 * i) + 1;`

==> **Jedes einzelne Vorkommen eines Parameters im Makro-Ersatztext immer einzeln in ( ) einschließen!!!**

Weitere Hinweise zu Makros:

- Möchte man einen Ersatztext eines Makros über *mehrere Zeilen* definieren, muss jede Zeile außer der letzten mit einem **** *unmittelbar gefolgt vom Zeilenvorschub* enden:

```
#define LONG_MACRO(a, b) \
     (someFunction(a) + \
      someFunction(b))
```

- Innerhalb des Ersatztextes haben ***# und ## Spezialbedeutungen***, auf die wir hier nicht weiter eingehen (Umwandeln eines Argument-Textes in eine Stringkonstante, Verschmelzen von zwei Namen zu einem).

  Das ist u.a. deshalb notwendig, weil Parameternamen, die im Ersatztext innerhalb von String- oder **char**-Konstanten vorkommen, *nicht* ersetzt werden.

- 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 (z.B. für Makros, die **printf** aufrufen).

- Es gibt auch die Anweisung

  #undef ***name***

  Sie *löscht* eine bestehende Makro-Definition für *name* wieder: *name* wird ab dieser Stelle nicht mehr ersetzt, sondern bleibt unverändert im Programmtext.

- Jeder Compiler bietet eine Möglichkeit, Makros beim Compiler-Aufruf *auf der Befehlszeile zu definieren* (um denselben Code in unterschiedlichen Varianten kompilieren zu können, ohne die Files jedesmal zu modifizieren), beispielsweise durch eine Option **–D** .

#if

- Die Zeilen zwischen **#if** und **#endif** werden *entfernt* (bzw. übersprungen), wenn der Ausdruck nach dem **#if** (der konstant und vom Präprozessor berechenbar sein muss) den *Wert 0* ergibt, und sonst unverändert belassen.

  Auch Präprozessor-Anweisungen innerhalb eines **#if** mit Bedingung 0 werden ignoriert, d.h. **#define** und **#include** in einem falschen **#if** haben keine Wirkung.

- Im Normalfall prüft die Bedingung Präprozessor-Konstanten bzw. -Makro's. Dafür gibt es ein spezielles Konstrukt (nur für **#if**-Bedingungen):

  defined(*name***)**

  Es ergibt 1, wenn *name* zuvor mit **#define** *als Makro definiert* wurde, und 0 sonst.

  Für **#if defined(***name***)** und **#if !defined(***name***)** gibt es auch Kurzformen:

  #ifdef *name* und **#ifndef** *name*

  Beispiel:

```
#ifdef __DEBUG__
printf("Schleife i = %d\n", i);
#endif
```

- Es gibt auch ein **#elif** und ein **#else** mit der naheliegenden Bedeutung.
- **#if** können geschachtelt sein.
- Ein häufiger Trick *zum Auskommentieren von Codeteilen* ist, die entsprechenden Zeilen in **#if 0** … **#endif** einzuschließen:
  0 ist immer falsch, der Code wird immer entfernt. Das funktioniert im Unterschied zu **/*** … ***/** auch dann, wenn der auskommentierte Teil Kommentare enthält.

#error

#error *text gibt beim Kompilieren eine Fehlermeldung aus*, die u.a. *text* enthält.

Verwendet wird das Konstrukt beispielsweise, um bei widersprüchlichen oder undefinierten Kombinationen von **#if**-Bedingungen Fehler anzuzeigen.

Beispiel:

```
#if defined(BIG_ENDIAN)
...
#elif defined(LITTLE_ENDIAN)
...
#else
#error Please define BIG_ENDIAN or LITTLE_ENDIAN !
#endif
```

Vordefinierte Makros

Der C-Standard definiert einige Makros, die in jeder Implementierung mit fixen Werten vordefiniert sein müssen (z.B. **__STDC_VERSION__** : Es hat einen numerischen Wert, der die C-Standard-Version angibt, die der Compiler unterstützt).

Weiters gibt es einige vordefinierte *Pseudo-Makros*, die *durch einen variablen Wert ersetzt* werden: **__LINE__** durch die aktuelle Zeilennummer, **__FILE__** durch den aktuellen Filenamen (ändert sich innerhalb von #include's), **__DATE__** und **__TIME__** durch die aktuelle Zeit beim Kompilieren.

Ein Beispiel dafür ist das Makro **assert(…)**, das mit Hilfe von **__LINE__** und **__FILE__** ausgibt, an welcher Stelle im Code der Fehler passiert ist.

Daneben definiert jeder Compiler viele *systemspezifische Makros*, die den Compiler (z.B. **__GNUC__**), das Betriebssystem (z.B. **__linux__**), die Hardware-Plattform bzw. die Prozessor-Fähigkeiten (z.B. **__x86_64__** oder **__SSE2__**), die Breite von Datentypen (z.B. **__LP64__** = "Long's and Pointers are 64 Bit") und vieles mehr anzeigen und **#if**'s zur Unterscheidung verschiedener Plattformen in universellem Code erlauben.

18. Kapitel: Aufteilung in mehrere Source-Files

Bei größeren C-Projekten wird der *Quelltext üblicherweise auf mehrere Files verteilt*
(bei wirklich großen Projekten können das einige tausend / zehntausend Files sein!).

Dafür gibt es viele Gründe, u.a. die Folgenden:

- *Verteilung auf mehrere Programmierer*, Vermeidung von Bearbeitungs-Kollisionen:
  Jeder bekommt ein paar Files, an denen im Normalfall nur er allein arbeitet.
- *Modulare, strukturierte Entwicklung*: Jede Komponente kann für sich entworfen, implementiert und teilweise auch getestet werden.
- *Übersichtlichkeit:* Einzelne Files bleiben in überschaubarer Größe.
- *Klare Schnittstellen:* Interne Variablen und Funktionen einer Komponente werden File-**`static`** deklariert und nicht nach außen bekanntgegeben;
  andere Komponenten können nur auf jene Dinge zugreifen, die explizit dafür freigegeben sind.
- *Compilezeit-Ersparnis*: Bei Änderungen müssen nur jene Files frisch kompiliert werden, die wirklich geändert wurden, nicht der gesamte Source. Beim Kompilieren des gesamten Projektes können mehrere Compiler gleichzeitig (parallel) arbeiten.
- *Leichtere Fehlersuche:* Wenn man von einem funktionierenden Stand aller Files ausgegangen ist, können neue Fehler nur in den seitdem geänderten Files stecken.
- *Versionsverwaltung:* Versionsstand und Änderungsgeschichte jedes Files (und damit jeder Komponente) lassen sich getrennt ermitteln, eine einzelne Komponente lässt sich im Notfall auf einen älteren Stand zurücksetzen, ohne dadurch die Änderungen in anderen Komponenten zu verlieren.
- Die *automatische Dokumentation* kann pro Komponente generiert / aktualisiert werden.

Wie wird in mehrere `.c` / `.h`-Files aufgeteilt?

In C++ ist das einfach:

- Ein **`.h`**-File und ein dazugehöriger **`.c`**-File **pro Klasse**
  (wobei der **`.c`**-File ev. leer sein und damit entfallen kann),
  ev. weitere **`.c`**/**`.h`**-Paare für Code und Daten, die zu keiner Klasse gehören.
- Ein **`.c`**-File für **`main`**.
- Ev. weitere **`.h`**-Files für zentrale projektglobale Konstanten, Typen usw..

Da es in C keine Klassen gibt, muss die Aufteilung sinngemäß erfolgen:
Statt einem **`.c`**/**`.h`**-Paar pro Klasse gibt es dann eben ein **`.c`**/**`.h`**-Paar pro Teilprojekt / Datenstruktur / Lösungsschritt / ...

Was gehört in den `.c`-File, was in den `.h`-File?

Das ergibt sich aus dem Zweck von **`.c`**- und **`.h`**-Files:

Technisch:

- Ein **`.h`-File** wird immer nur beim Kompilieren von **`.c`**-Files **inkludiert**, nie eigenständig für sich allein kompiliert.
- Ein **`.c`-File** wird immer **kompiliert**, nie inkludiert.

Organisatorisch:

- Der **`.h`-File** enthält die **öffentliche Schnittstelle** einer Komponente, d.h. er deklariert die Funktionen, Typen, Konstanten usw., die *für andere Komponenten sicht- und verwendbar* sind:

  Jede Datei, die eine fremde Komponente nutzen will, inkludiert deren **`.h`**-File (ein **`.h`**-File wird daher typischerweise *von vielen Files in einem Projekt inkludiert*).

- Der **`.c`-File** enthält die **Implementierung** einer Komponente, d.h. den *Code* der Funktionen, den Speicherplatz der *globalen Variablen* (d.h. zu jeder **`extern`**-Variablendeklaration im **`.h`**-File gibt es eine korrespondierende Deklaration ohne **`extern`** im dazugehörigen **`.c`**-File), und alle Deklarationen und Definitionen, die für die anderen Nutzer der Komponente nicht zugänglich sein sollen (*interne Konstanten, Typen und Funktionen*).

  Ein **`.c`**-File wird *niemals inkludiert* und geht auch sonst in keiner Weise direkt in die anderen Komponenten ein (er wird daher den Entwicklern anderer Komponenten gar nicht zur Verfügung gestellt, sondern ist mehr oder weniger Privatsache desjenigen, der diese Komponente baut): Alles zur Nutzung einer fremden Komponente Erforderliche steht in ihrem **`.h`**-File, der **`.c`**-File einer fremden Komponente wird zur Entwicklung der eigenen Komponente nicht benötigt.

  Erst im letzten Schritt der Erstellung eines ausführbaren Programmes sammelt der Linker die **`.o`**-Files, das sind die fertig compilierten **`.c`**-Files, aller Komponenten zusammen und verbindet sie zu einem Executable (direkt oder auf dem Umweg über Libraries).

Daraus folgt:

- Ein **`.h`**-File darf nur **`#define`- oder `enum`-Konstanten**, **Typdeklarationen** (**`typedef`**, **`struct`**, …), **Funktionsprototypen**, **`extern`-Deklarationen** für Variablen usw. enthalten.

  Er darf **weder ausführbaren Code enthalten** (Ausnahme: **`inline`**-Funktionscode, lernen wir erst mit C++) **noch Speicherplatz anlegen**: Würde er das tun, würde der Linker einen Fehler liefern, weil er diesen Code bzw. Speicherplatz unter demselben Namen *mehrfach* vorfindet (in jedem **`.o`**-File, dessen **`.c`**-File diesen **`.h`**-File inkludiert hat).

  Daher gilt: Vor jeder Variablen-Deklaration und jedem Funktions-Prototyp in einem **`.h`**-File muss **`extern`** stehen (bei Prototypen ist es heute technisch meist nicht mehr nötig, aber trotzdem üblich und sinnvoll).

Auch organisatorisch gilt:

- Eine Änderung im **.h**-File erfordert technisch eine *Neukompilierung aller* **.c**-*Files*, die diesen **.h**-File direkt oder indirekt inkludieren, und meist auch eine manuelle Code-Anpassung in diesen Files (weil sich im **.h**-File vermutlich irgendein Typ oder Funktionsaufruf geändert hat, der in den inkludierenden **.c**-Files verwendet wird).

  .h-Files werden daher in großen Projekten frühzeitig festgelegt und eingefroren, nachträgliche Änderungen erfordern die Genehmigung der Projektleitung o.ä. und sollten möglichst selten passieren.
- Änderungen in **.c**-Files können hingegen jederzeit ohne direkte Auswirkungen auf andere Komponenten gemacht werden, sie ziehen keine Folge-Änderungen in anderen **.c**- oder **.h**-Files nach sich.

Wer inkludiert was?

- *Jeder* **.c**-*File und jeder* **.h**-*File inkludiert nur genau jene* **.h**-*Files, aus denen er selbst Dinge direkt verwendet* ("auf Verdacht" zu viel zu inkludieren ist unklug!).

  Es ist in **.c**-Files nicht notwendig, darüber nachzudenken, welche Header die inkludierten Header intern brauchen, und diese auch zu inkludieren (denn die **.h**-Files inkludieren jene **.h**-Files, die sie intern brauchen, ohnehin selbst).

  Andererseits ist es auch nicht klug, sich in einem **.c**-File auf indirekte **#include**'s zu verlassen und Dinge zu nutzen, die aus einem solchen indirekt inkludierten Header stammen, ohne den Header direkt zu inkludieren.
- *Weiters inkludiert jeder* **.c**-*File den eigenen* **.h**-*File!*
  (u.a. damit man merkt, wenn **.c**-File und **.h**-File nicht zusammenpassen!)
- *Achtung:*

  #include's für Dinge, die nur im **.c**-File einer Komponente verwendet werden, aber nicht in ihrem **.h**-File, gehören in den **.c**-File und nicht in den **.h**-File!

  Sonst würden alle Verwender der Komponente sehen, was die Komponente intern benötigt, und das auch jedesmal inkludieren und davon abhängig werden!

Was ist bei einem .h-File sonst noch zu beachten?

Aus dem vorigen Punkt folgt, dass ein **.h**-File vom selben **.c**-File beim Kompilieren möglicherweise **mehrfach inkludiert** wird: Neben einem direkten **#include** kann es auch beliebig viele indirekte geben. Das kann zu *Fehlermeldungen wegen mehrfacher Deklarationen* führen (vor allem in C++) und ist außerdem ineffizient.

Jeder .h-File muss daher dafür sorgen, dass ab dem *zweiten* **#include** *nur mehr leerer Inhalt inkludiert* wird.
Dazu muss jeder **.h**-File **myfile.h** gleich *ganz am Anfang folgendes Konstrukt enthalten*:

```
#ifndef _MYFILE_H
#define _MYFILE_H 1
```

(und ganz am Ende des Files das dazugehörige **#endif**)

Das bewirkt, dass der Inhalt des Files *übersprungen* wird (weil das **#if** "falsch" ergibt), wenn der File *schon einmal inkludiert* wurde.

19. Kapitel: Reste und Überbleibsel…

Bitfields in `struct`'s

Für die exakte Abbildung des Inhaltes eines Hardware-Registers in einer **`struct`** kann man nach jedem Member angeben, *wie viele Bits es belegen soll*. Auch *namenlose Füll-Member* zum Überspringen von Bits sind erlaubt. Member mit Bit-Breite müssen vom Typ **`int`** oder **`unsigned int`** sein. Zum Packen mehrerer kurzer **`int`**-Werte in einen 32- oder 64-Bit-Speicherplatz eignen sich **`struct`**'s mit Bitfields ebenfalls.

Beispiel:

```
struct sensorData {
    unsigned int sec : 6;
    unsigned int min : 6;
    unsigned int hour : 5;
    int : 5;                /* 5 unbenutzte Bits */
    int valid : 1;
    int error : 1;
    int data: 8;
};
```

Achtung: Zahlreiche Details von Bitfields (z.B. ob Anordnung von links oder rechts) sind *implementierungsabhängig*! In der Praxis werden Bitfields daher nur ungern verwendet, wenn der Code portabel sein soll: Eine Implementierung von Bit-Zugriffen mit normalen **`int`**'s und Bit-Operatoren ist betreffend Portierung robuster.

`struct`'s variabler Größe

Das *letzte* Member einer **`struct`** (und nur dieses!) darf ein *Array unbekannter Größe* (deklariert mit **`[]`**) sein. Es wird beim **`sizeof`** nicht mitgezählt, d.h. das **`sizeof`** liefert nur die Größe des fixen Teils der **`struct`** ohne diesem Array (plus ein eventuelles Alignment vor dem variablen Array).

Derartige **`struct`**-Variablen können nicht fix deklariert, sondern *nur dynamisch angelegt* werden: Man muss beim **`malloc`** die gewünsche Größe des Arrays (Element-Anzahl * Element-Größe) explizit zum **`sizeof`** der **`struct`** dazuzählen.

`union`

`union`'s sind wie **`struct`**'s, mit einem wesentlichen Unterschied: Sie enthalten nicht alle angegebenen Member, sondern *zu jedem Zeitpunkt nur ein Member*, kommen also zur Anwendung, wenn man in einer Variable *wahlweise Werte verschiedenen Typs* speichern will (und zwar *jeweils nur einen Wert*, außer, das **`union`**-Member ist wieder ein Array oder eine **`struct`**). Der Speicher, den der Compiler für eine **`union`** reserviert, ist daher nicht die Summe des Speicherplatzes für die Members, sondern nur der *Speicherplatz des größten Members*.

Achtung: **union**'s sind ***ungeprüft:***

Das Programm "weiß" zur Laufzeit nicht, zu welcher der Möglichkeiten der in der **union** aktuell gespeicherte Wert gehört. Es gibt keine Prüfung und daher auch keine Fehlermeldung, wenn man mit einem falschen **union**-Member auf den Wert zugreift (z.B. ein **double**-Member liest, obwohl ein Pointer-Member gespeichert ist). Es liegt in der Verantwortung des Programmierers, zu wissen, wann welches Member in welcher **union** enthalten ist, und richtig darauf zuzugreifen.

In 90 % aller Fälle wurden **union**'s als Members für verschiedene Varianten innerhalb einer **struct** verwendet, wobei sich aus einem anderen Member derselben **struct** herauslesen läßt, welche Variante gerade in der **union** gespeichert ist.
Diese Anwendung von **union**'s wird in der objektorientierten Programmierung durch abgeleitete Klassen ersetzt und ist daher heute kaum noch anzutreffen.

In manchen Fällen greift man allerdings bewusst mit dem "falschen" Typ zu, beispielsweise, um denselben **int**-Wert abwechselnd als **int** und als Array von vier einzelnen Bytes (**char**) zu betrachten oder einen **double** als 64 Bit langen **int**.

Bei einer Oldstyle-Initialisierung (ohne explizite Feldnamen) einer **union** muss ein Wert für das *erste* Member angegeben werden.

enum

enum deklariert **Aufzählungstypen**, d.h. Typen, deren Wertebereich aus einer *Menge namentlich benannter, konstanter Werte* besteht. Beispiel:

```
enum color {black, red, yellow, green, cyan, blue, magenta, white};
enum color foreGround = black, backGround = white;
```

enum-Konstanten haben **Typ int** und können überall verwendet werden, wo eine **int**-Zahl erwartet wird. Der Compiler weist ihnen **implizit fortlaufende Werte von 0 aufwärts** zu (oben also 0 für **black** bis 7 für **white**). Man kann allerdings andere Werte angeben (sogar gleiche und negative Werte sind erlaubt):

```
enum color {
      black = 0x000000,
      red = 0xff0000,
      yellow = 0xffff00,
      green = 0x00ff00,
      cyan = 0x00ffff,
      blue = 0x0000ff,
      magenta = 0xff00ff,
      white = 0xffffff
};
```

enum-Variablen sind (nur in C, nicht C++!) ebenfalls **int**-*kompatibel*, aber ihr genauer Typ bzw. ihre *Größe im Speicher* (ob **char**, **short** oder **int**) ist *implementierungsabhängig*.

Wenn es nur um die Konstanten und nicht um einen eigenen Typ geht, oder wenn das **enum** sowieso in einem **typedef** steht, kann man den Namen nach dem **enum** auch weglassen:

```
typedef enum {TRUE = 1, FALSE = 0} BOOLEAN;
```

`const`, `volatile` und `restrict`

In einer Variablen- oder Parameter-Deklaration kann man *vor dem Typ* drei Schlüsselworte angeben:

- **`const`:**

  `const` bedeutet, dass der Inhalt dieser Variablen **konstant** ist: Die Variable bekommt durch die **Initialisierung** in der Deklaration (oder bei Parametern durch die Argument-Übergabe) *einmal* einen Wert (eine **`const`**-Variablen-Deklaration ohne Initialisierung ist daher *sinnlos*), und dieser **darf nicht mehr geändert werden**:

  - Man darf auf eine **`const`**-Variable *nicht zuweisen*.
    Auch ++, −− usw. sind *verboten*.
  - Bei Pointern ist zu unterscheiden, ob *der Pointer selbst konstant* ist oder *der Wert, auf den er zeigt* (siehe Beispiel im Kapitel Pointer). "Pointer auf **`const`**" dürfen nur auf "Pointer auf **`const`**" zugewiesen oder als Argument übergeben werden, nicht auf normale Pointer (umgekehrt schon).
  - Das **&** einer **`const`**-Variablen liefert einen "Pointer auf **`const`**", d.h. man darf die Adresse einer **const**-Variable nur in einem "Pointer auf **`const`**" speichern, nicht in einem normalen Pointer. Daher darf man eine **`const`**-Variable auch nur an solche Funktionen "by reference" übergeben, bei denen der Parameter als "Pointer auf **`const`**" deklariert ist.
  - *String-Konstanten sind per Definition konstant*,
    d.h. haben den Typ **`const char []`** bzw. **`const char *`**.

- **`volatile`:**

  `volatile` (**"flüchtig"**) bedeutet, dass der Wert der Variable jederzeit "von außen" (d.h. ohne Wissen des Compilers) geändert werden kann. Beispiele sind:

  - Globale Variablen, die **durch parallel laufenden Code modifiziert** werden können.
  - Variablen, die nicht im Speicher liegen, sondern **Hardware-Register darstellen** (das realisiert man durch "Pointer auf **`volatile`**", die mit der Adresse der Hardware initialisiert werden).

  `volatile` *verbietet dem Compiler jegliche Optimierung* dieser Variablen: Jeder Zugriff auf eine **`volatile`**-Variable muss genau an der Stelle erfolgen, an der er im Programm steht. **`volatile`**-Variablen dürfen nicht in CPU-Registern oder temporären Zwischenvariablen gehalten werden, und es dürfen nicht mehrere Zugriffe darauf zusammengefasst oder aus Schleifen herausoptimiert werden.

- **`restrict`:**

  `restrict` ist nur *im Zusammenhang mit Pointern* relevant und sagt dem Compiler sinngemäß, dass er sich beim Optimieren darauf verlassen kann, dass *kein zweiter Pointer* auf denselben Wert zeigt. Das erlaubt **zusätzliche Optimierungen**, die ohne **`restrict`** unzulässig wären.

  `restrict` wird derzeit noch praktisch nie verwendet.